

# Programming Languages Through Translation

– A Tutorial Approach –

August 16, 2014

**Jerud J. Mead**

Computer Science Department

Bucknell University

Lewisburg, PA 17837

# Preface

## For the student

The primary goal of this text is to provide you with an introduction to the principles and practices of programming language translation or, as it is more commonly known, compiling. The mechanism for reaching this goal is a set of innovative, flexible, hands-on tutorials and projects that guide you to a practiced understanding of how a translator works, how a translator can be decomposed into components, and how those components work and cooperate to carry out the translation process. The material assumes you have taken or are currently taking a course in the principles of programming languages, and that you are familiar with object-oriented programming and Java.

The goals of the text are multifaceted, going beyond the primary goal of illuminating the inner workings of a compiler. Thus, as you make your way through the text you will profit in several ways.

- You will acquire practical models of programming language principles through a dynamic mix of study and implementation.
- In the carefully guided approach of the tutorials and projects, you will apply object oriented techniques to a large system. Through this experience you will emerge a more confident and capable (object oriented) programmer.
- You will see in every aspect of translator development the strong tie between theory and application. Most importantly, you will come to understand and appreciate the centrality of the target language's grammatical structure on translator structures.
- In studying code generation, you will gain valuable insights into the software/hardware boundary.
- Finally, by implementing a translator front end, you will gain a better understanding of the effective use of the compilers you use.

It is important that you will gain experience and understanding of language translation, not through a simple sequence of assigned problems, but by working through tutorials and projects that have been designed with specific goals in mind. Of central importance is the extensive tutorial in Chapters 11-15, which leads you through a self-paced implementation for a compiler front end. It is this tutorial which provides the basic implementation strategies that you will apply repeatedly in exercises and projects.

The goals of the text are realized through the following structural and topical aspects of this text.

- The structure of the text follows the structures of a translator, with the discussion of principles and implementation separated to provide flexibility in how your instructor uses the text. Though discussions of a principle and its implementation are separated, they are drawn together through the use of a simple programming language.

- Tutorials and projects are supported by an extensive software base. The use of this base is especially important in the topical tutorials in Chapters 12-15. The parser tutorial, for example, begins with the study of the implementation of a parser for a subset of the target language and ends with a guided sequence of activities through which you extend the supplied parser to a completed parser for the target language.
- Object oriented techniques are used throughout the text. While they are not the target of study, their consistent use in the text and in the tutorials and projects provides a valuable mechanism for learning. There are three object oriented features that distinguish the text.
  - All translator error handling (compiler error reporting) is implemented via exception handling. A strategy for defining an appropriate exception hierarchy is developed in the tutorial part of the Analysis Phase.
  - To facilitate a first line of debugging capability, an object oriented debugging facility is an integral part of each tutorial and project. This facility allows you to enable different sets of debugging statements in the system through command line arguments.
  - UML class diagrams are used throughout to describe the structure of whole systems and components.

## For the instructor

Back in the early 90's I realized that my students were learning abstract programming principles but having trouble linking them to the practical use of programming languages. It seemed that a simple language recognition or translation project would provide a unifying context within which the students would find real (implementation) models for programming principles and the behavior of compilers they use. It took 15 years of evolution for the tutorials to arrive at their current form.

The PDef tutorials (Chapters 11-15) allow the student to progress step-by-step through the implementation of a recognizer for the very simple language PDef, defined in the first tutorial chapter. In these tutorials students see explicitly how syntax is checked and how errors are identified and recovered from; they are lead through the development of an appropriate syntax tree structure and the generation of that structure by the parser; tutorial steps lead them through the development of syntax tree traversal algorithms for displaying the syntax tree and for static semantic checking.

It is the PDef tutorials that set the tone for the entire text, with principles being discussed at a higher level and then being implemented in the context of a specific recognizer or translator. The text is structured following the basic structure of a translator as follows.

### **Prelude :**

The prelude presents an overview of the nature of language translators and their structures as well as an overview of programming languages and the formal and informal mechanisms used for describing them. The remaining parts of the text are divided between the front-end and back-end.

### **Front End – Analysis Techniques :**

This part comprises seven chapters, the first of which defines the language PDef. The next four focus on the four critical components of an analyzer: tokenizer, parser, syntax tree, semantic checker. The first parsing chapter presents an overview of parsing techniques and then focuses on top-down parsing and the particular problems encountered in designing a recursive descent parser. The sixth chapter extends the earlier parsing chapter, discussing

LR parsing, including LR(0), SLR, and LALR. The final chapter introduces attribute grammars as a formal mechanism for describing static semantics.

### **Front End – Tutorials :**

The chapters in this part of the text are paired with those from “Front End – Analysis Techniques.” The introductory chapter, which describes the tutorials and defines the language PDef-*lite* (a subset of PDef), is followed by the four main tutorial chapters. The tokenizer tutorial applies earlier principles (from its paired tokenizer principles chapter) to design and implement a PDef tokenizer. The parser, syntax tree, and static semantics tutorials follow in a similar manner, each applying techniques from its earlier paired chapter and building on the work from the preceding tutorials. Each tutorial has two parts – the first part investigates the implementation of, for example, the parser for PDef-*lite*, while the second part, always titled **Guided Development**, lays out a sequence of activities for the student to extend the PDef-*lite* implementation to a PDef implementation: this basically involves adding arithmetic expressions to PDef-*lite*.

The final chapter is paired with the earlier LR Parsing chapter and discusses the implementation (from scratch) of an LR(0) parser for PDef and the strategy for modifying the parser so that the syntax tree is also generated.

### **Back End – Synthesis Techniques :**

Unlike the first half of the text, where the techniques chapters are separate from the implementation chapters, the Back End chapters are split in half, with the first half discussing principles and the last half implementation. The first two chapters of this part introduce the (very simple) functional language FP and discuss its implementation in the form of an interpreter. The second chapter discusses this implementation, focusing on code generation, whose target machine is a simple stack machine.

The last four chapters focus more specifically on code generation problems related to intermediate code generation, final code generation (for the MIPS architecture), and optimization techniques that are applied to intermediate code (common subexpression elimination) and to MIPS code (register allocation).

### **Back End – Compiler Project :**

This part of the text focuses on the imperative language IP and the implementation problems associated with translating it to MIPS code. The five chapters are very much project oriented and meant to lead the student through new problems such as mixed-mode arithmetic (integer and floating point) and reference parameters for function and procedure calls.

A final comment for the potential instructor. A text such as this requires a certain commitment in time and effort. But it is important to understand that the same structure, tutorials and projects, that facilitate learning in the student, will make the transition smoother. In addition, the supplied base of software, includes solutions to the FP and IP projects.

## **How to use the text**

There are several contexts in which this text can be used, varying from a standard programming languages course to a junior/senior level introduction to compiler course. It is important to make appropriate use of the software base available with the text. The importance of the PDef tutorials has been described earlier, but each project is also provided in two forms: a completed front end and a completed compiler. In this way, where appropriate, students can be given the front end for a project so that they can focus on

implementing the back end. Following are some possible course structures and the sequences of chapters that would be appropriate.

### **Programming Languages Course: Labs:**

The first half of the text can be used in a programming languages course, perhaps in a closed laboratory context. The expectation is that the labs would be scheduled to coincide (or trail) the coverage of corresponding material in lecture. Also, each lab would require the student to do some preparatory work from Chapters 1 and 2. The chapters would be covered in pairs as follows (a laboratory is assumed to be two hours):

Lab	Topic	Readings
1	tokenizer	preparation: Chapter 5, laboratory: Chapter 12
2	parser	preparation: Chapter 6, laboratory: Chapter 13
3	syntax tree	preparation: Chapter 7, laboratory: Chapter 14
4	semantic checker	preparation: Chapter 8, laboratory: Chapter 15

Experience indicates that Labs 3 and 4 might require two laboratory sessions each.

### **Programming Languages Course: Labs and Interpreter Project:**

The strategy in this context is to use the laboratories as described above and then to assign a semester long out of class course project – the implementation of an interpreter for FP. This project is discussed in Chapters 16 and 17. The labs and the project have been routinely assigned in my programming languages course for many years. The project can be laid out as follows: tokenizer (1 week), parser (1 week), syntax tree (2 weeks), semantic checker (2 weeks), code generation (2 weeks).

To facilitate this project I supply the code for the symbol table and for the code table and FP machine implementation. This means the students can focus on the critical programming languages issues. While the project times indicated above may seem short, they will have the experience of the tutorials to build on – new challenges are few and discussed at length where they occur.

### **Programming Languages and Language Translation Course:**

The course is an amalgamation of the programming languages course, with topics from programming languages integrated with compiling strategies. So the the Prelude and the PDef tutorials would make up the first portion of the course, with LR parsing and attribute grammars being optional topics. Selected topics from Back End – Synthesis Techniques would complete the course. With the example of FP, an introduction to the language Haskell would be an appropriate final topic for the course.

### **Compiler Course I:**

This course is an elective level course where the students have no prior experience with language translation. In this case it should be possible to work through the Analysis Phase in the first half of the course, having the students do the PDef tutorials on their own outside of class. The instructor can then choose a path through the Synthesis Phase that suits the kinds of problems they want to deal with. The FP interpreter could be ignored in favor of the MIPS implementation.

### **Compiler Course II:**

If the course is to be more aggressive, then students can cover the Analysis Phase more quickly and be provided with a front-end for an IP translator. In this way more than half of the semester might be applied to intermediate code generation and optimization, and object code generation and optimization.

**Senior Project Course:**

This course can follow one of the previous two structures, but focus more directly on the object oriented design the software.

It will not have escaped the reader's attention that there is no mention of compiler tools, such as YACC and LEX or their relatives. The reason is in line with the goal of the text, that students understand what goes on inside the compiler. This is gained, not by having a tool generate the parser and syntax tree, but by having the student write the implementation themselves. It may be unusual to do a hand implementation of an LR parser, but doing so gives a good understanding of the technique and why it is so convenient for automatic generation. Students with this experience will have solid models in mind that will serve as a basis for the more complex issues addressed in a graduate level compiler course.



# Contents

Preface	i
Prelude	1
<b>1 How a Translator Works</b>	<b>3</b>
1.1 Variations in Programming Languages . . . . .	3
1.2 Variations in Translators . . . . .	5
1.3 Translators: An Intuitive View . . . . .	7
1.4 Translators: A Data-Driven View . . . . .	10
1.5 Translators: An Object-Oriented View . . . . .	11
<b>2 Describing Languages</b>	<b>17</b>
2.1 An Informal View of Language . . . . .	17
2.2 A Formal View of Syntax . . . . .	22
2.3 Problems of Ambiguity . . . . .	36
2.4 A Formal View of Semantics . . . . .	40
<b>The Front-end – Analysis Principles</b>	<b>50</b>
<b>3 Introduction</b>	<b>53</b>
<b>4 An Example Language – PDef</b>	<b>57</b>
4.1 Informal Definition of PDef . . . . .	57
4.2 PDef Syntax . . . . .	58
4.3 PDef Static Semantics . . . . .	60
<b>5 Lexical Analysis – Theory and Practice</b>	<b>63</b>
5.1 Finite-state Machines . . . . .	63
5.2 Designing a Finite-state Machine . . . . .	66
5.3 Implementing a Tokenizer . . . . .	69



<b>6</b>	<b>Syntax Analysis – Theory and Practice</b>	<b>73</b>
6.1	An Overview of Parsing Techniques . . . . .	73
6.2	Top-down Parsing . . . . .	82
6.3	Parsing Structured Lists . . . . .	92
6.4	Implementing a Parser . . . . .	94
<b>7</b>	<b>Syntax Tree – Theory and Practice</b>	<b>97</b>
7.1	An Overview . . . . .	97
7.2	A Syntax Tree Example . . . . .	98
7.3	Designing Syntax Tree Nodes . . . . .	100
7.4	Building a Syntax Tree . . . . .	108
7.5	Syntax Tree Traversal . . . . .	116
<b>8</b>	<b>Semantic Analysis – Theory and Practice</b>	<b>121</b>
8.1	An Overview . . . . .	121
8.2	Symbol Table Structure . . . . .	126
8.3	Semantic Checking . . . . .	132
<b>9</b>	<b>LR Parsing – Theory and Practice</b>	<b>137</b>
9.1	Understanding LR Parsing . . . . .	137
9.2	A Machine for LR Parsing . . . . .	142
9.3	SLR Parsing . . . . .	154
9.4	Guided Development . . . . .	161
<b>10</b>	<b>Attribute Grammars – Theory and Practice</b>	<b>163</b>
10.1	Overview . . . . .	163
10.2	Attributes and Attribute Migration . . . . .	165
10.3	Synthesized Attributes . . . . .	167
10.4	Inherited Attributes . . . . .	169
10.5	Static Semantics via Attribute Grammars . . . . .	171
	<b>The Front-end – Analysis Tutorials</b>	<b>177</b>
<b>11</b>	<b>Tutorial Overview</b>	<b>179</b>
11.1	The Language PDef- <i>lite</i> . . . . .	180
11.2	Programming Strategies . . . . .	181
<b>12</b>	<b>PDef Tokenizer Tutorial</b>	<b>189</b>
12.1	Tokenizer Structure . . . . .	190

12.2	The Supplied Java Code . . . . .	198
12.3	Completing the PDef- <i>lite</i> Tokenizer . . . . .	205
12.4	Guided Development – PDef Tokenizer . . . . .	209
<b>13</b>	<b>PDef Parser Tutorial</b>	<b>217</b>
13.1	Parser Structure . . . . .	219
13.2	The Supplied Java Code . . . . .	221
13.3	Dealing with Parse Errors . . . . .	225
13.4	Recovering from Parse Errors . . . . .	231
13.5	Guided Development – PDef Parser . . . . .	237
<b>14</b>	<b>PDef Syntax Tree Tutorial</b>	<b>239</b>
14.1	Supplied Java Code . . . . .	240
14.2	Syntax Tree Structure . . . . .	243
14.3	Traversing the Syntax Tree . . . . .	247
14.4	Enhancements to the Class <code>Parser</code> . . . . .	248
14.5	Pretty-printing for the Syntax Tree . . . . .	250
14.6	Guided Development – PDef Syntax Tree . . . . .	253
<b>15</b>	<b>PDef Semantics Tutorial</b>	<b>259</b>
15.1	Symbol Table Structure . . . . .	261
15.2	Extending the Exception Structure . . . . .	264
15.3	The Supplied Java Code . . . . .	266
15.4	Implementing Semantic Checking . . . . .	269
15.5	Guided Development – PDef Semantics . . . . .	276
<b>16</b>	<b>PDef LR Parser Tutorial</b>	<b>281</b>
16.1	The Design of an LR Parser . . . . .	281
16.2	The Supplied Java Code . . . . .	283
16.3	Error Handling and Recovery . . . . .	294
16.4	Guided Development – LR Parser for PDef . . . . .	294
16.5	Syntax Tree Generation by an LR Parser . . . . .	295
16.6	The Supplied Java Code . . . . .	302
16.7	Guided Development . . . . .	306
	<b>The Back-end – Synthesis Principles</b>	<b>308</b>
<b>17</b>	<b>Introduction</b>	<b>311</b>

<b>18 An Example Language – FP</b>	<b>313</b>
18.1 FP Syntax . . . . .	314
18.2 FP Static Semantics . . . . .	316
18.3 FP Dynamic Semantics . . . . .	318
<b>19 An FP Interpreter</b>	<b>327</b>
19.1 Syntax Analysis . . . . .	327
19.2 Static Semantic Analysis . . . . .	334
19.3 Synthesis . . . . .	339
19.4 Generating Linear Code . . . . .	345
<b>20 Intermediate Code Generation</b>	<b>357</b>
20.1 Three-address Code . . . . .	358
20.2 Code Optimization – Eliminating Common Sub-expressions . . . . .	369
<b>21 FP<sup>+</sup> to MIPS Code Generation</b>	<b>385</b>
21.1 Translating Three-address code to MIPS Assembler Code . . . . .	385
21.2 Generating the MIPS Assembler Code . . . . .	389
<b>22 MIPS Code Optimization</b>	<b>395</b>
22.1 Strategies for Register Allocation . . . . .	395
22.2 Implementing Register Allocation . . . . .	405
<b>The Back-end – Compiler Project</b>	<b>417</b>
<b>23 Project Overview</b>	<b>419</b>
23.1 The Language IP . . . . .	419
23.2 Project Overview . . . . .	424
<b>24 Syntax and Static Semantics of IP</b>	<b>427</b>
24.1 Parsing IP . . . . .	427
24.2 IP Syntax Tree Hierarchy . . . . .	431
24.3 Static Semantic Checking . . . . .	434
<b>25 IP Intermediate Code Generation</b>	<b>437</b>
25.1 Phase 1 . . . . .	437
25.2 Phase 2 . . . . .	445
25.3 Phase 3 . . . . .	449
<b>26 IP Common Subexpression Elimination</b>	<b>455</b>

26.1 Step 1 – Extending FP <sup>+</sup> Elimination to IP Expressions . . . . .	455
26.2 Step 2 – Extending Subexpression Elimination I . . . . .	458
26.3 Step 3 – Extending Subexpression Elimination II . . . . .	463
<b>27 IP Object Code Generation</b>	<b>473</b>
27.1 Phase 1 – The Base Implementation . . . . .	473
27.2 Phase 2 – Adding Reference Parameters . . . . .	475
27.3 Phase 3 – Integrating Floating-point Capabilities . . . . .	475
27.4 Peephole Optimizations . . . . .	480
<b>Appendices</b>	<b>482</b>
<b>A Parse Table for FP</b>	<b>483</b>
<b>B FP Interpreter Driver Code</b>	<b>489</b>
<b>C Review of MIPS Assembly Language</b>	<b>493</b>
<b>Index</b>	<b>499</b>

# Prelude



# Chapter 1

## How a Translator Works

In the computing context, a *translator* is a program or component whose job it is to convert data in a source language to an equivalent form in a target language. Compilers and interpreters are the best known, but by no means the only, translators in the the programmer’s toolkit. You have probably had quite a bit of experience using one or more of these fascinating software systems, most likely in the form of a C++ or Java compiler.

While compilers and interpreters are well known, other common language translators almost escape notice. If you ask your web browser to print a page, some component will have to translate the HTML description of the target page into an equivalent form required by your printer (pdf form is common). Translation from one file format to another is also very common, for example converting jpeg graphics files into eps or png form, movie files from divx to dvd form. Text editors associated with computing environments, such as Emacs or the Eclipse editor, also contain translators: the user program code is input from the keyboard or a file as a sequence of characters and the editing system displays the data in a useful way on the screen, using syntax coloring and indentation.

Programming language translators have been studied for many years and their structure is well understood. In this chapter we will look at the functioning of a translator with the goal to expose its internal structure. Our examination will be driven first by an intuitive view of translation as a sequence of activities. Then we will turn to a more detailed, data-driven view of the translator structure, in which we examine how the transformation of the input data can help add clarity to our understanding of a translator’s activities. Finally, because we are ultimately interested in how to implement a translator, we will apply the data-driven view to understand the object-oriented structure of a translator.

### 1.1 Variations in Programming Languages

We will start with a brief step back. A modern computer is an electronic device whose purpose is to solve problems by taking a set of input data and computing a result. We will use the term *computation* to refer to the process carried out by the computer in solving a problem. This notion of computation is obviously very broad and would include a simple program for computing the standard deviation of a set of data and a complex program running on a mobile phone to determine the phone’s current location. Every video game can be described as a computation. So the purpose of a programming language is to facilitate writing descriptions of computations – we casually refer to “writing a description of a computation” as programming.

In the very early days when computers were newly on the scene, programming a computer involved, at

first, rearranging connectors in the processing unit to hard-wire a particular sequence of machine instructions. Even after the introduction of automatic loading of programs (from paper tape, for example), the programs were still written in a binary form (perhaps octal or hexadecimal<sup>1</sup>).

*Assembly language* was an important advancement because it freed programmers from having to deal with hardware details such as machine instruction codes and physical memory addresses. In an assembly language program we would see mnemonic names rather than binary machine instructions, and variable names rather than binary memory addresses. An *assembler* is a program which processes the assembly language program description and produces a (binary) machine language form that can be loaded directly into memory and executed. The assembly language makes it easier for programmers to develop correct programs because the assembler can catch errors that would otherwise have to be caught at run-time. In a certain sense, the assembler acts as a filter, filtering out invalid programs and letting pass those programs that meet its particular correctness standards.

Assembly language is not perfect. First, each processor architecture requires its own assembly language, which means problems solved for an Intel architecture would have to be completely re-written for a MIPS architecture in a MIPS assembly language. So portability is not a strength of assembly languages. Second, assembler programmers must keep track of all variable names used and must make sure that data stored in the variables is properly manipulated. If a floating point add operation is applied (by mistake) to two values stored in integer format, then the addition will be carried out assuming the values are in floating point form – the result will obviously be incorrect. In addition to data manipulation problems, assembly programmers must manage the logical sequencing of program statements via appropriately applied branch or jump instructions. These control flow problems can be quite difficult to track down.

### 1.1.1 High-level Languages

While assembly languages are still used in certain situations, writing device drivers, for example, most programming today is carried out using what are called *high-level languages*. The idea behind these languages is to decouple the language from the underlying execution hardware. This decoupling facilitates not only a more effective programming capability but also portability across hardware platforms.

Decoupling also makes it possible to implement languages based on alternative computational models. The most familiar model is the imperative programming model in which a computation is described as a sequence of instructions. Java, C++, and Fortran are examples of imperative programming languages. An important characteristic of imperative languages is that they are used to describe how a computation is to be carried out – i.e., what is the sequence of instructions whose execution will correctly solve the problem at hand.

So the most common and widely used programming languages facilitate describing the *how* of a computation. But this suggests the possibility that we could also describe the *what* of the computation. That is, describe what is to be computed rather than how it is to be computed. Such languages are said to be *declarative*. The declarative languages can be naturally divided into two categories determined by the computation model. With a functional programming language a computation is described in terms of two things: a set of function definitions and an expression whose evaluation will yield the desired result – importantly, the expression is written making use of the functions defined as part of the program. Example languages are Lisp and Haskell. The other computation model comes from mathematical logic and is based on mathematical relations rather than mathematical functions. This model is referred to as *logic programming*. Data base query languages are often based on this model. A query is really a request for those data entries which satisfy a particular relationship. So in logic programming, the “what” of a computation is

---

<sup>1</sup> *Octal* refers to numbers represented using base-8 notation, while *hexadecimal* (hex for short) refers to base-16 notation.



described by a list of relation definitions and the computation is driven by a query. Notice in both the functional and logic programming models the focus is on “what” is to be computed; exactly “how” it will be done doesn’t enter in.

The programming languages that exist, regardless of the computation model, are not natural languages as are our spoken languages. While natural languages have evolved to their current forms, programming languages are carefully designed and formally defined. The design decisions have two purposes. First, the language must be able to express all computations, so the design is dictated by the particular computation model. Second, languages are designed to facilitate two opposite goals. The language should be flexible to allow the programmer as much expressive power as possible. But the language should also restrict the programmer so that certain kinds of programming errors are either prevented or at least made less likely to occur. In the next section we will formalize these design decisions in the form of principles that guide the design of translators for these programming languages.

## 1.2 Variations in Translators

Programming languages, which we have associated with three computation models, can also be divided into categories based on the nature of the target execution environment. A language is *compiled* if the target is a physical processor. In this case the translated form will be in the machine language of the processor. The target execution environment can also be a virtual machine – a software system – and in this case the language is *interpreted*.

Three examples will be illustrative. C++ is a compiled language. If you develop a program on an Intel based machine and then compile it for execution on that same machine, the executable form will be a file containing an equivalent machine language program. In order to execute the program you ask the operating system to run the program in the particular file. In this case once the compiler produces the executable form, it has no other part to play in the execution of the program.

The functional language Haskell, on the other hand, is an interpreted language. The program is a sequence of function definitions and an expression that defines what is to be computed. The translator will translate each function definition into a form that can be executed on the Haskell execution environment – these translated functions are stored in a table. The translator translates the expression in the same way and then, assuming no errors, executes the expression’s translated form on the Haskell execution environment. A Haskell environment is usually an interactive one, where the function definitions are translated initially and then the environment enters a cycle where the user can enter an expression that the environment will translate and execute – this cycle can repeat as long as the user has expressions to evaluate (the same expression form but with different arguments, for example).

These two examples show the tradeoffs between compiled and interpreted languages. The Haskell environment provides a significant level of flexibility, in that you can enter the program and then interactively evaluate various expressions of interest. It would also be possible, in such an environment, to edit or delete functions or add new ones during the execution session. On the other hand, everything done after translation must be done with the cooperation of the Haskell environment which, remember, is an executing program. This scheme is far less efficient execution-time wise. The compiled C++ program is already in machine code and has probably been optimized to make effective use of hardware features (general purpose registers, for example). The C++ program execution provides none of the flexibility of the interpreted language.

A third possibility is to combine the two environments. Java does this. Java is interpreted in the sense that the Java translator generates an executable form in terms of the Java virtual machine. At execution

time the virtual machine acts as a kind of controller of execution, facilitating additional error checking and late binding. But when a segment of code is to be executed, it is compiled to machine code and that form is saved in case it is executed again – this is known as just-in-time compilation.

One important point to remember here is that whether working with a compiled, interpreted, or hybrid language, the translator must still provide the same functionality: analyze the source program for static (pre-runtime) errors and, if none are found, generate an appropriate executable form. In the next section we consider principles that should help us understand how to design and implement programming language translators.

### 1.2.1 Implications for Language Designers

Regardless of the nature of the programming language, the user of a translator should have certain expectations for the behavior of the translator. In this section we look at four principles that guide the design and implementation of translators. These *translator design principles* will guide our discussions in the remainder of this book.

#### The Correctness Principle

*The runtime behavior of a translated form must be that described by the input being translated.*

This principle is an absolute requirement. It is clear that when you write a program and understand its behavior you expect the translated form to behave in that understood way. The real problem in guaranteeing this principle is having an accurate definition of the language being translated – a definition that is clearly understood by the programmer (user) and the implementor of the translator. Since clarity in language definition is critical we will have to rely on language definition techniques that are as formal (theoretical) as possible. The formal mechanisms for describing the syntax and semantics of a programming language are an important thread through this book.

#### The Early-warning Principle

*Both syntax and semantic errors should be identified and reported at the earliest possible point in the translation process.*

The whole idea of the translator early warning capability is to find a maximum of errors at translation time. This is critical because finding errors at run-time is difficult and not guaranteed to be complete. It is not surprising that syntax errors must be eliminated before translation can be carried out. But once the syntax errors have been identified and corrected, code could be generated. This is where the early warning system digs a bit deeper and finds errors that are not errors of form, but errors of use – i.e., semantic errors. So the early warning capability of a translator needs to extend error checking from the syntactic level into the semantic level to the extent possible.

Languages can be categorized according to the time (either translation or execution) when a type error can be identified – a process called *type checking*. Languages designed for translation time type checking are said to be *statically typed*, while languages requiring execution time type checking are said to be *dynamically typed*. So translators for statically typed languages will provide greater early-warning support than those for dynamically typed languages.

### The Efficiency Principle

*A translator must insure that a translated form makes sensible and efficient use of the computational resources in the execution environment.*

If a program is translated in a straightforward way there are certain inefficiencies that will appear in the translated form. For example, if the expression  $(a + b)$  appears several times, where the values of  $a$  and  $b$  remain unchanged, the value of the expression will be recalculated each time. It would be better to reuse the already calculated value. Another related problem has to do with strategies for using general purpose processor registers. So if the value  $a$  is used several times, it would be best for it to remain in the same register, rather than being loaded from memory for each use. The translator must be designed with these efficiency problems in mind.

### The Portability Principle

*A translator should be designed so that it can be ported to a new execution environment with a minimum of effort.*

The Efficiency Principle implies that specific hardware features be taken into account during translation. But some optimizations, such as multiple appearances of the same expression, are hardware independent. This means that the translator would be best designed to do hardware-independent optimization first, and hardware-dependent optimization last. If we do this, then the cost of porting a translator to a new hardware environment should be minimized. Imagine we have a Java implementation of a C++ compiler for an Intel processor, but that a new compiler is required for a system running on an ARM processor. If the ARM system has Java already installed, then the Java C++ compiler should execute on the new system. But of course the executable file will have Intel not ARM code. To finish the porting of the Java C++ compiler will simply require the recoding of those components that generate the processor-specific code; the rest of the compiler will not have to be changed. Delaying the generation of processor code will be important when we talk about code generation.

## 1.3 Translators: An Intuitive View

Because our interest is in understanding the general characteristics of compilers, we want to focus on the most common language settings. Since the most common languages are high-level, imperative, and statically typed, we will be focusing on the properties of their compilers. From this point on we will assume in our discussions languages with these characteristics – the standard examples being C++ and Java. We now want to see what our intuitive understanding of compilers can tell us about their structure.

When you compile a C++ program or a Java program you expect the compiler to produce one of two results: it will either generate error messages or a translated version of the program code. These two outcomes are characteristic of translators generally, reflecting the fact that a translator works as a sequence of two phases.

In the first phase, the *analysis phase*, the source code is analyzed to see if it conforms to requirements of form: e.g., are program statements formed correctly, are semi-colons in the right places, are identifiers declared and used appropriately? The second phase, the *synthesis phase*, carries out the actual translation activity, synthesizing an executable form whose behavior is equivalent to that inherent in the source code. This two-phase structure is illustrated in Figure 1.1.

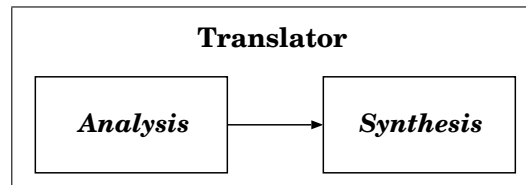


Figure 1.1: The Two Phases of Translation

## The Analysis Phase

The analysis phase, guided primarily by the Correctness and Early-warning Principles, focuses on two aspects of program structure that can be determined through the analysis of the source code itself: syntax and static semantics. *Syntax analysis* is based on two linguistic structures: words and phrases. The words of a programming language, also called *tokens*, correspond to the smallest character sequences that have meaning — these include punctuation and operation symbols, key-words, literal values, and identifiers (names). Phrases are sequences of tokens in well-defined structures. Syntax analysis focuses on these two linguistic structures, identifying character strings that aren't tokens and identifying token sequences that are not allowed in the source language. If no such errors (*syntax errors*) are detected then *semantic analysis* is carried out.

*Semantic analysis*, which we know more commonly as **type checking** focuses on the names (identifiers) and literal values appearing in source code and checks to see that they are used in ways allowed by the definition of the source language. These semantic properties are singled out because, while they can be analyzed in the context of the source code, they cannot be analyzed with the same techniques used in syntax analysis. Semantic errors are of two basic kinds. First, are identifiers declared in accordance with the source language requirements and are identifiers and literal values used in phrases in accordance with the source language requirements? Second, are there execution-time problems that can be identified by analyzing the source code; for example, is there code that cannot be reached, are there variables that have not been initialized? The diagram in Figure 1.2 depicts this more detailed two-phase structure of the analysis phase.

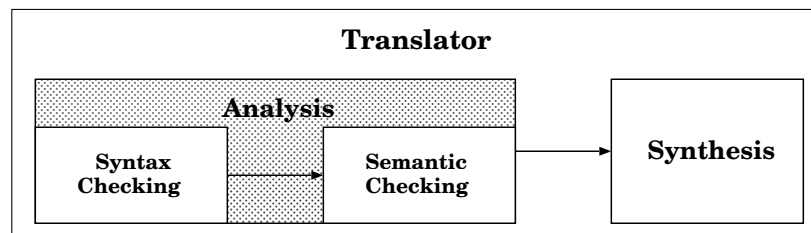


Figure 1.2: Recognition Phase Expanded

The syntax analysis phase can be split, not surprisingly, into two phases corresponding to the analysis of tokens and phrases. The first phase, called *lexical analysis*, identifies tokens in the source code; this identification process, of course, also identifies *lexical errors*, a sequence of characters corresponding to no source language token. The second phase retains the name *syntax analysis*. It analyses the source code as a sequence of tokens, looking for errors of form. Our diagram, then, can be redrawn as in Figure 1.3.

Each of the three phases represented in Figure 1.3, lexical analysis, syntax analysis, and semantic analysis, has a name associated with the checking process: *tokenizing* for lexical analysis, *parsing* for

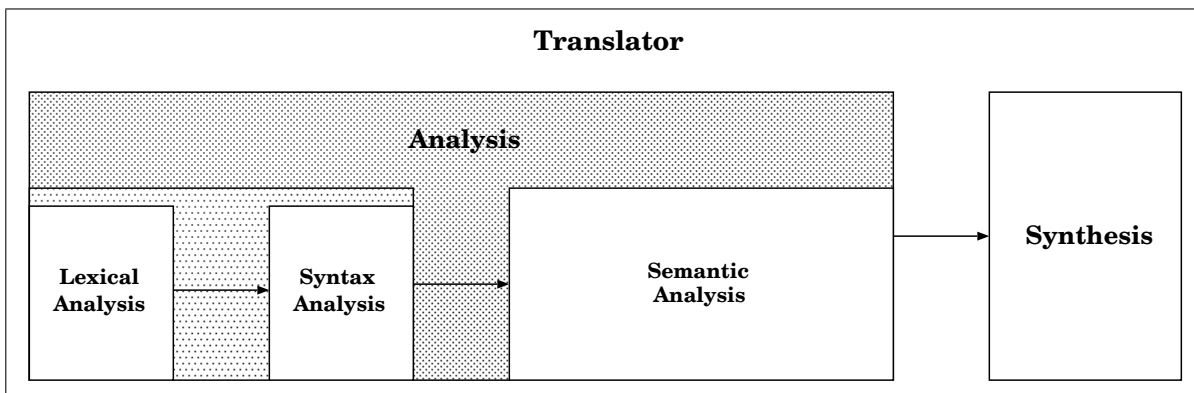


Figure 1.3: Sequence of Phases for a Translator

syntax analysis, and *semantic checking* for semantic analysis. Each of these processes will ultimately be associated with a particular component of a language translator front-end.

### The Synthesis Phase

The synthesis phase begins once the analysis phase has found the source code to be free of lexical, syntactic, and semantic errors. The synthesis phase, guided by the Correctness, Efficiency, and Portability Principles, produces an executable form whose behavior is equivalent to that inherent in the source code. This is a clear reflection of the Correctness Principle (see the Translator Principles in Figure 1.2.1). But the second Translator Principle, the Efficiency Principle, is important as well. The Efficiency Principle says that care must be taken in synthesizing an executable form so that the executable form makes efficient use of its computational resources.

The Correctness and Efficiency principles highlight the two key phases in synthesis – *code generation* and *code optimization*. The diagram in Figure 1.4 shows the translator phases, with the synthesis phase expanded into these two sub-phases.

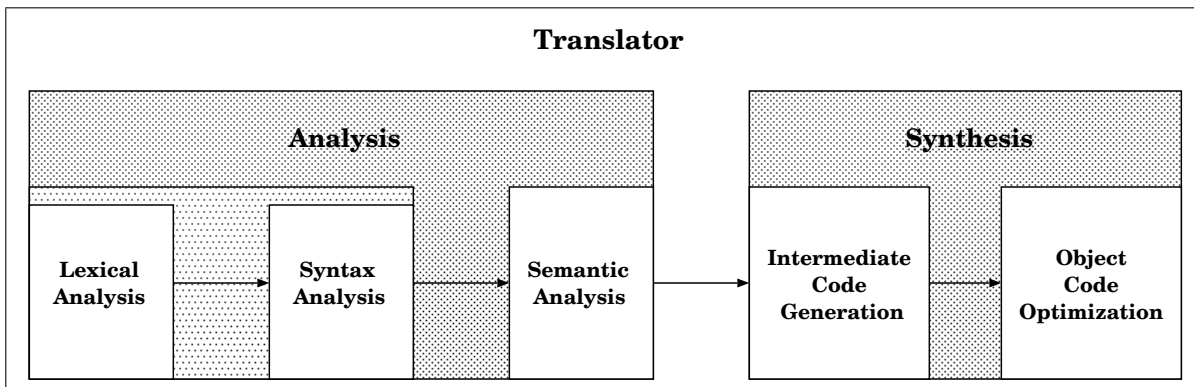


Figure 1.4: Synthesis Phase Expanded

## 1.4 Translators: A Data-Driven View

A translator transforms the input source program into a final executable form. Put in a perhaps more familiar way, the translator converts the input character stream (the source program) to an equivalent program executable on some target machine. What is not clear from the “phases” analysis of the last section is how that transformation from source program to executable form actually takes place. In this section we will look at a translator as a sequence of transformations, with each transformation generating a new representation of the original source program.

The sequence of transformations can also be seen in terms of data flow between the translator phases of the previous section. The question is, if we look at the diagram in Figure 1.4, what transformation takes place in each phase? What new form does the program take after each transformation? We will answer this question for each phase in turn.

**Tokenizer:** The tokenizer, which implements the lexical phase, converts the character-based input stream into a token-based stream. While the phase diagram may give the impression that the source program is converted to token form before the syntax analysis begins, in fact, the tokenizer works in synchrony with the second phase, providing the next token when it is requested.

**Parser:** The parser has two basic responsibilities. First, it checks that the tokens it reads from the token stream conform to the grammatical structure of the input language. Second, it generates a *syntax tree* and *symbol table*. The syntax tree is an internal representation of the content and structure of the input stream. The content can include identifier names as well as constant values, numeric values, for example. The symbol table is a table containing entries for the identifiers appearing in the input. The symbol table is implemented to make looking up names easier – without the symbol table the syntax tree would have to be searched, which is far less efficient than a table lookup.

**Semantic Checker:** The semantic checker checks that the data in the syntax tree and symbol table appear in accordance with the language’s static semantic rules. So while this phase doesn’t actually transform the syntax tree and symbol table, it does guarantee to the synthesis phase that the input is syntactically and static semantically correct.

**Intermediate Code Generator:** The *intermediate code generator* uses the data and structure represented in the syntax tree and symbol table to generate an intermediate form that can be executed on a virtual machine. The virtual machine architecture is chosen to provide a convenient half-way point toward a final executable form. Intermediate code generation is also a convenient place to take care of code optimizations which are independent of the final target machine architecture.

**Object Code Generator:** The *object code generator* translates the intermediate code to a final executable form called *object code*. This translation phase performs both code generation and optimization, taking into account the characteristics of the target processor architecture.

The sequence of transformations just described is summarized in Figure 1.5, which is an augmented version of Figure 1.4. In the diagram the fat arrows represent the direction of data flow while the skinny arrows represent, as before, the sequencing of phases in the translation process. It should be mentioned that the diagram does not reflect the complexities that can occur in code generation and optimization. There may be other intermediate forms and data structures created in the course of code optimization.

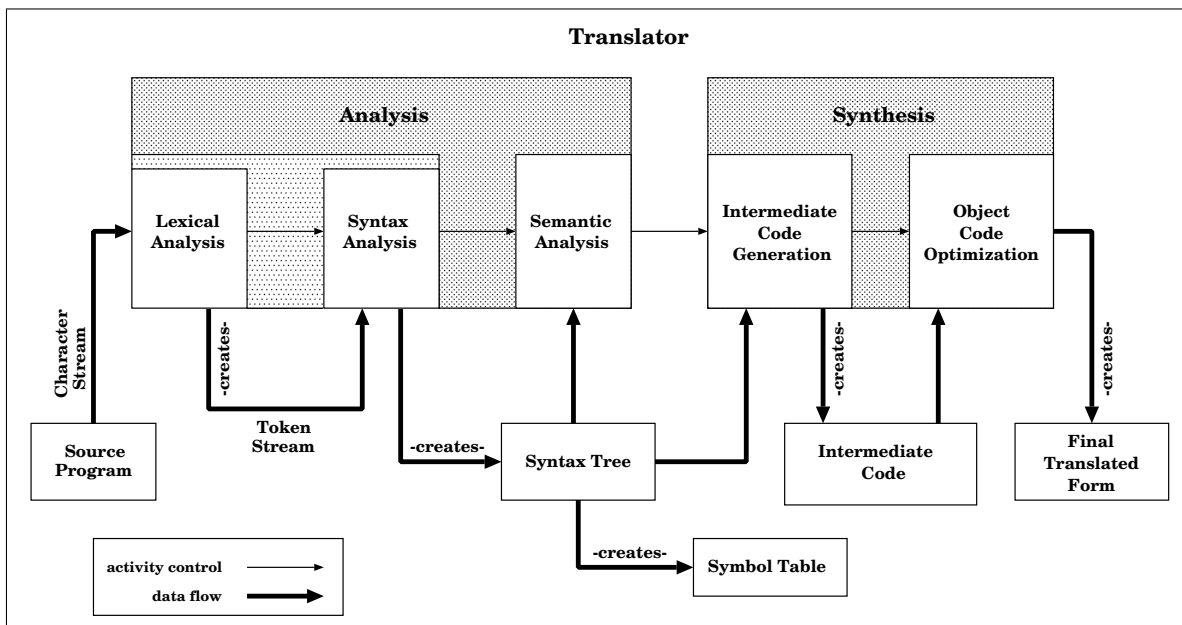


Figure 1.5: A Data Flow View of Translation

## 1.5 Translators: An Object-Oriented View

To this point we have looked at the translation process from two perspectives, the intuitive point of view and the data-driven point of view. If we are to progress to the implementation of a translator, however, we should consider an additional view, which will bring us closer to an implementation structure. Accordingly, this section presents an object-oriented view of a translator. Remember that in discussing an implementation we use the terms front-end and back-end for the implementations of the analysis and synthesis phases.

The data-driven view of the translator, given in the previous section, draws attention to the active and passive agents in a translator and their interaction mechanisms. These agents and their interactions are key to the object oriented structure we seek. We will look first at the object structure for the front-end (the analysis phase) and then the back-end (the synthesis phase).

### 1.5.1 Front-end Structure

We begin by focusing on the syntax analysis. The two active agents in this context are the tokenizer and the parser, with the parser being the controlling agent. That is, it is the parser that initiates all activity. When the parser needs to see a token, it sends a request for one to the tokenizer. The tokenizer, in turn, requests a sequence of characters from the input stream and returns the next (longest) token to the parser. This activity implies a structure of four classes: input stream, token, tokenizer, and parser.

The simple UML class diagram in Figure 1.6 shows the relationships among these four classes and also details an access mechanism for all but the token object, whose structure will be described shortly. The methods included in the diagram indicate the public interface of each class. In a real translator the front-end is activated via parser method `parseProgram`; the parser requests the next token via the tokenizer method `getNextToken`, and the tokenizer retrieves the next character from the input stream via the input stream method `getChar`.

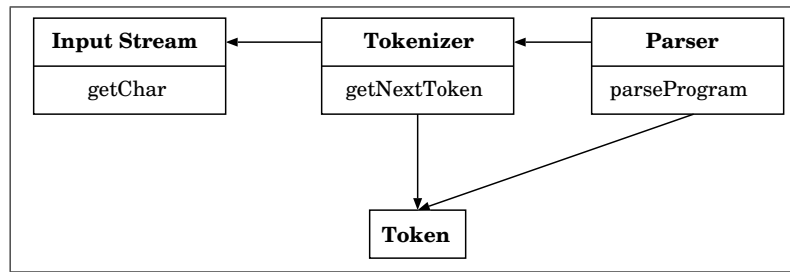


Figure 1.6: Object Structure for the Syntax Phase

The token class is a kind of class, which occurs commonly in object oriented programs, that takes data from an input source. Often a program's first responsibility is to take input and turn it into a form that is useful in the program context. In the case of the token class, we input data, in the form of strings of characters, and want to store them in a form that reflects the keywords, punctuation symbols, literals and identifiers we know in the program code. The token class, then, abstracts a type of data which represents these textual program elements. Each instance of a token, then, should contain within it the string it represents (the characters that make up the identifier, for example) and a name that the program can use to identify the particular kind of token it is, identifier, literal, etc. The 'token type' of a token value can be used by the parser to determine if this token is in a syntactically acceptable position while the 'token name' (the string of characters) can be used by the semantic checker to insure that the token is used appropriately in the syntactic structure. In order to be able to make useful error messages (think Early-warning Principle) it is also useful to know where in the input stream (line number and position) a particular token appears – this facilitates locating the position of an error in the program text.

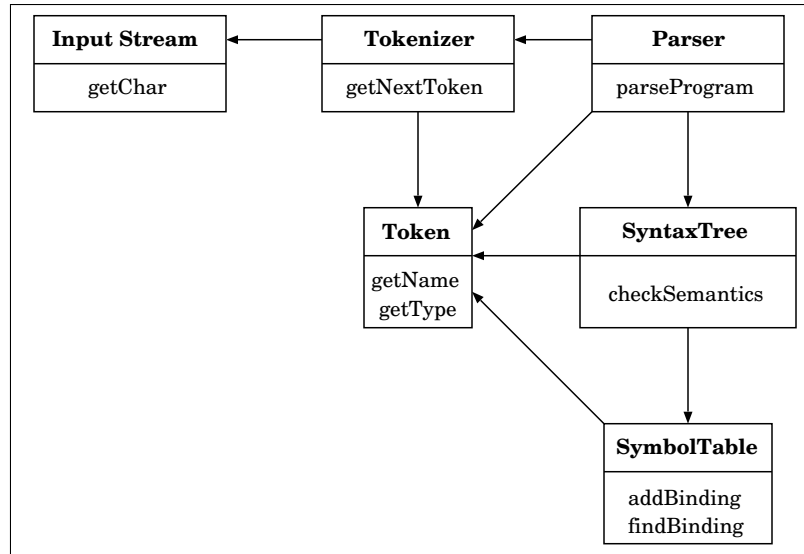


Figure 1.7: UML Class Diagram for a Translator Front-end

Turning to the semantic analysis half of the front-end, we note two relevant new objects in the data-driven view – namely the syntax tree and the symbol table. The syntax tree is generated by the parser via the call to `parseProgram`. The semantic analysis, then, is driven by requests made to the syntax tree, namely requests to generate a symbol table and then a request to check the semantics. The symbol table



is generated by the syntax tree and is actually integrated into the syntax tree structure. The UML class diagram in Figure 1.7 shows a complete architecture for the translator front-end; it is simply the diagram from Figure 1.6 with the syntax tree and symbol table integrated.

### 1.5.2 Back-end Structure

The back-end of the translator depends completely on the data supplied by the front-end in the form of the syntax tree and symbol table. In intermediate code generation the syntax tree is traversed, with appropriate intermediate code segments being generated as the traversal proceeds. Certain processor-independent optimizations are also applied to the intermediate code.

The object code generation focuses simply on the intermediate code, with occasional references to the symbol table. The conversion of the intermediate code to object code form and the application of various optimization techniques results in the object code. This optimization process can be simple or quite complex, depending on the demands of the translator, with additional data structures and/or intermediate forms being generated. The UML class diagram in Figure 1.8 combines the intermediate code and code table structures with the class diagram from Figure 1.7, giving a class diagram for the complete translator.

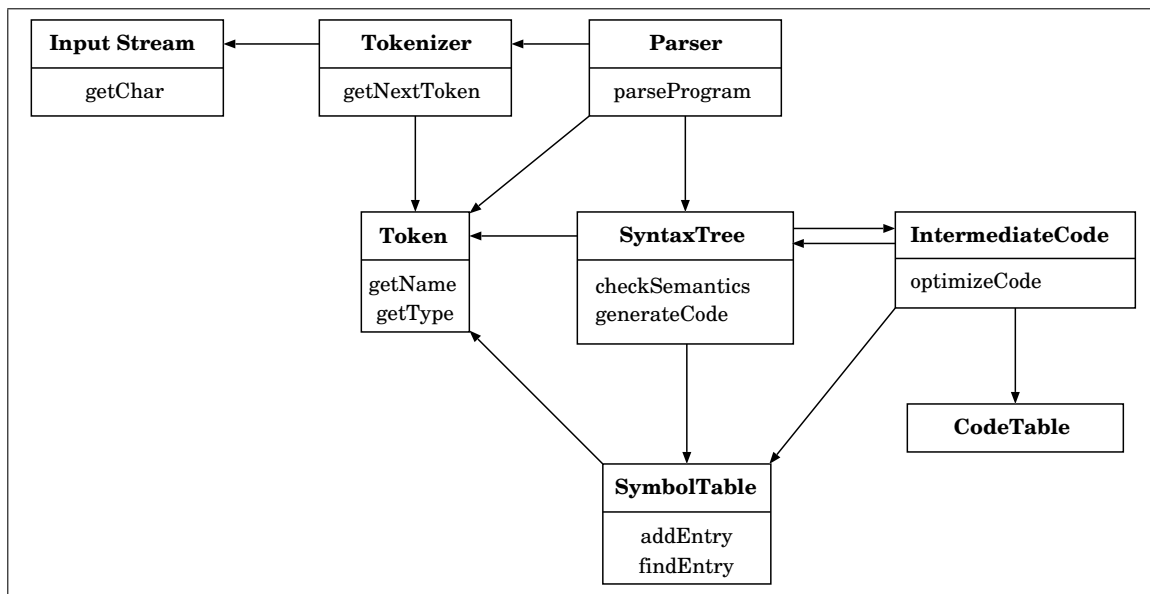


Figure 1.8: UML Class Diagram for a Translator

### 1.5.3 A Translator in Action

We close this section with two diagrams, each giving a slightly different view of the complete translation process. In Figure 1.9 we show a more active view of the translator, where boxes correspond to objects and arrows indicate control; labels on the lines indicate the method of the referenced object being used for the access.

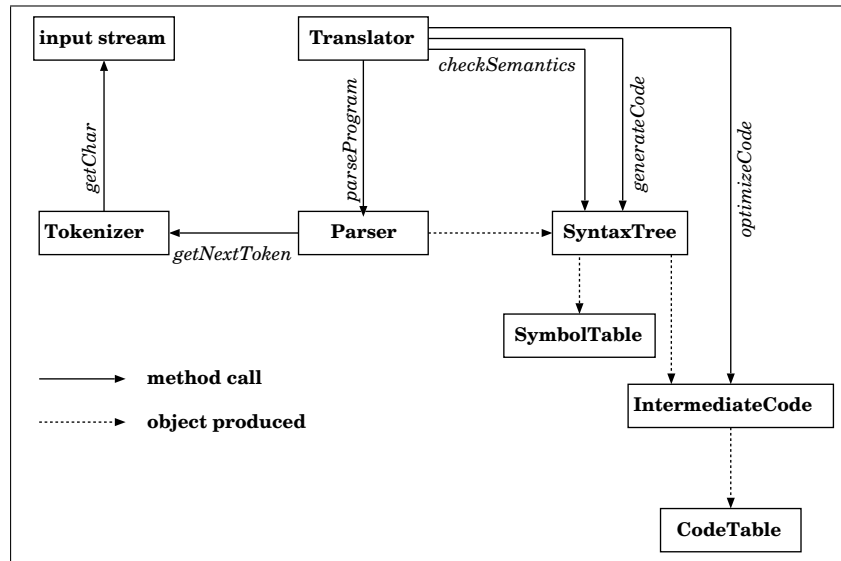


Figure 1.9: How the Translator Works

The second diagram, Figure 1.10, is a UML sequence diagram and shows how the major objects of a translator interact sequentially. It should be noted in this diagram that the symbol table and intermediate code have been left out in the interest of clarity.

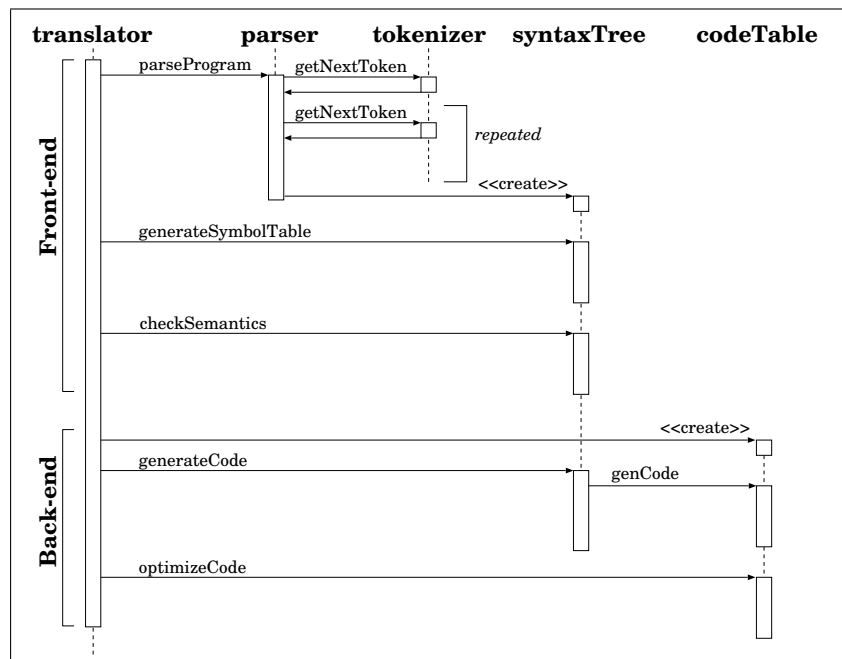


Figure 1.10: UML Sequence Diagram for a Translator



## Chapter 2

# Describing Languages

An artificial language is one that results from a process of definition rather than evolution. The languages we speak, English, French, Hindi, etc., are referred to as natural languages. While natural languages are of interest to those computer scientists working in the area of *natural language processing*, this book focuses on the artificial languages and from this point on we will simply refer to *languages*, usually dropping the modifier “artificial”.

Artificial languages are typically defined with some specific purpose in mind. Programming languages, such as Java, Fortran and Cobol, for example, are meant to be used for describing algorithmic solutions to general computational problems; HTML and LaTeX are two text-formatting languages for describing how text should be displayed in a web-browser, in the case of HTML, and in printed form, in the case of LaTeX. SQL, on the other hand, was designed to be used for describing database structures and queries. And there are hundreds of other artificial languages in use today.

When a language is created there are two questions that must be answered before we can claim to know what the language is:

- What are the legal forms for phrases in the language? We call this the *syntax* of the language.
- What meaning is associated with these legal forms? We call this the *semantics* of the language.

We are interested in these questions because their answers are required for and are central to the construction of language translators. In this chapter our interest is in understanding the basic strategies for providing both informal and formal answers to these two questions.

### 2.1 An Informal View of Language

To uncover the answers to the two basic questions above we will try decomposing the syntax and then the semantics into natural parts. The syntax of a language, for example, falls naturally into three levels: the alphabetic level which specifies the set of atomic symbols (characters) on which the language is based, the lexical level which specifies the shortest meaningful sequences of alphabet characters in the language, and the grammatical level which specifies the set of phrases (programs) expressible with the lexical elements in the language. The semantics of a language, on the other hand, can be looked at from two perspectives: meanings that can be determined for lexical and grammatical elements at translation time (static semantics) and at run time (dynamic semantics).

At an informal level we should be able to learn quite a bit from studying a simple segment of program code. The following segment of Java code will serve our purposes nicely.

```
int counter = 0;
double sum = 0.0;
while (counter < numberOfCharges) {
    sum = sum+cost[counter];
    counter = counter + 1;
    System.out.println("Sum so far = " + sum);
}
System.out.println("Total cost = " + sum);
```

We will look in more detail first at syntax and then semantics.

### Syntax: Alphabetic Level –

At the lowest level of detail we have the alphabetic symbols of the language. These symbols have no meaning of their own in the language and are present merely for defining higher level language structures. From our Java example we could certainly produce part of the Java alphabet, but a single example is unlikely to contain all legal alphabetic characters.

### Syntax: Lexical Level –

At the lexical level we have the words of the language – i.e., the sequences of alphabetic symbols that are indivisible in the language. These indivisible sequences are called *tokens* and include, for Java, the punctuation symbols of the language, identifier names, literal values, keywords, and operator symbols. From our example we can identify the following Java tokens – the table lists them by category.

Punctuation	Operator	Literal	Identifier	Keyword
{	+	0	counter	int
}	<	0.0	sum	double
(		"Sum so far = "	numberOfCharges	where
)		"Total cost = "	cost	
[				
]				
;				
=				

There is a more subtle problem at the lexical level. Describing the tokens in isolation is easy. But there is also the problem of layout. When tokens appear in a program, how are they spaced? On the fourth line of the example above the following sequence appears ‘sum+cost[counter];’ – there are 7 tokens but no spaces between any of them. On the preceding line there are also 7 tokens, but most are separated by blanks. What are the rules here? In fact, the spacing rules are essential for properly identifying tokens. We will talk about these rules focusing not on what can separate tokens, but what can terminate them. This detail is used by the tokenizer to determine when an entire token has been seen.

### Syntax: Grammatical Level –

The grammatical level consists of the legal phrases of the language – i.e., appropriately structured sequences of tokens. Since the purpose of a programming language is to describe

solutions to problems, those solution descriptions must be the largest possible legal phrases, that is, Java programs. But Java programs are pretty complex, so we should think in terms of those structures that can be combined in particular ways to produce programs. In object oriented programming languages such as Java, some of the common structures are *statements*, *methods* and *classes*. In our Java example above lines 1,2,4,5,6, and 7 are all examples of statements; the code on lines 3-7 also counts as a statement. But the exact structure of statements can be quite varied and each type must be carefully defined if a translator is to be implemented. One problem evident from the example is that each of the statements covering just one line is terminated by a semicolon, where the statement on lines 3-7 is terminated by a curly brace! We have to be able to describe precisely the rules for terminating statements. This termination problem is just one example of the sort of complex grammatical rule that is difficult to describe clearly and correctly without resorting to formal description methods. These methods will be introduced shortly.

The semantics of a language relates to how programs will behave at run-time. This means, if we provide a certain set of inputs to the program, what outputs will be produced – we refer to this as *run-time behavior*. Rather than determining this behavior for each program individually, it is important to have a systematic way of describing run-time behavior based on the behavior of its components. So the approach to language semantics is to understand what “behavior” means for each language component and how “behavior” can be systematically described for these components.

We address run-time behavior at two levels: behavior that can be deduced at translation time (static semantics) and behavior that can only be seen at run-time (dynamic semantics). Once again we will focus on the Java example at the beginning of the section as a guide.

### Semantics: Static Level –

If semantics focuses on meaning, then when we look at our Java example we should ask the question, what do the various syntactic elements in the code sample mean? Remember that the relevant syntactic elements are the tokens and the grammatical structures – the alphabet characters don’t enter in since they have no meaning on their own.

So we can start by focusing on tokens. If we look at the first line of the example we see 5 tokens: `int`, `counter`, `=`, `0`, and `;`. Of these tokens the last, `;`, is a punctuation token and has no meaning – it simply marks the end of the statement. So for semantic purposes punctuation tokens can be ignored. Working backwards, the token `0` is a literal token, that is it literally represents its value – in this case the integer value zero. The next token `=` is an operator token whose meaning will be discussed shortly. The remaining two tokens `int` and `counter` would seem to be built on the same rule: finite sequences of letters – also referred to as identifier tokens. But in most programming languages there are sequences of letters that are set aside for special uses – we call them *key word* or *reserved word* tokens. In this case `int` is reserved to represent a *type*, that is a set of values – in this case `int` represents the set of integer values. The name `double` (on the second line) also a type name – in this case the set of real numbers. Finally, we have `counter` which is an identifier token and names a memory location that will exist at run time – we refer to such identifiers as *variables*. As shown in the table above, each token in the example can be described in a similar way.

When we look at combinations of tokens we get a higher-level view of semantics. The first line of our example begins with the combination `int counter`. This means that the memory location associated with `counter` will be restricted to holding data that is in

integer format (a hardware characteristic). We refer to `counter`, then, as an `int` variable. The rest of the line gives us more information about the variable `counter`. The sequence `counter = 0` shows why `=` is an operator – it specifies that the variable `counter` is assigned the value `0`. This first line taken as a whole is a declaration statement in which `counter` is declared to be an `int` variable with initial value of `0`.

To be honest, each of the first two lines of our example, while being very standard Java, can actually be written as two statements. The first line is equivalent to the following.

```
int counter;
counter = 0;
```

Now the first is a plain declaration statement and the second is an assignment statement, in which the value to the right of the `=` token is stored (at run time) at the memory location named on the left. These two statements illustrate the declaration of a variable and also the *use* of a variable.

The statement on line 5 of our example illustrates the two ways that a variable can be used.

```
counter = counter + 1;
```

The use of `counter` on the left of the assignment is a reference to its location, its use on the right is a reference to the value stored at `counter`'s location. These uses of a variable can occur in other situations as well, but all uses are either references to the variable's location or its value. We use the terminology *L-value* to refer to a variable's location and *R-value* to reference its value. These names can be remembered because the L and R refer to the left and right sides of an assignment statement.

These example lines illustrate the issues involved at the static semantic level. First, since identifiers (in our examples we've talked about variables) must be declared before use, we should be able to tell at translation time if the use of an identifier can be matched to a previous declaration for the same identifier. In our sample code segment there are two identifiers, `numberOfCharges` and `costs`, for which we see no previous declarations. Since our sample is only a segment of code, it may be that earlier in the program appropriate declarations occur. If they do we are okay; if they do not then we would have a semantic error to report.

There is a second related issue that can be dealt with at translation time. Consider the fourth line in our example code.

```
sum = sum + cost[counter];
```

Our focus here is on the expression on the right side of the assignment. The token `+` is the binary addition operator. The problem is that the addition operator expects specific types of arguments. If `cost` has been declared to be a `double` array then at run-time the 64-bit addition operation will be carried out on the two 64-bit values (that's how a `double` value is usually represented on a 32-bit machine). If `cost`, on the other hand, has been declared as an integer array, then we would expect that the integer value of `cost[counter]` would be converted to an equivalent `double` form and then the operation would be carried out as usual, again producing a *double* result.

But what if, by some mistake, `cost` is declared to be a `boolean` array? In this case the addition operator doesn't make sense! So a semantic error would be expected, indicating that `cost[counter]` has an invalid type for the `+` operator. Thanks to the Java requirement for variable declarations, the three cases of variable use we have discussed can be distinguished and handled correctly at translation time.



So the static semantics of a language hinges on an understanding of declaration requirements for identifiers as well as the language's rules for using identifiers. These requirements are often referred to as a language's **type system**. The static semantics of a language is implemented in a translator in the form of a *semantic checker* or *type checker*, which generates semantic errors when identifiers are used but not declared, or are used in a way inconsistent with the context and the declared properties of the identifier.

A detail ignored to this point is the use of literal values. Since each literal value has an implicit type, determined by the language, that type can be used when assessing the use of literal values – just as the type of a variable is used to assess its uses. So literals fit very conveniently into the implementation of a semantic checker.

### Semantics: Dynamic Level –

If the static semantics of a program deals with properties associated with identifiers and the ways in which identifiers and literals are used, then the dynamic semantics of a program deals with the behavior of the program when it is executed. The behavior of a program can be thought of from two points of view: from outside the program – the *black-box view* – and from inside the program – the *state-change view*.

If a company comes to you and asks you to write a program for them, it is the black box view that they will specify in the contract. It's what they want to see when they run the program. This is fine for the customer, but how can you be sure you have produced a program that will meet the contractual black box view? The answer is that you can use the semantic description of the programming language you use for implementation and determine the state-change view semantics of the program you have written.

The internal view of a program, the state-change view, is based on the *program state*, i.e., to the list of the variable/value pairs that exist at a particular time during the program's execution. Using these terms we can think of the dynamic semantics of a program as the way in which its program state changes during execution.

Let's look once again at the example Java program – here it is again. (Remember that the variables `cost` and `numberOfCharges` have been declared elsewhere.)

```
int counter = 0;
double sum = 0.0;
while (counter < numberOfCharges) {
    sum = sum+cost[counter];
    counter = counter + 1;
    System.out.println("Sum so far = " + sum);
}
System.out.println("Total cost = " + sum);
```

When we reach the `while`-loop, the variables in the program state include `numberOfCharges`, `cost`, `counter` and `sum`. We know the values of `counter` and `sum` since they have just been initialized; but the values of `cost` and `numberOfCharges` are, at least as far as the example goes, unknown – but they do have values. The important idea to draw from the example code is the way in which the program state changes as subsequent statements are executed. For example, when the assignment statement

```
sum = sum + cost[counter];
```

is executed for the first time, the value of `sum` (initially zero) will be added to the value of `cost[0]`; the resulting value will be the new value of `sum`. So each assignment statement

changes the state value of the variable on its left side and these changes “accumulate” as execution proceeds.

So the execution of an assignment statement has a predictable impact on the program state. But what about the execution of the `while`-loop? At the end of the loop it would appear that the value `counter` must be equal to that of `numberOfCharges`, since to terminate the loop the loop condition must be false. But more interestingly, the value of `sum` will be the sum of the values it has encountered through the repeated executions of the loop’s body – i.e., the value of `sum` will be equal to

```
cost[0] + cost[1] + ... + cost[numberOfCharges - 1].
```

This means that after the loop we could assert the following condition.

```
counter == numberOfCharges AND
sum == cost[0]+...+cost[counter-1]}
```

It is this condition that defines the final state of the program (assuming that `cost` and `numberOfCharges` have not changed).

In this way, the state change due to executing structured statements, such as repetition and selection statements, depends on the changes resulting from the execution of the associated blocks of code. So dynamic semantics can be thought of recursively, with assignment and IO statements at the base, and then structured statements being defined in terms of these base statements and the structured statements.

## 2.2 A Formal View of Syntax

In this section we will discuss two formal mechanisms for describing syntactic aspects of a language: *regular expressions* provide a formal mechanism for describing the lexical level of a language, while *context free grammars* make it possible to formally describe the grammatical level of a language. We begin the section more broadly by discussing exactly what makes a language a language.

### 2.2.1 What’s in a Language

We start at the beginning with an alphabet, i.e., a finite set of symbols. If we have an alphabet  $\Sigma$  then our basic interest, linguistically, is in finite sequences of symbols from  $\Sigma$  – we often use the term *string* in place of finite sequence. As an example, if  $\Sigma = \{a, b, c, d\}$  then the following line has five strings over  $\Sigma$ :

```
a  ab  b  abbad  dcba
```

Each string has a length equal to the number of symbols in its sequence: in the examples above the lengths (from left to right) are 1, 2, 1, 5, and 4. We also single out the string  $\epsilon$ , which denotes the *null string*, whose length is zero – i.e., it is the string that contains no characters. We use the notation  $\Sigma^*$  to denote the set of all strings over  $\Sigma$ .

This  $\Sigma^*$  may sound a bit strange. What must be its elements? Well, ‘ $\epsilon$ ’ is the shortest string we can have so it must be in  $\Sigma^*$ . Next shortest will be strings with just one element: `a`, `b`, `c`, `d`, to be precise. Then we have the strings of length two: `aa`, `ab`, `ac`, `ad`, `ba`, `...`, well you get the idea – all possible combinations with two alphabet characters. And we can continue in this way to write out the strings in  $\Sigma^*$ . One obvious thing is that there is an infinite number of strings in  $\Sigma^*$ , which may or may not be reassuring. Now that we have a better idea of what  $\Sigma^*$  looks like we can turn to defining what we mean by a *language over an alphabet*  $\Sigma$ .

## DEFINITION 1 (LANGUAGE)

If  $\Sigma$  is an alphabet, then  $L$  is a language over  $\Sigma$  if and only if  $L \subseteq \Sigma^*$ .

One interesting consequence of this definition is that the set of all languages over  $\Sigma$  corresponds with the set of all subsets of  $\Sigma^*$ , in other words, the *power set of  $\Sigma^*$* , usually denoted  $\mathcal{P}(\Sigma^*)$ .

It should not come as a surprise that, since they are sets, we can combine languages (over the same alphabet) in particular ways. The combining operations provide a mechanism for combining languages, but also for exposing internal structures in a particular language, for example by showing a language can be written as a combination of other languages. Two of the language combining operations depend on the notion of string concatenation, which we define next.

## DEFINITION 2 (STRING CONCATENATION)

Assume that  $\Sigma$  is an alphabet and  $a_1, \dots, a_n, b_1, \dots, b_m \in \Sigma$ , with  $n > 0, m > 0$ .

If  $u = \epsilon$ , then  $uv = v$ .

If  $v = \epsilon$ , then  $uv = u$ .

If  $u = a_1 \dots a_n$  and  $v = b_1 \dots b_m$ , then  $uv = a_1 \dots a_n b_1 \dots b_m$ .

With this string concatenation operation in hand we can define the promised three language combining operations.

## DEFINITION 3

Let  $R$  and  $S$  be languages over an alphabet  $\Sigma$ . Then we define

**concatenation:**  $R \cdot S = \{uv \mid u \in R \text{ and } v \in S\}$ ,

**union:**  $R \cup S = \{u \mid u \in R \text{ or } u \in S\}$ ,

**Kleene star:**  $R^* = \{\epsilon\} \cup \{u_1 \dots u_n \mid u_1, \dots, u_n \in R, n \geq 1\}$

So the concatenation of two languages is simply a new language (set) containing all the strings you can get by concatenating a string from the first language with one from the second language. Notice that if  $\epsilon$  appears in the  $R$  (or  $S$ ) then every string from the  $S$  ( $R$ ) is in the  $R \cdot S$ . The union of two languages is a new language containing all the strings appearing in one or the other of the two languages. For the Kleene star of a language  $R$ , you can think of it as an infinite sequence of unions as follows:

$$R^* = \{\epsilon\} \cup R \cup R \cdot R \cup R \cdot R \cdot R \cup \dots$$

### 2.2.2 Regular Expressions

We can now apply some of the ideas defined above – in particular the operations on strings and languages – in defining the lexical level of a programming language. To focus our attention we will work on the definition of an *integer expression language* (IEL) that will allow us to describe integer-based expressions. The legal strings for this language include any algebraic expression based on integer operations and integer literals, for example,  $2 + 3$ ,  $(5 - 2) * 6$ , or  $(4 - 7) + ((10 / (4 + 3)) \% 3)$ . At the lexical level it is pretty clear that the tokens of our language will be the operation symbols, integer literals (sequences of digits), and the grouping symbols, which are the left and right parenthesis symbols.

The language IEL is simple enough that we can describe all the tokens informally. But we want a formal approach that will allow us to describe the lexical level of more complex languages as well. We also

want a formal way to describe the lexical level of a language so that it will be easier to write a program to recognize just the strings in the language – i.e., the tokens of a language.

To get started correctly we will point out the IEL alphabet is the set  $\{0123456789+-*/%()\}$ ; this is clear from the previous paragraph. We now need a way to formally define the set of tokens for IEL – i.e., the IEL token level. To facilitate this definition we define the notion of *regular expression*. A regular expression is a mechanism for describing a certain class of languages, called *regular languages*. The definition of regular expression is based on the language combining operations described in the previous section.

**DEFINITION 4 (REGULAR EXPRESSION)**

*Assume  $\Sigma$  is an alphabet. Then*

1. *the symbol  $\phi$  (not an element of  $\Sigma$ ) is a regular expression;*
2. *the symbol  $\epsilon$  (not an element of  $\Sigma$ ) is a regular expression;*
3. *for each  $a \in \Sigma$ ,  $a$  is a regular expression;*
4. *if  $R$  and  $S$  are regular expressions then*
  - (a)  *$R \cdot S$  is a regular expression (the concatenate operation);*
  - (b)  *$R \mid S$  is a regular expression (the union operation); and*
  - (c)  *$R^*$  is a regular expression (the Kleene star operation).*

While regular expressions are quite abstract, their utility is as descriptors – each regular expression describes a particular set of strings over the given alphabet. If  $R$  is a regular expression, then we use the notation  $\mathcal{L}(R)$  to mean the set of strings described by  $R$ .

**DEFINITION 5 ( $\mathcal{L}(R)$ )**

*Let  $\Sigma$  be an alphabet.*

1.  *$\mathcal{L}(\phi) =$  the empty set (also denoted  $\phi$ ).*
2.  *$\mathcal{L}(\epsilon) = \{\epsilon\}$  (the second  $\epsilon$  is the null string).*
3. *For each  $a \in \Sigma$ ,  $\mathcal{L}(a) = \{a\}$ .*
4. *If  $R$  and  $S$  are regular expressions then*
  - (a)  $\mathcal{L}(R \cdot S) = \mathcal{L}(R) \cdot \mathcal{L}(S)$
  - (b)  $\mathcal{L}(R \mid S) = \mathcal{L}(R) \cup \mathcal{L}(S)$
  - (c)  $\mathcal{L}(R^*) = (\mathcal{L}(R))^*$

Notice that  $\mathcal{L}(R)$  is a language over  $\Sigma$ . In fact, we associate a special name with languages that can be described by regular expressions.

**DEFINITION 6 (REGULAR LANGUAGE OVER  $\Sigma$ )**

*If  $\Sigma$  is an alphabet, then a language  $L$  over  $\Sigma$  is a regular language if and only if there is a regular expression  $R$  over  $\Sigma$  such that  $L = \mathcal{L}(R)$ .*

There are four basic ways to make a regular expression: a singleton symbol from the alphabet, the union or concatenation of two expressions, or the Kleene star of an expression. Using these we can define any token. Singleton tokens, such as in IEL the tokens  $+$  and  $*$ , are represented by themselves in a sense. Imagine the keyword `while` from Java; this keyword is the concatenation of five singleton regular expressions as follows:  $w \cdot h \cdot i \cdot l \cdot e$ . And further, this regular expression converts, via the function  $\mathcal{L}$ , to the set containing the string `while`:

$$\mathcal{L}(w \cdot h \cdot i \cdot l \cdot e) = \{ \text{while} \}$$

Since concatenation of singleton regular expressions is common in token descriptions it is customary to drop the dot-operation and simply use the string as both the token and the regular expression. From this point, unless clarity is an issue, we will drop the dot-operation from concatenations.

When it comes to defining all the tokens for a language it should be possible to union the regular expressions for the individual tokens. We illustrate this point by defining a regular expression over the alphabet of IEL describing the set of all the tokens of IEL. As pointed out above we can define the one character tokens of IEL by taking the regular expressions defined by the token symbols individually. We can make a preliminary attempt at the IEL token set by taking the union of these singletons as follows.

$$\begin{aligned} \mathcal{L}( + \mid - \mid * \mid / \mid \% \mid ( \mid ) ) &= \mathcal{L}(+) \cup \mathcal{L}(-) \cup \mathcal{L}(*) \cup \mathcal{L}(/) \cup \mathcal{L}(\%) \cup \mathcal{L}( ( ) \cup \mathcal{L}( ) ) \\ &= \{ + \} \cup \{ - \} \cup \{ * \} \cup \{ / \} \cup \{ \% \} \cup \{ ( ) \cup \{ } \} \\ &= \{ +, -, *, /, \%, (, ) \} \end{aligned}$$

So the regular expression ‘ $+ \mid - \mid * \mid / \mid \% \mid ( \mid )$ ’ describes the set of the singleton tokens for IEL. Just as with the concatenation operation, the inclusion of the union operator makes some regular expressions less clear. An extended notation is designed for exactly this situation. If a sequence of alphabetic characters is enclosed within square brackets, then the characters are assumed to be unioned. Thus, the regular expression above can be abbreviated to ‘ $[ + - * / \% ( ) ]$ ’.

The bracket notation just introduced has another abbreviation. The expression  $[0123456789]$  specifies the set of singleton strings, each composed of one of the listed digits. If we wanted to specify a similar list of alphabetic characters the lists could become quite long. When the characters listed are all characters between the first and the last (in ASCII code order), we abbreviate the list by designating the first and last as follows:  $[0 - 9]$ . The list of all upper and lower-case letters would be designated  $[a - zA - Z]$ .

We now turn to the only tokens still missing from our IEL token set, the integers. There is a basic problem, however: there are an infinite number of integer tokens over the IEL alphabet, since the length of the integer literals is unbounded. The Kleene star operation is designed specifically to describe such infinite sets of tokens. If an integer literal is any finite sequence of digits, it would seem we can apply the Kleene star directly to the union of the digit tokens – that is, using the abbreviated list technique,

$$[ 0 - 9 ]^*$$

But the one problem with this solution is that it allows the null string  $\epsilon$  as a token. Instead, we want to guarantee each token has length at least 1. In this case we can simply pull out a copy of the union of digits and concatenate it with the Kleene star expression – as such.

$$[0 - 9][0 - 9]^*$$

This being a bit long and awkward there is a convenient and standard abbreviation to define sequences of length at least 1. We replace the Kleene star by a ‘+’. So our integer tokens can be defined by the regular expression

$$[0 - 9]^+$$

There is one final interesting issue here. It may be that the designer of IEL has indicated that integer literals should not have leading zeros – except for the integer zero itself, of course. Such literals will have to start with a non-zero digit and then be followed by zero or more digits (including zero). The special case of zero will have to stand alone, apparently. So the integer tokens starting with non-zero digits must be described as follows.

$$[1 - 9][0 - 9]^*$$

Throwing in the special case of zero, then, gives us a regular expression that describes the integer tokens which don't have leading zeros.

$$0 \mid [1 - 9][0 - 9]^*$$

The tokens described so far make sense for the IEL. But in order to make the IEL more interesting later on when we talk about semantics, we will add a new token class for floating point literal values. When we talk about a floating point value we will mean a sequence of 1 or more digits separated by a period (decimal point). As in the case of integer literals we can either allow or disallow leading zeros for floating point literals. The following are examples of floating point literal tokens

0.4            5.1234            120.0004            0.0

while the sequence 024.024 is only a token if leading zeros are permitted.

Apparently, the sequence preceding the decimal point has the same structure possibilities as an integer, and the sequence following the decimal point can be any sequence of one or more digits. The following two regular expressions matches this description.

Leading Zeros Allowed	No Leading Zeros Allowed
$[0 - 9]^+ . [0 - 9]^+$	$(0 \mid [1 - 9][0 - 9]^*) . [0 - 9]^+$

With the regular expressions describing the integer and floating point tokens, we can write out a complete regular expression for the IEL token set as follows.

$$[ + - * / \% ( ) ] \mid [0 - 9]^+ \mid (0 \mid [1 - 9][0 - 9]^*) . [0 - 9]^+$$

While this regular expression definitely describes the IEL tokens, it is a bit clumsy. What we can do is make a table containing the name of each token class along with its defining regular expression. The table in Figure 2.1 illustrates this description technique. Notice that the table includes the second more restrictive version of the regular expression describing integer literals and the similarly restrictive definition for the floating point token class.

An important idea is hidden in the IEL regular expression table. The idea of a *token class*. A token class is a set of token values that are indistinguishable (or interchangeable) in a grammatical sense. All the integer literals can be used in the same way in the IEL – so `intT` names a token class. It could be that all the arithmetic operators are interchangeable, but we'll see in the section which follows that this isn't quite true. Finally, even though `lpT` and `rpT` are both punctuation, they are not interchangeable! After all the sequence `' )2+3('` just doesn't make mathematical sense! So `lpT` and `rpT` are not in a token class together.

Category	Token Class	Regular Expression
Operations	addT	+
	subT	-
	multT	*
	divT	/
	modT	%
Punctuation	lpT	(
	rpT	)
Literals	intT	$0 \mid [1 - 9][0 - 9]^*$
	fltT	$(0 \mid [1 - 9][0 - 9]^*) \cdot [0 - 9]^+$

Figure 2.1: Regular Expressions Describing the IEL Tokens

### 2.2.3 Context Free Grammars

Regular expressions are fine for describing structures that are either choices or repetitions, but they cannot be used to describe nested structures, as we see frequently at the grammar level of programming languages. In Java and many languages we can nest, for example, `while` statements with `if` or other `while` statements, which produces such nested structures. A more sophisticated language description mechanism is required.

The theory of context free grammars was developed in the late 1950's and early 60's by Chomsky, Naur, Backus and others as a mechanism for describing the nested structure of programming languages<sup>1</sup>. A context free grammar consists of a set of recursively defined rules, which make it possible to describe the syntactic structures such as nested structures that we are used to seeing in the programming languages we use. How might we approach a more formal description of this nesting. Let's think about the following rule-based approach.

a program *statement* : an assignment statement or a while statement or an if statement a  
*statement list* : a statement or a (statement ';' statement list) a *while statement*: `while` '('  
 Boolean expression ')' statement list

What we see is that every while statement contains a statement list consisting of one or more statements. But a statement can be a while statement, so while statements can have other while statements nested within. We will see this rule-based idea applied in the following section.

#### Definition of Context Free Grammar (CFG)

Formally a context free grammar consists of several components.

DEFINITION 7 (CONTEXT FREE GRAMMAR)

A context free grammar has four components:

- a finite set  $\Sigma$  of terminal symbols – the alphabet of the grammar,
- a finite set  $V$  of non-terminals symbols,
- a special symbol  $S \in V$ , called the start symbol, and

<sup>1</sup> Actually, Chomsky's work was aimed at describing syntactic structures for natural languages; Naur and Backus applied these ideas to describing structures for artificial languages, which, at the time, meant programming languages.

1	$S \rightarrow S', X$
2	$S \rightarrow X$
3	$X \rightarrow 'a'$
4	$X \rightarrow 'b'$

Figure 2.2: A Simple Context Free Grammar

- a finite set of grammar rules, with each rule having the form  $X \rightarrow Y$ , where  $X$  is a non-terminal and  $Y \in (\Sigma \cup V)^*$ .

The rules are obviously very important in a context free grammar and there are terms and assumptions that must be understood. We say that a rule with left side  $X$  is a “rule for  $X$ ,” i.e., that it in part defines the non-terminal  $X$ . It is also important that there can be multiple rules for the same non-terminal and that the right-hand side of a rule can be  $\epsilon$ . We also assume that every non-terminal has at least one rule defining it.

### An Example

As an example, consider the grammar  $G$  given in Figure 2.2. In this grammar the non-terminal symbols are  $S$  and  $X$ , while the terminal symbols are  $'a'$ ,  $'b'$ , and  $'.'$ . As is customary, the non-terminal  $S$  is the start symbol for the grammar. The rules are obviously those numbered 1-4 above, though the numbers are only used to identify a particular rule. It should be pointed out that the quote marks are not part of the grammar, but are used to highlight the terminal symbols.

The idea here is that the grammar describes a language and by that we mean strings of the terminal symbols. What language does this grammar describe? We can begin to understand by starting with  $S$  and then systematically applying rules to eliminate non-terminal symbols – basically, this means replace a non-terminal with one of its right-hand sides. Of course the resulting string will often still have a non-terminal so the process continues. We call this a derivation based on the grammar.

To illustrate this derivation process, we start with  $S$  and see if we can apply rules in a sequence so that when all non-terminals are gone we are left with the string  $'b, a, b'$ .

$$\begin{array}{l}
 S \xrightarrow{1} S, X \\
 \xrightarrow{4} S, b \\
 \xrightarrow{1} S, X, b \\
 \xrightarrow{2} X, X, b \\
 \xrightarrow{3} X, a, b \\
 \xrightarrow{4} b, a, b
 \end{array}$$

The number above each arrow in the derivation is the rule applied in the replacement in that step. So the first step involves replacing  $S$  by the right-hand side of rule 1. Notice that what is happening in the derivation is a continual rewriting of the previous string, where the rewriting is constrained to involve the replacement of one non-terminal. This is precisely what we described as the meaning of  $'\implies'$ .

The derivation above is proof that the string  $'b, a, b'$  is described by the grammar. But what other strings are so described? If we start with  $S$  and apply rule 2 and then either rule 3 or 4, then we get



either ‘ $a$ ’ or ‘ $b$ ’. Any other derivation will generate a comma-separated list of these two symbols. So the following five strings are in  $\mathcal{L}(G)$ .

$$a \quad a, b, b \quad b \quad a, a, a, a \quad b, b, b, a, a$$

### The Language Described by a Context Free Grammar

These are examples of applying a grammar rules in order to produce strings of terminal symbols. It is this derivation process which is the model for defining what we mean by the language defined by a CFG. On the notation side, if we have a context free grammar  $G$  then we write  $\mathcal{L}(G)$  for the language described by  $G$ . Our goal is to formalize what we mean when we say that a string of terminals  $w$  is described by the rules of a grammar  $G$ , or in more mathematical terms, when  $w \in \mathcal{L}(G)$ .

We start by formalizing the method of applying a grammar rule illustrated in the example. If we have a string  $w$  of terminal and non-terminal symbols then we can transform  $w$  into a new string by choosing one of the non-terminal symbols in  $w$  and then applying one of the grammar rules for that non-terminal. If  $w$  contains no non-terminal symbols then it cannot be transformed, of course. But imagine that  $w$  has the form

$$\alpha A \beta$$

where  $A$  is a non-terminal and  $\alpha$  and  $\beta$  are strings of terminal and non-terminals (i.e.,  $\alpha, \beta \in (V \cup \Sigma)^*$ ). We transform  $w$  into a new form by replacing the  $A$  in  $w$  by the right-hand side of one of the rules for  $A$ . Suppose we choose a rule of the form  $A \rightarrow \gamma$ ; then the transformed form of  $w$  would be

$$\alpha \gamma \beta.$$

For notational purposes it is convenient to express this transformation as follows.

$$\alpha A \beta \implies \alpha \gamma \beta, \text{ or in other terms } w \implies \alpha \gamma \beta$$

The symbol ‘ $\implies$ ’ denotes one transformation step, i.e., the substitution in  $w$  for one occurrence of one of its non-terminal symbols.

Bringing the start symbol together with ‘ $\implies$ ’ gives us the key to defining the strings described by a grammar. We say that a string  $w$  of terminal symbols is described by a context free grammar if there is a sequence of transformations starting with the start symbol and terminating with the  $w$  – we call such a sequence of transformation steps a *derivation* of  $w$  from  $S$ .

$$\begin{aligned} S &\implies w_1 \implies \cdots \implies w_n \implies w \text{ or in shorter form} \\ S &\xRightarrow{*} w \end{aligned}$$

So the notation  $S \xRightarrow{*} w$  can be read as “ $w$  is derived from  $S$  in one or more steps.” Looking at the previous line we can also deduce that the string  $S_n$  must have exactly one non-terminal and that the last transformation applies a rule whose right-hand side is all terminal symbols.

As a final point of notation, we can combine what we have described above into a more compact definition of the terminal strings described by a grammar  $G$  – i.e., the language of  $G$  over the alphabet  $\Sigma$ .

$$\mathcal{L}(G) = \{ w \mid w \in \Sigma^* \text{ and } S \xRightarrow{*} w \}$$

## Left-most and Right-most Derivations

Another point to make about derivations, and one which will be crucial in our discussion of parsing techniques, is that there can be different derivations for the same string. If at every transformation we always choose the left-most non-terminal for substitution, then we have a left-most derivation; if we always choose the right-most non-terminal, then we have a right-most derivation. The derivation above follows neither of these strategies since the third step transforms the left-most non-terminal ( $S$ ) while the first two steps transform the right-most non-terminals. Here are two new derivations of the same string, the one on the left being left-most and the one on the right, right-most.

### Left-most Derivation

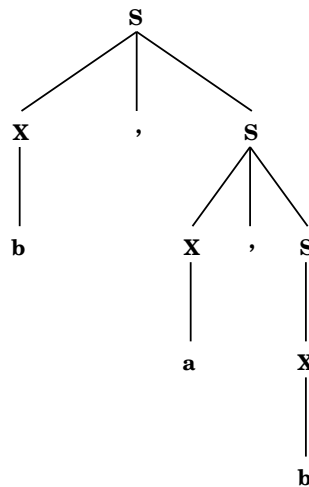
$$\begin{aligned}
 S &\xrightarrow{1} S, X \\
 &\xrightarrow{1} S, X, X \\
 &\xrightarrow{2} X, X, X \\
 &\xrightarrow{4} b, X, X \\
 &\xrightarrow{3} b, a, X \\
 &\xrightarrow{4} b, a, b
 \end{aligned}$$

### Right-most Derivation

$$\begin{aligned}
 S &\xrightarrow{1} S, X \\
 &\xrightarrow{2} S, b \\
 &\xrightarrow{1} S, X, b \\
 &\xrightarrow{3} S, a, b \\
 &\xrightarrow{2} X, a, b \\
 &\xrightarrow{4} b, a, b
 \end{aligned}$$

## Parse Tree for a Target String

The derivations illustrated in the previous two sections provide a way to determine if a target string is in the language of a grammar. But there is more information in these derivations than can be seen immediately. That hidden information can be exposed if we represent the derivation for a target string in an alternative form. What we do is draw a graph that represents how a derivation progresses. Here is such a graph for the string ‘ $b, a, b$ ’.



Each non-terminal in the grammar defines a subgraph determined by one of its rules. Not surprisingly all such graphs rooted with  $S$  are trees and we call them *parse trees*.

One interesting thing you might notice in the parse tree above is that at the second level there is no information to indicate whether the derivation converts the non-terminal  $X$  or  $S$  first. But this is one of the nice properties of the tree – it in fact summarizes several possible derivations in one tree.

### 2.2.4 A Grammar-level Example

Having defined the token level structure for the IEL (see Figure 2.1), we turn to the grammar level of the same language. We want to define a context free grammar whose language will be all the integer expressions as described in Section 2.1. We need to identify the sets of terminal and non-terminal symbols, the start symbol, and the set of grammar rules.

When defining a grammar for a language it is usually the case that the terminal symbols will be derived from the token level description of the same language. For IEL we can look back at the table of regular expressions in Figure 2.1. For each row in the table there is a regular expression and an associated name. It is the “Token Class” names that we will take as our terminal symbols in the IEL grammar. This choice is particularly convenient for `intT` and `fltT`, which name not single token values, but sets of literal token values – i.e., the sets of integers and real numbers.

Identifying non-terminals is more of a design problem and can be tangled up with the grammar rule design problem. What we want is a set of non-terminal names, where each name identifies a particular structure for the language. The start symbol should always be a name that defines full language strings. For IEL, since each string of the language is an expression, we will use the name `Exp` as a non-terminal and as the start symbol. One way to proceed is to ask how can an expression be seen structurally? The answers to this question will tell us about grammar rules defining `Exp`.

What we want to do now is to gain an understanding of the structure of an expression. We already know informally what expressions look like, so by examining some example expressions we should be able to gain insight into what grammar rules are necessary. Here are three example expressions we saw earlier on page 23.

$$\underline{2} + \underline{3} \qquad \underline{(5 - 2)} * \underline{6} \qquad \underline{(4 - 7)} + \underline{((10 / (4 + 3)) \% 3)}$$

Notice that in each example we have underlined the two components being added or multiplied. If we look at each component on its own, what we see is that each is what we would call an expression. What we learn from this examination is

- that integer literals are expressions (from the first example),
- that what falls within matched parentheses is an expression (from the second two examples), and
- that two expressions combined by an arithmetic operation is also an expression (from all three examples).

We also see in the third example that we can have deeply nested parenthesized expressions – this leads us to expect some recursion in order to define the nested structure.

We will address each of the three identified structures one at a time. The first structure is clear – there must be a rule that defines `Exp` as an integer literal token.

$$\text{Exp} \rightarrow \text{intT}$$

The second structure implies a rule defining `Exp` as `Exp` enclosed in matching parenthesis tokens.

$$\text{Exp} \rightarrow \text{lpT Exp rpT}$$

Notice that this rule will, at least in part, facilitate the nesting of parenthesized expressions. Finally, the third structure above indicates there should be a rule defining `Exp` to be two `Exp`'s linked by an operation token. In fact, since each of the five operation tokens is possible it would seem we need the following five rules.

```

Exp → Exp addT Exp
Exp → Exp subT Exp
Exp → Exp multT Exp
Exp → Exp divT Exp
Exp → Exp modT Exp

```

But since all of these rules are the same except for the operation token, we can use a special notation that indicates “choose one of the following”. The notation really defines itself – note that the vertical bar | should be read “or”.

```
Exp → Exp (addT | subT | multT | divT | modT) Exp
```

```

Exp → Exp (addT | subT | multT | divT | modT) Exp
    → intT
    → lpT Exp rpT

```

Figure 2.3: CFG for the Integer Expression Language

The list of grammar rules is summarized in Figure 2.3. One thing to notice here is that from the list of grammar rules we can deduce the other components of the context free grammar – at least if we use a consistent notional strategy: each non-terminal starts with an upper-case letter, each terminal symbol starts with a lower-case letter, the start symbol is on the left of the first grammar rule.

We finish this example showing how the second example expression above is derived with this grammar. The following is a left-most derivation of  $(5 - 2) * 6$ .

```

Exp   $\xRightarrow{1}$   Exp multT Exp
       $\xRightarrow{3}$   lpT Exp rpT multT Exp
       $\xRightarrow{1}$   lpT Exp subT Exp rpT multT Exp
       $\xRightarrow{2}$   lpT intT subT Exp rpT multT Exp
       $\xRightarrow{2}$   lpT intT subT intT rpT multT Exp
       $\xRightarrow{2}$   lpT intT subT intT rpT multT intT
              (5 - 2) * 6

```

### 2.2.5 BNF and EBNF Grammar Forms

The grammar rules we have seen to this point are in a standard form called BNF, for *Backus-Naur Form*, obviously named for two pioneers in programming language description (see the footnote on page 27). While BNF is quite useful, there are structures that require multiple rules and for which the definitional intention is not clear. Consequently, an extended form of BNF, referred to as EBNF, was introduced by Niklaus Wirth to facilitate readability<sup>2</sup>.

The principle mechanisms introduced in EBNF are for describing optional grammar structures and repeated grammar structures. Recall the simple grammar of Figure 2.2.

<sup>2</sup>Niklaus Wirth published a short communication in the November 1977 issue of *Communications of the ACM* in which he laid out the principles of EBNF.

- 1  $S \rightarrow S', X$
- 2  $S \rightarrow X$
- 3  $X \rightarrow 'a'$
- 4  $X \rightarrow 'b'$

We can see that the first two rules really describe a structure which must end in  $X$  but may be preceded by ' $S,$ ' – i.e., the sequence ' $S,$ ' is optional. In EBNF these two rules can be written as follows.

$$S \rightarrow [S,] X$$

The square brackets surround the optional part.

But there is another possibility here. If we think about a derivation based on the repeated application of rule 1, we can see that what we get is a comma-separated list of  $X$  structures. So the first  $X$  can be followed by zero or more copies of a comma followed by  $X$ . This repetition structure is described in EBNF with the following structure.

$$S \rightarrow X \{ , X \}$$

In this notation the curly braces surround the segment that is repeated zero or more times.

As a side note, one of the advantages to using names for tokens is that there won't be confusion between square brackets and curly braces as tokens and as grouping symbols for regular expressions.

While EBNF is useful for description, it cannot be used in the context of derivations. If we have a grammar in EBNF and want to investigate derivations in the grammar, then we will have to generate a BNF version of the grammar. For example, if we have the EBNF grammar rule

```
If → ifT Exp Statement [ elseT Statement ] endifT
```

we can introduce a new non-terminal for the optional part and have the following equivalent BNF rules.

```
If          → ifT Exp Statement ElsePart endifT
ElsePart    → elseT Statement
            → ε
```

In the next section we give a BNF equivalent form for the EBNF repetition.

### 2.2.6 Describing Lists

One of the more common programming language structures is the list. We see this in lists of statements, lists of formal and actual parameters, lists of declarations, etc. Not surprisingly the description of lists is common in context free grammars. There are two basic BNF forms for describing lists.

$$\begin{array}{ll}
 S \rightarrow S, X & S \rightarrow X, S \\
 \rightarrow X & \rightarrow X
 \end{array}$$

Notice that if we repeatedly apply the first rule on the right then our list emerges from left to right, where the rules on the left will cause the list to emerge from right to left. There are also EBNF forms which were introduced in the previous section. The following list rules make use of the EBNF optional structure.

$$\begin{array}{ll}
 S \rightarrow [ S, ] X & S \rightarrow X [ , S ]
 \end{array}$$

The BNF and EBNF rules on the left are written in left-recursive form and the ones on the right in right-recursive form. What we would like to know is what is the real difference between the two approaches?

A crucial point turns out to be the importance of the syntactic element separating the list components. The separators in a sequence, coming between two list elements, can be seen as binary operations. This means they really behave in the same way arithmetic operations work as well. It is the nature of these binary operations that is crucial to understanding lists structures.

### Left-recursive Lists

If we focus on the left-recursive structure we see the list built from left to right. It's important to understand what this means – if we have three elements 'a', 'b', and 'c' then working from left to right we will first form a list 'a,b' and then add the third element to the end of that list, giving 'a,b,c'. So the left-recursive form describes a list where the next element is added to the end of the list. Thinking in terms of the separators, in this syntactic form the separators behave as left-associative operations.

This form has direct application to describing arithmetic expressions since arithmetic operators are left-associative. Notice in this case that the tokens separating the elements of the list are crucial to our being able to understand the list. For example, the lists 'a+b-c' and 'a-b+c' have the same list elements but the separators are crucial.

### Right-recursive Lists

With the right-recursive form we see the list built from right to left – in which case we would form 'b,c' and then add the first element to the beginning of that list, giving 'a,b,c'. Notice that in the left-recursive form we build the list by adding the next element to the end of an already created list, while in the right-recursive form we add the next element to the head of a yet to be created list (of the rest of the elements). Thinking once again in terms of the separators, in this syntactic form the separators behave as right-associative operations.

It is interesting that in some languages, the functional languages Haskell and scheme for example, there is a specific notation for this recursive list structure. In these languages there is a built-in binary operator (in Haskell it is ':') which combines an element and a list. In Haskell notation we would write 'a:A' to denote the list with first element 'a' and the rest of the list 'A' – a singleton list is denoted 'a:[ ]', where '[ ]' denotes the empty list. Notice that the operator ':' takes an element as its left argument and a list as its right argument. Using this notation the list 'a,b,c,d' is described by the following expression.

$$a : (b : (c : (d : [ ]))),$$

Notice that the operation ':' is right associative.

The reason this right-recursive form is important is because in a language such as Haskell the lists are represented at run-time as right-recursive structures like these, which are critical to the computational characteristics of the language.

### Unstructured Lists

Sometimes in a language a syntactic list is simply a list, that is, a linear sequence of elements in which order is the only important factor. In such cases the elements separating the list elements are purely punctuational – they have no operational importance at all. Examples are lists of statements, lists of

arguments, lists of formal parameters. In these cases the recursive nature of the lists is not important – i.e., the list is simply a first element  $X$  followed by a sequence of ‘,  $X$ ’s. This repetition structure, as opposed to the recursive structure above, is conveniently described in EBNF as follows (remember we saw this structure in the previous section).

$$S \rightarrow X \{ , X \}$$

### 2.2.7 A Grammar Approach to Lexical Description

While regular expressions and context free grammars seem quite different, there is a connection between the two. We know that regular expressions describe regular languages (by Definition 6). But it turns out that context free grammars, with appropriate restrictions on the form of rules, also describe regular languages. The restrictions are not difficult to describe.

#### DEFINITION 8 (REGULAR GRAMMAR)

Let  $G$  be a context free grammar.  $G$  is a left(right) regular grammar if every grammar rule of  $G$  satisfies one of the following two criteria.

- The right side of the rule contains no non-terminals.
- The right side of the rule contains exactly one non-terminal and it is the left(right)-most symbol.

If  $G$  is left regular or right regular we say  $G$  is a regular grammar.

It is also important, from a theoretical point of view, that there is an algorithm which can convert any regular expression into an equivalent (i.e., describes the same language) regular grammar; there is also an algorithm which can convert any regular grammar into an equivalent regular expression.

Figure 2.1 displays a regular expression for the IEL (described in Section 2.2.2). If we think about the token description problems presented by IEL, the single symbol tokens are easy to describe grammatically.

```
Operation  → ‘+’ | ‘-’ | ‘*’ | ‘/’ | ‘%’
Punctuation → ‘(’ | ‘)’
```

Notice that neither of these rules has a non-terminal on the right, so both qualify as rules in a regular grammar. The literals in IEL present a more interesting problem, but if we watch for repetition patterns we can use list descriptions to form the required grammar rules. The following (regular) grammar rules define integer and floating point literals.

```
IntLit    → ‘0’ | PosDigits
FloatLit  → ‘0’ ‘.’ Digits
          → (‘1’ | ... | ‘9’) DFloat
DFloat    → ‘.’ Digits
          → (‘0’ | ... | ‘9’) DFloat
PosDigits → (‘1’ | ... | ‘9’)
          → (‘1’ | ... | ‘9’) Digits
Digits    → (‘0’ | ... | ‘9’)
          → (‘0’ | ... | ‘9’) Digits
```

It is easy to see that each rule above satisfies the requirement for a right-regular grammar.

This grammar may seem a bit complex, but the complexity is due to the desire in this case to give a right-regular grammar. In fact, we can write an equivalent non-regular grammar, which is much clearer, but that grammar wouldn't demonstrate that a regular grammar can describe the IEL tokens.

The advantage of describing the tokens of a language with grammar rules is it allows the language designer to use a single mechanism, a context free grammar, to give a complete syntactic description of a language. The advantage of using a regular expression to describe the tokens, on the other hand, is the regular expression is the form required by programs such as LEX, which automatically generate an implementation of a lexical analyzer.

## 2.3 Problems of Ambiguity

While the CFG in Figure 2.3 gives a precise definition of our language IEL, it is in a form that is not appropriate for deriving an implementation. The reason is the syntax does not carry enough information to accurately specify the desired order in which operations should be applied. The IEL grammar has two related problems: *operator precedence* and *operator associativity*.

### 2.3.1 Operator Precedence

Consider the integer expression '1 + 10 \* 4'. We learn in algebra courses that we do multiplication before we do addition, so this expression would evaluate to 41. But what if we were supposed to do the addition first. Then the expression would evaluate to 44! The problem with the grammar for the language IEL is that it does not indicate which of these evaluation orders is appropriate; in this case we say that the grammar is *ambiguous*.

This can be seen more clearly if we look at derivations of this expression in the IEL grammar. Here are two derivations with identical parse trees.

$$\begin{array}{l}
 \text{Exp} \xrightarrow{1} \text{Exp addT Exp} \\
 \xrightarrow{2} \text{intT addT Exp} \\
 \xrightarrow{1} \text{intT addT Exp multT Exp} \\
 \xrightarrow{2} \text{intT addT intT multT Exp} \\
 \xrightarrow{2} \text{intT addT intT multT intT} \\
 1 + 10 * 4
 \end{array}
 \qquad
 \begin{array}{l}
 \text{Exp} \xrightarrow{1} \text{Exp addT Exp} \\
 \xrightarrow{1} \text{Exp addT Exp multT Exp} \\
 \xrightarrow{2} \text{intT addT Exp multT Exp} \\
 \xrightarrow{2} \text{intT addT intT multT Exp} \\
 \xrightarrow{2} \text{intT addT intT multT intT} \\
 1 + 10 * 4
 \end{array}$$

These two derivations have the parse tree shown on the left of Figure 2.4. The following derivation has a different parse tree, that shown on the right of Figure 2.4.

$$\begin{array}{l}
 \text{Exp} \xrightarrow{1} \text{Exp multT Exp} \\
 \xrightarrow{2} \text{Exp addT Exp multT Exp} \\
 \xrightarrow{1} \text{intT addT Exp multT Exp} \\
 \xrightarrow{2} \text{intT addT intT multT Exp} \\
 \xrightarrow{2} \text{intT addT intT multT intT} \\
 1 + 10 * 4
 \end{array}$$



The two parse trees for our expression ‘1 + 10 \* 4’ illustrate that the difference in the two trees is determined by which operation, multiplication or addition, is given precedence. The parse tree on the left, and the first two derivations above, indicate that the multiplication of 10 and 4 must be done before the addition can be carried out – so multiplication has precedence over addition, which is our normal way of thinking about it. The tree on the right indicates the opposite, that the addition of 1 and 10 must be done before multiplication can be carried out – in this case addition has precedence. We would like to be find a way to distinguish these two cases in the IEL grammar.

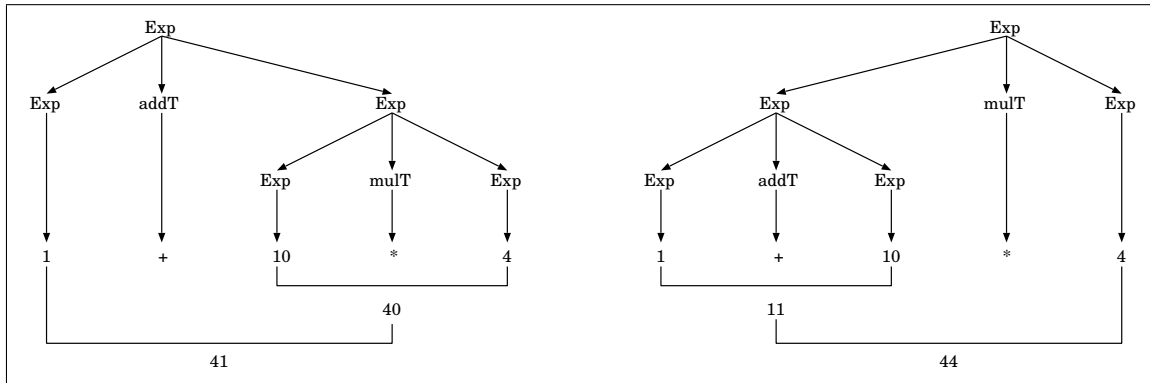


Figure 2.4: Ambiguity of Operator Precedence

When there is a string such as ‘1 + 10 \* 4’ derivable from a CFG which has two different parse trees, as illustrated in the diagrams, we say the grammar is *ambiguous*. In specifying the syntax of a language it is necessary to communicate to the implementor of the language which operator ordering is the correct one. Fortunately, as we will see shortly, it is possible to modify the IEL grammar so that this ambiguity disappears. The other ambiguity, dealing with operator associativity, is discussed below.

Just to drive home the definition of ambiguity. Having two different derivations for a target string is NOT evidence of ambiguity, since it is possible for two derivations to have the same parse tree – remember the first two derivations for ‘1 + 10 \* 4’ above. In order for a grammar to be ambiguous there must be two different *parse trees* for the same target string – there only has to be a single example string. If the grammar is non-ambiguous then for a target string there can be at most one parse tree, which embodies all possible derivations of the target string.

### 2.3.2 Operator Associativity

Another important example of ambiguity comes from the IEL grammar. This time it has to do with the order in which a sequence of ‘-’ operations is carried out. Consider the simple expression ‘10 - 4 - 3’. If we carry out the operations from left to right and then from right to left we get two different answers: 3 in the first case and 9 in the second. We can once again look at derivations for this expression to see if there are two parse trees.

#### ☛ Activity 1 –

Give two different derivations of the string ‘10 - 4 - 3’ in the IEL grammar. The derivations should have the two parse trees shown in Figure 2.5.

---

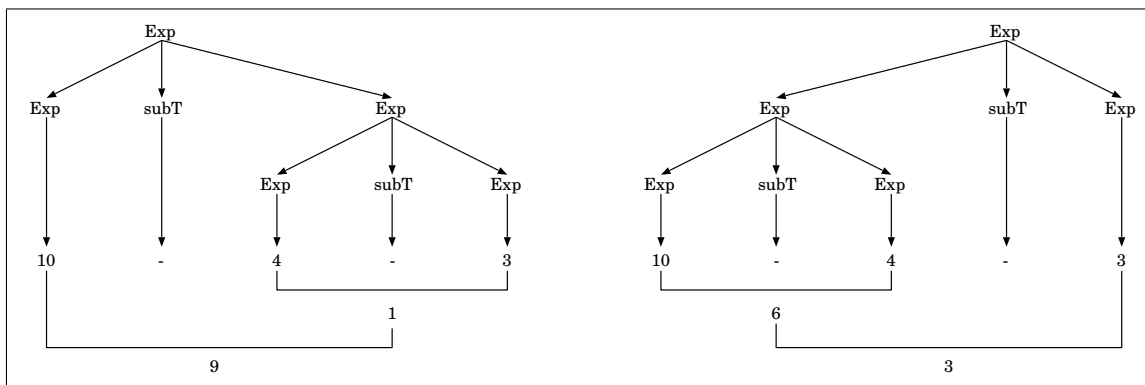


Figure 2.5: Ambiguity of Operator Associativity

The two parse trees in Figure 2.5 generate the same string, so we have an ambiguity. It is interesting to note that, even with an expression such as ‘10 + 4 + 3’, for which the result is the same regardless of order of operation, there are two parse trees. We again are driven by our expectations from years of using mathematics to expect that operations at the same precedence level be carried out from left to right. Once again it will be necessary to resolve this ambiguity by reforming the IEL grammar.

### Activity 2 –

Prove that the grammar

$$\begin{aligned} S &\rightarrow S \text{ ', ' } S \\ &\rightarrow x \end{aligned}$$

is ambiguous by finding a string that has two parse trees – notice that the strings can only be comma-separated lists of  $x$ 's!

### 2.3.3 Resolving Ambiguity

We want to discover a new grammar for the IEL language – a grammar that is not ambiguous. To motivate our discovery process let's consider a new IEL string:

$$3 + 4*(2+3) - 10\%4$$

Since a grammar is meant to describe the structure of strings in a language, we need to see if we can understand the structures in this example string. Before we go too far we will simply say that a string in IEL is an *expression* – that gives us a category we can always start with. So the natural question is: What is an expression? Thinking again about algebra class we know that when we see an expression like our example above we carry out multiplication-like operations before addition-like operations – and we carry out parenthesized expressions even earlier. So when we look at the expression above we naturally see it as a sequence of things linked by ‘+’ and ‘-’ signs. The terminology from algebra is *term*: an expression in IEL is a combination of one or more terms, where the terms are combined via additions and subtractions. This of course begs the question: What is a *term*? In our example string there seem to be three terms.

$$3 \qquad 4*(2+3) \qquad 10\%4$$

These three terms are examples of kinds of “products” of things – that is, sequences of things combined via ‘\*’, ‘/’, and ‘%’. The 3 is all by itself, so we can consider it to be a term of only one thing. The things that make up a term must have a name as well so we can talk about them – we will use *factor* to name these things. In the following we have written, below each term, the factors comprising it.

	3	4*(2+3)	10%4
factors:	3	4 (2+3)	10 4

But once again this begs a new question: What is a *factor*? Now that’s a good question. Apparently an integer can be a factor, since the stand-alone 3 has to be a factor, and the same is true for the other terms where 4 and 10 appear. What else? Well in the middle term we have a parenthesized thing – that must be a factor as well. But what goes between the pair of parentheses? Again from algebra, we know that in such a position we can put any expression at all – something simple, like a single integer, or complex, containing nested pairs of parenthesis. So a factor can be an expression surrounded by matched parentheses. We can summarize what we have learned as follows.

```
Exp:   Term addT Term addT ... addT Term
Term:  Factor multT Factor multT ... multT Factor
Factor: intT or lpT Exp rpT
```

What have we done in these rules? We have taken the form of an expression in IEL and broken it into natural grammatical layers based on our knowledge of arithmetic expressions from algebra. The first two layers resolve the precedence ambiguity we saw earlier by separating the additive and multiplicative aspects of an expression. Imagine drawing a parse tree for this “grammar”: at the top you would have terms combined with ‘+’ and ‘-’, and any occurrences of the multiplicative operations would be lower down the tree and in particular at the bottom. Things at the bottom of the tree have to be done before things further up the tree can be done, so our layering gives precedence to multiplicative operations as it should.

What the “rules” don’t do yet is deal with associativity ambiguity – i.e., the order in which operations of the same category are carried out. To deal with that we need to think about turning the first two “rules” into proper CFG notation. It’s clear that *Exp* describes a list of *Terms*. The smallest *Exp* is a single *Term*. But for longer lists we want to make sure our additive operations are left-associative – that means we should adopt the left-recursive structure for the *Exp* rule. But the multiplicative operations are left-associative also, so the rule for *Term* gets the same left-recursive treatment. The result below is the IEL CFG we have been after – describing strings in the IEL to give multiplicative operations precedence over additive operations and to force the additive and multiplicative operations to be left associative.

```
Exp   → Exp (addT | subT) Term
      → Term
Term  → Term (multT | divT | modT) Factor
      → Factor
Factor → intT | lpT Exp rpT
```

### 2.3.4 Removing Left-recursion

From the point of view of syntax description, the left-recursion in this grammar is not a problem. But we will see, when we talk about parsing, that some parsing strategies cannot be applied to grammars with

left recursion. For this reason it is important that there are techniques for removing left recursion from a grammar – that is, to give an equivalent grammar without left-recursive rules.

Left recursion can indeed be removed from our grammar, but only by making the grammar more complex. The idea is to replace the rules like

$$\begin{aligned} \text{Exp} &\rightarrow \text{Exp addT Term} \\ &\rightarrow \text{Term} \end{aligned}$$

with a new pair of rules that include a new non-terminal, as follows.

$$\begin{aligned} \text{Exp} &\rightarrow \text{Term E}' \\ \text{E}' &\rightarrow \text{addT Term E}' \mid \epsilon \end{aligned}$$

To fix our IEL grammar we would make a similar change for the **Term** rules. The resulting grammar is not as appealing as the earlier ones and will make certain translation operations less intuitive.

### ☛ Activity 3 –

Transform our IEL grammar with left recursion to an equivalent grammar without left recursive rules.

---

### ☛ Activity 4 –

With the non-left recursive grammar rules, write a derivation of the expression  $3 + 4*(2+3) - 10\%4$ . Be sure not to leave out any steps and label each derivation step with the rule number applied.

---

## 2.4 A Formal View of Semantics

Formalizing the syntax of a language is fairly straightforward, but to do so for the semantic side is more complex. The problem is that the static semantic properties that interest us (e.g., are identifiers appropriately declared, are identifiers used in accordance with their declared properties) cannot be described in a context free way – they are inherently *context sensitive*. To facilitate a discussion of formal semantics we will introduce a new language CL whose structure, while still simple, will allow us to focus on the basic semantic issues. In this section we will present the static semantics for CL as a list of informally-stated semantic rules. While the rules lack the formality of, say, a context free grammar, the language CL is so simple that the descriptions can be quite precise.

### 2.4.1 CL – The Calculator Language

The calculator language CL provides two familiar facilities: variable declaration statements and assignment statements. A program consists of a list of such statements. Our interest in this section is to define the CL tokens, via appropriate regular expressions, and the CL grammar, via a context free grammar. To motivate these definitions here are two example CL programs.

```
{ int a, float b, a = 3, b = a*4.0 }
{ float pi, pi = 3.14, float area, int diam, diam = 35, area = pi*diam/2 }
```

Okay, to call each of these lines a program may stretch the imagination but they have the advantage that we can focus on semantic issues without a lot of other baggage that we would have if we tried to focus on the semantics of Java for example.

CL has the following properties (informally given).

1. CL has two keywords `int` and `float` used to name the two possible data types in the language.
2. A CL program is a comma-separated list of declaration and assignment statements bracketed by curly braces.
3. A declaration statement is a type name followed by an identifier, and serves to bind a type to the identifier.
4. An assignment statement has the usual form: an identifier followed by the assign symbol (`'='`) followed by an arithmetic expression: this specifies that at run time the expression will be evaluated and the result stored at the variable named on the left.
5. An arithmetic expression has the same form as expressions in the IEL language, but allows variable names as well as integer and floating point literals to occur as arguments.

## Formal CL Syntax

The CL tokens are easily extended from those for IEL. The additions are mostly singleton tokens (`;` `=` `{` `}`) but also include type names and identifiers. Because the two type names are interchangeable syntactically, we put them into the same token class (`typeT`). Identifiers consist of one or more letters, either upper or lower case. The appropriate regular expressions defining each of these new tokens can be seen on the left in Figure 2.6. The grammatical level can be described by applying the techniques of the previous section. In order to organize the grammar we will focus on several abstract structures. A **Program** can be seen as a list of statements `StmtList` – enclosed in curly braces. A `StmtList` consists of a sequence of statements – `Stmt` – each of which is either a declaration statement – `Declaration` – or an assignment statement – `Assignment`. Looking on the right side of Figure 2.6 we can see the rules which defined formally each of these abstract structures. The declaration statement structure is easily defined as a sequence of two tokens, but the assignment statement structure involves a reference to the `Exp` structure defined for the IEL, but extended for use in CL. The only modification to the three rules defining `Exp` is the inclusion of identifiers and floating point literals as possible structures for `Factor`.

When you look at the token description above you should notice that there is a bit of an ambiguity problem. Notice that the strings `int` and `float` appear to belong to both the `typeT` class and the `identT` class. Eliminating this problem formally would seem easy enough – just indicate for `identT` that the two type names aren't identifiers. But doing this in the form of a regular expression is not easy. What happens when we implement a tokenizer (see Section 1.4) for real is to use the definition of `identT` above and then when an identifier is found to compare it to any reserved word (such as `int` or `float`, in this case). If an identifier matches a keyword then an error is given.

An aside: You might look at the description in Figure 2.6 and say to yourself “That’s a lot of complication just to formally describe what the two example CL programs seem to describe perfectly well!” In fact, for very simple languages a few examples may very well give a good idea of the structure of the language. But remember the problems we had with ambiguity? These problems can lurk even in seemingly simple structures. More importantly, since we want to eventually write an implementation of the language, the examples will not help us understand how to structure that implementation. Only the regular expressions and grammar can serve as a guide to the structure of an implementation.

Token Class	Regular Expression	Grammar Rules
addT	+	Program → lcbT StmtList rcbT
subT	-	StmtList → Stmt   Stmt commaT StmtList
multT	*	Stmt → Declaration   Assignment
divT	/	Declaration → typeT identT
modT	%	Assignment → identT assignT Exp
commaT	,	Exp → Exp (addT   subT) Term
assignT	=	→ Term
lpT	(	Term → Term (multT   divT   modT) Factor
rpT	)	→ Factor
lcbT	{	Factor → intT   floatT   identT   lpT Exp rpT
rcbT	}	
typeT	int   float	
intT	0   [1 - 9][0 - 9]*	
fltT	(0   [1 - 9][0 - 9]*) . [0 - 9]+	
identT	[a - zA - Z]+	

Figure 2.6: Syntax Description of the Calculator Language

### 2.4.2 Specifying Static Semantics

One of the first things we look for when learning a new language, after the general syntactic structure, is the set of rules dictating the use of identifiers. For languages such as Java and C++ identifiers are used to name not only variables but methods and classes as well. Each identifier is associated with a particular set of *attributes* which represent its static and dynamic properties – we say the attributes are *bound* to the identifier. For example, a variable name will be bound to a type, a memory address, and a value. Of course, the address and value attributes will be determined (bound) at run time – i.e., they are dynamic attributes. But the type attribute, for most familiar languages, is static and is specified in a program by a declaration statement of some sort.

The static binding of attributes to identifiers is one concern of the static semantics of a language, but it is also concerned with the ways in which identifiers are employed in a program. In arithmetic expressions, for example, identifiers and literals are combined via arithmetic operators. But some operators have restrictions – the mod operation requires integer arguments. The *type system* of a language specifies what types are in the language, which types are compatible, and how types are determined for expressions.

To specify the static semantics for CL we specify four things: CL’s built in and definable types, identifier declaration requirements, how to compute the type of an expression, and which pairs of types are assignment compatible. The following rules define the CL semantics.

**CL Types:** The only types are the two that are built in, `int` and `float`.

**Identifier Declaration:** Identifiers can be declared only once and a declaration of an identifier must appear in the text of the program before the identifier’s first use.

**Expression Type:** The type of an expression is defined as follows:

1. If the expression is simply an identifier or a literal, then the expression type is the type of the literal or the type bound to the identifier in its required declaration.
2. If the expression has the form `exp1 op exp2` and `op` is not ‘%’, then its type is `int` if both arguments have type `int` and `float` if at least one of the arguments has type

- float.
3. If the expression is `exp1 % exp2`, then both arguments must have type `int` and the expression type is `int`.

**Type Compatibility:** The type `int` is assignment compatible with both `int` and `float`; the type `float` is assignment compatible only with itself. In an assignment statement the type of the expression (on the right) must be assignment compatible with the identifier on the left.

We should point out that the identifier declaration rule above specifies that CL uses *static type checking*, which can be carried out by the static semantic checker. If the rule were written as follows

Identifiers must be declared at run time before their first use.

then CL would use *dynamic type checking*, which, of course, can only be done at run time. While dynamic type checking is rare, it does occur in well known languages such as Smalltalk, Perl, and Python.

### 2.4.3 Specifying Dynamic Semantics

The purpose of a *dynamic semantics* description is to formally specify the behavioral characteristics of the language – from this description it should be possible to determine, for any valid program, the output it will produce given a particular sequence or collection of input values. Not surprisingly such descriptions depend on the grammatical description of the language, just as the static semantic description does.

There are three well-known formal methods for describing the dynamic semantics of a language. *Axiomatic semantics* is described in terms of a set of axioms based on the syntactic structures of the language. The idea is, given a program and its input, you can apply the axioms and deduce the result of executing a program. *Denotational semantics* describes the behavioral characteristics of the language in terms of mathematical objects. One form of denotational semantics, for example, is based on definitions of functions between domains – functions are defined to describe the behavior of the various syntactic elements.

*Operational semantics* is the more accessible of the dynamic semantics description techniques. Operational semantics is based on the explicit translation of the language’s syntactic structures into the language of a specific machine architecture. We will use this approach to define the semantics of our example language CL. The approach will be in two steps. We will start by defining an abstract machine, a simple stack machine in this case, and then define a translation function based on the structure of the CL grammar, indicating for each (relevant) non-terminal the sequence of machine instructions to be generated. When a compiler is implemented for the language, its operational semantics will be used as a blueprint for the code generation implementation.

Notice that in the operational approach, the goal of describing the behavior of a program on a set of input is somewhat indirect. The translation function describes how to convert a CL program into an executable form for a specific (but abstract) machine; we have to run that program on the given input in order to determine the output. What we will do is to transfer the dynamic semantics problem for CL to the target machine, whose dynamic behavior is known. We know how the machine behaves because we can give it instructions and watch it work.

As indicated, in order to define a dynamic semantics for CL we will define a function `trans` on the set of (relevant) CL non-terminal grammar symbols. A relevant non-terminal is one whose structure somehow involves computation. For CL the only “irrelevant” non-terminal is **Declaration**. Declarations are present only for static semantic purposes. However, declarations do have a role in dynamic semantics since it is

the type of a variable which determines, for example, whether to apply a floating point add or an integer add to a pair of arguments.

## The CL Stack Machine

The CL machine is a very simple stack machine. The machine has a memory to hold program instructions and a separate stack for computation. Programs will always be loaded into memory at offset zero. There are two registers for the machine: the PC register (program counter) holds the address of the next instruction to be executed, while the TOS (top of stack) register is the offset in the stack of the next free stack slot. The software architecture expects space starting at the base of the stack for program variables – each variable will have an address that is its offset on the stack. Computation will take place by pushing values onto the stack and operating on them with arithmetic operations. It is important to realize that the stack can hold both integer and floating point values but that arithmetic operations require that their operands have the same type. So that mixed mode arithmetic can be implemented, the machine provides an instruction `convert` which converts an integer value to floating point form. The table in Figure 2.7 summarizes the CL machine instruction set.

Instruction	Behavior
<code>iadd, isub, imul, idiv, imod</code> <i>the operations work only if p1 and p2 are int</i>	<code>pop stack → p1</code> <code>pop stack → p2</code> <code>stack → (p1 op p2):stack</code>
<code>fadd, fsub, fmul, fdiv</code> <i>the operations work only if p1 and p2 are float</i>	<code>pop stack → p1</code> <code>pop stack → p2</code> <code>stack → (p1 op p2):stack</code>
<code>push x</code>	<code>stack → x:stack</code>
<code>load</code>	<code>pop stack → d</code> <code>stack → stack[d]:stack</code>
<code>store</code>	<code>pop stack → v</code> <code>pop stack → d</code> <code>v → stack[d]</code>
<code>convert</code> <i>v must be int</i>	<code>pop stack → v</code> <code>stack → convert2float(v):stack</code>
<code>halt</code>	<code>stop fetch/execute cycle</code>

“`stack → x:stack`” means that the value `x` is pushed onto the stack. Names `p1`, `p2`, `x`, `v`, `d` represent hidden special purpose registers. `convert2float` is a built in function to convert from integer to float form.

Figure 2.7: CL Machine Instruction Set

So what does a program look like for this machine? Consider the example CL program we have referenced several times above.

```
{ int a; float b; a = 3; b = a*4.0 }
```

This program describes two computations for two variables ‘`a`’ and ‘`b`’. We can implement this on the CL machine by setting aside the lowest two positions on the stack for the two variables, ‘`a`’ at offset 0



and 'b' at offset 1, and then computing the two expressions in the assignment statements and storing the resulting values at the appropriate offsets. Here is an annotated CL program for this computation.

```

0  push 0      allocate space for 'a' at offset 0
1  push 0      allocate space for 'b' at offset 1
2  push 0      offset for 'a'
3  push 3      push 3 before storing at 'a'
4  store       a = 3
5  push 1      b = a*4.0
6  push 0
7  load
8  convert
9  push 4.0
10 fmul
11 store
12 halt       stop fetch/execute cycle
    
```

Notice in this program that when execution reaches instructions 5 and 12 that the execution stack will contain only the reserved space for 'a' and 'b'. Also, the `convert` instruction at address 8 is necessary since the value loaded from 'a' is in integer form but must be multiplied (at instruction 10) by the value 4.0 which is in float form – the instruction `fmul` requires both of its arguments to be in float form. Figure 2.8 shows a trace of the execution of this program – remember when looking at the diagram that `a` and `b` are at stack offsets 0 and 1, respectively. Notice that the stack displayed above a particular PC value is the state of the stack before the execution of the instruction, so the result of an execution is reflected in the stack above the next PC value.

4														
3										4.0				
2														
1														
<b>Stack 0</b>	0	0	0	0	3	3	3	3	3	3	3	3	3	3
<b>PC</b>	0	1	2	3	4	5	6	7	8	9	10	11	12	

Figure 2.8: Trace of CL Program Execution

### Preparation for Translation

So far we have talked about syntax and static semantics without reference to how our languages, IEL and CL, are actually processed by a compiler. We've been able to ignore this aspect because we have been interested only in how to describe the static view of our languages. But in the current discussion of dynamic semantics we should no longer ignore this. The back-end of a compiler, in which the code generation is done, takes place after the input string has already been parsed with no errors found. The back-end, or synthesis phase, does not look at the input string. Rather, it looks at a representation of that string that has been built in memory by the parser; the representation is called syntax tree – see the discussion in Section 1.4.

As described in Section 1.4 both semantic checking and code generation (dynamic semantics) is carried out by traversing the syntax tree rather than by re-reading the input string. The fact that semantic checking and code generation take place after parsing is done, means that we have the opportunity to use a more compact grammar for the syntax tree. The parser generates the syntax tree as it carries out the parsing of the input string. So as the parser validates the syntax of the input string based on the non-ambiguous grammar, the parser can construct a more compact syntax tree by eliminating all non-data carrying elements and by squeezing out the level information (**Term** and **Factor**) from the expression description part of the grammar.

Let's start with the first two rules of the grammar.

```
Program → lcbT StmtList rcbT
StmtList → Stmt | Stmt commaT StmtList
```

Let's apply the strategy just discussed. We see in the first rule we have two punctuation tokens **lcbT** and **rcbT** and in the second rule there is one such token **commaT**. So we will eliminate them from our new grammar. But when we do the elimination, the first two rules indicate that **Program** is simply the same as **StmtList** – so we could make the two rules into one. But it is usually the case that there is some special code to be generated at the beginning of a program, so it will be best if we leave the two rules as is. The rules for **Stmt** and **Declaration** remain unchanged, but the rule for **Assignment** has the **assignT** token which can be eliminated. In this way the first half of the grammar is reduced to the following.

```
Program → StmtList
StmtList → Stmt | Stmt StmtList
Stmt → Declaration | Assignment
Declaration → typeT identT
Assignment → identT Exp
```

The remaining rules describe arithmetic expressions and here we simply undo all the layering we did earlier – adding **Term** and **Factor** to eliminate precedence ambiguity. If we return to our original arithmetic expression grammar structure (see Figure 2.3 on page 32) we will change the five expression rules in the grammar shown in Figure 2.6 as follows.

```
Exp → Exp (addT | subT | multT | divT | modT) Exp
     → intT | floatT | identT | lpT Exp lpT
```

But once again we have two punctuation tokens in the second rule (**lpT** and **rpt**) so they can as well be eliminated. But then, as with **Program** and **StmtList** earlier, we have a rule equating **Exp** and **Exp** – this can obviously be removed. So we end up with the following compact grammar that can define the structure of our CL syntax tree.

```
Program → StmtList
StmtList → Stmt | Stmt StmtList
Stmt → Declaration | Assignment
Declaration → typeT identT
Assignment → identT Exp
Exp → Exp (addT | subT | multT | divT | modT) Exp
     → intT | floatT | identT
```

With this definition we can then generate a syntax tree for our example string – it is displayed in Figure 2.9. Now imagine doing a depth-first traversal of this tree. If we think about generating code then

each time we visit a node requiring code generation that can be done. In the next section we will use the syntax tree structure, represented by the new compact grammar, to formally define the operational semantics of CL.

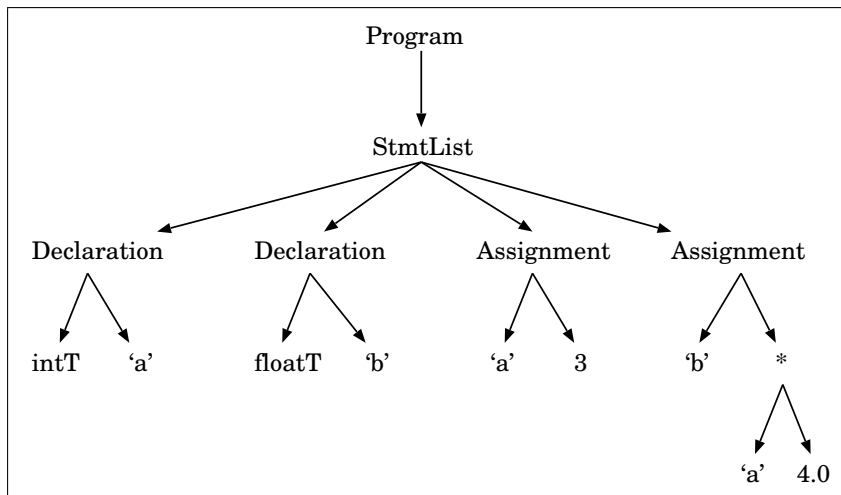


Figure 2.9: Syntax Tree for `int a, float b, a = 3, b = a*4.0`

There is one more important capability the syntax tree must provide during code generation. When there is an assignment such as `b = a*4.0` the translation must know what the type of `b` is and what the type of the expression `a*4.0`. Similarly, when the expression `a*4.0` is encountered the translation must know the type of each argument for the multiplication; with this information the `a` can be converted to floating point format and the correct machine instruction, `fmul` (floating point multiply), can be generated. What we assume is that the static semantic checking has already taken place and that, based on the information about identifier types, types have been assigned to each expression and subexpression in the parse tree. We will see this strategy implemented in the Syntax Tree Tutorial in Chapter 14.

### The CL Translation Function `trans`

While the strategy implied by the translation example above can be easily applied for other by-hand translations, what we want is a formal definition that can be used for automatic translation. For CL the translation function `trans` is a recursive function defined on the set of non-terminal symbols in our compacted grammar; the value for a particular non-terminal is determined by the right side(s) of the rule(s) for the non-terminal. For example, we would define `trans(Assignment)` by considering the appropriate pattern of code based on the right side of the corresponding grammar rule. The code for an assignment would require to first push onto the stack the address of the target variable, then generate the code for the statement's expression and finally generate a store instruction whose execution will pop the expression's value and target variable address from the stack and store the value at the specified address:

```

if identT.type = float and Exp.type = int then
trans(Assignment) = push identT.offset, trans(Exp), convert, store
else
trans(Assignment) = push identT.offset, trans(Exp), store

```

Notice how we take advantage of the types of the identifier and expression to generate the `convert` instruction when necessary. Also realize that the case when the identifier has type `int` and the expression has type `float` is not mentioned because we are assuming static semantic checking has been done – this case would be flagged as an error! The resulting code sequence, of course, can only be given completely after the evaluation of `trans(Exp)`. This is the strategy followed for each non-terminal of the grammar.

The translation function `trans` can now be easily defined in terms of the instruction set given in Figure 2.7. The definition is given in the following sequence of recursive definitions. Notice that the translation for the first rule illustrates the special code generated for the `Program` rule.

- 1 `Program`  $\rightarrow$  `StmtList`  
`trans(Program) = push 0, ..., push 0, trans (StmtList) , halt`  
*where the number of push 0 instructions = the number of declarations*
- 2 `StmtList`  $\rightarrow$  `Stmt`  
`trans(StmtList) = trans (Stmt)`
- 3 `StmtList`  $\rightarrow$  `Stmt StmtList`  
`trans(StmtList) = trans (Stmt) , trans(StmtList)`
- 4 `Stmt`  $\rightarrow$  `Assignment`  
`trans(Stmt) = trans (Assignment)`
- 5 `Declaration`  $\rightarrow$  `typeT identT`  
`trans(Declaration) = null`
- 6 `Assignment`  $\rightarrow$  `identT assignT Exp`  
*if Exp.type is int and identT.type is float then*  
`trans(Assignment) = push identT.offset, trans (Exp) , convert, store`  
*else*  
`trans(Assignment) = push identT.offset, trans (Exp) , store`
- 7 `Exp`  $\rightarrow$  `Exp1 addT Exp2` (follow same pattern for `subT`, `multT`, `divT`.)  
*Exp.type = float if Exp<sub>1</sub>.type = float and Exp<sub>2</sub>.type = float then*  
`trans(Exp) = trans (Exp1), trans (Exp2), OP2`  
*elseif Exp<sub>1</sub>.type = int and Exp<sub>2</sub>.type = float then*  
`trans(Exp) = trans (Exp1), convert, trans (Exp2), fadd`  
*lseif Exp<sub>1</sub>.type = float and Exp<sub>2</sub>.type = int then*  
`trans(Exp) = trans (Exp1), trans (Exp2), convert, fadd`  
*else*  
`trans(Exp) = trans (Exp1), trans (Exp2), iadd`  
`Exp.type = int`
- 8 `Exp`  $\rightarrow$  `Exp1 addT Exp2`  
`trans(Exp) = trans (Exp1), trans (Exp2), imod`  
`Exp.type = int`
- 9 `Exp`  $\rightarrow$  `intT`  
`trans(Exp) = push intT.value`  
`Exp.type = int`

```

10 Exp → floatT
    trans(Exp) = push floatT.value
    Exp.type = float

11 Exp → identT
    trans(Exp) = push identT.offset, load
    Exp.type = identT.type

```

### Applying the Function trans

The definition of `trans` presented above clearly involves a considerable amount of recursion. Since the recursion can obscure to some extent the ultimate translation that is generated, we will track an application of `trans` to a small example CL program – in fact the same example we have looked at several times already.

```
{ int a, float b, a = 3, b = a * 4.0 }
```

There is a small bit of preliminary data we must derive from the text of the example program. First, we need to know how many variables are defined in the program – in this case there are two. We also must assign a stack offset to each identifier to serve as its address in the translated code. The following table includes the necessary information that can be collected by the analysis phase of a translator.

number of variables	2	
variable	type	offset
a	int	0
b	float	1

Based on this data we can generate the following CL stack machine code. Notice that the sequence of evaluations is determined by the grammar rules used in a derivation of the string being translated. The rule number on a particular line in the evaluation sequence indicates the number of the `trans` definition line applied to get the code following the '=' sign. In addition, the current application of `trans` on each line is underlined. Finally, the two applications of `trans` rule 5, `trans(Declaration) = null`, result in the empty string being generated.

```

rule  trans( { int a, float b, a = 3, b = a * 4.0 } )
1     = push 0, push 0, trans( { int a, float b, a = 3, b = a * 4.0 } ), halt
3     = push 0, push 0, trans( int a ), trans( { float b, a = 3, b = a * 4.0 } ), halt
5     = push 0, push 0, trans( { float b, a = 3, b = a * 4.0 } ), halt
3     = push 0, push 0, trans( float b ), trans( { a = 3, b = a * 4.0 } ), halt
5     = push 0, push 0, trans( { a = 3, b = a * 4.0 } ), halt
3     = push 0, push 0, trans( a = 3 ), trans( { b = a * 4.0 } ), halt
6     = push 0, push 0, push 0, trans( 3 ), store, trans( { b = a * 4.0 } ), halt
9     = push 0, push 0, push 0, push 3, store, trans( { b = a * 4.0 } ), halt
2     = push 0, push 0, push 0, push 3, store, trans( b = a * 4.0 ), halt
6     = push 0, push 0, push 0, push 3, store, push 1, trans( a * 4.0 ), store, halt
7     = push 0, push 0, push 0, push 3, store, push 1, trans( a ), convert,
      trans( 4.0 ), fmul, store, halt
11    = push 0, push 0, push 0, push 3, store, push 1, push 0, load, convert,
      trans( 4.0 ), fmul, store, halt
10    = push 0, push 0, push 0, push 3, store, push 1, push 0, load, convert,
      push 4.0, fmul, store, halt

```



**The Front-end**  
**– Analysis Principles –**





# Chapter 3

## Introduction

The Prelude discussed the two basic topics that are interwoven in this text: *theory* and *practice*. Practice, which was the topic of Chapter 1, refers to everything dealing with the implementation of a programming language translator. The discussion included three ways of thinking about such a translator: the intuitive view (what are the critical pieces), the data-driven view (how does it function), and the object-oriented view (what is the data-access structure). These views together give a very useful view of the architecture of a translator. But how can we be sure that the implementation we build is valid – can we guarantee that the translator will translate only correctly structured programs written in the target language?

Theory is important because it can provide formal descriptions of our target language, descriptions that describe the form as well as the static and dynamic semantics. In Chapter 2 we looked at both intuitive and formal methods for describing various parts of a target and found that the formal (theoretical) descriptions were better because they could embody descriptions of subtle requirements, requirements difficult to describe completely and unambiguously in an informal way.

The rest of the book is in three parts. The current part, *The Front-end – Analysis Principles*, gives a complete description of the theory and practice applied to the front-end of a translator. A new language, named PDef, is defined and used throughout as the target language in implementing the components of the front-end of a translator for PDef. Since our interest is only in the front-end, there will be no back-end created for the language and we will refer to our software system as a PDef *recognizer* – i.e., a recognizer is simply a front-end without the back-end.

The next part in the book, titled *The Front-end – Analysis Tutorials*, is a companion to the current part. It contains a sequence of tutorials, each focusing on the implementation of a particular component of the PDef recognizer. In fact most of the chapters in this part are paired with a tutorial: you read Chapter 6, *Syntax Analysis – Theory and Practice*, and then work through the tutorial in Chapter 13, *PDef Parser Tutorial*. The chapters are paired as follows.

Chapter	<i>Analysis Principles</i>		Chapter	<i>Analysis Tutorials</i>
4	<i>An Example Language – PDef</i>			<i>no tutorial</i>
5	<i>Lexical Analysis</i>	⇒	12	<i>PDef Tokenizer Tutorial</i>
6	<i>Syntax Analysis</i>	⇒	13	<i>PDef Parser Tutorial</i>
7	<i>Syntax Tree</i>	⇒	14	<i>PDef Syntax Tree Tutorial</i>
8	<i>Semantic Analysis</i>	⇒	15	<i>PDef Semantics Tutorial</i>
9	<i>LR Parsing</i>	⇒	16	<i>PDef LR Parser Tutorial</i>
10	<i>Attribute Grammars</i>			<i>no tutorial</i>

The third and final part of the book is titled *The Back-end – Synthesis Techniques* and covers all the

basic aspects of code generation and code optimization.

The chapters in this part take us right through the basic elements of the recognizer. Here we give a quick overview of each. Because the recognizer design principles discussed in Section 1.2.1 are so important to the discussions to come we will mark each chapter with relevant principles.

### **An Example Language – PDef :**

The work discussed in the chapters and tutorials to follow focus on the implementation of the recognizer. This requires that we choose a target language for recognition – the language is called PDef and is in a direct evolutionary line from IEL and CL. While PDef is no more exciting than its two ancestors, it does have all the basic syntactic and static semantic structures that need to be addressed in real languages. The advantage of a language like PDef is that it is as small as possible but still has the structures we want to study. In this way there are no extraneous structures that will add little to understanding the principles but can be a big distraction. The goal here is for you to come out of the tutorial chapters with a very clear idea of how a front end works, how it is structured, and how it is implemented.

### **Lexical Analysis:** [Correctness]

This chapter introduces the *finite state machine*, as an alternative formalism to regular expressions. Conveniently, for each regular expression there is an equivalent finite state machine and the finite state machine turns out to be easier to implement in software. We discuss designing a finite state machine for a token language and address basic strategies for their implementation in Java. Because the finite state machine is easy to define and implement the techniques discussed support the Correctness Principle.

### **Syntax Analysis:** [Correctness, Early Warning]

This chapter presents an overview of parsing techniques for context free grammars. We discuss two forms of parsing algorithms – top-down parsing and bottom-up parsing. In this chapter we focus on top-down parsing and discuss strategies for implementing a parser for a given CFG in Java. A small set of six grammar rule forms is identified and implementation patterns are developed for each rule form. These forms will be useful in the Parser Tutorial. The ease of translating structure from a CFG to a Java implementation supports the Correctness Principle. In addition, the top-down strategy allows error recovery so that errors can be identified and reported as soon as possible – this directly supports the Early Warning Principle.

### **Syntax Tree:** [Correctness]

This chapter discusses the design and implementation of a syntax tree in Java. The structure of the syntax tree is driven by the structure of a language's grammar, so the design strategy presented is based on the six grammar rule forms given in the Syntax Analysis chapter. This data structure provides an internal representation of the input program's syntactic structure – the way the structure is determined and implemented further extends support for the Correctness Principle.

### **Semantic Analysis:** [Correctness, Early Warning]

This chapter introduces and discusses the basic notion of *scope of a declaration* and how this notion guides the design and implementation of a *symbol table*. The symbol table is integrated into the syntax tree and, in this way, facilitates the identification of static semantic errors. Again, the structure of the symbol table and its integration into the syntax tree means that the work supports the Correctness Principle. Also, the fact that static semantic errors are identified and reported before run-time means there is also support for the Early Warning Principle.

**LR Parsing:** [Correctness, Early Warning]

In Chapter 6 we discuss the structure and implementation of a top-down parser. But when language translators are automatically generated (using tools such as Lex and Yacc) the parsing technique used is bottom-up parsing. In this chapter we discuss the structure and implementation of a bottom-up parser. As with the top-down parser, the implementation of a bottom-up parser supports both the Correctness and Early Warning Principles.

**Attribute Grammars:** [Correctness]

When we have discussed static semantics, we have used less formal methods for description. In this chapter we discuss the notion of *attribute grammar* as a mechanism for formally describing the static semantics of languages.



## Chapter 4

# An Example Language – PDef

We begin by defining the language PDef, which will be our example language for the chapters dealing with the analysis phase. PDef is designed to be simple but structurally similar to familiar languages such as Java or C++, thus presenting the standard design problems of a compiler front-end. PDef is, in fact, an extension of the language CL, which was introduced in Section 2.2. In brief, the language PDef includes at its base the list of declaration and assignment statements seen in the CL, but in addition allows these lists to be nested, i.e., expands the definition of a statement to include lists themselves. The simple structure of PDef allows us to focus on the nesting, which is the source of syntactic and semantic complexity.

In the rest of this chapter we will give an informal as well as a formal view of the language PDef, including formal descriptions of its lexical, grammatical, and static semantic characteristics. Since these chapters and their paired tutorials are meant to focus on analysis and not synthesis, we do not define a dynamic semantics for PDef.

### 4.1 Informal Definition of PDef

The name PDef stands for **P**arenthesized **D**efinitions. The best way to see this is to think of PDef as CL with an additional statement type, namely, the list. The following examples illustrate the syntactic structures of the language (beware that these examples are syntactically correct but may contain semantic errors).

```
{ int xyz }
{ float a, a = 2.3, { float b, b = a*2, int a, a = 3, { b = a * 3 } } }
{ float a, a = 2.3, { int b, b = 3.1, a = b*(a+2) }, a = a * b }
{ float a, a = 2.3, { b = 3.1, a = b*(a+2) }, int b, a = a * b }
```

Remember that without the nesting, PDef is identical to CL. But in order to have a unified definition of PDef, we redefine these basic level entities here.

- A declaration statement is a type name, one of **int** or **float**, followed by an identifier.
- An assignment statement consists of an identifier and an expression separated by the assignment operator ‘=’.
- A list statement is a pair of curly braces, ‘{ }’ containing a comma separated list of declaration statements, assignment statements, and lists – there must be at least one element in any list.

There are a few restrictions in forming lists – these fall in the category of typing requirements. First, a list cannot contain two declaration statements for the same identifier. In addition, identifiers must be declared before they are used in assignment statements. The declaration can occur before the assignment in the same list or in a containing list, as long as the declaration comes textually before the assignment. If there is more than one such declaration, then the type for the identifier is determined by the closest declaration, in terms of nesting level.

The assignment statement `a = b*(a+2)`, in the third example above, is OK since `a` is declared in the outer list before the nested list occurs; the assignment `a = a*b` in the same example, on the other hand, is not OK, since there is no declaration of `b` in the list and it occurs in the outer most list. The fourth example is just the opposite: the statement `a = b*(a+2)` is illegal because there is no declaration of `b` textually before the expression, but `a = a*b` is OK because both `a` and `b` are declared before the assignment in the outer list in which this assignment occurs.

The second example is more interesting. In the second assignment statement `b = a*2` the `a` gets its type from the declaration statement in the outer list while the `b` gets its type from the declaration preceding it in the same list. The `a` in the assignment `a = 3` has two declarations preceding it – but the `int a` is textually closest, so that is the declaration which determines the type binding for `a`. Finally, the `a` in the assignment `b = a*3` in the inner most list also has two declarations preceding it – but as in the previous case, it is the `int a` in the enclosing list that dictates its type, since it is textually closest.

The type rules for PDef are exactly the same as for CL. The type associated with a literal is exactly as in CL – an integer or floating point literal has type `int` or `float`, respectively. When two elements are combined by an arithmetic operation, the type of the combination is determined by the types of the two elements. When an arithmetic operation acts on two `int` values (identifiers or literals) the combination has type `int`; if the operation is other than `%` and at least one of the elements has type `float`, then the combination has type `float`. As a special case, the mod operator `%` can be applied only to elements of type `int` and the combination has type `int`. These three combination rules make it possible to determine the type of any expression.

The final rule relates the types of the identifier and expression on either side of an assignment statement. First, if the type of the identifier on the left is `int` then the right side must be any expression whose type evaluates to `int`. If the identifier on the left has type `float`, then the right side must be an expression whose type evaluates to either `int` or `float`.

## 4.2 PDef Syntax

The formal description of PDef applies the techniques discussed in Chapter 2. Accordingly, this section will present a regular expression describing the PDef tokens, a context free grammar describing the syntax of PDef, and finally a simplified attribute grammar describing the static semantics of PDef.

### The PDef Alphabet

We could omit this level of description, since the alphabet should be easily extracted from the lexical level description. But for completeness we include the following listing of the PDef alphabet.

```
alphabet = { a ... z
             A ... Z
             0 ... 9
             .
```

```

    =
    (
    )
    {
    }
    ,
}

```

## The PDef Tokens

You may have noticed that the languages PDef and CL have the same set of tokens. Consequently, we use the regular expression from CL to describe the tokens of PDef. Since our interest in PDef is to actually implement a recognizer, we need to go beyond our discussion of tokens for CL. Here's the problem. If you are reading a text file and trying to pick out the tokens as they appear, what will you think if the next eight characters are `intfloat`? You might want to say that there are two tokens, `int` and `float`. But how can you be sure? What we have to specify for each token is what characters can immediately follow each token.

It is important to understand that this issue of what can follow a token is **not** a grammar issue – it is purely an issue for the tokenizer. If the sequence `'int+=23'` appears in a file the tokenizer will report four tokens: `typeT`, `addT`, `assignT`, `intT`. The parser, on the other hand, would complain that a token `typeT` cannot be followed by the `addT` token – the problem is a grammatical judgement! This has already been addressed in Section 2.1 (see page 18), where the spacing between tokens was discussed. In the table in Figure 4.1 we present the regular expression descriptions of the PDef tokens and an indication of what characters can follow each token.

Token Class	Regular Expression	Termination Characters
<code>addT</code>	<code>+</code>	any character
<code>subT</code>	<code>-</code>	"
<code>multT</code>	<code>*</code>	"
<code>divT</code>	<code>/</code>	"
<code>modT</code>	<code>%</code>	"
<code>commaT</code>	<code>,</code>	"
<code>assignT</code>	<code>=</code>	"
<code>lpT</code>	<code>(</code>	"
<code>rpT</code>	<code>)</code>	"
<code>lcbT</code>	<code>{</code>	"
<code>rcbT</code>	<code>}</code>	"
<code>typeT</code>	<code>int   float</code>	non-letter
<code>intT</code>	<code>0   [1 - 9][0 - 9]*</code>	non-digit
<code>fltT</code>	<code>(0   [1 - 9][0 - 9]*) . [0 - 9]+</code>	non-digit
<code>identT</code>	<code>[a - zA - Z]+</code>	non-letter

Figure 4.1: Regular Expression for PDef Tokens

There is an ambiguity here. The token class `typeT` has two values, `int` and `float`, but each of these is also a value of the token class `identT`. We could reflect this fact by adding a note to the `identT` regular

expression that `int` and `float` are not `identT` values. We will see when we talk about implementation that these special cases are easily handled.

## The PDef Grammar

While the tokens of PDef are the same as those of the CL, the grammar must be changed to reflect the fact that a statement list can also be a list entry. While we could add the non-terminal `Program` to the rule for `Stmt` in the CL grammar, it adds clarity to the grammar if the start symbol `Program` does not serve a dual purpose. We therefore add a new structure to describe the statement list structure. The resulting grammar is displayed in Figure 4.2.

```

Program    → Block
Block      → lcbT StmtList rcbT
StmtList   → Stmt { commaT Stmt }
Stmt       → Declaration | Assignment | Block
Declaration → typeT identT
Assignment → identT assignT Exp
Exp        → Exp (addT | subT) Term
           → Term
Term       → Term (multT | divT | modT) Factor
           → Factor
Factor     → intT | floatT | identT | lpT Exp rpT

```

Figure 4.2: Context Free Grammar for PDef

### Activity 5 –

There are a couple of questions that come to mind after seeing the earlier examples and then the formal PDef grammar in Figure 4.2.

1. Are empty lists allowed by the grammar?
2. Can extra commas be inserted between list entries or at the end of a list?
3. Are commas required after nested lists?

In other words, are the following phrases legal according to the PDef grammar?

```

{ }
{ float a, a = b, int b,,, a = b, }
{ char a, { a = b } int b, { a = b, float b, b = a }, a = b }

```

For each question show how derivations in the grammar support your answers.

---

## 4.3 PDef Static Semantics

With PDef being a derivative of CL, it will not be surprising that the PDef static semantics is derived from that for CL. The crucial difference between the two languages is that in PDef we can have nested lists. In



a PDef program, then, the most deeply nested lists will actually be CL lists – and at that level they have the same properties. We organize our description of the PDef static semantics as we did the description for CL, as a list of semantic rules.

Before listing the semantic rules we need to establish a bit of terminology. When we talk about a *statement in a PDef list* we mean one of the assignment or declaration statements directly in the list – a list within a list will be called a *statement list*. So in the following program

```
{ float a, a = 3, { int b, b = 4, a = b*a }, a = a+4.0 }
```

the assignment ‘`b = 4`’ is a statement in the inner list but NOT in the outer (enclosing) list; the outer list has just three statements, ‘`float a`’, ‘`a = 3`’, and ‘`a = a+4.0`’, and one statement list, ‘`{ int b, b = 4, a = b*a }`’.

### Types:

The only types in PDef are the two that are built in, `int` and `float`.

### Identifier Declaration:

1. An identifier can appear in only one declaration statement in a list. Note that this is a restriction on declaration statements in a list, but does not prevent an identifier being re-declared in a nested list.
2. When an identifier is used in an assignment statement that statement must be preceded, in the same list or in an enclosing list by a declaration for that identifier. If there are two or more such declarations the one appearing in the same list or the one appearing in the nearest enclosing list will determine the type attribute of the identifier.

### Expression Type:

The type of an expression is defined as follows:

1. If the expression is simply an identifier or a literal, then the expression type is the type of the literal or the type bound to the identifier in its required declaration.
2. If the expression has the form `exp1 op exp2` and `op` is not ‘`%`’, then its type is `int` if both arguments have type `int` and `float` if at least one of the arguments has type `float`.
3. If the expression is `exp1 % exp2`, then both arguments must have type `int` and the expression type is `int`.

### Type Compatibility:

In an assignment statement the type of the expression (on the right) must be assignment compatible with the identifier on the left. The type `int` is assignment compatible with both `int` and `float`; the type `float` is assignment compatible only with itself.

Notice that the Expression Type and Type Compatibility entries for PDef are unchanged from those for CL. The real characteristic of interest for PDef is the way a type is associated with an identifier.

### ☛ Activity 6 –

Consider the following PDef program.

```
\{ float b, float a, b = 3, \{ int b, b = 4, a = b \},
  \{ b = 5.0, int b, b = 2 \}, b = b*5 \}
```

Indicate which declaration statement determines the type of each identifier in each assignment statement.

---

## Chapter 5

# Lexical Analysis – Theory and Practice

Referring back to the UML class diagram in Figure 1.7, which describes the object-oriented structure of a translator front-end, we see that lexical analysis is implemented by a tokenizer – an object that extracts from the input stream the sequence of tokens for the language. Since the formal description of the tokens to be recognized is given by a regular expression, as discussed in Section 2.2.2, we must find a mechanism that provides an algorithmic link between a regular expression and a tokenizer.

In this chapter we will introduce the *finite-state machine* as the algorithmic link between regular expressions and tokenizers – it is also the key to the tokenizer’s implementation of the Correctness Principle. In the process, however, we will uncover unanticipated problems of token description and recognition.

### 5.1 Finite-state Machines

A finite-state machine is a simple but powerful concept that can be used to model complex systems, such as token recognizers, and simple systems such as traffic lights. A single traffic light has three possible states: red, green, amber. As time goes by the state of the light changes, but always in a particular pattern: green to amber to red and back to green. In France the cycle is reversed: red to amber to green and back to red. The state transitions occur when a particular amount of time has passed. In more sophisticated systems the transition from red to green can be triggered by the arrival of a car (a weight sensor in the road bed) or by a pedestrian pressing a button.

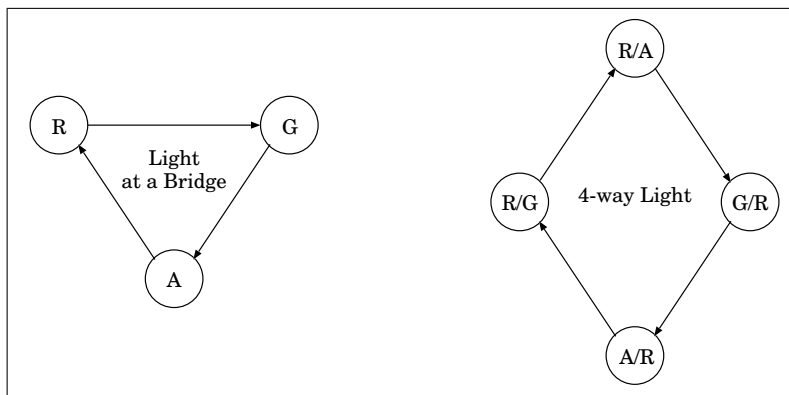


Figure 5.1: Finite-state Machines Modeling Traffic Lights

The two diagrams in Figure 5.1 are standard finite-state machine diagrams modeling, on the left, one direction of a two direction stop-light and, on the right, the two-direction stop-light. Looking at the finite-state diagram on the left we see the three states, circles labeled with the color names, and the arrows indicating direction of transition. The machine will exist in a single state at a time and transition from one state to the next based on passage of time. The diagram on the right models the light combinations present in a two-direction stop-light. There are four states here, with each labeled by the pair of colors present in the two directions.

Another simple application of the finite-state machine model can be seen in an algorithm for counting the number of words in an input text file – for our purposes we will make the simplifying assumption that punctuation is never separated from a word by whitespace (blank, tab, and new-line characters). Figure 5.2 shows a finite-state machine diagram which carries out the word counting. The idea is that the word counting program goes through two states – it is either currently reading letters (i.e., is scanning a word) or currently reading whitespace characters. The two states are appropriately named in the diagram. Notice that in this diagram that each transition is labeled by a category of characters. The diagram indicates that if the machine is in the **Word** state and a letter is read then the machine transitions back to the **Word** state; if a whitespace character is read then the machine transitions to the **Space** state. The **Space** state behaves in a reverse way, with a transition back to itself if a whitespace character is read and a transition to **Word** is a letter is read. One interesting thing on the **Space** → **Word** transition is that the transition label has an associated *action*, which is carried out each time the transition is taken. This action facilitates the word counting. A final point is the wedge shape marking the top of the **Space** state – this designates the **Space** state as the start state for the machine.

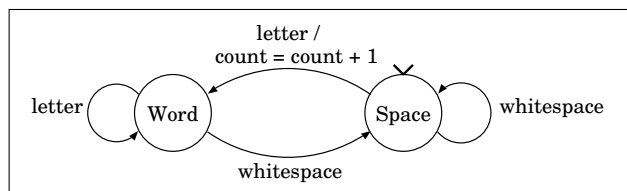


Figure 5.2: Finite-state Machine for Word Counting

The examples discussed above are simple and informal – notice for the stop-lights the finite-state machine diagrams have no designated states nor transition labels and there is no indication of how a machine terminates. The notion of finite-state machine is formally defined as a mathematical structure whose definition follows.

#### DEFINITION 9 (FINITE-STATE MACHINE)

A finite-state machine over an alphabet  $\Sigma$  is defined by the following components:

1. a finite set of states  $Q$ ,
2. a state  $S$  in  $Q$  designated as the start state,
3. a subset  $F$  of  $Q$  being the set of final states,
4. a transition function  $\lambda$  defined as follows:  
if  $s \in Q$  and  $a \in \Sigma$  then  $f(s, a) = t$ , for some  $t \in Q$ .

Notice that in condition 4 of the definition there is an implication that given a state and an input character

there will always be a transition to another state. We call this kind of finite-state machine *deterministic*, since its actions are completely specified – i.e., for every state and every alphabet character there is a single defined transition.

Figure 5.3 shows a standard finite-state machine diagram. The states are represented by circles and transitions by arrows. The character driving a particular transition is displayed toward the middle of the transition's arrow. The start state is marked by a wedge shape (in this case it is the state labeled S). Final states have a concentric circle inside. What may not be apparent from the diagram is the alphabet for the finite-state machine, but we can assume it is the set of all the transition characters – so for this machine the alphabet is  $\{0, 1\}$ . This is all summarized on the right of the diagram. Notice that the transition function  $\lambda$  is conveniently defined using a table.

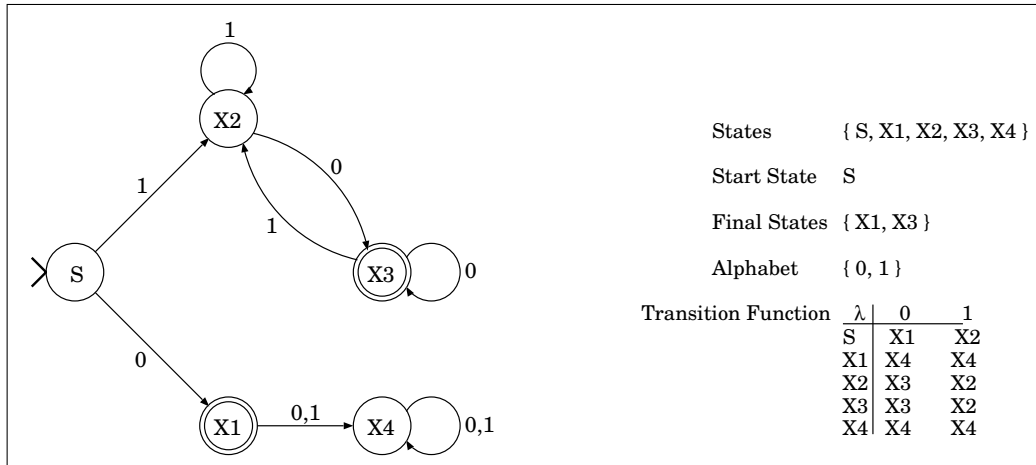


Figure 5.3: A First Finite-state Machine

This definition describes the static side of finite-state machines. But we can see in the diagram a clear notion of movement – if we are in the state S and we see a 0 then we should end up in the state X1. In fact we can imagine the string ‘1101’ driving the machine from S to X2 to X2 (again) to X3 and finally back to X2. The idea of the final states is to distinguish those strings that the machine accepts and those it rejects – if it ends in a final state the machine accepts and otherwise it rejects.

### Activity 7 –

For the finite-state machine in Figure 5.3 trace the sequence of states visited for each of the following strings, assuming the machine starts in its start state. For each also indicate whether the string is accepted or rejected. Give your answers as illustrated in the first problem.

- 1101

Answer:  $S \xrightarrow{1} X2 \xrightarrow{1} X2 \xrightarrow{0} X3 \xrightarrow{1} X2$  – rejected

- 0

- 1

- 0100

- 10111000

Try to characterize (describe) the set of all accepted strings (think of the strings as binary numbers).

---

You should have noticed that the state **X4** in our finite-state machine doesn't do much – it seems like a black hole, sucking up all characters it encounters. Definition 9 above indicates that for every state there must be a transition for each symbol in the alphabet. The state **X4** is required because we only want to accept a string starting with '0' if the string has length 1 – otherwise we reject the string by transitioning to the state **X4**.

We need a way to talk about this notion of strings accepted by a finite-state machine. We will use the notation

$$(X, w) \xRightarrow{*} (Y, \epsilon)$$

to indicate that if we start in state  $X$  with the string  $w \in \Sigma^*$  then there is a sequence of transitions that will drive the machine to the state  $Y$  with the input consumed. So we can define the set of strings accepted by a finite-state-machine as follows.

$$\mathcal{L}(M) = \{ w \in \Sigma^* \mid (S, w) \xRightarrow{*} (Y, \epsilon), \text{ where } Y \in F \}$$

So given a finite-state machine we can talk about the language it accepts – this sounds quite useful. One of the important results for us is the fact that regular expressions describe exactly the same languages the finite-state machines accept. This fact is often stated as a theorem.

#### THEOREM 1

*Assume that  $\Sigma$  is an alphabet and that  $L$  is a language over  $\Sigma$ . Then there is a regular expression  $R$  such that  $\mathcal{L}(R) = L$  if and only if there is a finite-state machine  $M$  such that  $\mathcal{L}(M) = L$ .*

So this means that with each regular expression  $R$  we can find a finite-state machine that accepts the language described by  $R$  – exactly what we need to link lexical description to an implementation.

Of course, this begs the question how do we find the finite-state machine corresponding to a particular regular expression. There are two answers. First, there is an algorithm, which is described in the proof of the theorem above, that converts a regular expression to an equivalent finite-state machine. This is nice to know generally, so that if confronted with a complex regular expression, there is an automatic way to generate a tokenizer. The second answer is that it is typically easy to fabricate by hand a finite-state machine directly from the regular expression. We will investigate this by-hand approach in the PDef Tokenizer Tutorial (Chapter 12).

## 5.2 Designing a Finite-state Machine

In this section we will design a finite-state machine that accepts tokens of the PDef language. We will find two things: some tokens are very easy to handle (those of fixed length) and others present significant problems – even problems that require slight adjustments to our notion of finite-state machine.

Rather than trying to describe a more general strategy at the outset, we will dive right into the regular expression which describes the PDef token set. For convenience here is that regular expression for PDef – it also appears in Figure 4.1.

Token Class	Regular Expression	Termination Characters
addT	+	any character
subT	-	"
mulT	*	"
divT	/	"
modT	%	"
commaT	,	"
assignT	=	"
lpT	(	"
rpT	)	"
lcbT	{	"
rcbT	}	"
typeT	int   float	non-letter
intT	0   [1 - 9][0 - 9]*	non-digit
fltT	(0   [1 - 9][0 - 9]*) . [0 - 9] <sup>+</sup>	non-digit
identT	[a - zA - Z] <sup>+</sup>	non-letter

We focus first on the tokens in the table beginning with `addT` and continuing through `rcbT`. Each of these tokens is paired with a single-character regular expression. The idea for PDef, of course, is that any one of these characters is a token by itself – as mentioned earlier, this reflects the union ‘|’ operator. A finite-state machine to accept these tokens is quite simple – there can be two states, as start state and one final state and a transition from the start to final state labeled by each of these token symbols. The diagram in Figure 5.4(a) shows this machine in the usual form. Notice we have labeled the two states `Start` and `Done`. In Java, and many other languages, we have a comparison operator consisting of two

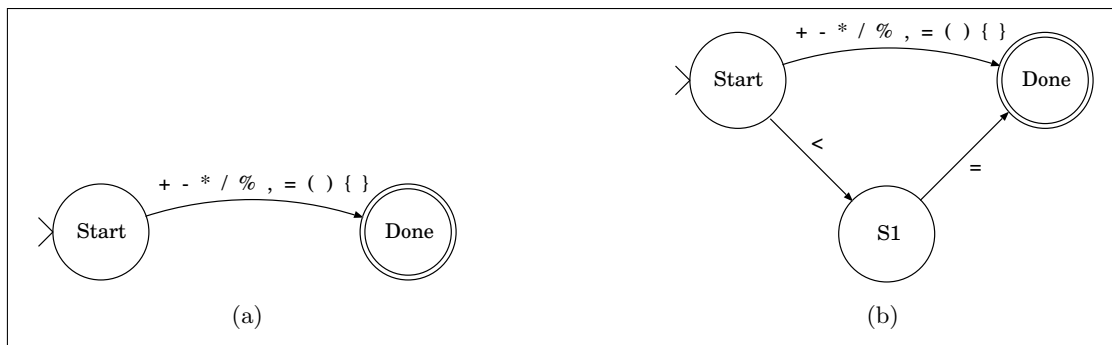


Figure 5.4: Simple Token-recognizing Finite-state Machines

characters – the less-than-or-equal operator ‘<=’. If we want to recognize such an operator we need to add a new state, transitioning to it from the start state on ‘<’, and then transitioning from it to the final state on ‘=’. The diagram in Figure 5.4(b) shows this finite-state machine. Notice in these two finite-state machines that the only way to get to the state `Done` is to see a correct token sequence.

### A Confession

OK, the diagrams in Figure 5.4 aren’t *really* finite-state machines, because there isn’t a transition for each state for each alphabet character. In fact, these diagrams are common and assume that there is a “black-hole” state, such as `X4` in Figure 5.3, to which all invalid character transitions would lead. Not surprisingly, adding in such a state and the transitions to interesting

finite-state machines would considerably complicate the diagrams – this is why such a state is often left out of a finite-state machine diagram.

The next natural token to add to our PDef finite-state machine would be `typeT`, which has two values `int` and `float`. But you should remember from our earlier discussion of these tokens that, since they also match the definition for `identT`, we must deal with them in a special way. Basically, our strategy will be to recognize an identifier and then at the last minute look to see if that identifier happens to be either `int` or `float`.

### Accepting Identifiers

We continue then with the next easiest token class to recognize, which is `identT`, which brings its own interesting recognition problems. Consider the string "count". This should certainly count as an `identT` token value. But there are more tokens buried within it – in fact, every substring of "count" is also an `identT` token value: "c", "cou", "nt", "t" to list just four of the 15 possible. Now, we can probably ignore those substrings not starting with 'c', but of the five that do start with 'c', which is the appropriate token value to pull from the input stream? This problem is actually more general and applies to any situation where two or more tokens begin with the same initial substring. In such cases we adopt the following token recognition rule:

**Longest Substring:** Starting with the current character, the longest string that qualifies as a token is chosen as the next token.

So assuming "count" is followed by a blank, then it should be taken as the next token value. This means that, in recognizing an `identT` value, the finite-state machine should not enter a final state until the entire identifier value has been seen. An almost complete finite-state machine of this form is given in Figure 5.5.

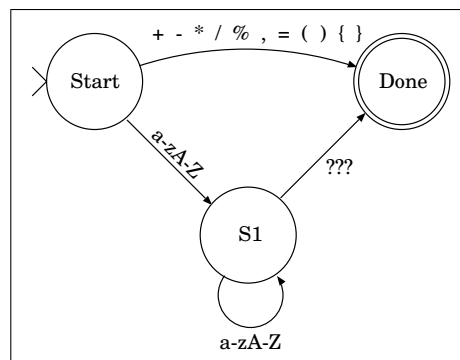


Figure 5.5: Recognizing PDef Identifiers

The finite-state machine seems fine, except it's not clear how to get to the final state. The reason is subtle and drifts over into the implementation. Imagine an input string such as

```
sum = sum+cost[count];
```

and focus on the two occurrences of `sum`. In the first occurrence `sum` is followed by a blank character. It would seem, then, that we could have a transition from state `S1` to `Done` on the blank character – note that the blank character would be consumed! On the other hand, the second `sum` is immediately followed



by a single character token '+'. When we see a single character token that could be consumed and send us to the state `Done`. But in the process we would lose the token value '+', which is critical. So in this case we want to transition to the `Done` state without consuming a character.

### Non-deterministic Finite-state Machines

This idea of transitioning without consuming a character is critical to implementing a tokenizer, so we allow a new kind of transition where nothing is consumed – being more correct we would say that the null string ( $\epsilon$ ) is consumed. We indicate this in a finite-state machine diagram with  $\epsilon$  as a transition label. This concession has theoretical consequences. First of all, when we use such a transition we no longer have an ordinary finite-state machine, rather we have a *non-deterministic finite-state machine*. A non-deterministic finite-state machine has three differences with ordinary (i.e., deterministic) finite-state machines.

1. There can be state/character pairs with no transition specified.
2. There can be transitions on  $\epsilon$ .
3. A state can have more than one transition for a given alphabet character.

Fortunately, we can get by with finite-state machines which only allow the  $\epsilon$  transition.

### Completing the PDef Finite-state Machine

So the  $\epsilon$  transition is necessary for a finite-state machine that can recognize identifiers. But the same situation exists for integer and floating point literals – we will need to use  $\epsilon$  transitions to recognize these token classes as well. We follow the same strategy for literals as we did for identifiers except that our token definition requires that no integer, other than zero, begin with a zero. A new finite-state machine is displayed on the left of Figure 5.6 which recognizes integer literals as well as identifiers. Notice the uses of the  $\epsilon$  transition. The diagram on the right of the diagram is augmented to recognize floating point literals, again assuming that the integer part can only start with zero if it is zero itself. The finite-state machine on the right of the diagram is the one we would use to drive a tokenizer implementation for PDef. Remember that we have explicitly ignored the token type `typeT` since its token values are keywords – the finite-state machine will identify them as identifiers, but in the implementation we will check all identifiers and adjust the type of those with the keyword values.

Something that should be evident at this point is that, with the use of  $\epsilon$  transitions, we can define a non-deterministic finite-state machine for a regular expression that has a single start state (that's required) and a single final state that has no transitions. This fact simplifies things when we turn to implementing a tokenizer, as we do in the next section.

## 5.3 Implementing a Tokenizer

In this section we focus our attention on strategies for implementing finite-state machines and, in particular, how these strategies can be applied to implementing lexical analyzers. The implementation strategy must, of course, focus on the two aspects of a finite-state machine, the static and the dynamic. On the static side we must implement the states and transition function, while on the dynamic side we must implement a mechanism for animating the cycle of state transitions.

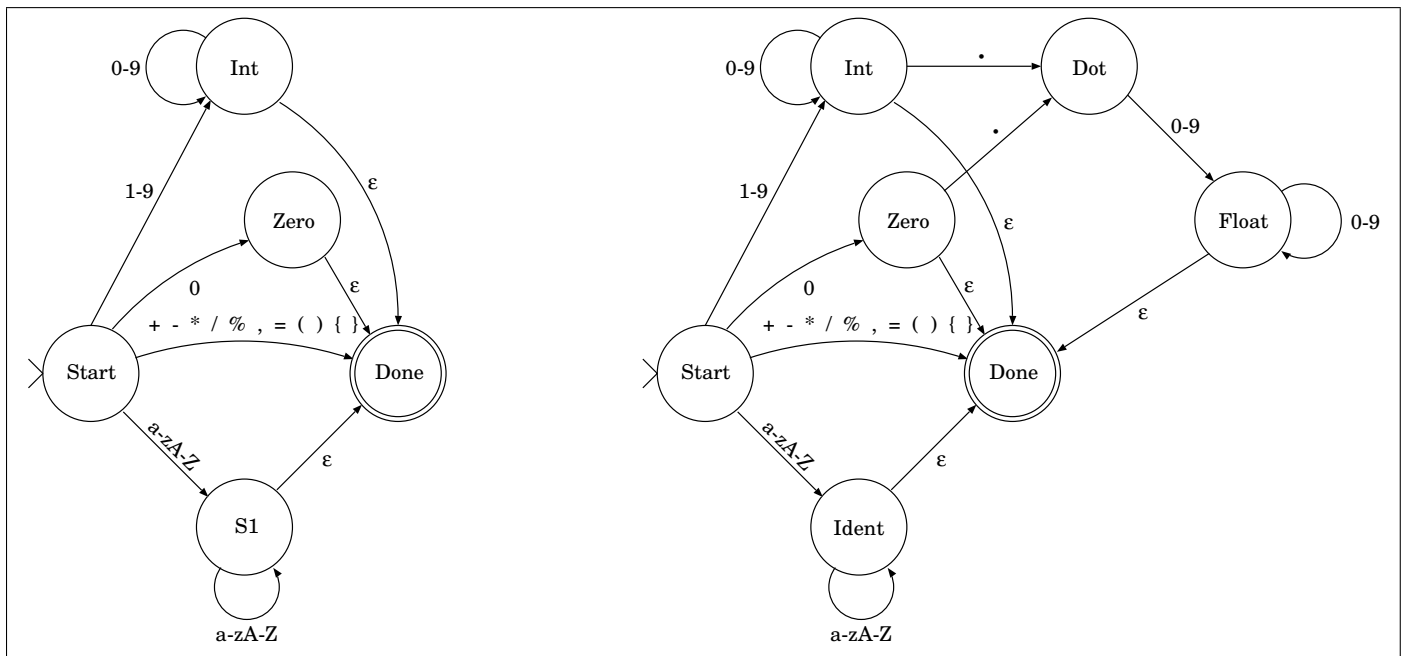


Figure 5.6: Recognizing PDef Identifiers

## States and Transitions

The static side really requires two components, the definition of a data type that describes the set of states and a description of the transition function. The first is easy since we only have to give a list of unique state names. For our algorithmic purposes we will assume the following state names (assuming there are  $n > 0$  states other than the start and final state).

`Start`, `S1`, `S2`, ..., `Sn`, `Done`

Of course, in this listing `Start` and `Done` are the finite-state machine's start and final states.

The implementation of the transition function takes a bit of thought. If we think about the transition function we would probably describe its operation to someone as follows.

If the current state is `Start` then determine the next state based on the current input symbol;  
 if on the other hand the current state is `S1` then determine the next state based on the current input symbol;  
 ...  
 if on the other hand the current state is `Sn` then determine the next state based on the current input symbol;

[Remember, that the final state has no transitions, so it needn't appear in the description.] So the transition function, ranging through the different states, is a selection and can be implemented using the standard `switch` structure as follows.

```
switch (state) {
```

```

case Start: // select next state based on current input symbol
case S1:    // select next state based on current input symbol
    ...
case Sn:    // select next state based on current input symbol
}

```

The action for each of these switch cases is, of course, dependent on the particulars of the transition function. It might appear that each case can itself be implemented as a switch structure, they are usually more appropriately implemented with nested `if...else` statements. For state `Int` in the right-most finite state machine in Figure 5.6, for example, the following code is an appropriate implementation.

```

case Int:
    if (isdigit(ch)) state = Int;
    else if (ch == '.') state = Dot;
    else {
        // implement the epsilon transition by putting back ch on the input stream
        input.putBack(ch);
        state = Done;
    }
    break;

```

Notice that each transition for the state `Int` has its own selection case, with the ‘ $\epsilon$ ’ transition being the default `else` as the end. This is a standard structure when ‘ $\epsilon$ ’ transitions are present.

### Cycling Through the States

Cycling through the states is the easier part of the implementation scheme. The algorithmic structure is clearly a repetition that begins in the state `Start` and continues until the state `Done` is reached. But there is also the issue of the “current symbol” that drives each transition. A reasonable strategy is for each cycle to first input a new symbol for the new state and then to apply the transition function to that state and symbol. This suggests the following algorithm.

```

state = Start;
while (state != Done) {
    ch = input.getSymbol();
    // determine the next state based on ch and state
}

```

The algorithm structure given in Figure 5.7 is a combination of the two parts we have defined. Notice in the switch statement that a final case has been added for the `Done` state – this is included so that every state is listed in the switch – a completeness consideration.

#### ☛ Activity 8 –

Work through the PDef Tokenizer Tutorial in Chapter 12. Before embarking on the Tokenizer Tutorial be sure to read the Tutorial Overview, Chapter 11, which provides important information and strategies employed in the PDef Tutorials.

---

```
state = Start;
while (state != Done) {
    ch = input.getSymbol();
    switch (state) {
    case Start: // select next state based on current input symbol
    case S1:    // select next state based on current input symbol
        ...
    case Sn:   // select next state based on current input symbol
    case Done: // should never hit this case!
    }
}
```

Figure 5.7: FSM Algorithm

## Chapter 6

# Syntax Analysis – Theory and Practice

The syntax recognition process, a clear reference to the Correctness Principle, is called *parsing* and involves reading characters from an input stream to determine if the characters appear in an order specified by a particular grammar. Actually, part of the parsing problem has been solved in the previous section, where we discussed the theory and practice of lexical analysis and its implementation in the form of a tokenizer. In this section we will assume that we always have an appropriate tokenizer in hand, which means we can restrict the attention of a parser to the sequence of tokens in the input string. The parser also reflects the Early Warning Principle in its responsibility to identify syntax errors when they appear in the input string.

In this chapter we will first discuss the two major strategies for syntax analysis – top-down and bottom-up parsing. We will then focus our attention more specifically on basic strategies for designing the most common type of top-down parser – a *recursive descent parser*.

### 6.1 An Overview of Parsing Techniques

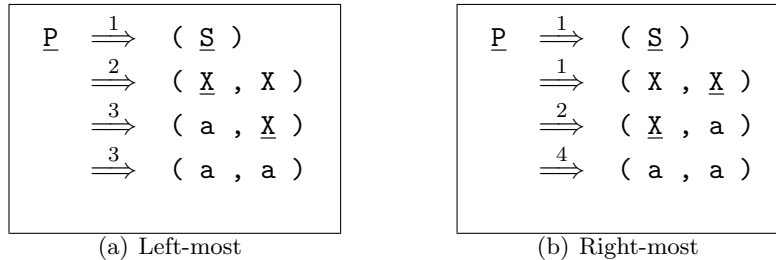
Recall from Section 2.2.3 that each context free grammar defines a language and that for each string in the language there is a derivation based on the rules of the grammar. From this point of view, when a parser is given a particular string, it should check to see if the string has a derivation based on the grammar rules. But how can we do this? We could have the parser simply generate all possible derivations, watching at the end of each derivation cycle to see if the target string was produced. OK – that’s going to require lots of computing time and if the string contains a syntax error the process won’t be able to give any feedback! A better approach is to study the derivation process to see if there is a way to synchronize the steps of the derivation with a scan through the string of tokens.

Remember that when we discussed string derivations we focused on two derivation strategies: left-most and right-most. While the distinction between these two strategies may have seemed uninteresting back in Section 2.2.3, we will now see that they hold the key to the two basic syntax recognition strategies, referred to as *top-down parsing* and *bottom-up parsing*. In order to investigate the properties of these two derivation strategies we will use the following simple context free grammar  $G$ .

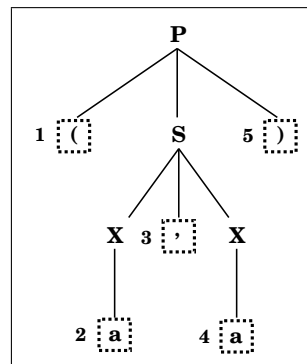
We will focus on the target string ‘( a , a )’, which is in the language of  $G$ . To show that the string is in the language of  $G$ , and for the convenience of our investigation, we display in Figure 6.2 two derivations of the target string. In each derivation step the non-terminal to be replaced is underlined and each arrow is annotated with the number of the grammar rule applied.

Doing a derivation for ‘( a , a )’ is easy, but how can we automate (i.e., come up with an algorithmic

1	$P \rightarrow '( S )'$
2	$S \rightarrow X ', X$
3	$X \rightarrow 'a'$

Figure 6.1: Grammar Rules for  $G$ Figure 6.2: Derivations for  $'( a , a )'$ 

strategy for) the discovery of the derivation of this string? A clue to how to proceed can be found in the parse tree for our two derivations, which is shown in Figure 6.3. Notice in this figure that the leaf nodes of the tree have been numbered in depth-first order – this will prove to be useful.

Figure 6.3: Parse Tree for  $'( a , a )'$ 

So we have two derivations and a parse tree for our string  $'( a , a )'$ . But if we are presented with this target string for parsing, we will have none of these artifacts in hand. In a certain way, it is the job of the parsing process to reproduce one of these – either one of the derivations or the parse tree. Now the connection to the ordering of the leaf nodes in the parse tree may become clear: if the parser is to somehow reproduce the parse tree, then the order in which the target string appears on input matches the sequence of leaf nodes as visited in depth first order. This means that whenever we want to parse a target string we want to produce a parse (or one of its derivations) in which the target string appears as leaf nodes in depth-first order.

### 6.1.1 Parsing Top-down

In parsing our target string we will read the tokens in order and, at the same time, trace out the corresponding parse tree. There are two reasonable approaches to traversing the parse tree: from the top down or from the bottom up. In traversing from the top down we focus on non-terminals and what they tell us about what to expect on the input stream. For example, the start symbol  $P$  for our grammar describes the structure of any string in the grammar's language. In  $G$  the one rule for  $P$  says every string must start and end with left and right parentheses, respectively, with the string in the middle described by the non-terminal  $S$ . So working from the top down we know the grammar requires '(' to appear first in the target string; notice that this is exactly what the top of the parse tree in Figure 6.3 tells us. The following parsing pattern for the target string '( a , a )' emerges: we have a pattern, composed of terminals and non-terminals, and a target string, composed of tokens.

At each point in the process we look at the first element of the pattern: if it is a non-terminal symbol we replace it with the right-hand side of one of its rules; if it is a terminal symbol we check the target string – if the first token in the target string matches the terminal symbol the two symbols are removed, otherwise there is an error.

Here is the process applied to '( a , a )'.

1. Pattern is ' $P$ ' and target is '( a , a )'.  
The pattern begins with the non-terminal  $P$ , so replace it with the right side of its grammar rule  $P \rightarrow ( S )$ .
2. Pattern is '(  $S$  )' and target is '( a , a )'.  
The pattern begins with the terminal '(' and target begins with the same terminal, so we eliminate the two '('s and continue.
3. Pattern is ' $S$  )' and target is 'a , a )'.  
The pattern begins with the non-terminal  $S$ , so replace it with the right side of its grammar rule  $S \rightarrow X , X$ .
4. Pattern is ' $X , X$  )' and target is 'a , a )'.  
The pattern begins with the non-terminal  $X$ , so replace it with the right side of its grammar rule  $X \rightarrow a$ .
5. Pattern is 'a ,  $X$  )' and target is 'a , a )'.  
Since the pattern and target begin with the same terminal 'a', we eliminate the two 'a's and continue.
6. Pattern is ',  $X$  )' and target is ', a )'.  
Since the pattern and target begin with the same terminal ',', we eliminate the two ','s and continue.
7. Pattern is ' $X$  )' and target is 'a )'.  
The pattern begins with the non-terminal  $X$ , so replace it with the right side of its grammar rule  $X \rightarrow a$ .
8. Pattern is 'a )' and target is 'a )'.  
Since the pattern and target begin with the same terminal 'a', we eliminate the two 'a's and continue.

9. Pattern is ‘)’ and target is ‘)’.

Since the pattern and target begin with the same terminal ‘)’’, we eliminate the two ‘)’s and continue.

10. The pattern is now empty and the target is empty, so the parse succeeds.

But what if we started with a string that is not in the language of  $G$ ? Let’s try the process on the string ( a a ).

1. Pattern is ‘P’ and target is ‘( a a )’.

The pattern begins with the non-terminal P, so replace it with the right side of its grammar rule  $P \rightarrow ( S )$ .

2. Pattern is ‘( S )’ and target is ‘( a a )’.

Since the pattern begins with the terminal ‘(’ and target begin with the same terminal, we eliminate the two ‘(’s and continue.

3. Pattern is ‘S )’ and target is ‘a a )’.

The pattern begins with the non-terminal S, so replace it with the right side of its grammar rule  $S \rightarrow X , X$ .

4. Pattern is ‘X , X )’ and target is ‘a a )’.

The pattern begins with the non-terminal X, so replace it with the right side of its grammar rule  $X \rightarrow a$ .

5. Pattern is ‘a , X )’ and target is ‘a a )’.

Since the pattern and target begin with the same terminal ‘a’, we eliminate the two ‘a’s and continue.

6. Pattern is ‘, X )’ and target is ‘a )’.

Since the pattern and target begin with different terminals, ‘,’ in the pattern and ‘a’ in the target string, we have an error and the parsing stops.

In this way the process identifies syntax errors and can identify where they occur.

It is important that the parsing process represented above corresponds to the steps in the left-most derivation in Figure 6.2(a). To more clearly illustrate this point, we display in Figure 6.4 three related representations of the top-down parsing of our target string. In subfigures 6.4(a) and 6.4(b) we see the parsing steps side by side with the steps in the left-most derivation. The derivation steps are lined up with the corresponding parse steps. In subfigure 6.4(c) we have the parse tree for our target string, with the labeled downward pointing arrows corresponding to the replacement actions in the top-down parsing sequence. This explains why this parsing technique is called top-down.

While we have been using the name “top-down” to refer to this parsing method, there are other names that are used as well. Any parsing algorithm that reads its target string from left to right and traces out the left-most derivation of the target string is referred to as an *LL* parser. The significance of the LL should be clear – **L**eft to right, **L**eft-most derivation.

Another name derives from a special situation which can arise during parsing. A grammar such as  $G$ , where there is a single grammar rule for each non-terminal, is said to be LL(0) – the reason for the 0 will become clear shortly. The name LL(0) begs the question, “What would LL(1) mean?” To see the answer consider the grammar  $G'$  described in Figure 6.5. This grammar is slightly more complex than  $G$  but still there are only four strings in its language. The big difference, from the point of view of the process, is that the non-terminal X has two rules. How do we deal with the input string (b, a)? Let’s follow the process.





And now we have a problem! The pattern starts with a non-terminal  $X$  which has two rules – how do we know which right-hand side to substitute? You have probably seen the solution already. We look at the string – sort of sneaking a peek – and see if the next token matches the first symbol in the pattern. Sneaking a peek we see  $b$  and so choose to replace  $X$  with the right-hand side of its second rule.

### ☛ Activity 9 –

Complete the process started and make a note in each step where a peek was necessary.

---

So, if we have a grammar in which some non-terminal  $X$  has multiple rules, when the non-terminal is encountered a decision must be made as to which rule to apply. To resolve the choice we look ahead at the next token in the target string and check if that token can appear first on the right side of the rules of  $X$ . If the first tokens for all possible rules of  $X$  are different, then we can resolve the choice by looking ahead just one token on the target string – i.e., a *one token lookahead*. In the previous example there were no non-terminals with multiple rules so no lookahead is required – the grammar is LL(0). What we have just demonstrated is that the grammar  $G'$  is an LL(1) grammar.

### ☛ Activity 10 –

- Following the discussion above, argue that the following fragment is LL(1).

```

Stmt      → While
             → If
             → Assignment
While    → whileT ...
If       → ifT ...
Assignment → identT '=' ...

```

Think in terms of what left-most derivations must look like if **Stmt** is the start symbol.

- What can you conclude if we extend the grammar fragment above with the following rules?

```

Stmt      → FunctionCall
FunctionCall → identT '(' ParamList ')'

```

---

Finally, since in a top-down parser there is the possibility of having to *predict* which of multiple rules to apply, by doing a token lookahead, a top-down parser is also referred to as a *predictive parser*.

## 6.1.2 Parsing Bottom-up

The top-down parsing process is grammar rule driven, that is, it uses the grammar rules to reveal what should be expected on the target string. The bottom-up parsing process is driven by the target string, with grammar rule applications being determined by identifying a right hand side and replacing it by its defining non-terminal. For example, based on the grammar  $G$ , if we see ' $a$ ' at the head of the target string, we would identify the right side of grammar rule 3 and consequently replace the ' $a$ ' by the rules defining non-terminal ' $X$ '.

Thus, bottom-up parsing can be understood as a process of transformation or rewriting applied to the target string. We can think of the target string as a tube of toothpaste from which we squeeze out one token at a time. But as tokens are squeezed out they are transformed in the following way. When a token is squeezed out it remains unchanged; however, before squeezing another token, the transformed string is examined to see if the most recent symbols form the right side of some grammar rule – if such a right hand side is found it is replaced by the rule’s defining non-terminal (on the left side of the rule). In fact, there are long accepted terms for the two operations just described. What we described as squeezing a token is traditionally referred to as the *shift* operation, while replacing a right hand side by its defining non-terminal is called the *reduce* operation.

Using our same target string, ‘( a , a )’, we can talk through how this process plays out. Initially we squeeze out the first token, that is, we *shift* ‘(’. Before doing another shift we look at the symbols that have been shifted to see if there is a right hand side that can be reduced. While ‘(’ is the first character on the right of rule 1, it doesn’t constitute an entire right side, so we leave the token unchanged. We proceed to shift the next token, which is ‘a’. Now when we look at the shifted string we see that the token ‘a’ is the right side of rule 3, so we *reduce* the token ‘a’ to ‘X’. We can describe our current status as follows.

$$( X_{\uparrow}, a )$$

where the up arrow marks the boundary between the emerging transformed string ‘( X’ and the remaining target string ‘, a )’.

We continue the process, looking for a grammar rule with ‘( X ’ (the bit to the left of the up arrow) as the right-hand side. Since there are no such rules, we shift the next token from the target string to the left, giving the following status.

$$( X ,_{\uparrow} a )$$

Another point of terminology: when we look for a grammar rule’s right hand side we look to the left of the arrow for a *handle* – a handle begins in the string to the left of the up arrow and contains all symbols up to the up arrow.

At this point we look at three handles: ‘(X,’ , ‘X,’ , and ‘,’ , none of which appears as a grammar rule right hand side. Since there is no handle to reduce we shift the next terminal from the target string, giving the following status.

$$( X , a_{\uparrow} )$$

The one handle that can be reduced is ‘a’, as we saw earlier, so we reduce ‘a’ to ‘X’, giving the following status.

$$( X , X_{\uparrow} )$$

Now this is more interesting. The handle ‘X , X’ is the right side of grammar rule 2, so we reduce it to ‘S’.

$$( S_{\uparrow} )$$

Having no reducible handle, we shift ‘)’, yielding the following.

$$( S )_{\uparrow} \epsilon$$

The handle ‘( S )’ reduces to the start symbol ‘P’, which itself cannot be reduced. Since the target string is empty, the parse is complete and the parse is successful. We summarize this sequence of actions in the following table.

Step	Rewritten string	↑	Action	Comment
1	$\epsilon$	↑	( a , a )	shift No handle so <i>shift</i> ‘(’
2	(	↑	a , a )	shift ‘(’ can’t be reduced, so <i>shift</i> ‘a’
3	( a	↑	, a )	reduce <i>reduce</i> ‘a’ to ‘X’ (grammar rule 3)
4	( X	↑	, a )	shift No reducible handle, so <i>shift</i> ‘,’
5	( X ,	↑	a )	shift No reducible handle, so <i>shift</i> ‘a’
6	( X , a	↑	)	reduce <i>reduce</i> ‘a’ to ‘X’ (grammar rule 3)
7	( X , X	↑	)	reduce <i>reduce</i> ‘X , X’ to ‘S’ (grammar rule 2)
8	( S	↑	)	shift No reducible handle, so <i>shift</i> ‘)’
9	( S )	↑	$\epsilon$	reduce <i>reduce</i> ‘( S )’ to ‘P’ (grammar rule 1)
10	P	↑	$\epsilon$	accept P is start symbol and input is empty

It is important that the parsing process represented above corresponds to the *reversed* steps in the right-most derivation in Figure 6.2(b). To more clearly illustrate this point, we display in Figure 6.6 three related representations of the top-down parsing of our target string. In the subfigures 6.6(a) and 6.6(b) we see the parsing steps side by side with the steps in the left-most derivation. Notice that the derivation is given in reverse order. In subfigure 6.6(c) we have the parse tree for our target string, with the reduce parse steps labeled with upward pointing arrows, which are numbered in the order they are encountered in the process of parsing. This illustrates two important ideas: (i) that the parsing strategy does a bottom-up traversal of the parse tree and (ii) why the parsing technique is called bottom-up.

Just as with top-down parsing, there are various names by which bottom-up parsing is known. Bottom-up parsing reads its target string from **Left** to right (as does top-down parsing); but a bottom-up parser traces a **Right-most** derivation (in reverse). So a bottom-up parser is referred to as an *LR* parser – **Left** to right, **Right-most** derivation. Again as with top-down parsers, there are grammars for which a bottom-up parser will encounter conflicts – basically, situations where both shift and reduce actions seem appropriate. These shift/reduce conflicts are resolved again by referencing the target string. Depending on the extent of token lookahead required, a grammar can be LR(0), LR(1), LR(2), etc.

Following the names of the two actions used in bottom-up parsing, bottom-up parsers are often referred to a *shift/reduce parsers*.

### 6.1.3 Comparing Top-down and Bottom-up Parsing

The two parsing methods we have discussed couldn’t be more different and, not surprisingly, they each have pluses and minuses when it comes to choosing a parsing technique for a compiler. In fact, most production compilers use the bottom-up approach. There are important reasons. First, a compiler with a bottom-up parser can be automatically generated. The target language’s semantic properties are defined in the context of a context-free grammar for the language. The tools for automatic generation have been referred to as *compiler compilers* and the best known tools for automatic generation are Lex and Yacc (which, amusingly, stands for “yet another compiler compiler”). Besides the automatic generation, the bottom-up approach allows grammars to have left-recursive rules – for example, to describe the left associativity of arithmetic operators.

The top-down method is easy to understand since the algorithms reflect the form of the grammar rules being parsed. A top-down parser is a convenient basis on which to construct both the static and dynamic

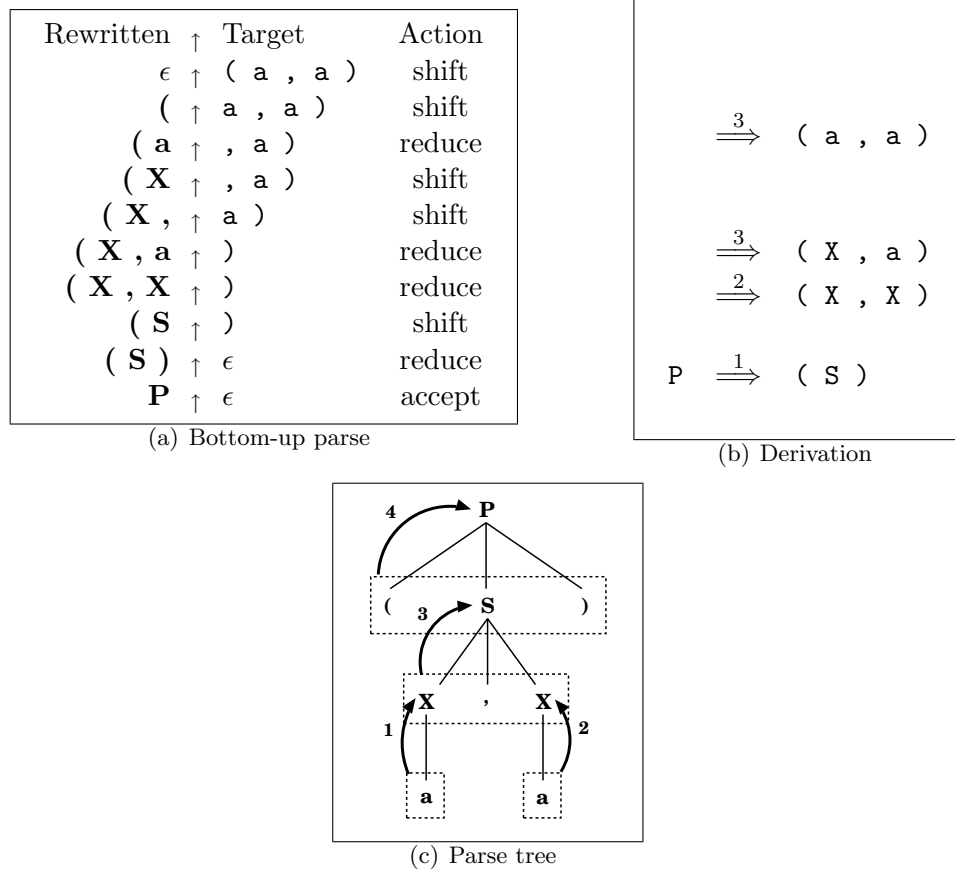


Figure 6.6: Understanding the bottom-up parsing of ‘( a , a )’

semantic components of a compiler. So for students unfamiliar with the structure of compilers, a top-down parser is a convenient basis for understanding the inner workings. Also, implementation of a top-down-based compiler is easier to construct from scratch than is a bottom-up-based compiler. One major problem for top-down parsers is that they cannot deal directly with left-recursive grammar rules. You can see this if you think about the process discussed above – we have at the head of the pattern a non-terminal defined by a left recursive rule. Of course, left-recursive means that the first element on the right side of the rule is the non-terminal defined by the rule! So the process devolves into simply replacing the non-terminal at the head of the pattern with the right-hand side of the rule and once again the same non-terminal is at the head of the pattern. Fortunately, it is possible to replace a left-recursive rule by rules with no left recursion. Unfortunately, the result is a less intuitive grammar, which is a great advantage for top-down parsing.

In this book we have chosen to focus on translators constructed on top-down parsers because they are easier to implement and emphasize in each component the connection to theory – i.e., the structure of a context free grammar. Once a student understands how a translator fits together and works, the transition to more advanced approaches to compiler writing is smoother and more fruitful.

## 6.2 Top-down Parsing

As illustrated in the previous section, top-down parsing focuses on non-terminals which, when encountered, are replaced by the right side of the defining grammar rule – terminals are matched against the head of the target string as they appear. In this section we will delve deeper into this parsing strategy, first looking at an important special case which occurs during parsing, and then turning to a particular top-down parsing implementation strategy called *recursive descent parsing*.

### 6.2.1 A Closer Look

From the top-down parsing example in Section 6.1 we see that top-down parsing can be defined in terms of the three actions listed in Figure 6.7.

1. If the current pattern starts with a non-terminal, then replace the non-terminal with the right hand side of its grammar rule.
2. If the current pattern starts with a terminal and it matches the next token on the target string, then consume the token from the target and remove the terminal from the pattern. If the match fails there is an error.
3. If both the pattern and target strings are empty, then the parse succeeds.

Figure 6.7: Top-down Parsing Actions

Of the three top-down actions, it is action 1 that exposes the special case we are interested in. The statement of action 1 for top-down parsing goes out of its way to indicate that each non-terminal has a single grammar rule. That was fine for the grammar  $G$ , but is not true for our new grammar  $G'$  – its rules are listed in Figure 6.8. This grammar exposes the special case immediately: since  $G'$ 's start symbol  $S$  has two grammar rules, how should we choose the right hand side to complete the rewriting? The answer is that eventually we have to use the target string to resolve the conflict. In fact, in looking at the target string, since there is a comma token in it, it must be that we use the first rule for  $S$  in the first rewriting. But that's cheating! We want to restrict our interaction with the input to the head of the string (if possible).

1	$\underline{S} \rightarrow X \text{ ', ' } S$
2	$\rightarrow X$
3	$X \rightarrow \text{'a'}$
4	$X \rightarrow \text{'b'}$

Figure 6.8: Grammar Rules for  $G'$

So how to we proceed? It is important to notice that each rule for  $S$  begins with  $X$ , and, in fact, from our discussions in Section 2.2.5, we could rewrite the rule in the form

$$S \rightarrow X [ \text{' , ' } S ] .$$

So apparently we can start by rewriting  $X$  and then checking to see if there is a comma at the head of the target string. In the example below, when we encounter an  $S$ , we will replace it by the string  $X [ \text{' , ' } S ]$ , which indicates that a choice must be made in the future based on the existence of a comma at the head of

the input string. Actually, there is a second situation where a judicious choice of grammar rule is required. When  $X$  is encountered we must choose whether to replace it by ‘a’ or ‘b’, since there are two such grammar rules for  $X$ . When we encounter an  $X$  for rewriting, then, we must look at the head of the input string and let that tell us which rule to use for the rewriting. Based on this analysis, then, we can produce the following top-down parsing of the target string ‘b , a , b’.

Pattern string	Target string	Action	Comment
S	b , a , b	1	Replace S by X with option of adding ‘ , S’
X [, S]	b , a , b	1	Use rule $X \rightarrow b$ since b is at head of input
b [, S]	b , a , b	2	Cancel the two b’s
			At this point we have to decide what to do with the [, S].
			Since the next token on input is comma, we keep the , S.
, S	<del>b</del> , a , b	2	Cancel the two ‘, ’s
S	<del>b</del> / a , b	1	Replace S by X with option of adding ‘ , S’
X [, S]	<del>b</del> / a , b	1	Use rule $X \rightarrow a$ since a is at head of input
a [, S]	<del>b</del> / a , b	2	Cancel the two a’s
			At this point we have to decide what to do with the [, S].
			Since the next token on input is comma, we keep the , S.
, S	<del>b</del> / <del>a</del> , b	2	Cancel the two ‘, ’s
S	<del>b</del> / <del>a</del> / b	1	Replace S by X with option of adding ‘ , S’
X [, S]	<del>b</del> / <del>a</del> / b	1	Use rule $X \rightarrow b$ since b is at head of input
b [, S]	<del>b</del> / <del>a</del> / b	2	Cancel the two b’s
			Once the b’s are cancelled the input is empty. Since there is no comma on the input we do not use the , S.
$\epsilon$	<del>b</del> / <del>a</del> / <del>b</del>	3	Parsing is successful

### 6.2.2 Implementing a Recursive Descent Parser

Having studied the basics of the top-down parsing strategically, we now turn to the question of implementation. The top-down process illustrated in the previous sections is clearly algorithmic, but a good implementation doesn’t always arise immediately from the abstract description. In this section we will focus on the top-down parsing technique, and leave a fuller study of bottom-up parsing until a later chapter.

There are two basic strategies for implementing a top-down parser: *table driven* and *recursive descent*. The table-driven approach is appropriate when a parser will be generated automatically. The recursive descent approach, while it can be chosen for automatic generation, is always chosen when the parser will be built by hand. Since our interest in this book is to expose the inner workings of a compiler, the “by hand” approach will be our preferred parsing method. The approach will be applied in the Front-end Tutorial.

In the previous section we saw, in Figure 6.4, an annotated parse tree illustrating the order in which non-terminals were replaced by their defining rule’s right hand side. It is this diagram that holds the key to the structure of a recursive descent parser. A useful way to envision recursive descent parsing is as a sequence of questions and answers. Each symbol (either terminal or non-terminal) has an associated question: Does the next sequence of tokens on the input stream match the named symbol? If the question is for a terminal symbol the answer is yes or no depending whether the next token matches the terminal symbol. If the question is for a non-terminal symbol then the answer depends on whether the next sequence of tokens matches the structure defined by the right-hand side of a rule for the non-terminal in question. For example, if the non-terminal is `Block` from the PDef grammar, then since the associated rule is

```
Block ::= lcbT StmtList rcbT
```

the answer to the question is yes or no depending on whether the next tokens match the structure of the right-hand side. So the questions associated with `StmtList` are as follows (each subsequent question assumes a yes answer to its predecessor).

1. Does the next token match `lcbT`?
2. Does the next sequence of tokens match `StmtList`?
3. Does the next token match `rcbT`?

It is useful to notice that the nature of these two question forms (for terminals and non-terminals) correspond to the first two top-down parsing actions listed in Figure 6.7.

From a programming point of view it would seem reasonable to associate each question with a method that can answer the question. For this reason, in a recursive descent parser there is one parse method for each non-terminal in the grammar, with each parse method being responsible for recognizing strings of tokens that match the description given by the rules for that particular non-terminal. All terminal symbol questions are associated with a method called `consume`, which takes the expected token type as an argument. So if our current question is the non-terminal `NT1` with the following rule

$$\text{NT1} \rightarrow \text{tok1T NT2 lrt tok2T}$$

our new question will be “Is the next token `tok1T`?” The `consume` method answers the question about `tok1T`.

### 6.2.3 Strategy for Designing Parse Methods

The structure of a parse method for a non-terminal in a recursive descent parser is determined by the structure of the right side of the non-terminal’s grammar rule – remember that each non-terminal has a parse method associated with it. It is this strategy which insures in part the Correctness Principle for a translator. In this section we will take a general look at some common rule structures or forms that appear frequently in grammars for programming languages. We focus on six grammar rule forms which are summarized in the table shown in Figure 6.9.

Form 1:	$\text{NT} \rightarrow \text{tok1T tok2T tok3T}$
Form 2:	$\text{NT} \rightarrow \text{NT1 NT2 NT3}$
Form 3:	$\text{NT} \rightarrow \text{tok1T NT1 NT2 tok2T NT3}$
Form 4:	$\text{NT} \rightarrow \text{"rhs1"} \mid \text{"rhs2"} \mid \text{"rhs3"}$
Form 5:	$\text{NT} \rightarrow \text{NT1 [ tokT NT2 ] NT3}$
Form 6:	$\text{NT} \rightarrow \text{NT1 \{ tokT NT1 \}}$

Figure 6.9: General Grammar Rule Forms

Before considering how to parse these rules there is a kind of cart-and-horse problem to be discussed. The problem involves the strategy that will dictate exactly when new tokens are read from the input stream. Having not discussed parsing yet the problem may not be clear, but at the same time the discussion will be more complex if we put off confronting the problem. So we state the *Parser Invariant* now.

The Parser Invariant is an invariant for the parse method – that is, it must be true before a method is called and it must be true when a method returns after its execution. Each parse method must be carefully



We will assume an object called `currentToken` exists and that at any point during the parsing process the value of `currentToken` is the next token to be processed.

Figure 6.10: Parser Invariant

designed and implemented so that the invariant is guaranteed. In thinking about the implementation of the invariant, we must ask the following question:

*When is it that the value of `currentToken` does not satisfy the invariant?*

And the answer is:

*When the value of `currentToken` has been verified to be correct in the existing context.*

For example, when the method `consume` is called with a particular token type as argument, if the value of `currentToken` matches the argument type then the current token is correct. Once the value of `currentToken` is verified then it is no longer the “next token to be processed.” So before returning from `consume` the next token must be read from the input stream into `currentToken` – this will guarantee the validity of the Parser Invariant. This idea of when to update the value of `currentToken` will be referred to as the *Parser Rule*; it is stated in Figure 6.11

Immediately after verifying that a particular token type matches the value of `currentToken`, read the next token.

Figure 6.11: Parser Rule

The following defines the interface for the method `consume`, which, as we have just explained, is central to the proper functioning of a recursive decent parser.

```
void consume(Token t)
// Pre:  tokenStream == <t2,t3,...,tn> AND currentToken == t1
// Post: if (t == currentToken)
//       then currentToken = t2 AND tokenStream == <t3,...,tn>
//       else exit program
```

### Form 1: A sequence of terminals

$$NT \rightarrow tok1T \ tok2T \ tok3T$$

This rule poses three questions about terminal symbols. Since this is the first rule we will be a bit more verbose about the parse method design. The questions for this rule can be abbreviated as follows:

1. Is the token type of `currentToken` the same as `tok1T`?
2. Is the token type of `currentToken` the same as `tok2T`?
3. Is the token type of `currentToken` the same as `tok3T`?

Notice that there is an implicit assumption in these questions that the value of `currentToken` is updated when each question is answered. Since each question can be implemented by an appropriate call to `consume` the corresponding parse method is easy to write out – as follows.

```
void parseNT() {
    consume(tok1T);
    consume(tok2T);
    consume(tok3T);
}
```

Notice that since each call to `consume` maintains the Parser Invariant (assuming `consume` is properly implemented), the method `NT` will also maintain the invariant.

### Form 2: A sequence of non-terminals

$$NT \rightarrow NT1 \ NT2 \ NT3$$

This grammar rule begs a slightly different sequence of three questions.

1. Does the next sequence of tokens match `NT1`?
2. Does the next sequence of tokens match `NT2`?
3. Does the next sequence of tokens match `NT3`?

Assuming that we have parse methods `parseNT1`, `parseNT2`, and `parseNT3` that maintain the Parser Invariant, then the following parse method will implement the grammar rule above. Since each of the called parse methods maintain the invariant, this method will as well.

```
void parseNT() {
    parseNT1();
    parseNT2();
    parseNT3();
}
```

Before going on it is important to point out a special situation in this case. If the non-terminal `NT1` is the same as `NT` then we have a left recursive rule. We need only consider the questions above to see that this will cause trouble: since `NT1` is `NT`, each time we ask “Does the next sequence of tokens match `NT1`?”, we go back to this rule for `NT` and start again – we never make any progress. When implementing a recursive descent parser we will first have to remove the left recursion from left-recursive rules, which will give us two replacement rules which can be reclassified according to their forms (see Section 2.3.4, page 39).

### Form 3: A mix of terminals and non-terminals

$$NT \rightarrow tok1T \ NT1 \ NT2 \ tok2T \ NT3$$

The questions posed by this rule simply fold together the strategies of the previous two forms. The following parse method is appropriate, once again assuming that `parseNT1`, `parseNT2`, and `parseNT3` exist and maintain the Parser Invariant.

```

void parseNT() {
    consume(tok1T);
    parseNT1();
    parseNT2();
    consume(tok2T);
    parseNT3();
}

```

**Form 4: When there is a choice**

$$NT \rightarrow \text{"rhs1"} \mid \text{"rhs2"} \mid \text{"rhs3"}$$

Initially we will assume that the three options above are as simple as possible – i.e., that there are non-terminals NT1, NT2, and NT3 and the NT rule has the following form.

$$NT \rightarrow NT1 \mid NT2 \mid NT3$$

You should recognize this as the situation discussed at the end of Section 6.1.1 – we need to do a token lookahead to try to sort out which of the three non-terminals to parse – this highlights the notion of predictive recursive descent parsing (see page 78). So a grammar with such a rule is at least LL(1). In order to implement `parseNT` in this case we need to compute what is called the *first set* for each non-terminal on the right.

**The *first Set* –**

The important question to ask in this situation is “If we call `parseNT1` what will be the first token consumed, or more correctly what tokens (plural) can be the first consumed?” We need to know this for each of the non-terminals. To identify the tokens we define the *first set function*, which we denote *first*. It takes a sequence of terminals and non-terminals and returns the set of terminal symbols that will be consumed first when parsing the sequence. More formally we can define *first* as follows. The implementation of this function is defined later.

If  $w$  is a string of terminals and non-terminals then

$$\begin{aligned}
 first(w) = \{ x \mid & x \text{ is a terminal symbol and } w \Rightarrow^* x\alpha \\
 & \text{or} \\
 & x \text{ is } \epsilon \text{ and } w \Rightarrow^* \epsilon \}
 \end{aligned}$$

Some examples based on the PDef grammar (see Figure 4.2, page 60) displayed in the table in Figure 6.12. There are times, illustrated by the second entry in the table, when finding the first terminal symbol requires searching more than one grammar rule.

So returning to the current case, let’s assume that  $first(NT1) = \{tok1T\}$ ,  $first(NT2) = \{tok2T\}$ , and  $first(NT3) = \{tok3T\}$ , and that `tok1T`, `tok2T`, `tok3T` are all distinct. We can use a `switch`-statement to sort out which parse method to call. An important detail in this situation is that we must leave it up to `parseNT1`, `parseNT2`, and `parseNT3` to consume their own leading terminal symbols. The following implementation follows naturally.

rhs	<i>first</i> (rhs)
identT assignT identT	identT
Stmt	identT, typeT, lcbT
identT	identT
Block	lcbT

Figure 6.12: *first* Sets for PDef Sequences

```

void parseNT() {
    switch(currentToken.getType()) {
    case tok1T:
        parseNT1();
        break;
    case tok2T:
        parseNT2();
        break;
    case tok3T:
        parseNT3();
        break;
    default:
        error();
    }
}

```

It is important to recognize that the `default` case at the end of the `switch`-statements is essential. The call to `parseNT` was made because it was required by some other parse method. If the value of `currentToken` cannot start `NT1`, `NT2`, or `NT3` then it indicates an error – i.e. the next token doesn't support our call of `parseNT`. The default allows us to trap the error and take an appropriate action.

There are a couple of special cases we should mention. First, suppose we discover that  $first(NT1) = \{ tok1a, tok1b \}$  (and both these token types are disjoint from `tok2` and `tok3`), then we would have to call `parseNT1` if either of these token types turned up. The code for the first case in the switch would be modified as follows.

```

switch(currentToken.getType()) {
case tok1aT:
case tok1bT:
    parseNT1();
    break;
...
}

```

The second case involves the situation where, for example,  $first(NT1) \cap first(NT2)$  is not empty – i.e., that there is a token type that can start both `NT1` and `NT2`. This situation was mentioned earlier (see page 78) and illustrated with the following grammar.

```

Stmt      → While
          → If
          → Assignment
          → FunctionCall
While     → whileT ...
If        → ifT ...
Assignment → identT '=' ...
FunctionCall → identT '(' ...

```

In this case we would have to look even further ahead than the current token. We use a technique called a *peek*, which allows us to look at the input stream without actually consuming the token. The result is an implementation as follows – notice that the peek operation is represented by a function call `peek`.

```

switch(currentToken.getType()) {
case whileT: parseWhile(); break;
case ifT:    parseIf(); break;
case identT: switch( peek() ) {
               case '=': parseAssignment();
               case '(': parseFunctionCall();
             }
default error();
}
}

```

We should mention that this structure is LL(2) and commonly found in many modern imperative programming languages, for example Java and the C languages.

A third special case deserves mention. Suppose that one of the right hand sides is  $\epsilon$ ; how does this complicate things? Consider the following grammar rule, in which `tok1` and `tok2` are different.

```

NT → tok1 NT1
   → tok2 NT2
   →  $\epsilon$ 

```

One thing we can see is that if we call `parseNT` and don't see one of `tok1` or `tok2` we shouldn't consider this an error. So our switch statement in this case shouldn't have a default case.

But a more subtle condition exists here. Suppose that in our grammar there is another grammar rule of the following form.

```

N → NT tok2 NX

```

In this case `tok2` can follow `NT`. So now what should it mean, in `parseNT`, when our switch encounters `tok2`? Should we see it as an indication to call `parseNT2`? Or an indication to take the  $\epsilon$  path and consume `tok2` in `parseN`? This kind of conflict needs to be avoided, so in the presence of  $\epsilon$  as a choice we must make sure the tokens that start one of the choices cannot also follow the non-terminal `NT`. This requires the definition of another function, which we call *follow*, and which acts on non-terminals.

### The *follow* Set –

Unlike the *first* set, the *follow* set function is defined only on non-terminals. It is defined to be the set of terminal symbols, along possibly with  $\epsilon$ , that can follow immediately after a non-terminal in a derived string. In other words,

$$\text{follow}(\bar{X}) = \{x \mid x \text{ is a terminal symbol and } S \Rightarrow^* \alpha X x \beta$$

*or*

$$x \text{ is } \epsilon \text{ and } S \Rightarrow^* \alpha X \}$$

non-terminal	<i>follow</i> (rhs)
Program	$\epsilon$ (empty string)
Block	{ commaT, rcbT, $\epsilon$ }
StmtList	{ rcbT }
Stmt	{ commaT, rcbT }

Figure 6.13: *follow* Sets for Certain PDef Non-terminals

Thus, in the case where we have an  $\epsilon$  choice, and using the rule for NT given above, we will use the following implementation for `parseNT` under the assumption that

`tok1, tok2`  $\notin$  *follow*(NT).

With this assumption the parse method `parseNT` is implemented as follows.

```
void parseNT() {
    // Pre: currentToken.getType() not in follow(NT)
    switch(currentToken.getType()) {
    case tok1T:
        parseNT1();
        break;
    case tok2T:
        parseNT2();
        break;
    }
}
```

### Form 5: When something is optional

$$NT \rightarrow NT1 [ \text{tokT } NT2 ] NT3$$

This form is actually an abbreviation for a special case of Form 4.

$$\begin{aligned} NT &\rightarrow NT1 \text{ tokT } NT2 \text{ NT3} \\ &\rightarrow NT1 \text{ NT3} \end{aligned}$$

This is an interesting example from Form 4 as the two right sides actually start with the same non-terminal, a possibility we didn't discuss above. But it turns out that the optional form is more convenient for seeing the appropriate solution.

Let's talk our way through this one before trying to write out an algorithm. The questions posed by this rule can be stated as follows.

1. Does the next sequence of tokens match NT1?
2. If the next token is tokT, does the sequence of tokens following tokT match NT2?
3. Does the next sequence of tokens match NT3?

Notice the second step involves a lookahead to determine the course of action. This sequence of questions, while more involved than for the earlier forms, leads naturally to the following code.

```
void parseNT() {
    parseNT1();
    if (currentToken.getType() == tokT) {
        consume(tokT); // now consume tokT
        parseNT2();
    }
    // (A)
    parseNT3();
}
```

But there is a subtle assumption at work in this code. Do you see it? When we do the lookahead on `currentToken`, how can we be sure that the presence of `tokT` correctly distinguishes whether the optional part is present or not? The problem occurs if NT3 can also begin with `tokT` – i.e. that  $first(NT3)$  contains `tokT`! In this case we can't know what to do based on this comparison. So the hidden assumption in the parse method above is that `tokT` is not in  $first(NT3)$ .

But that isn't the only subtlety. What if  $\epsilon \in first(NT3)$  or if there is no trailing element (such as NT3)? In this case our concern is that `tokT` might be in  $follow(NT)$ ! These subtleties must be taken into account and should be included in the precondition of `parseNT`, assuming, of course, that they are true!

Finally, notice in `parseNT` implementation above that there is no input of the next token at the point of the comment (A). Why not? Because at that point we did not find an expected token – so the value of `currentToken` still contains the value of the next token of interest.

### Form 6: When there is a repetition

$$NT \rightarrow NT1 \{ \text{tokT } NT1 \}$$

This form is more specialized than Form 5. Where Form 5 can be used to define structures such the `if` with an optional `else`, in addition to recursive list structures, the repetition structure is used almost exclusively for defining unstructured lists. It is critical in this case to remember that the repetition notation is again an abbreviation for the following rule forms.

$$\begin{array}{ll} NT \rightarrow NT \text{ tokT } NT1 & NT \rightarrow NT1 \text{ tokT } NT \\ \rightarrow NT1 & \rightarrow NT1 \end{array}$$

Rules of these two forms are used to describe expression forms, where the form used depends on the associativity of the separator – left-associative means the left recursive form, right-associative means the right recursive form. In fact, any kind of list can be described using this rule form.

Our Form 6 grammar rule looks very similar in structure to the previous optional form. But where the optional part can appear zero or one times in Form 5, it can appear zero or more times in Form 6 – a subtle but crucial difference. For a parse method, then, we substitute the recursive parsing algorithm of Form 5 with a repetition algorithm – the idea being that we will continue to parse an NT1 as long as we continue to see the separator tokT.

```
void parseNT() {
// Grammar Rule: NT --> NT1 { tokT NT1 }
  parseNT1();
  while (currentToken.getType() == tokT) {
    currentToken = tokenStream.getNextToken();
    parseNT1();
  }
}
```

Because their structures are similar it should be clear that with Form 6 we have the same restriction as discussed in Form 5 for the separator token tokT; namely, that tokT  $\notin$  follow(NT).

There is a final special situation to consider. What if there is no separator – for example when a list is whitespace separated? Then, again following the analysis for Form 5, we must rely on the tokens in *first*(NT1) in order to determine how long to repeat parseNT1. Again, it is critical that the sets *first*(NT1) and *follow*(NT) be disjoint. Notice that this is the form we use for describing the non-terminal StmtList in the PDef grammar.

### 6.3 Parsing Structured Lists

Forms 1-6 are a good guide to converting grammar rules to parse methods. But a common parsing task deserves a bit more attention. In Section 2.2.6 (pages 33-35) we discussed three ways of describing lists, depending on whether they are inherently unstructured, left-associative, or right-associative. Form 6 above describes how to handle unstructured lists so in this section we look at the two left- and right-recursive lists, their grammar rules, and appropriate parse method structures.

#### Right-recursive List

Another possible list form is right-recursive. Our example here should not be confused with PDef, but rather it should be seen as a general right-recursive structure. Here is the grammatical structure for such a list.

$$\text{List} \rightarrow \text{Element} [ \text{tokT List} ]$$

This rule clearly has Form 5 structure. As we saw in the discussions of Form 5 above, we must assume here that tokT is not in *follow*(List). With that assumption the following parse method would be appropriate.



```

void parseList() {
    parseElement();
    if (currentToken.getType() == tokT) {
        consume(tokT);
        parseList();
    }
}

```

### Left-recursive List

The final list possibility is to describe it in left-recursive form (the strategy followed in the arithmetic expression grammar rules). Following the general setting of the right-recursive case above, consider the following left-recursive grammar description for `List`.

```

List → List tokT Element
     → Element

```

This rule form is problematic for a recursive descent parser for two reasons. First, notice that the  $first(List)$  and  $first(Element)$  are the same – this means that the following optional form

```

List → [ List tokT ] Element

```

is not a possibility. That only leaves Form 4 where we have two options. But again, the two options start the same. The only way to know which option to take is to look ahead to see if there's a comma after the first statement – if there is we use the first option, otherwise the second. While this amount of look ahead may be theoretically possible, it is impractical – we would have to parse a statement and then, if there's a comma, put back onto the input stream all the tokens parsed and then call `parseList`.

The second problem with this left-recursive form is that calling `parseList` will lead to an infinite loop of recursive calls. To see why, consider the following parse method, which follows naturally from Form 5 (notice we're ignoring here the first problem we just discussed).

```

void parseList() {
    if ( canStartList( currentToken.getType() ) ) {
        parseList();
        consume(tokT);
    }
    parseElement();
}

```

Notice that if the value of `currentToken` is in  $first(List) = \{ identT, typeT, lcbT \}$ , then we will immediately (without consuming a token) call `parseList` again. On this second call the value of `currentToken` will be unchanged so we will call `parseList` again, and again, and again..... An infinite recursion results. Anytime we have a rule with left-recursion its parse method will have this problem.

This is why left-recursion removal (see Section 2.3.4 page 39) is important. If we apply left-recursion removal to our rule we will arrive at the following form, in which the left-recursion is converted to right-recursion.

```
List      → Element RestOfList
RestOfList → tokT Element RestOfList | ε
```

How should we write parse methods for these rules. The first is easy since it is an obvious application of Form 2.

```
void parseList() {
    parseElement();
    parseRestOfList();
}
```

The second rule seems more difficult – but what the  $\epsilon$  really indicates is that the sequence ‘tokT Element RestOfList’ is optional. So assuming that tokT is not in *follow*(RestOfList), then the following parse method is be appropriate.

```
void parseRestOfList() {
    if (currentToken.getType() == tokT) {
        consume(tokT);
        parseElement();
        parseRestOfList();
    }
}
```

This is a curious parse method since if there is no tokT present, then nothing is done. But then, that’s exactly what the grammar rule indicates – parse the empty string if tokT is not present.

As a final point it is critical to remember that this solution is only valid if we verify that tokT is not in *follow*(RestOfList). It would be a surprising syntactic structure that specifies a list as terminated by the token that separates its elements, but such errors can occur when designing a grammar. In the PDef grammar we see that the list separators for **Exp** and **Term** are not in their follow sets, so this strategy will work for their left-recursive rules.

## 6.4 Implementing a Parser

The implementation of a parser for PDef is the subject of the PDef Parser Tutorial in Chapter 13. But to give a preliminary flavor of that implementation we will look at a couple of the PDef grammar rules and determine the appropriate code for corresponding parse methods. We will focus on the following two grammar rules.

```
Block      → lcbT StmtList rcbT
StmtList   → Stmt { commaT Stmt }
```

We will look at these two rules in turn.

**Block** → lcbT StmtList rcbT

The right side of this rule has a mix of terminal and non-terminal symbols, so matches Form 3. In this case we call the method `consume` on the terminal symbols and the parse method corresponding to the non-terminal – making sure we call them in the sequence specified by the rule. The parse method has the following form.

```

void parseBlock() {
    consume(Token.TokenType.LCB_T);
    parseStmtList();
    consume(Token.TokenType.RCB_T);
}

```

`StmtList`  $\rightarrow$  `Stmt { commaT Stmt }`

After the discussions for Form 6 and in Section 6.3, the implementation for this grammar rule is straightforward. The form of the grammar rule for `StmtList` implies that lists of statements are unstructured lists and so we can use a repetition-based (rather than a recursion-based) solution. According to the pattern established in Section 6.3 we will call `parseStmt` before the repetition and at the end of the repetition body. The repetition continues as long as the current token on returning from `parseStmt` is a comma. The implementation, then, is as follows.

```

void parseStmtList() {
    parseStmt();
    while (currentToken.getType() == Token.TokenType.COMMA_T) {
        consume(Token.TokenType.COMMA_T);
        parseStmt();
    }
}

```

#### ☛ Activity 11 –

Work through the PDef Parser Tutorial in Chapter 13. Before embarking on the Parser Tutorial be sure to review the Tutorial Overview, Chapter 11, and the work done for the Tokenizer Tutorial of Chapter 12.

---



## Chapter 7

# Syntax Tree – Theory and Practice

Once the parser has shown its input to be syntactically correct, it's up to the semantic analyzer to verify the semantic correctness of the input. The semantic analyzer, as described in Section 1.4, comprises three elements, the syntax tree, the symbol table, and the actual semantic checking algorithms, and its functionality is determined by the semantic rules for the target language. In this chapter we will discuss issues related to the design and implementation of the syntax tree for a grammar, thus providing the first element needed for semantic checking.

### 7.1 An Overview

A syntax tree is constructed at run-time by the parser and is designed to have the syntactic structure and data represented by the tokens on the input stream. A syntax tree, once constructed, remains in memory for use by the semantic checker and the code generator. Because different syntactic elements in a language have different structure and data content, there will have to be multiple node types, to accommodate the variance in structures. Imagine, for example, a node for an assignment statement

```
x = expression
```

versus a node for a while statement.

```
while ( condition ) Block
```

For the assignment statement we would need one entry for the identifier token (on the left of the assignment) and another entry for the expression (on the right of the assignment). We can describe the assignment structure as follows, using Java class notation.

```
public class Assignment {
    private Token target;
    private Exp  source;
    public Assignment(Token name, Exp exp) {
        target = name;
        source = exp;
    }
}
```

A while statement has a bit more punctuation than an assignment, but still has just two components; an appropriate tree node would have two entries, one for the statement’s condition expression and the other for its block of code. We can imagine the same sort of structure arising for the while statement.

```
public class While {
    private Exp    condition;
    private Block body;
    public While(Exp e, Block b) {
        condition = e;
        body = b;
    }
}
```

Of course, these two examples implicitly assume that structures for `Token`, `Exp`, and `Block` are defined based on their own syntactic structures.

## 7.2 A Syntax Tree Example

Before delving into the technical aspects of the syntax tree it will be useful to look at an example that illustrates the basic process of construction. The idea is to draw a diagram representing the structure and critical data of a particular PDef string. Before looking at a more interesting example we will consider, not a complete PDef string, but simply an assignment entry from a list – Figure 7.1(a). The process of abstraction is to associate a form with a name (the abstraction) and then to associate the name with its constituent parts. In this case we will associate the name `Assignment` with the assignment entry ‘`b = a`’ and then notice that the assignment has two `identT` tokens and one `assignT` token. In the program code the token `assignT` is necessary to indicate that the statement is an assignment statement. In the syntax tree the `assignT` is replaced by the name `Assignment` for the node representing the statement. The diagram in Figure 7.1(b) illustrates the abstraction, showing a structure containing two `identT` tokens associated with the name `Assignment`.

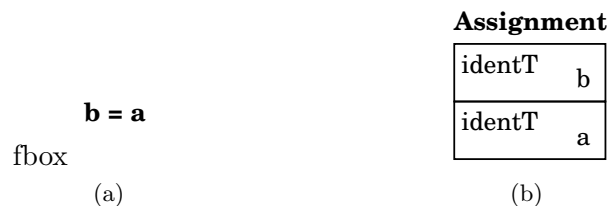


Figure 7.1: A PDef Assignment and its Syntax Tree Node

Seeing this example you might imagine that ultimately we will represent this structure by an object of a class named `Assignment` with two `Token` data components.

We now turn to the following larger example.

```
{int a, {float b, a = b}, int b, b = a}
```

The approach will be to continually rewrite this string in increasingly detailed form, but eliminating unnecessary details in the form of punctuation-type tokens. Each structure encountered can be replaced by an appropriately named box whose contents reflect the components of the syntactic structure. Notice that the names of boxes derive from non-terminal symbols from the PDef grammar corresponding to the particular syntactic structure.

The first thing we see in our example string is a list of four entries. In Figure 7.2(a) we see a box labeled **Block** that contains a reference to another box labeled **StmtList**. This matches the familiar grammar structure. The elements of the list are still in their original string form.

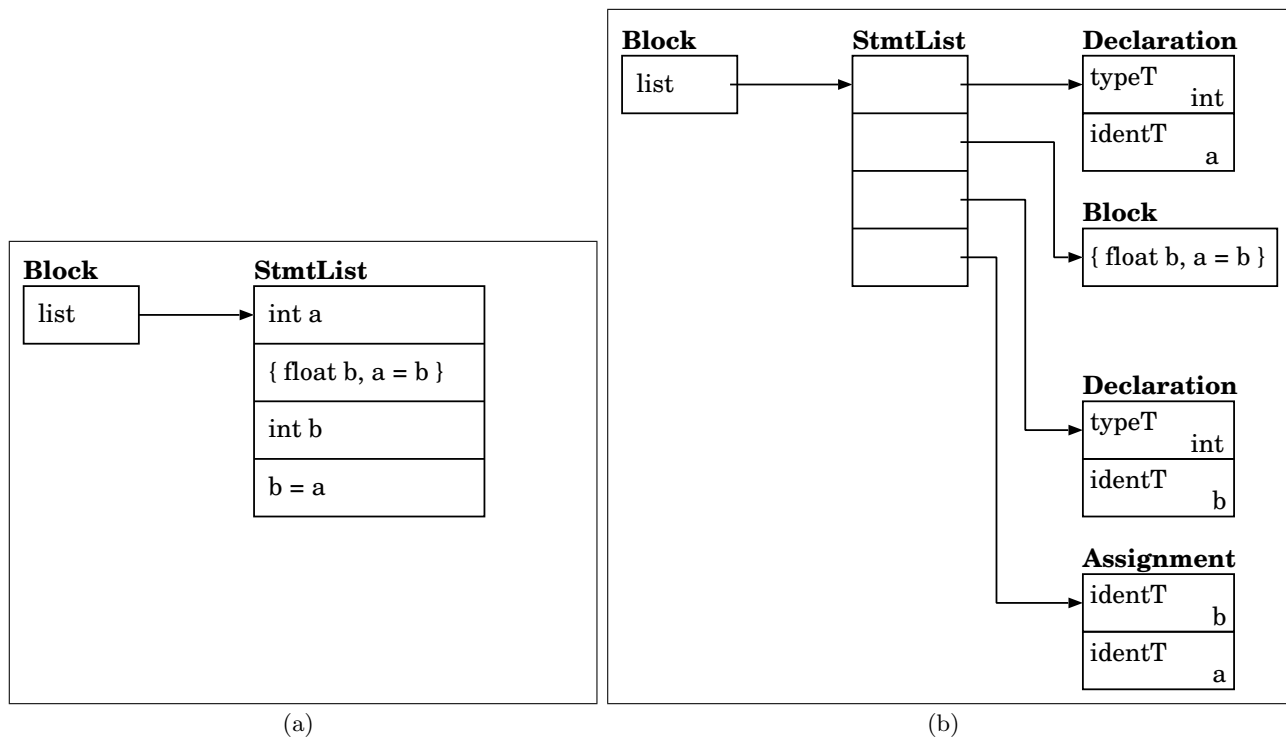


Figure 7.2: A Syntax Tree for `{int a, {float b, a = b}, int b, b = a}`

The next step, then, should be to expand these entries into their boxed representations. Figure 7.2(b) shows the second step in the expansion: the simple structures, such as declarations and assignments, have been put into a final form following the pattern we saw in Figure 7.1(b).

The final syntax tree is given in Figure 7.3 and has a form that should not be a surprise. It is useful to realize that the production of the syntax tree by the parser is analogous to the production of the stream of tokens by the tokenizer – the tokenizer takes its (character) input and produces a sequence of tokens, the parser takes its input (tokens) and produces a single syntax tree.

### Activity 12 –

Draw a syntax tree for the following PDef string, following the pattern established in this section. Remember, the syntax for PDef is defined in Figure 4.2.

`{{float x, x = a}, int a, {int b, b = a}, float x}`

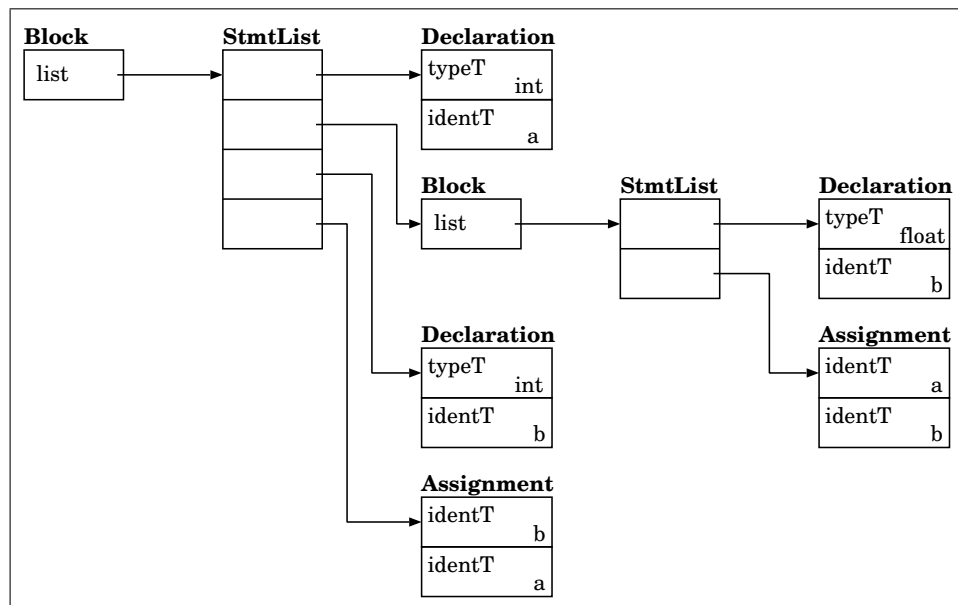


Figure 7.3: A Syntax Tree for  $\{int\ a, \{float\ b, a = b\}, int\ b, b = a\}$

### 7.3 Designing Syntax Tree Nodes

The intuitive examples above illustrate the strategy we will apply in defining the syntax tree structure for a language:

#### Syntax Tree Formation Rule

*For each non-terminal define a class (or class hierarchy) whose data members represent the non-terminals and meaningful tokens appearing on the right sides of the non-terminal's rules.*

Knowing that ultimately it is the parser that creates the syntax tree, we center our syntax tree design strategy on the six grammar rule forms that we considered for the recursive descent parser in the previous chapter. The six forms, which appear in Figure 6.9, are displayed again for convenience.

- Form 1:  $NT \rightarrow tok1T\ tok2T\ tok3T$
- Form 2:  $NT \rightarrow NT1\ NT2\ NT3$
- Form 3:  $NT \rightarrow tok1T\ NT1\ NT2\ tok2T\ NT3$
- Form 4:  $NT \rightarrow "rhs1" \mid "rhs2" \mid "rhs3"$
- Form 5:  $NT \rightarrow NT1\ [ tokT\ NT2 ]\ NT3$
- Form 6:  $NT \rightarrow NT1\ \{ tokT\ NT1 \}$

Remember that our expectation is that, in a given grammar, each non-terminal can be defined by a rule in one of these forms. We will examine each of the forms and describe the structure of a class appropriate for the syntax tree nodes for the form.



### 7.3.1 Our Strategy

#### Putting a Top on the Syntax Tree Hierarchy

For flexibility and orderliness it is advantageous to start with an abstract class at the top of the syntax tree hierarchy. This abstract class will be the repository for any common data values or methods which appear in later phases. In addition, subclass polymorphism will make it possible to treat objects of the subclasses as though they were objects of `SyntaxTree`. The structure of the abstract class initially will be mostly empty.

```
public abstract class SyntaxTree {
    public SyntaxTree() { }
}
```

#### Form 1: All terminal symbols

$$NT \rightarrow \text{tok1T tok2T tok3T}$$

First, while it is possible to have a rule in which all the terminal symbols on the right are punctuational, in a real language such a description is unlikely – a sequence of punctuation symbols could hardly be seen as an abstraction. So, while there must be at least one data token, some of the other tokens could be punctuation. It is easiest to assume, in dealing with this general form, that all three of the terminals in the rule carry meaningful data. In other words, `NT` is an abstraction for the collection of the three data values. In this case we will associate with `NT` a syntax tree class which stores each of the tokens on the right. The appropriate class structure would be the following.

```
public class NT extends SyntaxTree {
    // Class Invariant: tok1.getType() == tok1T AND
    //                  tok2.getType() == tok2T AND
    //                  tok3.getType() == tok3T
    private Token tok1;
    private Token tok2;
    private Token tok3;
    public NT(Token t1, Token t2, Token t3)
    // Pre:  t1.getType() == tok1T AND
    //       t2.getType() == tok2T AND
    //       t3.getType() == tok3T
    // Post: tok1 = t1 AND tok2 = t2 AND tok3 = t3
    { tok1 = t1; tok2 = t2; tok3 = t3; }
}
```

The class invariant in this here is important. Since objects of class `Token` can have various values, we must provide some guarantee that the data members receive values of the appropriate type: i.e., `tok1T`, `tok2T`, and `tok3T`. Notice that the precondition for the class constructor echoes the class invariant, which means that when an `NT` object is constructed, the programmer must ensure that the tokens passed to the `NT` constructor match the constructor's precondition.

Obviously, the strategy above is easily modified if a grammar rule has two or four or more meaningful tokens. The case of a single meaningful token on the right of a rule bears discussion.

When this happens in a grammar rule it means that the non-terminal on the left is acting as an abstraction for a particular token – an `identT` token, for example. In this case we proceed as above except that we will name the class `Tok1ST`, to emphasize the fact that the class abstracts that token. It is customary to refer to such classes as wrapper classes, as the class seems to be wrapped around the one data member.

### Form 2: All non-terminal symbols

$$NT \rightarrow NT1 \ NT2 \ NT3$$

This form is similar to the first in that there is a straightforward way to define the associated syntax tree class, but there’s also a twist in the case of a single non-terminal.

Given the form above we first must assume that, since `NT1`, `NT2`, and `NT3` are non-terminals, each is defined by a grammar rule and consequently will have an associated syntax tree class – let’s assume these classes are called `NT1`, `NT2`, and `NT3` (i.e., the class names match the non-terminal names). In an `NT` node we would expect to see references to nodes corresponding to the three non-terminals on the right – that means a class structure as follows.

```
public class NT extends SyntaxTree {
    private NT1 nt1;
    private NT2 nt2;
    private NT3 nt3;
    public NT(NT1 n1, NT2 n2, NT3 n3)
    // Pre: n1, n2, n3 != null
    // Post: nt1 = n1 AND nt2 = n2 AND nt3 = n3
    { nt1 = n1; nt2 = n2; nt3 = n3; }
}
```

The twist occurs when there is a single non-terminal (i.e.,  $NT \rightarrow NT1$ ). In such a case we could define our class `NT` just as above, except with just a single data member – say `nt1` of the class `NT1`. But now `NT` is really no different from `NT1` – it’s the same except the `NT1` reference is “wrapped” inside. In this case we will drop the invention of a class `NT` and simply associate the class `NT1` with the non-terminal `NT`.

While this last rule (for the case of a single non-terminal on the right) is often applied, there can be exceptions. For example, it can be that a rule  $NT \rightarrow NT1$ , in addition to the syntactic structure, can implicitly carry semantic data – a fact that might not emerge until the semantic checking phase. When this can be predicted, then we will go back to the original strategy and define a new class `NT`. This situation also occurs in the case of Form 3. This connection with semantic data will be discussed shortly in the context of the `PDef` non-terminal `Block`.

### Form 3: A mix of terminals and non-terminals

$$NT \rightarrow \text{tok1T} \ NT1 \ NT2 \ \text{tok2T} \ NT3$$

When we have a rule in Form 3 and assume that the terminal symbols `tok1T` and `tok2T` carry meaningful data, then we form the class `NT` by integrating the strategy of Form 1 with that of Form 2. The class definition for `NT` will have the following structure.

```

public class NT extends SyntaxTree {
    // Class Invariant: tok1.getType() == tok1T AND tok2.getType() == tok2T
    private Token tok1;
    private Token tok2;
    private NT1   nt1;
    private NT2   nt2;
    private NT3   nt3;
    public NT(Token t1, Token t2, NT1 n1, NT2 n2, NT3 n3)
    // Pre:  t1.getType() == tok1T AND t2.getType() == tok2T
    // Post: tok1 = t1 AND tok2 = t2 AND
    //       nt1  = n1 AND nt2  = n2 AND nt3 = n3
    {
        tok1 = t1; tok2 = t2;
        nt1  = n1; nt2  = n2; nt3 = n3;
    }
}

```

Notice that the class invariant as well as the precondition on the constructor behave as they did in the previous two Forms.

Again, there is a twist – this time in the case where there are punctuational tokens and a single non-terminal. Since the terminals can be ignored, this case is treated just like the one-non-terminal case in Form 2.

#### Form 4: When there is a choice

$$NT \rightarrow \text{"rhs1"} \mid \text{"rhs2"} \mid \text{"rhs3"}$$

A non-terminal whose grammar rule offers choices is the most interesting of the rule forms. This form, for example, is used for the non-terminals `Stmt` and `Factor` in the PDef grammar. As we did with Form 4 for analyzing parse methods, we will start out by simplifying the grammar rule so that each right side is a single non-terminal.

$$NT \rightarrow NT1 \mid NT2 \mid NT3$$

Our strategy in this situation is two fold: First define a new abstract class `NT` defined as follows.

```

public abstract class NT extends SyntaxTree {
    public NT() { super(); }
}

```

We then modify the class definitions for `NT1`, `NT2`, and `NT3` to specify each class as a subclass of `NT`. This defines the class hierarchy in Figure 7.4. This structure implies that in building a syntax tree, anywhere an `NT` object is called for we can substitute objects of any of `NT`'s subclasses.

Addressing the general case now becomes easy – we turn the general case into the special by rewriting the Form 4 grammar rule as follows.

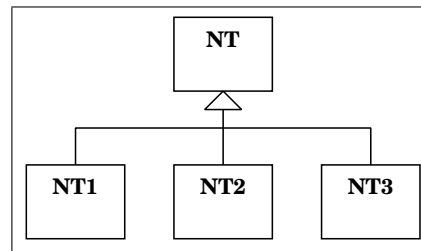


Figure 7.4: The UML Class Hierarchy Implied by Form 4

```

NT    → RHS1 | RHS2 | RHS3
RHS1 → "rhs1"
RHS2 → "rhs2"
RHS3 → "rhs3"
  
```

As an example of this general strategy consider the following simple grammar rule.

```

NT → tok1T NT1
    → tok2T
    → NT3 [ tok4T NT5 ]
  
```

The first step, after defining the abstract class `NT`, is to replace the three grammar rules above with the following four.

```

NT    → RHS1 | RHS2 | RHS3
RHS1 → tok1T NT1
RHS2 → tok2T
RHS3 → NT3 [ tok4T NT5 ]
  
```

Now we handle each of the last three rules one at a time. In the first rule, if we assume that `tok1T` is punctuational, then Form 3 above says we should use the class `NT1` in place of `RHS1` (assuming no semantic entanglements). In this case we would also replace `RHS1` in `NT`'s rule by `NT1`. If `tok1T` is not punctuational then in following Form 3 we define a new class `RHS1` – we also make it a subclass of `NT`.

For the second rule the right side is simply a token (`tok2T`). We see such a form in the `PDef` rule for `Factor`, where `identT`, `intT`, and `floatT` all appear alone as right side options. According to Form 1 above we should define a wrapper class for `tok2T` and name it `RHS2` – this new class will also be a subclass of `NT`.

Finally, we have the third rule, which has Form 5. We haven't discussed that form yet, but it will result in a new class named `RHS3`. Whatever class Form 5 tells us to define, we will call it `RHS3` and make it a subclass of `NT`. In this way we complete the `NT` class hierarchy.

### Form 5: When something is optional

```

NT → NT1 [ tokT NT2 ] NT3
  
```

A grammar rule with the optional structure is easily handled in a manner similar to that for Forms 1, 2, and 3. We define a class which matches the right side and assumes the optional

part is required. Then, when instantiating the class, we set the data member(s) representing the optional part to `null` if the structure is not present. An example would be the standard `if-else` structure in which the *else* part is optional. The class associated with the grammar rule above would have the following structure (assuming `tokT` acts as punctuation).

```
public class NT extends SyntaxTree {
    // if nt2 == null then the optional part of NT is not present
    private NT1 nt1;
    private NT2 nt2;
    private NT3 nt3;
    public NT( NT1 n1, NT2 n2, NT3 n3)
    // Pre:  n1 != null AND n2 != null AND n3 != null
    // Post: nt1 = n1 AND nt2 = n2 AND nt3 = n3
    {  nt1 = n1; nt2 = n2; nt3 = n3; }
    public NT( NT1 n1, NT3 n3)
    // Pre:  n1 != null AND n3 != null
    // Post: nt1 = n1 AND nt2 = null AND nt3 = n3
    {  nt1 = n1; nt2 = null; nt3 = n3; }
}
```

Notice the two constructor methods, one used when the optional part is present and the other used when the optional part is missing. We should note that there is a special situation where the optional structure is used to define a recursive list structure. Such recursive list structures can be handled in a convenient way, which will be discussed below.

#### Form 6: When there is repetition

$$NT \rightarrow NT1 \{ \text{tokT } NT1 \}$$

This form is used to describe unstructured lists, in this general case, of `NT1`'s. The token `tokT` is punctuation for separating the components of the list, so we will ignore it – in fact in some cases it may be missing, i.e., the list is whitespace separated. We could conjure up a class structure for representing a list of `NT1`'s, but in most object-oriented languages such a class is predefined – in Java we can use the generic class `LinkedList`. In using this class we must give an indication of what objects can be stored, so we use the form `LinkedList<NT1>`. So for unstructured lists, the syntax tree component requires no implementation.

### 7.3.2 Syntax Tree Structures for Structured Lists

In Section 2.3 we discussed at length the problems of ambiguity that can arise in defining context free grammars for lists. We looked at precedence ambiguity and associative ambiguity and discussed how to resolve the ambiguities – review the ideas in Section 2.3.3, in particular. Resolving associative ambiguity is important, but there was one side effect: the resulting grammar had left-recursion. Left recursion is a problem for recursive descent parsing and we had to rewrite the expression grammar to remove the left-recursion, making the grammar less intuitive. This is a problem for any list structure where structure counts, as in arithmetic expressions.

Our interest here is to understand how to define syntax tree nodes for structured lists. We know that the syntax tree nodes will be generated by corresponding parse functions, and that those parse functions are derived from the following grammar structures, depending on whether the recursion is left or right. The following table shows both alternatives.

	<i>Left-recursive</i>	<i>Right-recursive</i>
List <sup>1</sup>	→ Element RestOfList	List → Element tokT List
RestOfList	→ tokT Element RestOfList	→ Element
	→ ε	

You should review the parse methods, for these rules, that are discussed in Section 6.3. While these grammars are necessary for structuring the parse method `parseList`, it turns out the original ambiguous grammar structure is more convenient and completely adequate for structuring the corresponding syntax tree structure. Remember these ambiguous rules for structured lists.

```
List → List tokT List
      → Element
```

When you look at the ambiguous grammar as a basis for a syntax tree structure you would be justified in asking “How much of the list is to the left and how much to the right?” But since the nicely structured parse methods will determine the answer to this question when it parses an actual list, we know the ambiguous structure will have the left and right parts laying over the correct parts of the list.

So, using the ambiguous grammar rules for `List`, we see that it is Form 4 which we must follow in structuring our syntax tree classes. We will use the name `ListST` as the name of our abstract class, allowing the other classes to be more clearly associated with their corresponding grammar rules. One more important point before looking at the class definitions. Because these are structured lists, it is unlikely that the separator is merely punctuation. A good example would be the definition of `Term` in the expression grammar for `PDef` (see page 39). There the separator can be the plus sign or minus sign, each carrying vital semantic information. Here are the resulting class definitions.

```
public abstract class ListST extends SyntaxTree { }

public class List extends ListST {
    private ListST left;
    private ListST right;
    private Token sep;
    public List(ListST lexp, Token t, ListST rexp) {
        left = lexp;
        right = rexp;
        sep = t;
    }
}
```

---

<sup>1</sup>The use of `List` is meant to reflect the problem being solved. It is understood that the name is also a common class name in Java and other object oriented languages.

```
public class Element extends ListST {
    ... // appropriate structure for an Entry
    public Entry(...) { ... }
}
```

The strategy discussed above gives us a way of defining syntax tree structures that will work with both left and right-recursive lists. The strategy will be put to use in the Syntax Tree Tutorial (Chapter 14). Now we will look at examples from PDef of applying the Form-based strategy discussed above.

### 7.3.3 Examples from PDef

Our ongoing question is “How does this fit into the PDef translator?” Most of the answer to this question will be saved for the tutorial paired with this chapter (see Chapter 14). But to demonstrate how the Forms described above work for a real grammar we will look at a couple of the PDef grammar rules, namely for `Stmt` and `Block`.

`Stmt` → `Declaration` | `Assignment` | `Block`

This grammar rule is a clear example of Form 4. We have three alternatives, each a non-terminal. Applying Form 4 is in two parts. First we make an abstract class for the left side of the rule – we will call the class `StmtST`. The definition is very simple.

```
public abstract class StmtST extends SyntaxTree {
    public StmtST() { }
}
```

Then for each of the elements on the right-hand side we create classes based on the nature of each particular alternative. What we can say is that there will be three: `DeclarationST`, `AssignmentST`, and `BlockST`. And it is `BlockST` that is defined in our next example.

`Block` → `lcbT StmtList rcbT`

This grammar rule clearly falls into Form 3, and since `lcbT` and `rcbT` are both punctuational (they simply mark the beginning and end of a list of statements) the grammar rule falls into the special case discussed at the end of Form 3. Since we have one non-terminal and only punctuational terminals, we would expect to follow the rule that the syntax tree class associated with `Block` will be the same as for `StmtList`. But the syntax tree class associated with `Block` serves a special purpose in PDef when it comes to semantic checking. Namely, the syntax tree class associated with `Block` will end up holding a reference to a local symbol table. In this way, the grammar rule for `Block` holds a hidden reference to the symbol table – hidden in the sense that it does not appear in the grammar rule and one has to recognize the situation early on, i.e., at syntax tree design time.

The long and short of it is that in this case we will treat the grammar rule as though it had more than one non-terminal and, for now, define the syntax tree class associated with `Block` as follows.

```

public class BlockST extends SyntaxTree {
    LinkedList<StmtST> list;
    public Block(LinkedList<StmtST> l)
    { list = l; }
}

```

Notice the class name is `BlockST`. We add the `ST` to the non-terminal name so that the result will be easily identified as a syntax tree class. You can see this also in the one element not yet defined: `StmtST` is the name of the class for the non-terminal `Stmt` and it will be defined in the tutorial.

## 7.4 Building a Syntax Tree

If we define all the appropriate syntax tree node classes for a grammar we are still not finished, because the syntax tree nodes have to be generated by the parse methods. In this section we will look at strategies for generating syntax tree nodes, syntax tree generation for structured lists, and apply the techniques to the `PDef` examples we saw in the last section.

### 7.4.1 Generating Syntax Tree Nodes

So we have an idea of the structure of a syntax tree and how its components can be designed from a language's grammar, but how do we create one? Since the syntax tree is generated by the parser, and since the parser and the syntax tree nodes have been designed following the analysis of the same six grammar rule forms, we should be able to combine the two so that the parse method for each form generates the corresponding syntax tree node. In this section we will discuss how to alter the parse method for each grammar rule form, so that it generates the correct syntax tree structure.

What you will see in the sections that follow is a beautiful integration of the recursive nature of the parse methods and the production of the parse tree. We have seen that parse methods call other parse methods based on the structure of the governing grammar rule. To build its parse tree node, a modified parse method will gather up the parse tree nodes returned by the parse methods it calls, and then combine these returned nodes into the parse tree node it will return. So our approach will be to examine each parse method, determine the data returned by each called parse method, and then to combine the returned data into the required return value.

### Forms 1, 2, and 3: A mix of terminals and non-terminals

The first three forms can really be addressed as one and we will focus on Form 3, knowing that the other two forms will be special cases.

$$NT \rightarrow tok1T \ NT1 \ NT2 \ tok2T \ NT3$$

Here is the parse method for this form followed by the specification of the constructor for the corresponding syntax tree node `NT`.



```

void parseNT() {
    consume(tok1T);
    parseNT1();
    parseNT2();
    consume(tok2T);
    parseNT3();
}

NT(Token t1, NT1 n1, NT2 n2, Token t2, NT3 n3)
// Pre: t1.getType() == tok1T AND t2.getType() == tok2T

```

First, we assume that each method call in `parseNT` returns a reference to an object of its corresponding syntax tree node. **IMPORTANT:** During parsing, each meaningful token must be saved before `consume` is called, so the token value can be passed into the `NT` constructor later. The result is the following method.

```

NT parseNT() {
    Token t1 = currentToken; consume(tok1T);
    NT1 nt1 = parseNT1();
    NT2 nt2 = parseNT2();
    Token t2 = currentToken; consume(tok2T);
    NT3 nt3 = parseNT3();
    // Assert: t1.getType() == tok1T AND t2.getType() == tok2T
    return new NT( t1, nt1, nt2, t2, nt3 );
}

```

Notice that the assertion preceding the return statement is justified because we know that at that point, each `consume` was successful – i.e., that the token consumed has the same type as the argument to `consume`.

Since the three Forms were compressed into one, we should also mention the special cases mentioned for these Forms. First, what if the right side contains a single meaningful terminal symbol `tokT`. Then, as indicated earlier, we associate the wrapper class `TokST` with `NT`. The parse method would take on the following appearance (assume the grammar rule has the form  $NT \rightarrow tok1T tok2T$  where `tok1T` is assumed to be punctuation).

```

Tok2ST parseNT() {
    consume(tok1T); // we can ignore this one
    Token t2 = currentToken; consume(tok2T); // remember the meaningful token
    // Assert: t2.getType() == tok2T
    return new Tok2ST( t2 );
}

```

In a similar vein, let's assume the rule for `NT` has the form  $NT \rightarrow tok1T NT1 tok2T$ , where both terminal symbols are punctuational. In Form 3 earlier we indicated in this case we would associate the class `NT1` with the non-terminal `NT` (unless there are semantical entanglements). So, assuming no entanglements, we would adopt the following form for the parse method.

```

NT1 parseNT() {
    consume(tok1T); // we can ignore this one
    NT1 nt1 = parseNT1();
    consume(tok2T); // we can ignore this one as well
    return nt1;
}

```

Notice in this case that the value returned by `parseNT1` is simply passed through as the return value of `parseNT`.

#### Form 4: When there is a choice

$$NT \rightarrow \text{"rhs1"} \mid \text{"rhs2"} \mid \text{"rhs3"}$$

In the discussion above for generating a syntax tree structure for Form 4, we first analyzed a simplified case where each right side is a single non-terminal. In this case we defined `NT` to be an abstract class and each of the classes `NT1`, `NT2`, and `NT3` to be subclasses of `NT`. Thanks to this structure we can declare a variable of type `NT` at the beginning of the parse method and then assign to it the values returned by the parse methods for `NT1`, `NT2`, and `NT3`. Remember that this all works because we have defined a class hierarchy headed by `NT` – see Figure 7.4. The following extended implementation of `parseNT` naturally results.

```

NT parseNT() {
    NT value = null;
    switch (currentToken.getType()) {
    case tok1T:
        value = parseNT1();
        break;
    case tok2T:
        value = parseNT2();
        break;
    case tok3T:
        value = parseNT3();
        break;
    default:
        error();
    }
    return value;
}

```

At the end of our discussion of Form 4 in Section 7.3.1, we concluded that any grammar rule with options can be modified so that the strategy just laid out works, with the exception that if a choice is a single terminal symbol (a token), then we need to define a wrapper class for that token and it is that wrapper class that represents that token in the syntax tree hierarchy. If we assume in the example above that the second case has `tok2T` as the right side then we would implement that switch case as follows.

```

...
case tok2T:
    value = new Tok2ST(currentToken);
    consume(tok2T);
    break;
...

```

**Form 5: When something is optional**

$$NT \rightarrow NT1 [ tokT NT2 ] NT3$$

Here is the previously defined parse method for this grammar rule form followed by the constructor for the syntax tree class NT.

```

void parseNT() {
    parseNT1();
    if (currentToken.getType() == Token.TokenType.tokT) {
        consume(tokT); // now consume tokT
        parseNT2();
    }
    //
    parseNT3();
}

public NT( NT1 n1, NT2 n2, NT3 n3)
// Pre:  n1 != null AND n3 != null

```

The critical thing to notice is that the return statement we add at the end of the parse method will call the NT constructor. Thus, we will have to have a value to return of type NT2, whether or not the value is null. So we can declare a variable of type NT2, set it to null and then use the same variable to receive the return value from parseNT2. The following modification is appropriate.

```

void parseNT() {
    NT2 nt2 = null;
    NT1 nt1 = parseNT1();
    if (currentToken.getType() == Token.TokenType.tokT) {
        consume(tokT); // now consume tokT
        nt2 = parseNT2();
    }
    NT3 nt3 = parseNT3();
    // Assert: nt1, nt3 != null
    return new NT(nt1, nt2, nt3);
}

```

**Form 6: When there is repetition**

$$NT \rightarrow NT1 \{ \text{tokT } NT1 \}$$

As discussed in the last section, the appropriate syntax tree structure for `NT` in this case is the Java class `LinkedList`. The problem now is how to extend `parseNT` to generate the structure. First, each call to `parseNT1` should produce a new entry to be placed on the list. In order to place the first entry on the list, the `LinkedList` object will have to be instantiated first thing in the method. If we use the method `LinkedList.addLast` for adding elements, then at the end of the method the list should be ready to return. The following implementation is the natural result.

```
LinkedList<NT1> parseNT() {
    LinkedList<NT1> list = new LinkedList<NT1>();
    NT1 element = parseNT1();
    list.addLast(element);
    while (currentToken.getType() == Token.TokenType.tokT) {
        consume(Token.TokenType.tokT);
        element = parseNT1();
        list.addLast(element);
    }
    return list;
}
```

**7.4.2 Syntax Tree Generation for Structured Lists**

Forms 1-6 are a good guide to implementing parse methods that generate appropriate syntax tree structures. But a common task deserves a bit more attention. Sections 6.3 and 7.3.2 focused on parsing and syntax tree structures specific to structured lists. In this section we complete the sequence with a discussion of syntax tree generation techniques for structured lists.

In a structured list we have one of two forms for the expression `x op y op z`:

**left recursive:** `(x op y) op z`

**right recursive:** `x op (y op z)`

In the first we always add the next element to the expression already computed, while in right-recursive lists we add the current element to the rest of the list which is yet to be computed. Left-recursive lists are like the sum of a sequence of values – we add as we go along, so  $x+y+z = (x+y)+z$ . Right-recursive lists are like a value raised to multiple powers: like  $x^{y^z}$ . We apply the operations in right-to-left order – that is  $x^{y^z} = x^{(y^z)}$ . Constructing the syntax tree structures that distinguish these two possibilities is crucial.

**Right-recursive List**

When we construct a right-recursive list we use the structure ‘`a op A`’, where ‘`a`’ is the first element, or *head*, of the list and ‘`A`’ is the rest (also called the *tail*) of the list – this is consistent with the structure in the class `List` (see Section 7.3.2 page 106) if we use the variable `left` for

the element and the variable `right` for the list. Now the question is, how must we modify the method `parseList` (in this right-recursive case) in order to have it generate the appropriate structure? We display the appropriate parse method again for convenience.

```
void parseList() {
    parseElement();
    if (currentToken.getType == tokT) {
        consume(tokT);
        parseList();
    }
}
```

So where do we start? First, we need to recognize that this parse method will return a list of type `ListST` (the Forms above dictate the new name for the list class). According to the method we will see a element (the first) and then if there is a separator we will parse the rest of the list. So we see the first element and the rest of the list and we should combine them in that way (along with the separator). Notice that according to our node construction strategy, since `Element` is an option for `List`, the class `ElementST` will be a subclass of `ListST`. So, if there is no separator we can simply return the element we found as a `ListST`. The following modified parse method naturally follows.<sup>2</sup>

```
ListST parseList() {
    ListST list = null;
    Token tok = null;
    List elt = parseElement();
    if (currentToken.getType == tokT) {
        tok = currentToken;
        consume(tokT);
        list = parseList();
        list = new List(elt, tok, list);
    }
    else {
        list = elt;
    }
    return
}
```

### Left-recursive List

From the examples at the beginning of this section, when we construct a left-recursive list we use the structure ‘A op a’, where ‘A’ is the beginning of the list and ‘a’ is the tail (i.e., last)

---

<sup>2</sup>

You may be tempted to be more compact with your code and rewrite the `if`-block in `parseList` as follows.

```
tok = currentToken;
consume(tokT);
list = new List(elt, tok, parseList());
```

While this looks cooler, it destroys the connection between the parse method and the grammar rule from which it is derived. Control is important when applying theory to programming – make sure the theory is always evident, since it is a good indication that the code is correct.

element of the list – this is consistent with the structure in the class `List` (see Section 7.3.2 page 106) if we use the variable `left` for the list and the variable `right` for the element. How can we generate this using our two left-recursive parse methods? Here are they are for convenience.

```

void parseList() {
    parseElement();
    parseRestOfList();
}

void parseRestOfList() {
    if (currentToken.getType() == tokT) {
        consume(tokT);
        parseElement();
        parseRestOfList();
    }
}

```

Each of these methods will have to have `ListST` as the return type. But the internal structure is not quite as natural because of the left recursion removal.

In the method `parseList` we first recover a reference to an `Element` object. It would seem that we would recover a reference to a `ListST` object from `parseRestOfList` and then combine these two references – but this doesn't work. The final return value is constructed inside `parseRestOfList`. In order to do this we must pass it the reference to the `Element` object so it has access to it.

```

ListST parseList() {
    ListST elt = parseElement();
    ListST list = parseRestOfList(elt);
    return list;
}

```

The method `parseRestOfList` has more work to do, and that work depends on whether the null string grammar rule option is taken. We can tell by checking for an occurrence of the token `tokT`, the separator. If there is no separator then we've already seen the last element of the list – i.e., the argument `x` is the thing we want to return. If there is a separator then we need to carry out the work in the true branch of the selection: we must consume that separator, parse an element, make a new list from `x`, the separator, and new element. This new `list` is passed on to the call to `parseRestOfList`. What is returned by that call will be the return value in this case. Here's the appropriate implementation – notice the code depends as for right-recursive lists on the fact that `Element` must be a subclass of `ListST`

```

ListST parseRestOfList(ListST x) {
    // x represents the list that has been seen so far on the left
    ListST list = null;
    Token tok = null;
    if (currentToken.getType() == tokT) {
        tok = currentToken;

```

```

        consume(tokT);
        ListST elt = parseElement();
        list = new ListST (x, tok, elt);
        list = parseRestOfList(list);
    }
    else // we're at the end of the list
        list = x;
    return list;
}

```

### 7.4.3 Examples for PDef

Here we demonstrate how the PDef parser will construct a syntax tree. We focus on the grammar rules from Section 7.3.3, where we discussed syntax tree class examples. So in this section we will discuss the parse methods for `Stmt` and `Block`.

`Stmt` → `Declaration` | `Assignment` | `Block`

As seen in the Form 4 discussion above, converting the parse method is straightforward. We declare a `StmtST` variable and have it receive the value of each parse method call in the `switch` statement. It is that `StmtST` variable that carries the parse method's return value. Here is the appropriate definition. Notice the comments on the `switch` cases indicate what must be switched on.

```

StmtST parseStmt() {
    StmtST stmt = null;
    switch (currentToken.getType()) {
    case typeT: // Declaration must start with typeT
        stmt = parseDeclaration();
        break;
    case identT: // Declaration must start with identT
        stmt = parseAssignment();
        break;
    case lcbT: // Block must start with lcbT
        stmt = parseBlock();
        break;
    default: // should never reach this point!
    }
    return stmt;
}

```

`Block` → `lcbT StmtList rcbT`

The following parse method follows naturally from the discussion in the combined Forms 1-3 above.

```

BlockST parseBlock() {
    consume(lcbT); // ignore this token
    LinkedList<StmtST> list = parseStmtList();
}

```

```

        // we know this is the syntax tree class associated with StmtList
        consume(rcbT); // ignore this as well
        return new BlockST(list);
    }

```

Notice that the syntax tree element, `list`, is generated simply by having the parse method `parseStmtList()` return the list that it constructs. This is the beauty of the recursive descent parser: each parse method builds its part of the tree based on its own grammar rule and without really knowing how the other parts are constructed.

This is the very standard process applied to the parse methods so that they can generate appropriate syntax tree nodes. It is important to recognize that the object reference returned by `parseStmtList` in the `parseBlock` method above will be a reference to a subtree of nodes describing the statement list. That same list value will be returned by `parseBlock` in `parseStmt` and then be the return value from `parseStmt`. So as each parse method returns the node it constructed, they will be combined into a new node and passed back to the next calling method. In the end the very first parse method called, `parseProgram`, will return the actual completed parse tree.

## 7.5 Syntax Tree Traversal

We have chosen this syntax tree structure because it will simplify the implementation of the semantic part of the recognizer. Recall that the semantic part includes symbol table formation and display, static semantic checking, and code generation (where appropriate). It turns out that all of these activities, as well as syntax tree display, can be implemented based on a single algorithmic form, the depth-first traversal or, more commonly, depth-first search (DFS) of the syntax tree. With several of these DFS algorithms in your future, it will be useful to have a general strategy for designing and implementing one. The development of the method `traversal`, which follows in this section, will demonstrate the strategy. But note that the method `traversal` that we are about to develop does not actually appear yet in our PDef implementation.

Our DFS algorithms will all be based on a simple DFS algorithm that simply displays the class name of each node visited, with each name displayed on its own line. In other words, we visit each node in the tree in a depth-first order and use a `println` statement to print the class name of the node visited. To implement a traversal we do the following steps.

### Step 1:

*Determine a signature for the traversal method that will be used in each class of the hierarchy.*

It is important to realize that this step may require determining if there are parameters required in order to implement the algorithm.

### Step 2:

*Make a table listing the class of each node type in the tree structure and for each class give a description of the action when such a node is visited.*



**Step 3:**

*Specify for each class an algorithm that describes the action from Step 2.*

It is important to remember that our method must be implemented for each class in the tree structure; if the class is for a leaf node and has no action then the method could simply be an empty block of code. But the methods for internal (non-leaf) nodes must include code to traverse any subnodes of the node.

To illustrate the development of the traversal algorithm we focus once again on components from the PDef language as we have done in previous examples – see Sections 6.4, 7.2, and 7.4.3. In these previous example sections we used `Block` and `StmtList`. But the syntax tree `StmtList` has a special status since its corresponding syntax tree structure is `LinkedList<Stmt>`, which is in the Java library. We will see below how this class is handled specially. So rather than focusing on `StmtList` in this example, we will use the syntax tree node classes `Declaration` and `Assignment`.

**Step 1**

We will cleverly use the name `traverseST` for our traversal method and its signature is as follows.

```
public void traverseST()
```

In this case we will use the following signature in the super class `SyntaxTree`.

```
public abstract void traverseST()
```

Remember that when a method is declared in this way in a superclass it means that the method must be implemented in each of its (non-abstract) subclasses.

**Step 2**

In this design step we produce a table, listing for each of our example node-classes a description of the processing to take place when each corresponding node is visited.

class of node	layout description
<code>BlockST</code>	This is an internal node and all links to subtrees are stored in the data member <code>list</code> (of type <code>LinkedList&lt;Stmt&gt;</code> ). The first thing we do is to step through the <code>LinkedList</code> referenced by <code>list</code> and call <code>traverseST</code> on each of its elements in order – this will display each subtree referenced in the list. Then we display the name <code>Block</code> and return.
<code>DeclarationST</code>	This is a leaf node of the hierarchy so we display the name <code>Declaration</code> and return.
<code>AssignmentST</code>	This is a leaf node of the hierarchy so we display the name <code>Assignment</code> and return.

**Step 3**

The descriptions above are implemented by the following two segments of Java code. The implementations are straightforward except perhaps for the use of the special Java `for` statement in `Block.traverseST`.

```
// in class BlockST
public void traverse() {
    for (StmtST st : list)
        st.traverse();
    System.out.println("Block");
}

// in class DeclarationST
public void traverseST() {
    System.out.println("Declaration");
}

// in class AssignmentST
public void traverseST() {
    System.out.println("Assignment");
}
```

**Testing the Example**

If all appropriate `traverseST` methods, as defined in a complete **Step 3**, were added to their corresponding classes then we could test our implementation by executing the following code.

```
Tokenizer tokenStream = new Tokenizer(in, echo);
Parser parser = new Parser(tokenStream);

SyntaxTree syntaxTree = parser.parseProgram();
syntaxTree.traverseST();
```

When the last line is executed the program will perform a depth-first traversal of the syntax tree generated for the given input data and display the names of all nodes visited – they will be displayed in depth-first order. For example, given the input

```
{ int t, { int a, t = a }, t = 3 }
```

the program will produce the following output (annotations on the right are added to help connect the trace to the input).

```
Declaration          first declaration -- for t
Declaration          second declaration -- for a
Assignment           first assignment  -- for t = a
Block                the nested list
```

```
Assignment          second assignment  -- for t = 3
Block               the outer list
```

Of course, we could have `BlockST.traverseST` print `Block` before the `for`-loop and then the order of the lines would be changed. The traversal could also do depth-first right to left! It all depends on the purpose of the traversal. The strategy, illustrated here, provides considerable flexibility with this fairly easy sequence of steps.

A couple of other points to keep in mind: First, it is possible that, rather than printing the nodes as the traversal goes along, the requirement is for the traversal method to return the string that would have been printed. Then the traversal methods will have to assemble the return string from the pieces it gets from its own traversal. Second, most traversals do not print anything. Rather, they carry out some other kind of activity when visiting each node. A symbol table is constructed in this way – most nodes have no symbols, but when a node is visited with symbols they can be entered into the table. Another example is the building of a code table during code generation. So the traversal strategy is very useful. You should remember that what ties all these traversal algorithms together is the basic theory of context free grammars. All we have discussed to this point can be understood by understanding grammars.

### ☛ Activity 13 –

Work through the PDef Syntax Tree Tutorial in Chapter 14. Before embarking on the Syntax Tree Tutorial be sure to review the Tutorial Overview, Chapter 11, and review your work for the Tokenizer and Parser Tutorials (Chapters 12 and 13).

---



## Chapter 8

# Semantic Analysis – Theory and Practice

Syntax analysis (see Chapter 6) can go only so far in implementing the Early Warning Principle (see Section 1.2.1, page 6) – it can check for and report errors in form but that’s it. Semantic analysis is responsible for detecting and reporting misuse of identifiers, literals, and built-in operations (such as the arithmetic operators), that is, uses that break semantic rules. The semantic analyzer, as described in Section 1.4, comprises three elements, the syntax tree, the symbol table, and the actual semantic checking algorithms; its functionality is determined by the semantic rules for the target language. Having described how to design and implement a syntax tree in the previous chapter, we dedicate this chapter to an examination of strategies for symbol table design and implementation and semantic analysis.

### 8.1 An Overview

Since the syntax of a language is described with a context free grammar, it must be that the syntax checker can detect language errors that are independent of the context in which they occur. For example, in a PDef program the list entries ‘`int float`’ and ‘`a = int`’ will be seen as errors regardless of where they occur – the syntax definition says an entry starting with the token `int` must be followed by an identifier token, while an entry beginning with an identifier token must have a ‘`=`’ token followed by another identifier token. If we consider the list entry ‘`x = y + 3`’, on the other hand, we can rightly claim that it is syntactically correct – i.e., its form will be considered correct regardless of where it appears in a PDef list.

But we can also see that this same assignment entry could still be incorrect. Consider the following short PDef program.

```
{ int x, float y, x = y + 3 }
```

The program is syntactically correct, but when we consider the declaration entries we see that the assignment entry breaks a semantic rule *in this context* – i.e., we can’t have an expression on the right evaluating to a `float` value while the identifier on the left has type `int`. If we change the program as follows

```
{ float x, int y, x = y + 3 }
```

then it is both syntactically and semantically correct. So whether or not the assignment entry is semantically correct is context dependent – remember the static semantics of PDef presented in Section 4.3.

How can we detect these context sensitive errors? It would seem that to evaluate the correctness of an assignment entry we must have access to the types associated with the elements of the assignment *at the point where the assignment entry occurs*. So we must provide a mechanism that can carry the type information for identifiers to the site of each assignment entry. The mechanism we seek, of course, is the *symbol table*.

Semantic checking focuses on the uses of identifiers, checking to make sure that each identifier use is in accordance with the semantic rules specified for the target language. Before going too far, it is important to lay a groundwork of definitions so that we can understand what a symbol table carries and how it is used in conjunction with the syntax tree to facilitate semantic checking.

### 8.1.1 Identifiers, Literals, and Attributes

In the lifetime of a program, from translation through execution, identifiers have associated with them various *attributes*, some are determined at translation time (*static attributes*) and some are determined at run time (*dynamic attributes*). A variable name in a program has three attributes: type, address, value. In modern languages the address and value attributes are dynamic – determined at run time. In some modern languages, for example Java and C++, the type attribute is determined at translation time (supplied in the form of a type declaration for the variable) – we call such languages *statically typed*. For other languages, Python and Smalltalk are examples, the type attribute is not known until run time (in this case the type of the variable can vary and coincides with the type of the data the variable currently references) – we call such languages *dynamically typed*.

Our interest in identifiers, relative to semantic checking, is with their static attributes, since it is these that can be used in assessing static semantic correctness. The two kinds of identifiers we will focus on are those that name variables and those that name functions. We have mentioned that variables have one static attribute (type) and two dynamic attributes (address, value). For identifiers that name functions there are four static attributes: the return type, the number of parameters, the list of parameters (a type for each), and the code body, which is really the function name's value; a function name has a single dynamic attribute, its address, which is the address of the first instruction in its body of code.

There are, of course, other uses of names in various modern languages. In most statically typed languages there is a mechanism for defining new types based on the built in types and type constructors. When a type is defined it is given a name. The attributes for a type name will depend on the type constructors available. Basically, the attribute(s) of a type name must describe the structure of the data values for the type. A particular type constructor allows one to define *enumerated types* – an enumerated type is a type name associated with a list of new data value names. A classic example is given in C++ is the following

```
enum Color { red, green, blue }
```

which defines a new type name `Color` and three new literal values `red`, `green`, and `blue`.

Literal values also play a part in semantic checking. Each literal value is a value within some identified type: the literals 3 and 23 are integer values, 31.4 and 0.004 are floating point values,

`true` and `false` are Boolean values, while, as just described, `red`, `green`, and `blue` are values of type `Color`. Of course in some languages it is possible to define more complex literal values, such as string values and array values. In each case, a literal value has an implicit or explicit (in the case of enumerated types) type attribute associated with it. It is this type attribute that is used during semantic checking.

Some languages allow the definition of an abstract data type, which means a description of a data structure along with a list of operations for that structure. Of course there are also many modern languages that allow the definition of classes – depending on the language, the class structure is seen as part of the type system, as in statically typed languages such as Java and C++, or is a separate kind of structure in dynamically typed languages such as Smalltalk and Python.

### 8.1.2 Scope and Block Structure

Identifiers with static attributes can appear in a program in two basic contexts – where the identifier is declared (where static attributes are bound) and where it is used. The uses of variables appear in expressions, assignment statements, and input statements, by enlarge. The job of semantic analysis will be to check that the use of an identifier is in accordance with the attributes bound to it. But for a particular use of an identifier, how do we know what attributes have been bound to it?

Statements that bind static attributes to identifiers are called declaration statements – they are familiar in the form of variable, type, class, and function declarations.<sup>1</sup> In most programming languages it is possible for there to be, in a single program or system, multiple declarations for the same name. This means that it is important to be able to associate each use of an identifier with the appropriate declaration – and the association must be determined automatically by a translator. There are a couple of critical definitions that facilitate this association of identifier use to declaration.

#### Scope of a Declaration

The *scope of a declaration* is the segment of program code within which the attribute bindings of the declaration apply.

What is imprecise about this definition is the use of the phrase “segment of program code.” The word “segment” implies an uninterrupted sequence of program lines – but occurring where in the syntactic structure of a program? In fact, for each language the definition of “segment of program code” must be specified. Modern languages are, for the most part, *block structured*, which means there are code segments defined in the grammar of the language within which declarations can occur – these blocks define for each block structured language what is meant by a “segment of program code.”

In statically typed languages such as Java and C++ the notion of block coincides with most occurrences of matching curly braces. So in a `while` statement, the body of the loop is a block; the body of executable code associated with a function is a block; the code within curly braces of a class definition is a block. It is these blocks that are the focus of attention for defining the declaration scope. There are two standard definitions.

---

<sup>1</sup>Actually, though we use the term “declaration” for all four identifier groups, programmers usually refer to variable “declarations” but class, type, and function “definitions”.

**Declare before use:** In this case the scope of a declaration begins at the declaration and extends to the end of the block containing the declaration.

**Use anywhere:** In this case the scope of a declaration is the entire block within which the declaration occurs.

Declare-before-use is used in Java and C++ for variable declarations that occur in blocks of executable code, while use-anywhere is used in Java and C++ class declarations, where the order in which data member and method declarations appear doesn't matter. In C all declarations, even for function names, require define-before-use – this is why in C if a function is used before its declaration (in mutually recursive functions, for example), a special kind of function definition, a prototype, is required. The prototype consists of the function definition without the associated body of code – this allows checking the function calls without having the entire function definition.

An important point about the connection between declaration scopes and block structures is that in languages such as Java and C++ it is possible to have nested blocks, which means that we could have a declaration for `x` in one block and another declaration for `x` in the nested block. A simple illustration of this comes from PDef, in which blocks correspond to lists bounded by curly braces, blocks can be nested, and declare-before-use is the scoping rule.

```
{int b, {float b, b = 3}, int a, b = a}
```

In this example the use of `b` in the assignment `b = a` falls only in the scope of the declaration `int b`. The `b` in the assignment `b = 3`, on the other hand, is in the scope of two declarations for `b`. Of course, it is the inner declaration that governs the inner use of `b`. This overlapping of declaration scopes is central to semantic checking.

### 8.1.3 Semantic Rules

A major issue for a language designer is to define how identifier attributes are to be interpreted in a program. We have already introduced ideas such as *block* and the *scope of a declaration*, but rules for their use must be specified. There are a few general principles that govern how declaration scopes and identifier uses are associated and they hinge not only on the specific declaration scope strategy (define-before-use versus use-anywhere) but on the fact that blocks can be nested and, consequently, that the scope of a declaration can cover or include a nested block as well. The list of rules in Figure 8.1 defines more formally the general scoping rules for block structured languages; in particular, the rules indicate how to resolve the problem of overlapping declaration scopes.

Another important characteristic of scoping is that it is dynamic – that is, the set of identifier/attribute bindings currently in scope changes as a syntax tree is traversed, with the set currently in scope corresponding to the current set of nested blocks (and declaration scopes). This dynamic nature suggests the following alternative (but equivalent) definition for declaration scope.

#### Scope of a Declaration

The *scope of a declaration* is the segment of program code within which the attribute bindings of the declaration are *maintained*.



1. Two declarations for the same identifier cannot appear as entries at the same level in a block.
2. The scope of a declaration includes nested blocks (nested levels) that are covered by the scope – notice this refers to blocks nested within nested blocks as well.
3. Every use of an identifier must fall within the scope of a declaration for that identifier.
4. The declaration determining the type of an identifier use is the declaration for that identifier with the most deeply nested scope containing that use – this also corresponds to the closest declaration.

Figure 8.1: General Scoping Rules

### 8.1.4 The Symbol Table and Semantic Checking

The problem of overlapping declaration scopes, discussed in the last two sections, illustrates that if we are to implement semantic checking then care must be taken to insure that uses of identifiers are correctly associated with declarations. It is the job of the symbol table to keep track of identifier attribute bindings and to do so in a way that will allow the semantic checker to solve the overlapping declaration-scope problem we just talked about.

The symbol table can be thought of as an object (lets call it `symbolTable`) with two operations: `addBinding` and `findBinding`. Imagine doing a traversal of a PDef syntax tree. If we come upon a declaration node for `'float a'` then we should do something like

```
symbolTable.addBinding(TypeT.FloatT, "a")
```

where we assume it is up to the symbol table to organize the binding `<float,a>` according to the PDef declaration scoping rules.

On the other hand, what if we come across the assignment node `'a = b'`? One thing we should do is to check that the two identifiers have been properly declared and we should retrieve the type bound to each – something like this.

```
type4a = symbolTable.findBinding(a);
type4b = symbolTable.findBinding(b);
// now check that type4b is assignment compatible with type4a
```

In this way we can determine if the assignment statement is semantically correct – if it is not, then we can throw an exception to signal an error.

There is a subtle but critical problem lurking here in the relationship between the symbol table and semantic checker and it deals with when we can add bindings to the symbol table and when we can retrieve an identifier's attributes from the symbol table. If we use use-anywhere scoping, then

the symbol table must be created first (i.e., traverse the syntax tree adding entries to the symbol table as they are encountered) before semantic checking can be carried out. This is because an identifier might be used before it is declared and to check it is used correctly we must be able to retrieve the attributes bound to it – this only works if all identifier bindings have been added to the symbol table before semantic checking.

On the other hand, if we use define-before-use then we must not only add bindings to the symbol table as they are encountered, but we must also check semantics of identifier uses during the same traversal of the syntax tree. If we waited for a second pass then we couldn't tell if an identifier use occurred before or after its declaration. If we do symbol table building and semantic checking on the same traversal, when we encounter the use of an identifier and look it up in the symbol table, we will know if its declaration has been processed because there should be an entry for the identifier in the symbol table.

It is obviously critical that the symbol table be designed so that it correctly represents the declaration scoping rules of the target language and so that it can be used in semantic checking as just described. In the next section we address these design issues for the symbol table.

## 8.2 Symbol Table Structure

What we want to understand is how a symbol table structure can be defined so that the scoping rules described above can be easily evaluated. There are two issues that will drive the symbol table design: that blocks contain declarations and that blocks can be nested. Consequently, at a particular identifier use, the part of the symbol table needed to identify the binding should be readily available. We will begin this section with an example from PDef and let the example motivate our design of the symbol table.

### 8.2.1 An Example

In advance of our example we need to specify two characteristics of PDef scoping – what constitutes a PDef block and within a block what constitutes the scope of a declaration.

- A PDef block is the entire sequence of entries in a list – i.e., what is generated by the PDef non-terminal **Block** – notice the choice of the name **Block** is not accidental.
- PDef scoping follows the define-before-use rule.

With these definitions we can define the scoping rules for PDef. With the definition of block for PDef and the specification of how scoping is done in PDef we can now see how the general scoping rules in Figure 8.1 apply to PDef. The diagram in Figure 8.2 illustrates these rules using a simple PDef program. In the diagram an arrow points from each declaration to the horizontal line covering its scope. Notice in the diagram that the scopes for the declarations `int b` and `int a` overlap but are not the same – this is because PDef uses define-before-use scoping.

Another important point becomes apparent in the diagram in Figure 8.2. Focus on the assignment entry '`a = b`', which appears in the nested list. If we think about the uses of the two identifiers in that assignment, we can see that both uses fall in the scopes of two declarations (`int b` and `float b`), but also that neither scope is for a declaration of `a`. This situation is covered by

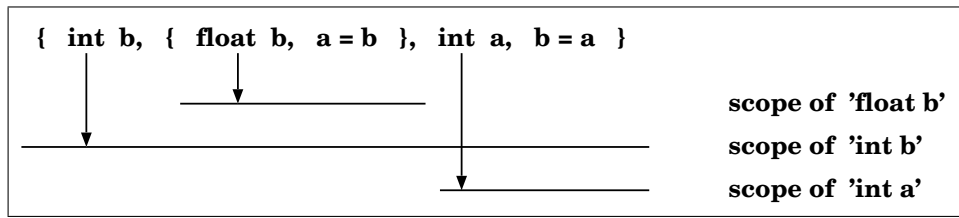


Figure 8.2: The Scope of Declarations for PDef

rule 4 in Figure 8.1 which would indicate that the declaration governing the use of this use of `b` is the declaration ‘float `b`’, which appears in the “most deeply nested scope containing the use.”

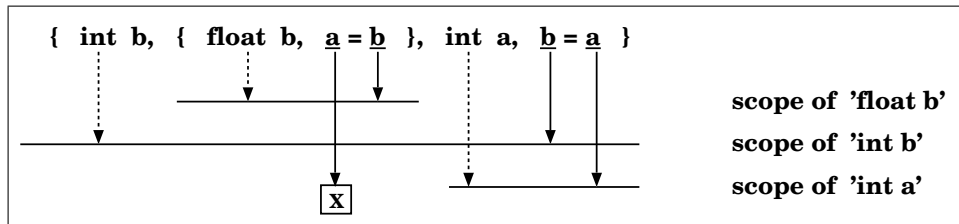


Figure 8.3: Determining Identifier Attributes

If we measure distance from a use to a declaration in terms of number of nesting levels, then for the use of `b` we have been discussing the distance to the scope of ‘int `b`’ is 1, while the distance to the scope of ‘float `b`’ is 0. The diagram in Figure 8.3 shows for each use of an identifier, which declaration determines its attributes; an arrow points from each use to the appropriate declaration scope. In the case of the first use of `a` we see in the diagram that the arrow hits the scope of no declaration for `a` – we indicate this by the box at the end of the box from `a`. In this case, the first use of `a` fails rule 3 in Figure 8.1 – a semantic error should result in this case.

#### Activity 14 –

It is interesting that we can define a different version of PDef (allowing use before declaration) by simply changing the first scope rule so that the scope of a declaration covers the entire block containing the declaration. For each of the following PDef programs

1. { float a, {int b, {float b, a = b}, int a, b = a}, int b, a = b }
2. { int a, {{float b, a = b}, { int a, int c, b = c }, int a, b = a}, int b, a = b }

draw two diagrams – the first based on define-before-use scoping and the second for use-anywhere scoping. In each diagram underline the scope of each declaration and draw an arrow from each identifier use to its governing declaration. If there is none, put a box at the end of the arrow, indicating a semantic error – see Figure 8.3.

### 8.2.2 Structuring a Symbol Table

So the question is “How do we structure and construct a symbol table?” The discussion in Section 8.1.4 and the examples we examined in the previous section emphasize the importance of blocks and their nesting. The following strategy seems to follow naturally: when we see the use of an identifier we should first check if it has been declared in the current block, if not then check in the containing block (one nesting level out), continuing this search pattern until we find a declaration or we fail to find a declaration. Since we have to search for declarations in each block it makes sense to build a local symbol table for each block. We can construct a local symbol table as a list of the name/attribute pairs determined by each declaration. Think about this in the context of a generated syntax tree.

Another design detail is critical: since searching the whole symbol table progresses from inner tables to outer ones, it would be good for one table to carry a link to its “parent” table one nesting level out. Figure 8.4 shows this symbol table structure for our example PDef program. The diagram shows the structure assuming all declarations have been processed.

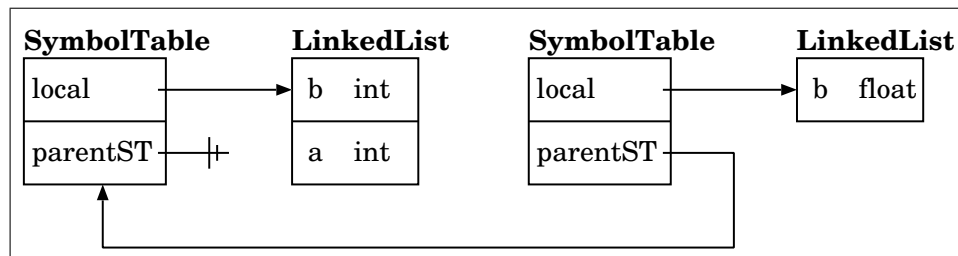


Figure 8.4: A Symbol Table Structure for  $\{\text{int } b, \{\text{float } b, a = b\}, \text{int } a, b = a\}$

#### Activity 15 –

Draw the complete symbol table for the two PDef programs in Activity 14, following the structure illustrated in Figure 8.4.

There’s a curious thing about the symbol table structure – it is an upside-down tree and since the tree can have many leaves, there is no single access point that guarantees access to the entire tree (this would be true only if there is a single leaf). So how can we make use of it? The answer can be seen if we think about how the symbol table will be used in semantic checking. Semantic checking will progress as a depth-first traversal of the syntax tree. When an identifier use is encountered it will lookup the identifier in the local symbol table for the current list – if it is not declared at that level it must look in the parent symbol table (up one nesting level). Since the symbol table must be accessed during a traversal of the syntax tree, it makes sense to integrate the local symbol table into the structure of the syntax tree. In PDef the non-terminal **Block** corresponds to the PDef block, we can put a reference to the local symbol table as an instance variable in the syntax tree class corresponding to the non-terminal **Block**.

Figure 8.5 illustrates how this works in the context of the example above. We will talk through the traversal of the syntax tree and the symbol table building that is carried out. Here is a list

of stops we make during the traversal – notice that in Figure 8.5 the number of each step also appears by the corresponding tree node. Also, the symbol table representations run across the page. Vertically, from the top, we have the symbol table as it appears through the traversal, with a new copy drawn each time a binding is added.

1. We begin at the top of the tree with the left-most **BlockST** node, which contains a reference to a list of statements and a reference (the dashed line) to an empty symbol table (with a null parent reference). The node contains the local symbol table – **LinkedList** – which has four elements.
2. Following the traversal, we next visit the declaration ‘**int b**’ labeled **2**. We add this binding to the local symbol table, which is redrawn and labeled **after 2**. Notice that this is the beginning of the scope of the declaration ‘**int b**’. Also note that the dashed arrow links the root block to its local symbol table.
3. We go back to the list and find the next element, which is a nested block. In visiting the new **BlockST** (labeled **after 3**) we first create an empty local symbol table, with the outer symbol table as its parent. Also note that the dashed arrow links the block to its local symbol table and the local symbol table has a link to the root’s local symbol table. Now we move to the first of two nested list items.
4. We are at the declaration ‘**float b**’ labeled **4**. We add this binding to the local symbol table – the symbol table entry is labeled **after 4** below the syntax tree (corresponds to the number **4** on this declaration node). This is the beginning of the scope of the declaration ‘**float b**’. We return to the nested list (at **3**) and move to the node labeled **5**.
5. We are at the assignment node labeled **5**. Here we need to check that the identifiers ‘**a**’ and ‘**b**’ are declared and determine if their declared types are assignment compatible. We look in the local symbol table and don’t find ‘**a**’ so we look in the parent symbol table and get the same result. Since the parent symbol table is now **null** we conclude ‘**a**’ is not declared. So this node fails the semantic check. For fun we will keep going. We return to block labeled **3** and see we have exhausted its list. We go back to the block labeled **1** and move to its third entry.
6. We are at the declaration node labeled **6**, which is a declaration ‘**int a**’. We add this binding to the root’s local symbol table – look at the symbol table row labeled **after 6**. This is the beginning of the scope of the declaration ‘**int a**’. We move back to the root list and visit the last list element.
7. This is an assignment node labeled **7**. We don’t have to add anything to the local symbol table, but we do have to look up the source and target identifiers. The target ‘**b**’ is found in the local symbol table with type **int**; the source ‘**a**’ is also found in the local symbol table also with type **int**. Since both identifiers are declared and their types are assignment compatible the assignment is correct. We return to the list and have no more elements to visit so we return to the block node labeled **1**, and our traversal terminates.

With the syntax tree and symbol table structures associated, it is not difficult to imagine traversing the syntax tree looking for other uses of identifiers. When an assignment entry is encountered, the semantic checker can look in the local symbol table associated with the assignment entry’s list;

if the search fails then the parent reference can be followed and the search repeated one level up. Notice that, according to our earlier discussion, the first binding we find for an identifier will match our notion of “closest” declaration. One more thing to gather from this excursion is that when we visit a declaration or assignment node we seem to have access to the local symbol table. Apparently when we write our semantic checking traversal, the local symbol table will have to be passed along as the traversal proceeds.

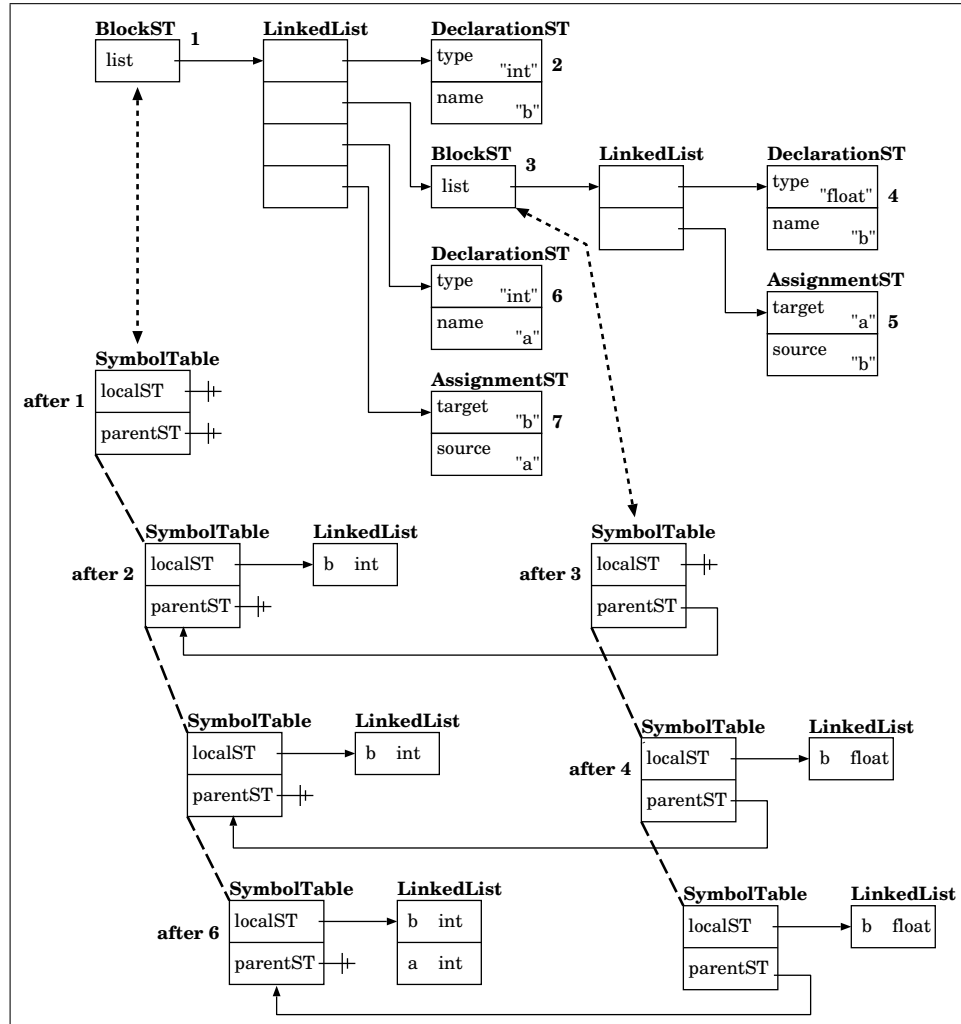


Figure 8.5: Symbol Table Growth Linked into Syntax Tree (see Figure 8.4)

### 8.2.3 Symbol Table Implementation

A production level compiler, such as for Java or C++, will be designed to work effectively on large software systems. For such compilers the amount of time spent adding entries to the symbol table and searching the symbol table can be considerable – so the Efficiency Principle must be considered (see Section 1.2.1, page 7).

There are two levels for the implementation of the symbol table. First, remember that for PDef we want to modify the syntax tree class **Block** as follows.

```

public class Block extends SyntaxTree {
    private LinkedList<StmtST> list = null;
    private SymbolTable localST = null;
    public Block(LinkedList<StmtST> l, SymbolTable parent) {
        list = l;
        localST = new SymbolTable(parent);
    }
}

```

In addition, a class template for `SymbolTable` can be described as follows – notice that the description, along with the definition of `BlockST` above, makes explicit how the parent of the symbol table is established and used.

```

public class SymbolTable {
    private LocalSymbolTable localST = null;
    private SymbolTable parentST = null;
    public SymbolTable(SymbolTable p) {
        localST = new LocalSymbolTable();
        parentST = p;
    }
    public void addBinding(String name, /* attributes */)
        // add the bindings to localST
    public ... findLocalBinding(String name)
        // search for a binding for name in localST
    public ... findBinding(String name)
        // search localST for name -- if not there search parentST
}

```

Notice that when a binding is added to the symbol table it is always added to `localST`. Also, there are two mechanisms for searching for the symbol table – `localST` can be searched specifically, or the search can be directed upward in the symbol table by searching `parentST`.

One implication of the design for `BlockST` in PDef is that the block structure for any block structured language should correspond to a syntax tree class – each block class can be similarly modified.

The second level of symbol table implementation is represented by the class name `LocalSymbolTable` above. The class can take various forms depending on the implementation of the structure that will hold the name/attribute bindings. The simple way to implement `LocalSymbolTable` is to use the standard list structure (`LinkedList` in Java, for example). While additions to such a structure run in constant time, symbol lookups would run in linear time – if the list is maintained in sorted order then additions are linear while lookups are  $\log_2(n)$ . But to see real performance gains, as would be required in a production level compiler, it will be best to implement `LocalSymbolTable` as a hash table.

### ☛ Activity 16 –

Work through the first half of the PDef Semantics Tutorial in Chapter 15 (Sections 15.1-15.4). Before starting be sure to review Tutorial Chapters 11-14.

---

### 8.3 Semantic Checking

Semantic checking is accomplished via depth first traversals of the syntax tree. There are two crucial activities, as discussed above, that take place during semantic checking, one checks that identifiers are properly declared, and the other checks that identifiers are used appropriately. Being more specific, there are four actions that must be performed during semantic checking.

1. For each declaration encountered, check that the target identifier is not already declared in the same block.
2. For each declaration encountered, add an appropriate entry for the target identifier into the current block's local symbol table.
3. For each identifier use encountered, check that it falls in the scope of a declaration.
4. For each identifier use encountered, check that the identifier is used in a way consistent with the attributes bound to the identifier by the governing declaration.

Figure 8.6: Actions in Semantic Checking

The idea behind the integration of the symbol table into the syntax tree, as hinted in Figure 8.5, is to make it possible to easily implement semantic checking via syntax tree traversals. Remember, however, that the declaration scope strategy, either define-before-use or use-anywhere, has an important impact on the implementation.

#### **Define-before-use:**

Actions numbered 1-4 are carried out in a single traversal. This means when an identifier use is checked for consistency, the symbol table will be only partly complete. However, if the identifier is properly declared (before the use) the declaration will already be represented in the symbol table.

#### **Use-anywhere:**

Actions numbered 1 and 2 must be completed in a single traversal before actions 3 and 4 can be carried out in a second traversal. This means the first traversal builds the symbol table and the second traversal checks the consistency of identifier uses.

#### 8.3.1 Semantic Checking for Literals

There is an important but easily forgotten aspect of semantic checking (we mentioned this earlier in Section 8.1.1). Identifier uses that occur in expressions will be processed according to action 4 above. But another kind of token that can appear in an expression is a literal. In PDef expressions both integer and floating point literals can appear. So in addition to insuring that identifiers are used consistently with their attributes, it is critical that semantic checking insure that literals are used



consistently with their attributes as well. This checking will not involve the symbol table, of course, since the translator will be able to determine the attributes of a literal when it is encountered.

### 8.3.2 Semantic Checking for PDef

At this point we should be able to design a semantic checking syntax tree traversal for PDef. We know that PDef uses define-before-use scoping, so we know we need a single semantic checking traversal algorithm. We also have the description of the PDef semantics – see Section 4.3, page 61, so that appropriate actions can be described. Rather than laying out a complete implementation of the semantic checker for PDef here we will leave that to the semantic checking tutorial in Chapter 15. However, we will illustrate the process by focusing the design of the semantic checking traversal on three syntax tree classes: `BlockST`, `DeclarationST`, and `AssignmentST`. It will be useful to look back at Figure 8.5 in order to view the semantic checking traversal from the point of view the declaration and assignment nodes.

**Step 1:** The first step is to choose a name for the traversal methods to be added to the classes in the syntax tree hierarchy – we will use the name `checkSemantics`. It will be important in the analysis in Step 2 to watch for situations where the method `checkSemantics` needs parameters to function properly.

**Step 2:** We need to look at each of our three example classes to determine the action that must be carried out when nodes are visited by calls to `checkSemantics`.

#### `BlockST`:

There are two things to do on arrival at a `BlockST` node. First, the local symbol table must be initialized. In order to initialize the local symbol table there must be a parent symbol table reference to pass to the constructor – this must be passed as a parameter to the call to `checkSemantics`. So the call to `BlockST.checkSemantics` must have a symbol table reference as parameter.

Second, we must drive the semantic checking down the syntax tree by calling `checkSemantics` on each reference on the list of statements (which is a data member of this node). The statement list can hold references to three kinds of statements: declaration, assignment, and block.

We also notice, here, that since the local symbol table is ‘local’ to `BlockST`, there will be no access to this symbol table in the list’s statement nodes – but those statement nodes will need access to this local symbol table. So the local symbol table must be passed as a parameter to the `checkSemantics` calls.

#### `DeclarationST`:

When `checkSemantics` is called on this node it will carry the local symbol table as an argument.

This node contains two data members, a type token and an identifier token. There are two things that need to be done by `checkSemantics` at this node. First, it must check that the identifier doesn’t have an entry in the local symbol table, and second, it must add a new entry to the local symbol table – that covers the semantic checking actions 1 and 2 listed above.

If we assume that the symbol table method `addBinding` will throw an exception if there is already an entry in the local symbol table, then both of these actions will be accomplished by calling `addBinding(name, type)`.

#### AssignmentST:

When `checkSemantics` is called on this node it will carry the local symbol table as an argument.

This node also contains two data members – one is an identifier token which is the target of the assignment, while the other is a reference to an expression node. At this node `checkSemantics` has two responsibilities: check that the target identifier is in the symbol table at some level (and retrieve the type bound by the symbol table to the identifier) and check that the type of the expression is assignment compatible with the type of the target identifier. The way to determine the type of the expression is to call `checkSemantics` on the expression reference and expect to get the expression type as the return value from the call.

There are two things we have discovered here: the method `checkSemantics` should be defined in all syntax tree classes with a parameter of type `SymbolTable`, and for classes in the expression hierarchy the method `checkSemantics` must also return a value of type `Token`, so the determined type value of the expression can be returned.

**Step 3:** We now revisit the three classes to determine the algorithms appropriate for the `checkSemantics` versions. For each, we present the class definition including the implementation of `checkSemantics` for the class. It should be clear that each method implementation matches the description above in Step 2. Because error handling can change from implementation to implementation, the implementations below ignore the error handling. We assume that the method `SymbolTable.addBinding` will throw an exception if there is already an entry for the identifier in the local symbol table; we also assume the same sort of exception throwing in the methods `SymbolTable.findLocalBinding` and `AssignmentST.checkAssignmentCompatible`.

#### BlockST:

```
public class BlockST extends SyntaxTree
    private LinkedList<StmtST> list;
    private SymbolTable localST;
    public DeclarationST(LinkedList<StmtST> l)
    { list = l; }

    public void checkSemantics(SymbolTable st) {
        localST = new SymbolTable(st);
        for (Stmt entry: list)
            entry.checkSemantics(localST);
    }
}
```

#### DeclarationST:

```
public class DeclarationST extends SyntaxTree
    private Token name;
    private Token type;
```

```
public DeclarationST(Token n, Token t)
{ name = n; type = t; }

public void checkSemantics(SymbolTable st) {
    st.addBinding(name, type);
}
}
```

AssignmentST:

```
public class AssignmentST extends SyntaxTree
private Token target;
private Expression source;
public DeclarationST(Token t, Expression e)
{ target = t; source = e; }

public void checkSemantics(SymbolTable st) {
    Token leftType = st.findLocalBinding(target);
    Token rightType = source.checkSemantics(st);
    checkAssignmentCompatible(leftType, rightType);
}
}
```

### Activity 17 –

Work through the second half of the PDef Semantics Tutorial in Chapter 15. Before embarking on the completion of the tutorial be sure to review the Tutorial Overview, Chapter 11, and review your work from the first half of the Semantics Tutorial.

---



## Chapter 9

# LR Parsing – Theory and Practice

In Chapter 6 we discussed the basic strategies involved in two parsing strategies: top-down (predictive or LL) parsing and bottom-up (shift/reduce or LR) parsing. At that time we also discussed in some detail a standard top-down implementation strategy called recursive descent parsing, which was used to implement the parser in Chapter 13 of the Front-end Tutorial. In this chapter we will study a second parser implementation strategy referred to as *table driven parsing*, which is the dominant strategy for implementing bottom-up parsers.

### 9.1 Understanding LR Parsing

The discussion of LR parsing<sup>1</sup> in Chapter 6 took a quite simplistic view – the aim having been to lay out the most basic elements of the strategy. In this section we will study additional small example grammars, fragments drawn from the PDef grammar, in order to expose in more detail what happens during LR parsing.

We will base our examples on a slightly simplified and abbreviated version of the PDef-*lite* grammar, which is of course a subset of the PDef grammar. The grammar has been annotated to indicate how non-terminal names have changed. The fragments will be introduced in each of the subsections which follow.

0	P	→	SL \$	P replaces Program
1	SL	→	lbT L rbT	SL replaces StatementList
2	L	→	L commaT S	L replaces List
3	L	→	S	
4	S	→	A   D   SL	S replaces Statement
5	A	→	identT assignT identT	A replaces Assignment
6	D	→	typeT identT	D replaces Declaration

You are probably also wondering about the symbol ‘\$’ at the end of rule 0. It has become traditional in compiling circles to add ‘\$’ in this way to explicitly represent the end of the token string – in fact, the PDef tokenizer returns `eotT` (end of tokens token) for just this reason. From this point we will follow the traditional notation, as illustrated in the grammar above.

---

<sup>1</sup>We will use this more traditional name from this point on.

**PDef fragment 1**

Our first PDef fragment describes a language that allows lists of a single statement, which must be an assignment statement. Here is the fragment we will use.

```

0 P  → SL $
1 SL → lbT S rbT
2 S  → A
3 A  → identT assignT identT

```

This grammar could be made simpler, but keeping some of the structure (in this case rules 2 and 3) will be helpful in seeing the connection from one fragment to another. The strings that can be generated by this grammar are very simple – all have the form

{ a = b }

The names of the identifiers can change but otherwise this is the form.

In Chapter 6 we worked through an example very similar to this one using both LL (see page 75) and LR (see page 79) strategies. The LR strategy centered on two operations, *shift* and *reduce*, used to carry out a kind of rewriting of the target string. Here is a rendering of that process for the current grammar and target string '{ a = b }'. There are two things to notice: first we have put a '\$' to mark the left end of the string left of the up arrow; second, notice that we have characters in the target string which, when *shifted* are represented as tokens

Action	Stack (top at the right) ↑	Target (after operation)
		\$ ↑ { a = b } \$
shift lbT		\$ lbT ↑ a = b } \$
shift identT	\$ lbT identT	↑ = b } \$
shift assignT	\$ lbT identT assignT	↑ b } \$
shift identT	\$ lbT identT assignT identT	↑ } \$
reduce rule 3	\$ lbT A	↑ } \$
reduce rule 2	\$ lbT S	↑ } \$
shift rbT	\$ lbT S rbT	↑ \$
reduce rule 1	\$ SL	↑ \$
reduce rule 0	\$ P	↑ \$
accept		

Using this example as basis we will make a slight course correction relative to the way we talk about an LR parsing. If we focus on the way the parsing manipulates the string to the right of the up arrow it becomes clear that that string is treated as a stack – a shift operation pushes a token on top of the stack and a reduce pops a substring at the top of the stack and pushes a non-terminal in its place. So in the future will refer to this string as the stack.

We will also expand slightly our use of the term *handle*, which we used in describing how the *reduce* operation is carried out (see 79). We will use the term handle to generally mean a substring in one of two contexts. A *stack handle* will refer to a substring beginning in the middle (or bottom) of the stack and continuing to the top of the stack. We will also use the term *handle* to refer to an

initial substring of the right hand side of a grammar rule. So in these terms we can restate the LR parsing process as one in which we continually reduce appropriate stack handles to corresponding defining non-terminals – and if no reduction is possible we shift the head of the target string to the stack. Notice that reductions are done as long as possible and, if a reduction is not possible, then shifts are performed as long as necessary (until there is a reduction). Here is the process in algorithmic form:

```
repeat
    if a stack handle matches a rule's right hand side
        then reduce
        else shift the head of the target string to the stack
until stack == "$ P" AND target string is "$" (i.e., empty)
```

## PDef fragment 2

In LL parsing problems occur when there is a non-terminal in the grammar with multiple rules – we use the character(s) at the head of the target (the lookahead) to resolve the conflict. The corresponding problem in LR parsing occurs when the stack handle matches a grammar rule's right hand side and appears as a handle of one or more other rules (even if the rules have different defining non-terminals).

The second fragment we will look at describes nested structures. Since nesting characterizes context free grammars, it is important to see how LR parsing deals with it. Here is the fragment we will use.

```
0 P → SL $
1 SL → lbT S rbT
2 S → D
3 S → SL
4 D → typeT identT
```

Notice that grammar rules 0 and 3 have the same right hand side, which means in an LR parsing when SL appears at the top of the stack, there will have to be some mechanism to resolve which rule to reduce.

To see how the nested structure is parsed we apply the LR parsing strategy to the target string "{ { int a } }".

Action	Stack	↑	Target
		\$	{ { int a } } \$
shift lbT	\$ lbT	↑	{ int a } } \$
shift lbT	\$ lbT lbT	↑	int a } } \$
shift typeT	\$ lbT lbT typeT	↑	a } } \$
shift identT	\$ lbT lbT typeT identT	↑	} } \$
reduce rule 4	\$ lbT lbT D	↑	} } \$
reduce rule 2	\$ lbT lbT S	↑	} } \$
shift rbT	\$ lbT lbT S rbT	↑	} \$
reduce rule 1	\$ lbT SL	↑	} \$

Since lbT is below the stack handle SL, we must reduce rule 3, leaving S on the stack.

reduce rule 3	\$ lbT S	↑	} \$
shift rbT	\$ lbT S rbT	↑	\$
reduce rule 1	\$ SL	↑	\$
reduce rule 0	\$ P	↑	\$
accept			

What we learn from this example is that when a handle has two possible reductions, they can be discriminated by understanding what appears below the handle on the stack. In our example the lbT below SL on the stack indicates that we are in the process of parsing the right hand side of rule 1. In order to complete that parsing it will be necessary to reduce rule 1, then shift the ‘}’ before we can reduce rule 1 to SL. So the apparent reduction conflict between rules 0 and 3 can be handled by information already on the stack – no lookahead reference to the target string is necessary. This means fragment 2 is LL(0).

### PDef fragment 3

Our final fragment focuses on the comma separated list of statements, as in the following fragment of the PDef-*lite* grammar. You will notice that this fragment represents the list in the right-recursive form. We use this form because it is more interesting in this context than the left-recursive form.

```

0 P → L $
1 L → S commaT L
2 L → S
3 S → D
4 D → typeT identT

```

We have expanded the PDef Rule 2 into two separate rules (1 and 2) since each represents a separate rule. This time we will process a longer input string.

```
int a, int b, float c
```

In anticipating the LR parsing, we see that there are two grammar rules (1 and 2) which start with the same handle – and for rule 2 that common handle is the complete right hand side. During the



parsing it would appear that this conflict will materialize as a conflict between reducing rule 2 or shifting a comma (to follow the *S* on the stack). We will watch to see how this conflict materializes during the parse that follows.

Action	Stack	↑	Target
		\$	int a , int b , float c \$
shift typeT		\$ typeT	↑ a , int b , float c \$
shift identT	\$ typeT	identT	↑ , int b , float c \$
reduce rule 4		\$ D	↑ , int b , float c \$
reduce rule 3		\$ S	↑ , int b , float c \$

Having gotten to this point we see a problem: there are two rules whose right hand sides start with *S*. So should we shift or reduce? What happens if we reduce by rule 2. We will have *L* on the stack with no reducible handle, so the comma on the target string will be shifted. This the string *L commaT*. But now we have *L* on top of the stack followed by *commaT* which we cannot allow – *commaT*  $\notin$  *follow(L)*. So we must do the shift at this point.

shift commaT		\$ S commaT	↑ int b , float c \$
shift typeT		\$ S commaT typeT	↑ b , float c \$
shift identT	\$ S commaT	typeT identT	↑ , float c \$
reduce rule 4		\$ S commaT D	↑ , float c \$
reduce rule 3		\$ S commaT S	↑ , float c \$

This is the same conflict we resolved above – so we shift.

shift commaT		\$ S commaT S commaT	↑ float c \$
shift typeT		\$ S commaT S commaT typeT	↑ c \$
shift identT	\$ S commaT	S commaT typeT identT	↑ $\epsilon$
reduce rule 4		\$ S commaT S commaT D	↑ \$
reduce rule 3		\$ S commaT S commaT S	↑ \$

Now we have *S* on the stack and the target string is empty. Since *S* is the only handle and there are no characters to shift, we must reduce rule 2 at this point.

reduce rule 2		\$ S commaT S commaT L	↑ \$
reduce rule 1		\$ S commaT L	↑ \$
reduce rule 1		\$ L	↑ \$
reduce rule 0		\$ P	↑ \$
accept			

## 9.2 A Machine for LR Parsing

We are confronted with a parsing strategy which we can apply with paper and pencil, but whose efficient implementation is not clear. What appears to be difficult is the identification of reducible handles. The shift operation doesn't seem to be a problem, but when to shift and what to reduce hinges on the ability to identify reducible handles. Fortunately there is a clever use of finite state machines that facilitates the implementation of LR parsers. In the rest of this chapter we will investigate this state-based implementation for LR parsers.

### 9.2.1 Transition Graph for an LR Parser

Our implementation will be centered on a finite state transition graph that we can construct from a context free grammar. The transition graph is structured to represent progress through the parsing process with the transitions indicating that some symbol has been seen. In this way the transition graph seems to be a finite state machine designed for LR Parsing.

The design of the transition graph is based on a kind of decorated grammar rule. We call such a decorated rule a *dotted form* and their form and interpretation is described here. We consider the PDef grammar rule 1 ( $SL \rightarrow lbT L rbT$ ).

We use a dot character on the right hand side of a rule to indicate the handle to the left of the dot has been matched to this point in the parsing process. For example, the following dotted form for rule 1

$$SL \rightarrow lbT L \cdot rbT$$

represents progress through a parse where 'lbT L' appears as the stack handle and the character '{' has not yet been shifted onto the stack. When the dotted form appears with the dot at the end, as in

$$SL \rightarrow lbT L rbT \cdot$$

we call it a *terminal dotted form*, which represents a handle that can be reduced. We use the term *left-most dotted form* in case the dot appears at the left-end of the rule's right hand side, in this case it appears as follows.

$$SL \rightarrow \cdot lbT L rbT$$

So for this grammar rule the following list has all possible dotted forms.

- a)  $SL \rightarrow \cdot lbT \cdot L rbT$
- b)  $SL \rightarrow lbT \cdot L rbT$
- c)  $SL \rightarrow lbT L \cdot rbT$
- d)  $SL \rightarrow lbT L rbT \cdot$

There is an interesting thing to point out in the sequence of dotted forms above. If we think of animating the parsing process then we would expect that when we *shift* a '{', then we would move from dotted form (a) to (b). But if the dot is before the L what does it mean to move the dotted form (c)? Since moving the dot should mean we have seen an L, it would seem we must shift our

attention to the L rules, starting with a left-most dotted form and continuing until we arrive at a terminal dotted form. At that point we can say we've seen an L and can return to the original form and move the dot after the L.

From this discussion you may have guessed that when a dot passes a token then we do a shift operation; when the dot passes an L, that means that we have seen all the right-hand side of one of L's rules, so a reduction is necessary.

### The Transition Graph for PDef Fragment 1

Now the fun construction starts. We will focus on the PDef fragment 1 grammar for our first example. We proceed in a top-down fashion creating sets of dotted forms – these sets will become the states of our transition graph. The set formation process has two phases. The first phase is to add certain initial dotted forms and the second phase is to form the *closure* of that set. The closure comes about by ensuring that if the dot appears immediately before a non-terminal, say X, in some dotted form, then we add to the set all initially dotted forms for rules with X as the left-hand side. That sounds complicated, but the example will illustrate the two phases repeatedly in the formation of the nine states of the fragment 1 transition graph.

1. The first set is always started in the same way: write down the first rule in its left-most dotted form.

$$P \rightarrow \cdot SL \$$$

We attach the \$ token to emphasize the requirement that we see the end of input before we can reduce to P.

Now we form the closure of this singleton set. Because the form we started with has its dot immediately before a non-terminal, SL, we add all left-most dotted forms of rules with that non-terminal on the left-hand side. So our list of dotted forms is as follows.

$$\begin{aligned} P &\rightarrow \cdot SL \$ \\ SL &\rightarrow \cdot lbT S rbT \end{aligned}$$

There are no other non-terminals preceded by a dot, so we have our closure and the first set.

2. We will now focus on the first set. We look at the right-hand sides of the forms and look at all the symbols immediately following a dot. We select one of these symbols (it doesn't matter which) and create a new set containing all the dotted forms from the first set with the chosen symbol after the dot. But in each form in the new set we move the dot to the position after the chosen symbol. For the current set we will choose SL. This means we should create the new set with the following form.

$$P \rightarrow SL \cdot \$$$

The next phase is to form the closure of the new set, but since the one form in the set has no non-terminal after the dot, the set is its own closure.

3. We continue in this way... We look again at the first set and choose the only other symbol following a dot – the lbT token in the second form. So we initialize the third set by adding the second form with the dot advanced one position – as follows.

$$SL \rightarrow lbT . S rbT$$

For the closure, notice that the non-terminal  $S$  has a dot before it. So we add left-most dotted form for any  $S$  rule – we add the following form.

$$S \rightarrow . A$$

But this new form also has the non-terminal  $A$  after the dot, so we add the left-most dotted form for the one  $A$  rule. So the closure of the our set is as follows.

$$\begin{aligned} SL &\rightarrow lbT . S rbT \\ S &\rightarrow . A \\ A &\rightarrow . identT assignT identT \end{aligned}$$

- We have now exhausted the possibilities from our first set, so we turn to the second set we constructed. The second set has a single form and the dot is before the  $\$$  token. We could create a new set with the dot advanced passed the  $\$$ , but the resulting set would be unnecessary – so we don't form any sets based on the second.

But the third set has several possibilities. We proceed as we did for the first set.

For the new set we will choose the non-terminal  $S$ , which means our initial set will contain the following.

$$SL \rightarrow lbT S . rbT$$

Since the dot doesn't precede a non-terminal the set is its own closure.

- We look at the third set again and this time choose the non-terminal  $A$ , which yields the following new set.

$$S \rightarrow A .$$

As in the previous case this set is its own closure.

- We can create one more set from the third set by choosing the  $identT$  token in the third form. A singleton set results which is its own closure.

$$A \rightarrow identT . assignT identT$$

- We continue in this way until there are no new sets to be made. We've exhausted the first three sets so we turn to the fourth. In the fourth set we have one rule and by advancing the dot we get the following singleton set which is its own closure.

$$SL \rightarrow lbT S rbT .$$

- The fifth set has one form and the dot is already at the end, so it can't be used to generate a new set. We turn to the sixth set which has one form. Advancing the dot one position gives us another singleton set which is its own closure.

$$A \rightarrow identT assignT . identT$$

9. Finally, we have just one set to consider – the eighth, which has one form and yields another singleton set which is its own closure.

$$A \rightarrow \text{identT assignT identT} .$$

Now the interesting thing about the construction just completed is that, in talking through it, we have actually described the transition graph for our LR parser. Since each set is to be a state of the machine, we constructed subsequent sets by asking, in essence, if we see a particular symbol in this set, what new set do we go to. The transition graph we have described in in Figure 9.1.

### ☛ Activity 18 –

You are to complete the following partially completed construction of the set of states for PDef fragment 2. When completed, draw out the transition graph following the structure in Figure 9.1.

1. We add to this first set the left-most dotted form of the first rule. But since that form has a dot before the L, we must add the left-most dotted forms for L and any subsequent non-terminals as well. This first set then has the following form.

$$\begin{aligned} P &\rightarrow . L \$ \\ L &\rightarrow . S \text{ commaT } L \\ L &\rightarrow . S \\ S &\rightarrow . D \\ D &\rightarrow . \text{typeT identT} \end{aligned}$$

2. If we see an L in the first set then we get just one new dotted form.

$$P \rightarrow L . \$$$

3. If we see an S in the first set we actually have two forms to advance. But adding those to this set will generate no additional forms, so this set is as follows.

$$\begin{aligned} L &\rightarrow S . \text{ commaT } L \\ L &\rightarrow S . \end{aligned}$$

4. You carry on from here .....

### ☛ Activity 19 –

Draw the transition graph for PDef fragment 3.

## 9.2.2 Relating the Transition Graph to Parsing

When we look at the transition graph in Figure 9.1 one thing sticks out: if we follow the path from  $St1$  to  $St9$ , when we get to  $St9$  we are stuck – there are no transitions from that state. Obviously,

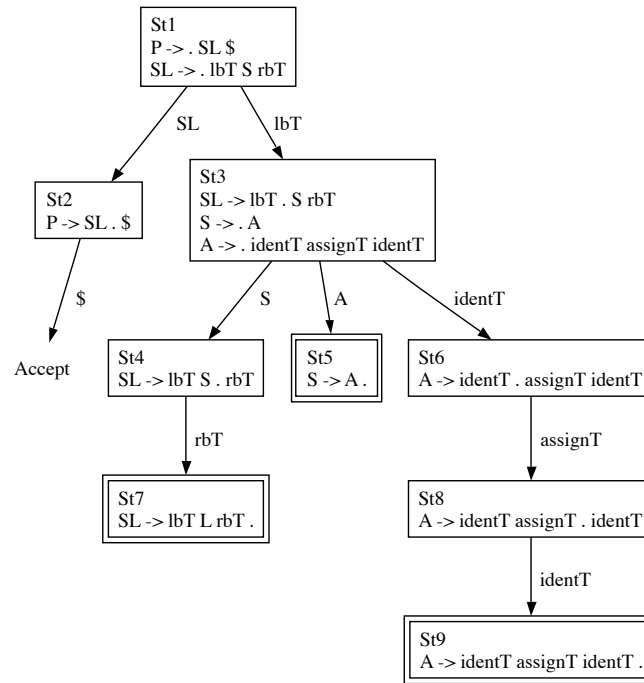


Figure 9.1: Transition Graph for PDef Fragment 1

there is more going on with the transition graph than we have discussed so far. One thing for sure, each transition labeled with a token will be associated with a shift operation. But what about those labeled with a non-terminal? Not surprisingly, they are associated with a reduce operation – but when? how?

The key to understanding how the transition graph relates to reduce operations is understanding the states that have a double-lined boundary. Notice that each of these contains a terminal dotted form. We know, from our earlier discussion, that when the right-hand side of a rule is on the stack then we pop off the elements of that right-hand side and replace them with the non-terminal on the left. If we are in **St9** we know that the top three elements on the stack must be the right-hand side for **A** – this must trigger a reduction. But if we pop the three tokens from the stack then it is as though we were back in **St3** having an **A** on the stack. So the transition from **St9** is actually determined by **St3** and the left-hand side of the rule in **St9**.

But here’s an interesting twist. Suppose there were a transition to **St7** (and thus on to **St9**) from some other state than **St3**. Then we would have to keep track of which state the reduction should return to. We can do this by keeping the state name with the symbol when it is put on the stack. We can see how this is done by going back to the trace of stack use in PDef fragment 1. We begin with  $\$(1)$  on the stack – meaning that we have a mark for the bottom of the stack and the state is **St1**.

Action	Stack	↑	Target
			\$ (1) ↑ { a = b } \$
1 shift lbT			\$ (1) lbT(3) ↑ a = b } \$
2 shift identT			\$ (1) lbT(3) identT(6) ↑ = b } \$
3 shift assignT			\$ (1) lbT(3) identT(6) assignT(7) ↑ b } \$
4 shift identT	\$ (1) lbT(3) identT(6) assignT(7) identT(9)	↑	} \$
5 reduce rule 4			\$ (1) lbT(3) A(5) ↑ } \$
6 reduce rule 3			\$ (1) lbT(3) S(4) ↑ } \$
7 shift rbT	\$ (1) lbT(3) S(4) rbT(7)	↑	\$
8 reduce rule 2			\$ (1) SL(2) ↑ \$
9 shift \$	\$ (1) SL(2) \$	↑	\$
10 accept			

If we follow from the top we can see that, when we did the reduction from step 4 to step 5 we popped off the top three elements from the stack (left in step 4) and then look at the state of the entry on top of the stack — in this case that stack entry is lbT(3) — and then take the transition on A (the reduced non-terminal) from the state St3.

Notice that in the reduction from step 4 to 5 that the lbT tells us nothing, it is the accompanying state which is important. The surprising consequence is that we can eliminate the symbols from the stack entries and use only the state number. In addition, when it comes to a reduction, we don't even have to check the stack content, since the only way to get to a state where there is to be a reduction is to have already pushed the required symbols onto the stack.

### 9.2.3 LR(0) Parse Tables

The LR(0) transition graph discussed in the last section is a useful tool for understanding the parsing mechanism, but the implementation is still not clear – however, the fact that only state names are needed on the stack seems important. In this section we will discuss the construction of the LR(0) parse table, which derives directly from the parse tree, and how it becomes the core of a *table driven parsing* algorithm.

The idea behind the parse table is to collect the information explicitly or implicitly contained in the LR(0) transition graph in a data structure more convenient than the transition graph. A 2-dimensional array (table) is a common data structure for implementing state machines, but, as pointed out above, these transition graphs have missing links that must be accounted for in the table structure. Our basic table structure, then, will use states to label the rows and the grammar's terminal and non-terminal symbols to label the columns (plus the new symbol '\$'). The table will store actions appropriate for each state/symbol combination, based on the transition graph: i.e., *shift actions*, *reduce actions*, and what we will call *goto actions*. Goto actions? We know about the shift and reduce actions from our earlier discussions, but what are "goto actions?"

When a shift takes place it is easy to determine the next state (the one stored with the shifted token). We look at the top entry on the stack and use that state and the token to determine the next state. But what is the next state when a reduce takes place? Remember the example above where we talked about the reduction that takes place from step 4 to step 5. The reduced symbol is the non-terminal A and we determine the next state by looking at the state in the top stack element (after the necessary elements are popped from the stack) and use that state and A to determine the

next state. The state and symbol identify a position in our parse table and the entry that goes into that position is called a "goto action", since it is used to determine the next state based on a state and a non-terminal.

An LR(0) parse table, then, is organized as a matrix, with one row for each state in the LR(0) transition graph and one column for each terminal and non-terminal symbol. The entry associated with a state and a terminal will either be a shift or a reduce operation, while the entry associated with a state and non-terminal will be a goto operation. The production of the table is quite straight forward once the transition graph is determined. We will build the parse tables for each of the three PDef fragments.

### LR(0) parse table for PDef fragment 1

The LR(0) transition graph for fragment 1 is given in Figure 9.1. We build our table one row at a time by focusing on the structure of the states and transitions. We will use the notation  $s5$ ,  $r(2)$ , and  $g(2)$  to denote shift, reduce, and goto actions:  $s5$  denotes a shift and new state 5,  $r(2)$  denotes the reduction of grammar rule number 2, and  $g(2)$  denotes that the new state (after a reduction) is 2. It is important to remember that the reduce notation is different. The action  $r(2)$  tells the parser to determine the length of the right-hand side of rule 2 (so it knows how many elements to pop) and to determine the non-terminal on the left of the same rule – so the appropriate goto action can be found.

#### St1

When we look at **St1** in Figure 9.1 we see that there are two transitions from the state, one on the non-terminal **SL** to state **St2**, a goto action, and the other on the terminal **lbT** to **St3**, a shift action. The appropriate entries are below.

State	Input (token)					Goto		
	identT	assignT	lbT	rbT	\$	SL	S	A
1			s3			g(2)		

#### St2

This state is a special case because the only way to arrive in this state is to have seen a syntactically correct input string (including the **\$** token). So we have a special action **Accept** for this entry.

State	Input (token)					Goto		
	identT	assignT	lbT	rbT	\$	SL	S	A
1			s3			g(2)		
2					Accept			

#### St3 & St4

From state **St3** we have three transitions, two on non-terminals (goto actions) and the other on a terminal (shift action). From **St4** we have one transition on the non-terminal **S**. These add the following lines to the table.



State	Input (token)					Goto		
	identT	assignT	lbT	rbT	\$	SL	S	A
1			s3			g(2)		
2					Accept			
3	s6						g(4)	g(5)
4				s7				

**St5**

State **St5** has the property that its only entry is a dotted form with the dot at the end of the rule. This means that no input should be processed, but that a reduction should take place. The rule to be reduced is the one specified in **St5**. If we look back at the grammar for fragment 1 on page 138, we see that the rule in **St5** is number 3. But in which column should we put the reduce action? Since the reduction will take place regardless of the input token, we put the reduce action in each token column.

State	Input (token)					Goto		
	identT	assignT	lbT	rbT	\$	SL	S	A
1			s3			g(2)		
2					Accept			
3	s6						g(4)	g(5)
4				s7				
5	r(2)	r(2)	r(2)	r(2)	r(2)			

**St6, St7, St8, St9**

The rest of the states fit situations we have already discussed for the earlier states. **St6** and **St8** involve shift operations and **St7** and **St9** involve reduce operations. The final table has the following structure.

State	Input (token)					Goto		
	identT	assignT	lbT	rbT	\$	SL	S	A
1			s3			g(2)		
2					Accept			
3	s6						g(4)	g(5)
4				s7				
5	r(2)	r(2)	r(2)	r(2)	r(2)			
6		s8						
7	r(1)	r(1)	r(1)	r(1)	r(1)			
8	s9							
9	r(3)	r(3)	r(3)	r(3)	r(3)			

**The error action**

During the implementation of the table above we came across a new action (the fourth) in **st2**, the **Accept** action, which indicates that the input string parsed correctly. There is a fifth action that we need to mention – in fact, you may already have spotted it. What if we find the state 3 at the top of stack and the next input token is **assignT**, what do we do then? Well, if **assignT** is the

next token it is a clear error, so perhaps we need an action that indicates an error has occurred. At each position in the LR(0) parse table (for a row and terminal symbol) we should assign the error action if there isn't already a shift or reduce action in place.

### Activity 20 –

Implement the LR(0) parse table for the PDef fragment 2. You will need access to the LR(0) transition graph you implemented in the exercise on page 145. So that you can check your result from that earlier exercise, the required transition graph is in Figure 9.2.

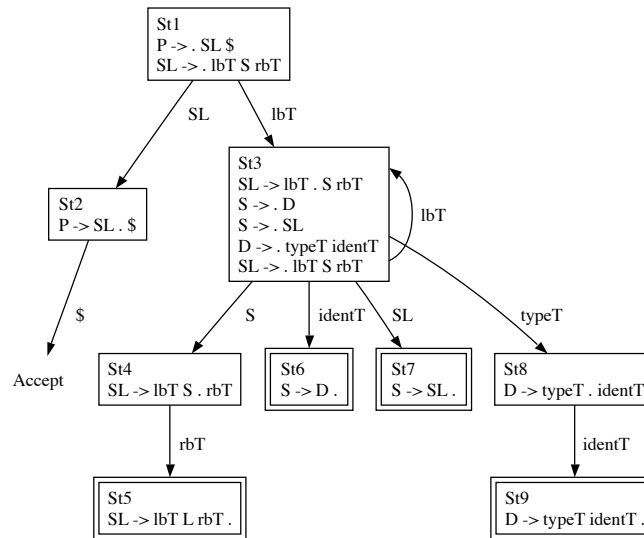


Figure 9.2: Transition Graph for PDef Fragment 2

### LR(0) parse table for PDef fragment 3

The PDef fragments 1 and 2 are simple grammars. Their characteristics are such that our parse table construction process works without problems. But, as we will see in this section, fragment 3 is not quite so well behaved. We will see that the construction of the table generates a conflict. But as we will see the conflict can be resolved by making a slight modification to the grammar.

Before developing the LR(0) parse table for fragment 3 we need the LR(0) transition graph, which was posed as an exercise earlier in this chapter. This is an opportunity to check your work against the correct transition graph, which is shown in Figure 9.3.

From the fragment 1 exercise we know how this should be done. We build the table one row at a time. The transition graph structure will tell us in which row and column to put each shift, reduce, goto, and accept action; all empty token column positions will then get an error action by default. But as we will see this time things don't work out quite the way we expect. Filling the entries for all the rows except row 3 is straight forward and here is the table (with row 3 blank). [Reference page 140 for the fragment 3 grammar.]

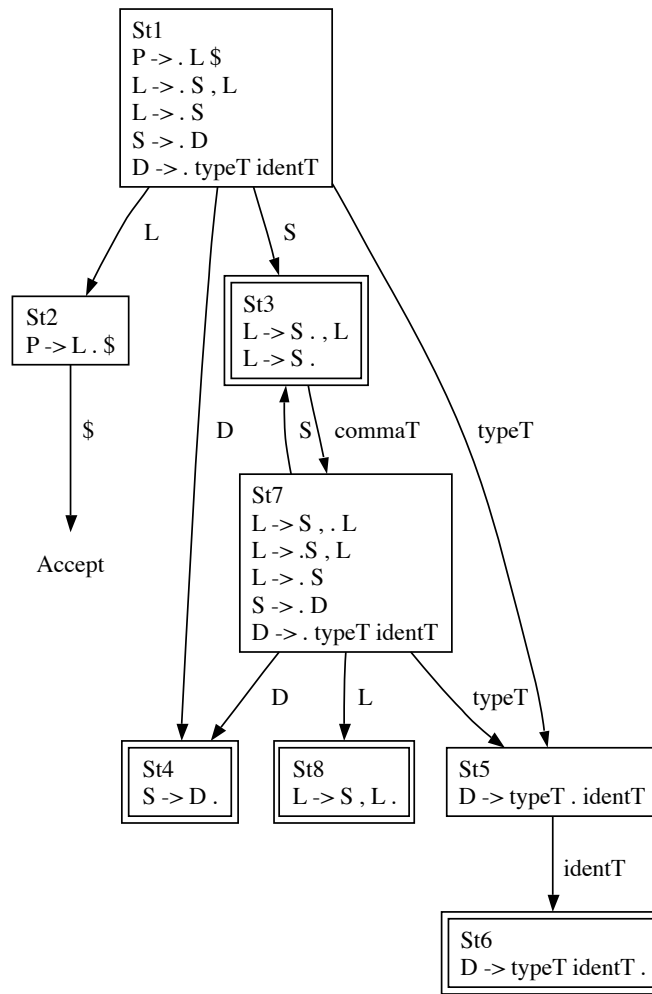


Figure 9.3: Transition Graph for PDef Fragment 3

State	Input (token)				Goto		
	identT	typeT	commaT	\$	L	S	D
1		s5			g(2)	g(3)	g(4)
2				Accept			
3	-	-	-	-	-	-	-
4	r(3)	r(3)	r(3)	r(3)			
5	s6						
6	r(4)	r(4)	r(4)	r(4)			
7		s5			g(8)	g(3)	g(4)
8	r(1)	r(1)	r(1)	r(1)			

Now, what is the problem in row 3? We go to state 7 on **commaT** and then we have to reduce rule 3. Let's see what happens when we do that. Here's completed table.

State	Input (token)				Goto		
	identT	typeT	commaT	\$	L	S	D
1		s5			g(2)	g(3)	g(4)
2				Accept			
3	r(2)	r(2)	s7,r(2)	r(2)			
4	r(3)	r(3)	r(3)	r(3)			
5	s6						
6	r(4)	r(4)	r(4)	r(4)			
7		s5			g(8)	g(3)	g(4)
8	r(1)	r(1)	r(1)	r(1)			

OK! We have two actions in one entry and that is probably not good. What does it really mean? When we are in state 3 and we see `commaT` on the input we have a conflict between consuming the `commaT` and shifting it onto the stack or and reducing grammar rule 3. The difference here is that, where PDef fragments 1 and 2 are LR(0) grammars (no lookahead), the grammar of fragment 3 is not LR(0) and requires some lookahead.

Remember that the PDef fragments we have been investigating come directly from the grammar we used in the PDef tutorial. Because that tutorial used a recursive descent parser, our grammar could not have any left-recursive rules, as this would lead an infinite loop involving calls to the parse method for the rule with left recursion. Here are the rules in PDef fragment 3 which are causing the problem.

```
L → S commaT L
L → S
```

These two rules describe the syntax for a comma separated, non-empty list of non-terminals `S`. But, since the associativity of the list forming operation is not really important, we could, in this case, try a left-recursive version of the first rule; i.e., use the following two rules.

```
L → L commaT S
L → S
```

Notice that now the two right-hand sides start with different non-terminals and that allows us to generate the following LR(0) parse table without conflicts.

### ☛ Activity 21 –

Construct the transition graph for the left-recursive version of PDef fragment 3 and verify that the table in Figure 9.4 is generated from it.

It is important to observe that this left-recursive version of the grammar produces a table with no conflicts. This is nice for LR parsing, since many repetitive language structures are naturally left-associative.

## 9.2.4 The LR(0) Parse Table for PDef-lite

When we examine closely the transition graphs and parse tables for the three fragments and relate them to the bottom-up parsing strategy one point of efficiency becomes clear. There are times when

State	Input (token)				Goto		
	identT	typeT	commaT	\$	L	S	D
1		s5			g(2)	g(3)	g(4)
2			s7	Accept			
3	r(2)	r(2)	r(2)	r(2)			
4	r(3)	r(3)	r(3)	r(3)			
5	s6						
6	r(4)	r(4)	r(4)	r(4)			
7		s5				g(8)	g(4)
8	r(1)	r(1)	r(1)	r(1)			

Figure 9.4: Parse Table for the Left-recursive Version of PDef Fragment 3

a reduction operation is called for but only a single element on the stack is reduced. We can see this in PDef fragment 3 (here slightly modified to use the left-recursive rule for L).

```

0 P → L $
1 L → L commaT S
2 L → S
3 S → D
4 D → typeT identT

```

There are two rules in this fragment, rules 2 and 3, for which the corresponding reduction involves just one stack element. The question is, can we reduce the grammar and in the process make the parsing process more efficient? The answer is “yes,” but care must be taken. For the reduction of rule 4, there will be two elements on the stack, an `identT` on top and a `typeT` below it. The reduction operation will pop the top two elements and then push an appropriate new state. But having reduced rule 4, the only thing that can happen next is to reduce rule 3; this sequence will always take place, reduce 4 and then reduce 3. If we replace the right-hand side of rule 3 with the right-hand side of rule 4 then we can eliminate rule 4. This will accomplish two things: the table will become smaller because a column will not be needed for the symbol `D`, and the double reduction sequence will become a single reduction.

Can we do the same optimization with rule 2? When we reduce rule 3 do we know that our next operation will always be to reduce rule 2? Here the answer is “no”, since it could be that rule 1 must be reduced. So for efficiency purposes we can adopt the following grammar for *PDef-lite*.

```

0 P → SL $
1 SL → lbT L rbT
2 L → L commaT S
3 L → S
4 S → identT assignT identT
5 S → typeT identT
6 S → SL

```

Show that the new PDef-*lite* grammar is LR(0) by constructing its LR(0) transition graph and demonstrating that it contains no shift/reduce conflicts.

---

## 9.3 SLR Parsing

Some grammars, such as PDef fragments 1, 2, and 3, are simple enough to be LR(0) grammars. This means that when we build the LR(0) parse tables there are no shift/reduce conflicts. But there are standard programming language structures, such as arithmetic expressions and possibly-empty parameter lists, which are not LR(0). In this section we will investigate a slight modification to the parse table construction process which allows us to build parse tables without shift/reduce conflicts for most programming language structures.

### 9.3.1 Non-LR(0) Grammar Structures

In this section we will investigate grammar fragments which generate standard programming language structures but have LR(0) parse tables with shift/reduce conflicts.

#### Arithmetic expressions

In Section 9.1 we investigated the fragment of the PDef-*lite* which describes lists of statements. We determined that as long as the recursive part of the grammar is left-recursive then the grammar is LR(0). Consider the following grammar for arithmetic expressions using only the ‘+’ operator.

$$\begin{array}{l} 0 \ P \rightarrow E \ \$ \\ 1 \ E \rightarrow E + \text{id} \\ 2 \ E \rightarrow \text{id} \end{array}$$

This grammar describes, in a left-recursive way, a plus-separated list of identifiers, just as the left-recursive fragment 3 grammar describes a comma-separated list of declarations. So the grammar above must be LR(0). But what happens if we add an operation with a higher precedence to grammar above; is it still LR(0)? Here is the grammar in question.

$$\begin{array}{l} 0 \ P \rightarrow E \ \$ \\ 1 \ E \rightarrow E + T \\ 2 \ E \rightarrow T \\ 3 \ T \rightarrow T * \text{id} \\ 4 \ T \rightarrow \text{id} \end{array}$$

The transition graph for this grammar is depicted in Figure 9.5 and we can see immediately that there are two states which contain terminal dotted forms but are not singleton states – states 3 and 6. The parse table generated from the graph follows.

State	Input (token)				Goto	
	id	+	*	\$	E	T
1	s4				g(2)	g(3)
2		s5		Accept		
3	r(2)	r(2)	s7,r(2)	r(2)		
4	r(4)	r(4)	r(4)	r(4)		
5	s4					g(6)
6	r(1)	r(1)	s7,r(1)	r(1)		
7	s8					
8	r(3)	r(3)	r(3)	r(3)		

The table clearly shows the shift/reduce conflicts inherent in the grammar. In state 3 when the ‘\*’ token appears on the token stream there are two possibilities: either reduce rule 2 or shift state 7 onto the stack. The same conflict exists in state 6, where the shift to state 7 is in conflict with reducing rule 1. These conflicts indicate that the grammar is not LR(0) and we can’t use the parse table to generate a parser.

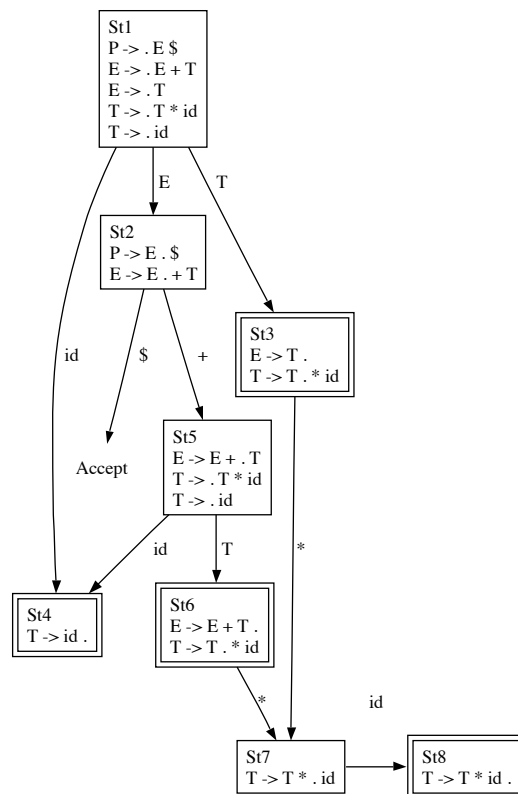


Figure 9.5: Transition Graph for Arithmetic Expressions with operations ‘+’ and ‘\*’

### Possibly empty lists

Another problematic grammatical structure is the possibly empty list. We have already seen that a list with at least one element is an LR(0) structure, but can the same be said for a possibly

empty list? Empty lists are common in programming language structures, empty statement lists and empty parameter lists, for example. Here is a grammar which isolates the empty list structure.

```

0 P → L $
1 L → C
2 L → ε
3 C → C , id
4 C → id

```

The transition grammar for this grammar is quite small and compact. The obvious new issue here is the  $\epsilon$  grammar rule. In a shift/reduce parser an  $\epsilon$  rule can be reduced at anytime.

### ☛ Activity 23 –

Construct the transition graph for the empty-list grammar. Identify any shift/reduce conflicts.

---

## 9.3.2 SLR Parse Tables

The existence of the non-LR(0) grammar structures may seem to be a problem, but, in fact, they lead us to a better parse table structure. The parse table construction process used in Section 9.2.3 indicated that we put a reduce action in each symbol entry for the state containing the terminal dotted form. The strategy would seem to imply that if the next symbol is incorrect then it will show itself later. This, in a sense, is the source of our shift/reduce conflicts. It may be that in a particular state for some symbols a reduce is appropriate while for others a shift is appropriate. SLR (Simple Left Right) parsing uses information about the next token to help decide what to do in states where there is a shift/reduce conflict. In this section we will formalize a process for resolving such shift/reduce conflicts.

To understand the process we will look again at the LR(0) parse table for possibly empty lists. States 1 and 3 are the sites of trouble for that grammar. In state 1, if an `identT` is seen then we don't know whether to reduce rule 2 or to shift to state 4, but on all other input the indication is to reduce rule 2. Is there a way to tell when to reduce rather than to shift?

If the decision is to reduce rule 2, then the parse algorithm will pop the stack no times (rule 2 being an  $\epsilon$ -rule) and then push onto the stack the state corresponding to the symbol (L) on the left-hand side of the rule. This implies that whatever the current token is it must be able to follow L on the stack or in a derivation.

## 9.3.3 First and Follow Sets

In Section 6.2.3, in the discussion of Form 4 grammar rules, it emerged that we had to know the first token that could start the right-hand sides of a set of grammar rules. To help us deal with this we introduced the notion of the first set of a string and defined a corresponding function *first*, which defines the first set for a particular string. In this section we focus on how to compute the



*first* function and its relative the *follow* function, which plays a critical part in forming an SLR parse table.

### The *first* Set Function

In Section 6.2.3 (see page 87) we introduced the following definition for the *first* function.

If  $w$  is a string of terminals and non-terminals then

$$\begin{aligned} \text{first}(w) = \{ \mathbf{x} \mid & \mathbf{x} \text{ is a terminal symbol and } w \Rightarrow^* \mathbf{x}\alpha \\ & \text{or} \\ & \mathbf{x} \text{ is } \epsilon \text{ and } w \Rightarrow^* \epsilon \} \end{aligned}$$

What we did not present at that time was an algorithm for computing values for *first*. That is our purpose in this section. To compute  $\text{first}(\alpha)$  for some string  $\alpha$  we first consider the case where  $\alpha$  is a single symbol  $\mathbf{X}$ .

1.  $\mathbf{X}$  is a terminal symbol: then  $\text{first}(\mathbf{X}) = \{\mathbf{X}\}$ .
2.  $\mathbf{X}$  is a non-terminal symbol.
  - (a) For each grammar rule  $\mathbf{X} \rightarrow \mathbf{xY}$ , where  $\mathbf{x}$  is a terminal symbol:  
Add  $\mathbf{x}$  to  $\text{first}(\mathbf{X})$ .
  - (b) For a rule  $\mathbf{X} \rightarrow \epsilon$ :  
Add  $\epsilon$  to  $\text{first}(\mathbf{X})$ .
  - (c) For each grammar rule  $\mathbf{X} \rightarrow \mathbf{X}_1 \cdots \mathbf{X}_n$  ( $n > 1$ ):  
Let  $\mathbf{k}$  be the largest integer  $1 \leq \mathbf{k} \leq n$  such that  $\mathbf{X}_1 \cdots \mathbf{X}_{\mathbf{k}-1} \Rightarrow^* \epsilon$ , then add all terminals in  $\text{first}(\mathbf{X}_i)$  to  $\text{first}(\mathbf{X})$  for each  $1 \leq i \leq \mathbf{k}$ .  
If  $\mathbf{X}_1 \cdots \mathbf{X}_n \Rightarrow^* \epsilon$ , add  $\epsilon$  to  $\text{first}(\mathbf{X})$ .

There is an important implication in step 2(c), that  $\text{first}(\mathbf{X})$  depends potentially on *first* of all non-terminals. Consequently step 2 above must be carried out on all non-terminals repeatedly until no terminals or  $\epsilon$  can be added to any first set.

Now for the general case in which  $\alpha$  contains more than one symbol. This case is actually very similar to the step 2(c) above. Assume that  $\alpha = \mathbf{X}_1 \cdots \mathbf{X}_n$  ( $n \geq 1$ ), where each  $\mathbf{X}_i$  is a terminal or non-terminal symbol. Let  $\mathbf{k}$  be the largest integer  $1 \leq \mathbf{k} \leq n$  such that  $\epsilon \in \text{first}(\mathbf{X}_i)$  ( $1 \leq i < \mathbf{k}$ ). Then add to  $\text{first}(\alpha)$  all terminals in  $\text{first}(\mathbf{X}_i)$  to  $\text{first}(\alpha)$  for each  $1 \leq i \leq \mathbf{k}$ . If  $\mathbf{k} = n$  and  $\epsilon \in \text{first}(\mathbf{X}_n)$ , then add  $\epsilon$  to  $\text{first}(\alpha)$ .

It may seem odd to allow  $\epsilon$  into the first set of something — after all, what could that possibly mean? Consider the situation where we are confronted with an LL parsing problem such as that presented in Form 4 back on page 87. How should we interpret the situation where  $\epsilon$  is in the first set for one of the choices? Consider the following grammar rules

$$\begin{aligned} \mathbf{X} &\rightarrow \alpha \\ &\rightarrow \beta \\ &\rightarrow \gamma \\ \mathbf{Y} &\rightarrow \rho \mathbf{X} \theta \end{aligned}$$

and assume that  $\epsilon$  is in  $first(\gamma)$  but not in  $first(\alpha)$  or  $first(\beta)$ . In parsing  $Y$  what do we expect to happen when we reach  $X$ ? If one of the terminals in  $first(\alpha) \cup first(\beta) \cup first(\gamma)$  is seen, then one of  $\alpha$ ,  $\beta$  or  $\gamma$  will be parsed. But if none of these terminal symbols match the next token, the fact that  $\epsilon$  is in  $first(\gamma)$  would seem to indicate that we can ignore the three options for  $X$ . So  $\epsilon$  in  $first(\gamma)$  indicates that in certain situations  $X$  is optional.

### Computing the *follow* Set Function

In Section 6.2.3 (see page 89) we introduced the following definition for the *follow* function – remember that this function is defined only on non-terminal symbols.

$$follow(X) = \{x \mid \begin{array}{l} x \text{ is a terminal symbol and } S \Rightarrow^* \alpha X x \beta \\ \text{or} \\ x \text{ is } \epsilon \text{ and } S \Rightarrow^* \alpha X \end{array} \}$$

As with the earlier introduction of the *first* function, we have not yet looked at the algorithm for computing the *follow* function. The algorithm follows a similar strategy to that for computing *first*.  $follow(X)$  is determined by following these three steps repeatedly on all terminals until no terminals or  $\epsilon$  are added to any follow set.

1.  $X$  is the start symbol:  
Add  $\epsilon$  to  $follow(X)$ .
2. For all rules of form  $A \rightarrow \alpha X \beta$ :  
Add to  $follow(X)$  all terminal symbols in  $first(\beta)$ .
3. For all rules of form  $A \rightarrow \alpha X$  or  $A \rightarrow \alpha X \beta$  with  $\epsilon \in first(\beta)$ :  
Add everything (including  $\epsilon$ ) from  $follow(A)$  to  $follow(X)$ .

We will see shortly that the follow set is critical in resolving the shift/reduce conflicts that can occur in some LR(0) parse tables.

#### 9.3.4 Constructing an SLR Parse Table

The construction of an SLR parse table follows the same general path as for an LR(0) parse table. We form the LR(0) transition graph as a basis and construct the follow set for each non-terminal in the grammar. We then form the SLR parse table as follows.

1. For state  $s$  and token symbol  $x$ , if there is a dotted form in state  $s$  with  $x$  following the dot, then put a ‘**shift t**’ into the  $(s, x)$  position in the table if there is a transition from  $s$  to  $t$  on the symbol  $x$ .
2. For state  $s$  and non-terminal symbol  $X$ , if there is a dotted form in state  $s$  with  $X$  following the dot, then put a ‘**goto t**’ into the  $(s, X)$  position in the table if there is a transition from  $s$  to  $t$  on the symbol  $X$ .
3. If in state  $s$  there is a terminal dotted form  $r$  with the non-terminal  $X$  on the left-hand side, and if the token symbol  $y$  is in  $follow(X)$ , then put a ‘**reduce n**’ into the  $(s, y)$  position in the table, where  $n$  is the number of rule  $r$ .

4. For all other pairs  $(s, x)$ , where  $s$  is a state number and  $x$  a terminal symbol, put an ‘error’ entry in the table at position  $(s, x)$ .

If the grammar is an SLR grammar then this table will have no conflicts of any kind— i.e., two entries in one table position.

### Possibly empty lists

In Section 9.3.1 we showed that the grammar for possibly empty lists is not LR(0). This structure is, however, SLR. To show the grammar is SLR we must construct a parse table, according to the SLR rules, with no conflicts. We first define  $follow(X)$  for each non-terminal  $X$  in the grammar. In Section 9.3.3 we determined that the follow sets for the possibly empty list grammar are as follows.

$X$	$follow(X)$
L	\$
C	commaT, \$

With this table in hand we can construct the SLR parse table. It will be helpful to have the LR(0) transition graph – it is displayed in Figure 9.6. The first thing to notice from the construction steps given above is that the goto entries and most of the shift entries are unaffected. This means that our existing table has the correct goto entries and also the correct entries in states 5 and 2. The new structure, which results from step 3 in the process, occurs in the states where there are reduce actions specified; these include the states in which there are conflicts between shifting and reducing. The reduction states are 1, 3, 4, and 6.

In state 1 the reduction is to rule 2, whose left-hand side is the non-terminal L. According to the table above, the follow set of L has one element – \$. So in state 1 there should be an  $r(2)$  in the \$ column and an  $s4$  action in the  $id$  column. We can resolve the conflict in state 3 in the same way. The reduction there is of the rule 1 whose left-hand side is also L. So, as in state 1, we put an  $r(1)$  in the \$ column and an  $s5$  in column  $id$ . While it may seem we are finished, we are not. We have resolved the shift/reduce conflicts, but there are two other states with terminal dotted forms, states 4 and 6. Each dotted form has the same left-hand side, the non-terminal C, which has a follow set including \$ and ‘,’. So we put appropriate reductions in the columns for the follow set elements.

State	Input (token)			Goto	
	id	,	\$	L	C
1	s4	r(2)		g(2)	g(3)
2			Accept		
3		s5	r(1)		
4		r(4)	r(4)		
5	s6				
6		r(3)	r(3)		

### Arithmetic expressions

To see that the grammar for arithmetic expressions is SLR we follow the same procedure as in the previous section.

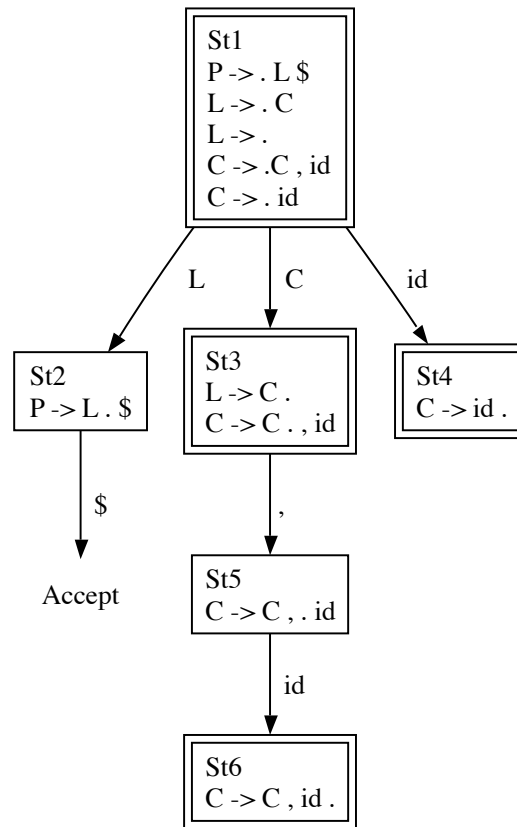


Figure 9.6: Transition Graph for Possibly Empty List Grammar

### Activity 24 –

Verify that the following table defines the follow sets for the non-terminals of the arithmetic expression grammar.

$X$	$follow(X)$
E	\$ +
T	\$ + *

### Activity 25 –

Verify that the arithmetic expression grammar is SLR and that the following table is its SLR parse table.

State	Input (token)				Goto	
	id	+	*	\$	E	T
1	s4				g(2)	g(3)
2		s5		Accept		
3		r(2)	s7	r(2)		
4		r(4)	r(4)	r(4)		
5	s4					g(6)
6		r(1)	s7	r(1)		
7	s8					
8		r(3)	r(3)	r(3)		

## 9.4 Guided Development

In this chapter we have focused on the LR parsing approach, with an emphasis on constructing LR(0) and SLR parse tables for various grammars and fragments. In this guided development you will apply the techniques to the construction of a parse table for the PDef grammar.

The PDef grammar can be seen as comprising two components, the list component which is almost identical to the PDef-*lite* grammar and the expression component. The two components come together when the expression grammar is plugged into the PDef-*lite* assignment structure, with the expression structure replacing the `identT` token on the right-hand side of the assignment statement. We have seen earlier that the PDef-*lite* component has an LR(0) grammar while the expression component has an SLR grammar. We would expect, then, that their combination would be SLR. Here is the grammar we will use for PDef for purposes of forming its parse table.

```

0 P → SL
1 SL → lbT L rbT
2 L → L commaT S
3 L → S
4 S → identT assignT E
5 S → typeT identT
6 S → SL
7 E → E addOpT T
8 E → T
9 T → T mulOpT F
10 T → F
11 F → intT
12 F → floatT
13 F → identT
14 F → lpT E rpT

```

Complete the following table, which is meant to define  $follow(X)$  for each non-terminal  $X$  in the PDef grammar above.

non-terminal $X$	$follow(X)$
P	
SL	
L	
S	
E	
T	
F	

---

### ☛ Activity 27 –

Construct the SLR parse table for rules 0-6 above. In this process treat **E** as a terminal symbol. Notice that, even though we know this grammar is LR(0), you will construct the SLR parse table. This means, of course, that you must be careful to assign reduce actions only for appropriate follow set symbols (see the table developed in Activity 26). Be sure to start by drawing a new LR(0) transition graph.

---

### ☛ Activity 28 –

Create a new grammar by taking the PDef rules 7-14 (retain that numbering) and adding as rule 0 (zero) the following.

$$P \rightarrow E$$

For this new grammar construct the SLR parse table – as in the previous activity, begin by drawing an LR(0) transition graph for the grammar. Again, make use of the follow table developed in Activity 26.

---

### ☛ Activity 29 –

Take the two transition graphs and “glue” them together. This is more subtle than it may seem. In gluing the two parts together the symbol **E**, which was treated as a terminal symbol, must now be treated as a non-terminal. There should be a state in the graph of the first component containing a dotted form with the dot immediately before the **E**. Now that **E** is to be interpreted as a non-terminal, it is necessary to reform the closure of that state. In the expression graph you will want to discard the portion of the graph dealing with the start non-terminal **P**. When you have correctly formed the PDef transition graph, construct the SLR parse table for PDef.

---

## Chapter 10

# Attribute Grammars – Theory and Practice

Formalizing the syntax of a language is fairly straightforward, but to do so for the semantic side is more complex. The problem is that the static semantic properties that interest us (e.g., are identifiers appropriately declared, are identifiers used in accordance with their declared properties) cannot be described using context free grammars – they are inherently *context sensitive*. Consequently, in Chapter 8 we described the static semantics of PDef using informal means. In this chapter we will see that there are ways we can enhance a context free grammar in order to include context sensitive information – these enhanced grammars are called *attribute grammars*. In this section we will investigate these *attribute grammars* and see how they can be used to formally describe static semantic properties.

### 10.1 Overview

The key to understanding the formal approach to describing static semantics is to understand how declaration information, which we call *bindings*, relates to expressions in the context of a parse tree. We will focus on the following simple CL program.

```
{ int a, float b, a = 3, b = a*4.0 }
```

The first thing we can say is that, based on the six static semantic rules of CL (see page 42), our program is semantically correct. Looking at the last assignment statement (`b = a*4.0`) we can see that both identifiers, ‘a’ and ‘b’, are declared earlier in the program and that, based on the bindings from those declarations and CL’s semantic rule 4, the expression ‘a\*4.0’ has type `float`. Since ‘b’ is bound to type `float` and `float` is assignment compatible with itself, we know the assignment statement is correct.

Obviously, in judging the correctness of the assignment statement we had to make use of the bindings from the declaration statements *at the point* of the assignment statement – i.e., where the declared identifiers are actually used. This is an easy thing for us to do visually, but not so easy if we think in terms of automating this way of checking correctness. We get a new perspective of the automatic correctness problem by looking at the parse tree for our example program – the parse tree is shown in Figure 10.1. What we see in the parse tree is that, at least from a graphical point

of vie, the declaration bindings are quite a distance from the assignment statements, where they are needed in order to judge correctness.

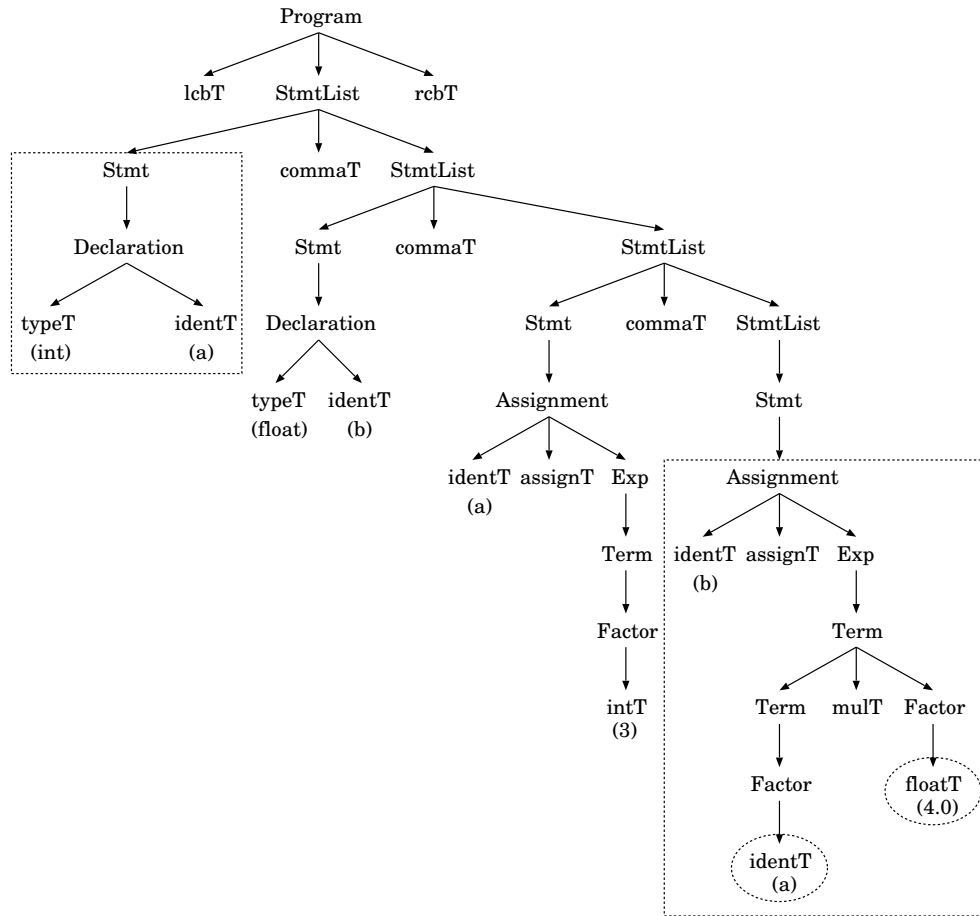


Figure 10.1: Parse Tree for  $\{ \text{int } a, \text{float } b, a = 3, b = a*4.0 \}$

Now in the context of the same parse tree, consider performing a depth-first traversal. It seems that when we visit the declaration subtrees we might be able to grab hold of that data, i.e., the bindings  $(a, \text{int})$  and  $(b, \text{float})$ , and “carry” them to the assignment statement subtrees. We could also think of it as a game where the game tokens are pieces of data and we move the game tokens from one node to another (up or down the tree) until the game tokens have arrived where it is needed. There are two problems uncovered here. First, we need to understand what data is associated, either explicitly or implicitly, with nodes in a parse tree – for example, bindings (such as  $(a, \text{int})$  and  $(b, \text{float})$ ) will be associated with `Declaration` nodes. Second, if we look at either of the assignment statement nodes we see that we will have to know the type of the identifier (on the left) and the type of the expression (on the right) in order to judge correctness. Apparently the bindings associated with the declaration nodes will have to be able to migrate to the assignment nodes and then to migrate through the nodes of the expression subtree.

In order to formalize descriptions of static semantics we will have to formally associate the data inherent in a parse tree with particular nodes and formalize the ways that data can be moved around the parse tree. This will be the focus of the next few sections.



## 10.2 Attributes and Attribute Migration

In the previous section we saw a clear connection between information in the text of a program and a parse tree, information such as variable and type names, literal values, and even binding pairs from declarations. When talking about semantics we use the term *attribute* to refer to information such as names, values, and bindings. The connection between attributes and a parse tree is made by associating the attributes with certain nodes in the parse tree. Two simple examples are indicated by the dashed ovals on the parse tree of Figure 10.1, where the terminal symbol `identT` has a name attribute ‘a’ and `floatT` has the value attribute ‘4.0’. We might also note that there are two other attributes that can be easily associated with these terminal symbols: in addition to a value attribute, `floatT` also has a type attribute `float`; the symbol `identT` also has a type attribute `int`. It is important to see here that the values of three of these attributes, the type and value of `intT` and the name of `identT`, are implicit in the tree, where as the value of the fourth, the type of `identT`, must be deduced from the first declaration in the program.

Figure 10.2 isolates the two boxed subtrees of Figure 10.1. These subtrees have been augmented to show the attributes associated with certain terminal symbols – they also demonstrate the dot-notation used to specify the particular attribute of a grammar symbol: for example, `identT.name` and `identT.type` denote the name and declared type for the token `identT`, while `intT.value` and `intT.type` denote the numeric value and type for the token `intT`. But the diagram also illustrates the fact that non-terminal symbols can have attributes. In particular, for the non-terminal `Exp`, just below the `Assignment` node, `Exp.type` implies that `Exp` has a type attribute with value `float` and the notation `Declaration.bind` implies `Declaration` has an attribute named `bind` whose value is the ordered pair shown.

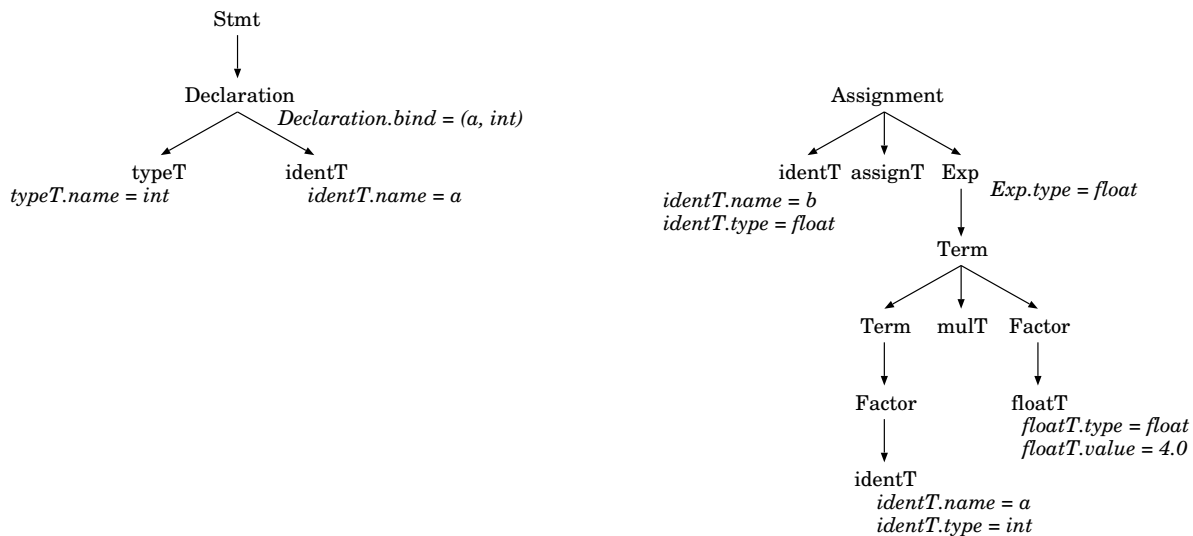


Figure 10.2: Parse Subtree for ‘`int a`’ and ‘`b = a*4.0`’

After our discussion of the previous section these attribute values should not be a surprise. But in the context of a parse tree there an important issue raised by three of the values: `identT.type` and `Exp.type` in the assignment subtree and `Declaration.bind` in the declaration subtree. While the values of `intT.value` and `intT.type` are either present in the token (`intT.value`) or implicit

(`intT.type`), the values of `Declaration.bind`, `Exp.type`, and `identT.type` must somehow be determined. These determinations hinge on the ability to move attribute values from one node in the tree to another.

The simplest of the three determinations involves `Declaration.bind`. The data needed to form the value of `bind` always (i.e. for all declaration subtrees) resides in the two components of the `Declaration` node. So by accessing the values `identT.name` and `typeT.name` (immediately accessible from the `Declaration` node) we can build – actually the traditional term is *synthesize* – a value for `bind` in the following form.

(`identT.name`, `typeT.name`)

This reflects our idea that a declaration statement binds a type to an identifier – here the binding is explicitly represented as an ordered pair. So we say that `Declaration.bind` is synthesized from component attribute values.

Deducing a value for `Exp.type` is not so straight forward, since its value depends on the type attribute values associated with its subtrees. We can see the answer to this question by looking more carefully at the parse tree for the expression ‘`a*4.0`’. Figure 10.3 has two versions of this subtree, with the one on the left being extracted from the boxed assignment subtree shown in Figure 10.1. Following the reasoning from the declaration problem, we can see that the value `Exp.type` should

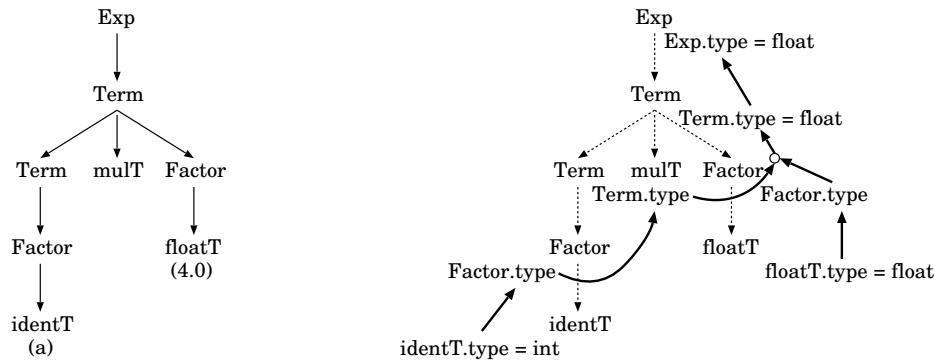


Figure 10.3: Annotated Parse Tree for `a*4.0`

be the same as that retrieved from the component of `Exp`, i.e., from the node `Term`. This, of course, implies that the non-terminal `Term` also has a type attribute, but the question “What is the value of `Term.type`?” is not as easily answered. As with the attribute `Declaration.bind`, the value of `Term.type` must be synthesized from the values of its components. The difference here is that the attribute values of the components must be combined in the right way in order to determine the value of `Term.type` – in fact, the synthesized value is dictated by the CL semantic rule 4.

This reasoning strategy continues to descend the tree, with the attribute value at each node being synthesized from the attribute value(s) of its component(s). Once this process reaches the leaves of the tree (`identT` and `floatT`) we actually get real answers which can then unwind their way back up the tree. This unwinding is represented in the annotated parse tree on the right side of Figure 10.3.

The one problem we haven’t addressed to this point is, at an `identT` node in an expression subtree, how is it that we can know the value of `identT.type`? It is certainly not inherent in

the identifier itself, so it must be that we have to have access to the corresponding declaration for `identT.name`! This attribute information cannot be found below `identT` since it is a leaf node. Instead, the attribute information must migrate down the tree to the `identT` node.

To describe this migration of bindings we need to establish a new attribute, call it `bindList`, at the `StmtList` nodes. This new attribute will pickup the `bind` attributes, which migrate from the `Declaration` nodes up to the `StmtList` nodes, and then be passed down the tree to the assignment subtrees. This new attribute value travels in the opposite direction from those of `bind` and `type`, discussed above. We say that attributes that move up the parse tree are *synthesized*, while those moving down the parse tree are *inherited*. In the next two sections we will focus on these two categories of attributes and how to describe their migration.

## 10.3 Synthesized Attributes

In this section we will focus on attribute values that travel upwards in a parse tree – our goal is to progress towards a more formal description of attribute migration. Our discussions above, dealing with attributes and attribute migration, appealed to the depth-first traversal of parse trees to understand the needs of semantic checking. Since the tree traversals visit each node in the tree, it is appropriate for us to formalize attribute migration in the same way, one node at a time. To this end we will focus on individual nodes and how attribute values originate at or pass through the nodes.

### 10.3.1 Declaration.bind

Let's consider the `Declaration` node depicted in Figure 10.2. The attribute `bind` is created there from values obtained from its component subtrees. We say that `Declaration.bind` is a *synthesized attribute* because the direction of travel of the attribute values is up the tree. We describe the attribute motion in the following formal way.

```
Declaration.bind = (identT.name, typeT.name)
```

We call such a form an **attribute assignment**, adapting the name of the corresponding familiar assignment statement. In this case the attribute `Declaration.bind` is assigned a value composed of attribute values from its components. Notice that our focus is on the node `Declaration` and that it is the name `Declaration` which appears on the left of the attribute assignment – this is what distinguishes a synthesized attribute. It is important to notice that the attribute assignment given above is not determined by a particular `Declaration` node, but rather describes the behavior of the `bind` attribute for all such nodes.

But there is more to uncover relative to values of the attribute `bind`. The `bind` value synthesized at a `Declaration` node must be passed on up the syntax tree in order to make the binding in the value available to other parts of the syntax tree. So, the node `Stmt` above each `Declaration` node would seem to need a `bind` attribute – the same, of course, is true for the node `StmtList`, since it is only these nodes that give access to other parts of the syntax tree. The following two attribute assignments define these relationships.

```
Stmt.bind = Declaration.bind
StmtList.bind = Stmt.bind
```

Again, notice that the form of these attribute assignments indicate that both `Stmt.bind` and `StmtList.bind` are synthesized attributes.

### 10.3.2 Synthesized Attributes in the Expression Subtree

We can revisit the `Assignment` subtree which is also depicted in Figure 10.2. What we would like to do is to generate a list of attribute assignments that provide a description of the migration of type values up the subtree. We will begin at the bottom of the tree and work our way upwards.

On the right side of Figure 10.3 we see an annotated version of the expression subtree. The bold arrows indicate the flow of type information. If we focus on the left branch of the subtree we see that the value `identT.type` is passed up to `Factor` then to `Type` and then, along with the value passed up the right branch, synthesized into a new value passed to the upper `Type` node. Just for the transfers mentioned so far, and following the declaration example, we can write down the following attribute assignments.

```
Factor.type = identT.type
Term.type = Factor.type
Termtop.type = a combination of Termbottom.type and Factor.type
```

*We have invented an ad hoc notation to distinguish the two Term nodes.*

```
Exp.type = Term.type
```

As in the case of the declaration subtree, notice that each attribute assignment is independent of the particular nodes involved. In fact, the second assignment above describes the corresponding `Type` node on the right branch of the expression subtree. Consequently, the only new attribute assignment that needs to be added to our list above is that involving the bottom `Factor` node on the right subtree branch. That assignment takes the following form.

```
Factor.type = identT.type
```

There is one final point to make about the third attribute assignment above. The phrase “a combination of” is not sufficiently specific to provide a useful rule. In forming an attribute assignment it would be nice to be able to write a mathematically clear description of the required synthesis. The table in Figure 10.4 defines a function taking three parameters, the operation and the types of the two arguments to the operation. We will use `table` as the name of the function. Now borrowing from familiar programming notation, we can rewrite the third attribute assignment as follows.

```
Termtop.type = table(multT, Termbottom.type, Factor.type)
```

In fact, following this same strategy we can anticipate other expression syntax tree forms for `Term` and `Exp` nodes (we add in the previous `multT` version above for completeness).

```
Exptop.type = table(addT, Expbottom.type, Term.type)
Exptop.type = table(subT, Expbottom.type, Term.type)
Termtop.type = table(divT, Termbottom.type, Factor.type)
Termtop.type = table(multT, Termbottom.type, Factor.type)
Termtop.type = table(modT, Termbottom.type, Factor.type)
```

operations	argument 1 type	argument 2 type	result type
+ - * /	int	int	int
	int	float	float
	float	int	float
	float	float	float
%	int	int	int

Figure 10.4: Result Type Attribute for CL Operations

## 10.4 Inherited Attributes

Synthesized attributes travel up a parse tree, as we've seen with the attributes `type` and `bind` in the declaration and expression parse trees. When attributes travel in the opposite direction, down a parse tree, we say the attributes are *inherited*. In this section we will discuss what makes the attribute at a particular node an inherited attribute and we will extend the attribute assignment notation so that it describes the behavior of inherited attributes.

Since we are interested in describing attribute migration down a parse tree, we will focus on the problem of getting binding data from a declaration node to assignment subtrees, where it can be used in semantic checking. The annotated parse tree in Figure 10.5 illustrates, via the bold arrows, the migration paths the `bind` attribute values must follow. Since we have already studied the movement of `bind` values we will look most carefully at the `StmtList` nodes and their attribute `bindList`. The `bindList` value is meant to accumulate all `bind` values encountered during a depth-first traversal of the parse tree.

If we begin at the top of the parse tree, at the `Program` node, and follow the bold arrows that lead downward, we see first that the initial empty list value is passed down to the `StmtList` node labeled ①. At that node the `bind` value (`a`, `int`) will be synthesized from below, which makes it available to be added to `StmtList.bindList`, which at that point is empty. The list will descend the tree to the `StmtList` node labeled ②. At that point the `bind` value (`b`, `float`), synthesized up from the second `Declaration` node, can be added to the list inherited from the ① node. This list, now containing the two declaration statement bindings, continues down the tree to the two `Assignment` nodes in turn. From an `Assignment` node the `bindList` value will descend through the expression graph. One such path is drawn for the second assignment statement and the left branch of its expression subtree.

Before discussing attribute assignments for inherited attributes, we need to orient ourselves properly. In reading an attribute assignment for a synthesized attribute it is easy to relate the assignment to the node and its component subtrees. We must think in the same way for inherited attributes. So if we focus on the attribute `StmtList.bind`, since it is inherited from above we must see `StmtList` as a component of `Program` and relate the value of `StmtList.bind` to some attribute value available at the node `Program`. Since the initial value of `bindList` is to be empty, we can adopt the following two attribute assignments.

```
Program.bindList = [ ]
StmtList.bindList = Program.bindList
```

While it is true that these two assignments could be collapsed to one, at this point it is best to

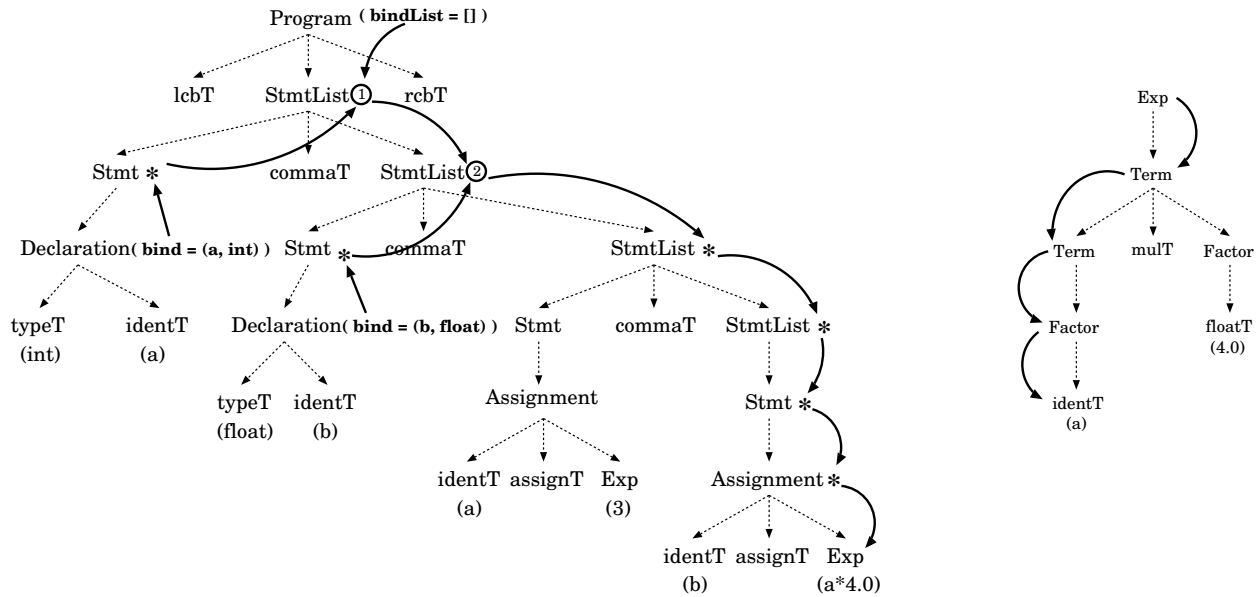


Figure 10.5: Annotated Parse Tree for `{ int a; float b; a = 3; b = a*4.0 }`

retain both to orient the assignments at the Program node.

The first attribute assignment above shows `Program.bindList` being assigned a constant value but indicates no migration of attributes – so `Program.bindList` is not a synthesized attribute, even though the node name is on the left of the assignment. On the other hand, the second attribute assignment is characteristic of an inherited attribute: the component node appears on the left and the node appears on the right – a clear indication of information moving down the tree. So `StmtList.bindList` is an inherited attribute.

With that start the corresponding attribute assignments for the other nodes of interest are not difficult to see. We wrote on page 167 that `StmtList.bind` synthesizes a value from `Stmt.bind`. Now we will see that that assignment is not really necessary. Instead we can replace it with a new assignment in which the value of `Stmt.bind` can be added immediately to the inherited value `StmtList.bindList`. Of course, one that `bind` value is added to the list, the list must be inherited by the lower `StmtList`. Here are appropriate assignments oriented at the `StmtList` node.

```

StmtList.bindList.addTo(Stmt.bind)
StmtListbottom.bindList = StmtListtop.bindList

```

The first line, while it doesn't look like an attribute assignment, actually is – we have simply used the familiar object-oriented notation `'addTo(..)'` to indicate adding the argument to the list.

The two attribute assignments we have just generated seem to describe what happens at the third and fourth `StmtList` nodes as well. But there is a bit of a hiccup. Do you see it? When the `Stmt` node has a declaration below it then we know we can synthesize a value for `Stmt.bind`. But what if there is an assignment instead? From the vantage point of `StmtList` there is no way to see what is below its `Stmt` node. So we employ a standard sort of programming trick. If we can't tell the difference, then force `Stmt` to always have a value – even if it is the value `null`. So in the cases when `Stmt` has an `Assignment` node below, we can simply assign a null value to `Assignment.bind`

and then synthesize that value up to `Stmt.bind` as usual. Here are the appropriate attribute assignments.

```
Assignment.bind = null
Stmt.bind = Assignment.bind
```

We have just one unsolved problem remaining: how do we get the `bindList` value to the points it is needed for semantic checking. In fact, semantic checking takes place not only in assignment subtrees, but also in declaration subtrees as well – after all, CL semantic rule 1 indicates an identifier can't be declared twice. So in a `Stmt` subtree each node must inherit the `bindList` value. So for example in the declaration subtree we can add the following attribute assignment.

```
Stmt.bindList = StmtList.bindList
Declaration.bindList = Stmt.bindList
```

We will see in the next section that it is not necessary to pass the `bindList` down to the `identT` node. Similar additions can be made to the expression nodes, in each case simply passing the inherited `bindList` value down to components. Here are the appropriate attribute assignments.

```
Assignment.bindList = Stmt.bindList
Exp.bindList = Assignment.bindList
Expbottom.bindList = Exptop.bindList
Term.bindList = Exptop.bindList
Termbottom.bindList = Termtop.bindList
Factor.bindList = Termtop.bindList
identT.bindList = Factor.bindList
Exp.bindList = Factor.bindList
```

These attribute assignments may seem a bit disconnected when seen in a list like this, but remember they must be understood from the point of view of a particular node type. Later we will see that these attribute assignments can be associated conveniently with grammar rules, which will provide a convenient organizing structure.

## 10.5 Static Semantics via Attribute Grammars

It should be obvious from the preceding sections that the attribute assignment forms are central to a formalism for describing the static semantics of a language. In fact they are one example of what are called *semantic rules* of a language. In this section we will discuss semantic rules and how they are used to define a new grammatical structure called an *attribute grammar*. It is attribute grammars that can be used to formally define a language's static semantics.

The attribute assignment forms discussed in the previous two sections have the advantage that they are independent of the parse tree to which they are applied. At the same time there is an inherent ambiguity that makes them difficult to sort out. For example, we saw several attribute assignments defined for the nodes labeled `Stmt` – here they are collected.

```
Declaration.bindList = Stmt.bindList
Stmt.bind = Declaration.bind
```

`Assignment.bind = null` – this is listed because it is meant to be paired with the following

```

Stmt.bind = Assignment.bind
Assignment.bindList = Stmt.bindList

```

But whether these are isolated or meant to be grouped is unclear. More seriously, it is not clear which node the assignments are to be associated with. If we associate the first with `Stmt` then `Stmt.bindList` is a synthesized attribute – if it is associated with `StmtList` then it is an inherited attribute. Clearly only one should be correct, but there’s no indication which one.

There is one organizing structure, however, that can bring the attribute assignments into a comprehensible order – the language’s context free grammar. Since the structure of each non-terminal node in the graph is determined by a grammar rule, we should be able to associate the appropriate attribute assignments with a particular grammar rule. Such an association of a grammar rule and attribute assignments is called an annotated grammar rule. It would seem that, to define the static semantics of a language we might be able define a new grammar structure in each rule of the context free grammar is represented by an annotated version of the same rule. In this way we get a description of what happens, attribute migration wise, at each possible node in any parse tree. The following definition formalizes these ideas.

#### DEFINITION 10

*Attribute Grammar* An attribute grammar is a context free grammar for which each rule,  $B \rightarrow w$ , has been augmented with a sequence of semantic rules derived from  $B$  and  $w$ . The new rules are called annotated rules and are written in the following form:  $B \rightarrow w : B_1, \dots, B_n$ , where  $n > 0$  and each  $B_i$  is a semantic rule.

### 10.5.1 Semantic Rules

There are two important characteristics of semantic rules. First, they are defined relative to a particular grammar rule. What this means is that the only elements that can appear in a rule are attributes of the terminal and non-terminal symbols appearing in the rule. The attribute assignments described above clearly meet this requirement. In fact, if we look at the assignments we listed above for `Stmt` we can associate them with grammar rules in the following way.

```

Stmt → Declaration : Declaration.bindList = Stmt.bindList,
                        Stmt.bind = Declaration.bind
Stmt → Assignment : Assignment.bindList = Stmt.bindList,
                        Assignment.bind = null,
                        Stmt.bind = Assignment.bind

```

Notice that the second two assignments for `Stmt → Assignment` can now be easily combined into one assignment, assigning `null` directly to `Stmt.bind`.

The attribute assignment form of semantic rule is appropriate for defining attribute movement, but it is not useful for semantic checking. In order to facilitate semantic checking we allow semantic rules to be any “algorithmic” statement involving the attributes specified for a rule. We have already seen an example of this for the non-terminal `StmtList` – here is the partially attributed rule.



$$\text{StmtList} \rightarrow \text{Stmt} : \text{StmtList.bindList.addTo}(\text{Stmt.bind})$$

This “method call” carries the implication that `bindList` is an object with various methods available. While this semantic rule still specifies attribute motion, the structure can be useful in semantic checking as well.

Remember the grammar rule  $\text{Declaration} \rightarrow \text{typeT identT}$ . The CL semantic rule 1 indicates that an identifier cannot be declared twice. So for this grammar rule we should specify that the value `identT.name` does not appear in a binding in `bindList`. Remember that `Declaration` inherits a value for `bindList` from `Stmt`, so we can write a semantic rule that searches `bindList` for `identT.name`.

$$\text{Declaration} \rightarrow \text{typeT identT} : \text{Declaration.bindList.hasNoEntry}(\text{identT.name}), \\ \text{Declaration.bind} = (\text{identT.name}, \text{typeT.name})$$

The first semantic rule is meant to specify a condition that must be true – in this case that there is no `bind` entry in the `bindList` that starts with `identT.name`. The implication is that if the condition fails then checking terminates. If we think of the semantic checking occurring during a parse tree traversal, then if the condition fails the traversal terminates. So the second semantic rule, the attribute assignment, only occurs if the first condition succeeds. An alternative form for these two rules would be to nest the second within a selection-type statement as follows.

$$\text{Declaration} \rightarrow \text{typeT identT} : \text{if} (\text{Declaration.bindList.hasA}(\text{identT.name})) \\ \text{Declaration.bind} = (\text{identT.name}, \text{typeT.name})$$

Another important thing about the two semantic rule form for the `Declaration` rule is that the order of the semantic rules is crucial. We do not want to send the `bind` value up the tree if the correctness condition is not true – so we check the condition first.

There is another situation that can occur in an annotated rule that we must mention – that is the situation where the same non-terminal appears on both sides of a rule. We have seen this problem occur in the attribute assignments for `StmtList`, `Exp`, and `Term`. In these cases we used subscripts ‘top’ and ‘bottom’ to distinguish the parse tree nodes. But it is possible for there to be several occurrences of a non-terminal in a grammar rule so we need a more flexible strategy. What we will always do is to number the occurrences of a non-terminal from left to right starting with one. So the following would be an appropriate annotated grammar rule for `Exp`.

$$\text{Exp} \rightarrow \text{Exp addT Term} : \text{Exp}_2.\text{bindList} = \text{Exp}_1.\text{bindList}, \\ \text{Term.bindList} = \text{Exp}_1.\text{bindList}, \\ \text{Exp}_1.\text{type} = \text{table}(\text{addT}, \text{Exp}_2.\text{type}, \text{Term.type})$$

The numbering certainly makes the semantic rules appear more complex, but in fact the rules are complex and the numbering is a reasonable way to manage that complexity. Again notice that the order in which the semantic rules appear is important, since we can’t expect to extract a type from `Exp2` until the value of `bindList` has been passed down the subtree. Remember, our perception of the semantic rules is that they describe part of a tree traversal: the first two rules assign attribute value inherited form up the tree; the third indicates the depth-first traversal of two subtrees (for `Exp2` and then `Term`), at the conclusion of which the corresponding type values are returned.

Finally, we see in Figure 10.6 a listing of an attribute grammar for the language CL. Repeated there are rules we have already examined, but also the appropriately attributed rules for the other

grammar rules of the CL context free grammar. Notice that synchronized and inherited attributes are distinguished in the attribute assignments, where synchronized attributes represent a migration of values from right to left relative to the grammar rule and inherited attributes represent a migration from left to right. There are a few of the attribute rules for CL whose semantic rules need discussion.

**Assignment**  $\rightarrow$  **identT assignT Exp** : In this rule we must retrieve an entry in the **bindList** for the **identT.name**. The call to **lookUp** will return an ordered pair (a **bind** value), of which we need only the type. The notation specifies the returned ordered pair being assigned to **(n,t)** – so ‘**n**’ must contain the name and ‘**t**’ the type from the pair. This form also appears in the entry for rule **Factor**  $\rightarrow$  **identT**.

**Exp**  $\rightarrow$  **Exp (addT | subT) Term** : This rule is actually two rules, one for each possible operation. Since the function **table** returns the same value regardless of the operation we simply pass **addT** even if the operation is **subT** – this saves writing the rule out again. A similar structure appears for the second rule for **Term**.

**Term**  $\rightarrow$  **Term modT Factor** : This rule is singled out because, while **mult** and **divT** produce the same results via **table**, the values returned **table** for **modT** are different.

The desire for a formal description of static semantic properties also relates to the desire of professional compiler writers to be able to automatically generate as much of the compiler as possible. For automatic generation, it would be attractive to be able to feed a program a regular expression, for the tokens, and a list of annotated context free grammar rules from which could be automatically generated a tokenizer, parser, and semantic checker (and maybe more).

An *attribute grammar* is a set of annotated context free grammar rules. The purpose of the annotations is to describe how attributes associated with various tokens (terminal symbols) and non-terminal symbols migrate around a parse tree so that static semantic properties can be described. Derived from our earlier discussions, the table in Figure 10.6 presents an attribute grammar for CL. Notice that some of the annotations are informal, since clarity is more important at this point than absolute formality. Also notice in some entries, where the **identT** token is involved, there is an annotation that describes an action or a property rather than indicating some transmission of attributes. Finally, it is important to focus on those rules, such as **Exp** and **Stmt**, for example, which have both inherited and synthesized attributes.

### 10.5.2 The PDef Static Semantics

The semantic characteristics of PDef are very similar to those of CL, with the only difference being the way in which **bind** attributes are managed. The attribute grammar for PDef, given in Figure 10.7, reflects this similarity in that there are only three new or modified entries. While all but three of the entries are the same as in the CL attribute grammar, there are a couple of important changes to terminology employed in the semantic rules. First, it is necessary to build into the attribute grammar knowledge of the different levels in the syntax tree. Thus in the first entry, for the grammar rule

**Program**  $\rightarrow$  **List**,

we specify the parent symbol table reference to pass to **List** as *null* – thus putting a top on the symbol table structure. Notice also in the second table entry, for the grammar rule

CL Grammar Rule	Annotations
$\text{Program} \rightarrow \text{lcbT StmtList rcbT}$	$\text{StmtList.bindList} = []$ (empty list)
$\text{StmtList} \rightarrow \text{Stmt}$	$\text{Stmt.bindList} = \text{StmtList.bindList}$ , $\text{StmtList.bindList.addTo}(\text{Stmt.bind})$
$\text{StmtList} \rightarrow \text{Stmt commaT StmtList}$	$\text{Stmt.bindList} = \text{StmtList}_1.\text{bindList}$ , $\text{StmtList}_1.\text{bindList.addTo}(\text{Stmt.bind})$ , $\text{StmtList}_2.\text{bindList} = \text{StmtList}_1.\text{bindList}$
$\text{Stmt} \rightarrow \text{Declaration}$	$\text{Declaration.bindList} = \text{Stmt.bindList}$ , $\text{Stmt.bind} = \text{Declaration.bind}$
$\text{Stmt} \rightarrow \text{Assignment}$	$\text{Assignment.bindList} = \text{Stmt.bindList}$ , $\text{Stmt.bind} = \text{null}$
$\text{Assignment} \rightarrow \text{identT assignT Exp}$	$(n,t) = \text{Assignment.bindList.lookUp}(\text{identT.name})$ , $\text{Exp.bindList} = \text{Assignment.bindList}$ , $\text{compatible}(t, \text{Exp.type})$
$\text{Declaration} \rightarrow \text{typeT identT}$	$\text{Declaration.bindList.hasNoEntry}(\text{identT.name})$ , $\text{Declaration.bind} = (\text{identT.name}, \text{typeT.name})$
$\text{Exp} \rightarrow \text{Term}$	$\text{Exp.type} = \text{Term.type}$ , $\text{Term.bindList} = \text{Exp.bindList}$
$\text{Exp} \rightarrow \text{Exp (addT   subT) Term}$	$\text{Exp}_2.\text{bindList} = \text{Exp}_1.\text{bindList}$ , $\text{Term.bindList} = \text{Exp}_1.\text{bindList}$ , $\text{Exp}_1.\text{type} = \text{table}(\text{addT}, \text{Exp}_2.\text{type}, \text{Term.type})$
$\text{Term} \rightarrow \text{Factor}$	$\text{Factor.bindList} = \text{Term.bindList}$ , $\text{Term.type} = \text{Factor.type}$
$\text{Term} \rightarrow \text{Term (multT   divT) Factor}$	$\text{Term}_2.\text{bindList} = \text{Term}_1.\text{bindList}$ , $\text{Factor.bindList} = \text{Term}_1.\text{bindList}$ , $\text{Term}_1.\text{type} = \text{table}(\text{multT}, \text{Term}_2.\text{type}, \text{Factor.type})$
$\text{Term} \rightarrow \text{Term modT Factor}$	$\text{Term}_2.\text{bindList} = \text{Term}_1.\text{bindList}$ , $\text{Factor.bindList} = \text{Term}_1.\text{bindList}$ , $\text{Term}_1.\text{type} = \text{table}(\text{modT}, \text{Term}_2.\text{type}, \text{Factor.type})$
$\text{Factor} \rightarrow \text{intT}$	$\text{Factor.type} = \text{int}$
$\text{Factor} \rightarrow \text{floatT}$	$\text{Factor.type} = \text{float}$
$\text{Factor} \rightarrow \text{identT}$	$(n,t) = \text{Factor.bindList.lookUp}(\text{identT.name})$ , $\text{Factor.type} = t$
$\text{Factor} \rightarrow \text{lpt Exp rpt}$	$\text{Exp.bindList} = \text{Factor.bindList}$ , $\text{Factor.type} = \text{Exp.type}$

Figure 10.6: CL Attribute Grammar

$\text{List} \rightarrow \text{lcbT StmtList rcbT}$ ,

that the symbol table for `List` is specified as the parent of the symbol table for `StmtList` (notice this new symbol table is initially empty. Finally, the new option for `Stmt`, the fact that it can be a `List`, is also accommodated – such a statement has no bindings for the local symbol table, so its entry matches that of `Assignment`.

CL Grammar Rule	Semantic Rules
List Rules	
◇ Program → List	List.bindList = <i>null</i>
◇ List → lcbT StmtList rcBt	StmtList.bindList = empty list StmtList.bindList.parent = List.bindList
StmtList → Stmt	Stmt.bindList = StmtList.bindList, StmtList.bindList.addTo(Stmt.bind)
StmtList → Stmt commaT StmtList	Stmt.bindList = StmtList <sub>1</sub> .bindList, StmtList <sub>1</sub> .bindList.addTo(Stmt.bind), StmtList <sub>2</sub> .bindList = StmtList <sub>1</sub> .bindList
Statement Rules	
Stmt → Declaration	Declaration.bindList = Stmt.bindList, Stmt.bind = Declaration.bind
Stmt → Assignment	Assignment.bindList = Stmt.bindList, Stmt.bind = <i>null</i>
◇ Stmt → List	List.bindList = Stmt.bindList, Stmt.bind = <i>null</i>
Assignment → identT assignT Exp	(n,t) = Assignment.bindList.lookUp(identT.name), Exp.bindList = Assignment.bindList, compatible(t, Exp.type)
Declaration → typeT identT	Declaration.bindList.hasNoEntry(identT.name), Declaration.bind = (identT.name, typeT.name)
Expression Rules	
Exp → Term	Exp.type = Term.type, Term.bindList = Exp.bindList
Exp → Exp (addT   subT) Term	Exp <sub>2</sub> .bindList = Exp <sub>1</sub> .bindList, Term.bindList = Exp <sub>1</sub> .bindList, Exp <sub>1</sub> .type = table(addT, Exp <sub>2</sub> .type, Term.type)
Term → Factor	Factor.bindList = Term.bindList, Term.type = Factor.type
Term → Term (multT   divT) Factor	Term <sub>2</sub> .bindList = Term <sub>1</sub> .bindList, Factor.bindList = Term <sub>1</sub> .bindList, Term <sub>1</sub> .type = table(multT, Term <sub>2</sub> .type, Factor.type)
Term → Term modT Factor	Term <sub>2</sub> .bindList = Term <sub>1</sub> .bindList, Factor.bindList = Term <sub>1</sub> .bindList, Term <sub>1</sub> .type = table(modT, Term <sub>2</sub> .type, Factor.type)
Factor → intT	Factor.type = int
Factor → floatT	Factor.type = float
Factor → identT	(n,t) = Factor.bindList.lookUp(identT.name), Factor.type = t
Factor → lpt Exp rpt	Exp.bindList = Factor.bindList, Factor.type = Exp.type

Figure 10.7: PDef Attribute Grammar

**The Front-end – Analysis Tutorials  
– Recognizing the PDef Language –**



# Chapter 11

## Tutorial Overview

The Front-end Tutorial focuses on the implementation of the analysis phase for a PDef translator – what we are referring to as a PDef recognizer. The tutorial is laid out as a sequence of mini-tutorials, with each focusing on the implementation of a particular front-end component: the tokenizer, parser, syntax tree generator, and semantic checker (and symbol table). The diagram in Figure 11.1 shows these components in the context of a UML class diagram for the front-end.

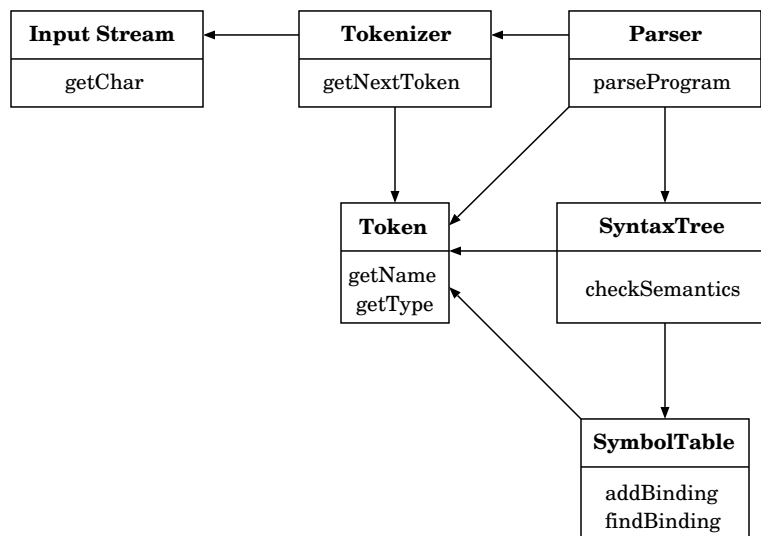


Figure 11.1: UML Class Diagram for a Translator Front-end

The Front-end Tutorial has as a central strategy to study a recognizer for a subset of PDef, called *PDef-lite*, and to extend it to an implementation for PDef. The heart of each tutorial is a step-by-step explanation of the implementation of the particular component for *PDef-lite*. At the end of each tutorial is a section titled Guided Development containing a sequence of activities leading to a PDef implementation. In the process of explanation the tutorials expose how theory links to implementation for each particular phase.

- The tokenizer tutorial discusses the implementation of a finite state machine for the regular grammar description of the tokens of *PDef-lite*.

- The parser tutorial discusses the implementation of a recursive descent parser based on the PDef-*lite* grammar.
- The syntax tree tutorial discusses the implementation of syntax tree classes appropriate for PDef-*lite* and the extension of the PDef-*lite* parser to implement the syntax tree.
- The semantic checker tutorial studies the implementation of a semantic checker for PDef-*lite* based on the attribute grammar for PDef-*lite* – this involves the implementation of the symbol table and syntax tree traversals that facilitate verifying semantic rules for PDef-*lite*.

The important underlying principle, which will be exposed in this sequence of tutorials, is the central role of the context free grammar as a design tool. In other words, having a formal description of the grammar leads, in a somewhat automatic way, to the final implementation. The rest of this chapter will include a formal description of PDef-*lite* and a discussion of programming strategies employed in the PDef-*lite* component to be discussed.

## 11.1 The Language PDef-*lite*

PDef-*lite* is a simple subset of PDef, retaining all the structure except for the assignment statement. The assignment statement in PDef-*lite* restricts the right hand side to being an identifier – general expressions are not allowed. The following are simple examples of PDef-*lite* programs.

```
{ int a, int b, b = a }
```

```
{ int this, float that, { float that, this = that }, { int this, that = this } }
```

```
{ int this, float that, { float that, { int this, that = this } }, that = this }
```

### The PDef-*lite* Tokens

You may have noticed that the languages PDef-*lite* and CL have the same set of tokens. Consequently, we use the regular expression from CL to describe the tokens of PDef-*lite*. Since we will be talking about PDef-*lite* in the context of implementation there is another issue to confront relative to the tokens for PDef-*lite* – in particular, how the tokens can appear in an input file. Since tokens have to be identified one at a time from an input stream it is critical to know what each token can be followed by. This has already been addressed in Section 2.1 (see page 18), where the spacing between tokens was discussed. In the table which follows we present the regular expression descriptions of the PDef-*lite* tokens and an indication of what characters can follow each token.

Token Class	Regular Expression	Termination Characters
commaT	,	any character
assignT	=	"
lcbT	{	"
rcbT	}	"
typeT	int   float	any nonletter
identT	[a - zA - Z] <sup>+</sup>	"



### The PDef-*lite* Grammar

While the tokens of PDef-*lite* are the same as those of the CL, the grammar must be changed to reflect the fact that a statement list can also be a list entry. While we could add the non-terminal **Program** to the rule for **Stmt** in the CL grammar, it adds clarity to the grammar if the start symbol **Program** does not serve a dual purpose. We therefore add a new structure to describe the statement list structure. The resulting grammar follows.

```

Program    → Block
Block      → lcbT StmtList rcbT
StmtList   → Stmt { commaT Stmt }
Stmt       → Declaration | Assignment | Block
Declaration → typeT identT
Assignment → identT assignT identT

```

### The PDef-*lite* Static Semantics

PDef-*lite* is a statically typed, block structured language, where blocks are described in the grammar by the non-terminal **Block**. The semantic rules for PDef-*lite* are as follows.

#### Types:

The only types in PDef-*lite* are the two that are built in, **int** and **float**.

#### Identifier Declaration:

1. An identifier can be declared only once in a block (statement list). Note that this is a restriction on declaration statements in a block, but does not prevent an identifier being re-declared in a nested block.
2. When an identifier is used in an assignment statement that statement must be preceded by a declaration for that identifier, in the same block or in an enclosing block. If there are two or more such declarations the one appearing in the same block or the one appearing in the nearest enclosing block will determine the type attribute of the identifier.

#### Assignment Compatibility:

In an assignment statement the type of the identifier on the right must be assignment compatible with the identifier on the left. The type **int** is assignment compatible with both **int** and **float**; the type **float** is assignment compatible only with itself.

## 11.2 Programming Strategies

This section is a catch-all for advice and advanced warning on implementation strategies that are encouraged or used in the two tutorials that follow. The section is divided into three parts. The first discusses useful software design strategies that are used in the software that accompanies the tutorials. The second concerns various aspects of Java that will be used extensively in the tutorials. The third discusses more detailed aspects of the IO and data structures available with Java that will be critical to the tutorials. The discussion here is preliminary, with details being added in the tutorials as needed.

### 11.2.1 Design Strategies

Because you will be writing a complex piece of software in the course of these tutorials there are some important programming principles that you should recognize and practice. Hopefully, you will find them useful enough to add them to your programming routine. Each principle has been given a name so that it can be easily referenced later.

**Theory is Best** – *When possible apply theory to the design process.*

There are many situations in problem solving where, with careful thought, one can see the problem as an instance of some theoretical result. The advantage of applying theory is that there is often a clear, known path to an algorithm that can be almost automatically generated.

A good example of this is the theoretical result that the language of a regular grammar can always be recognized using a finite state machine. Since token grammars are always designed to be regular, and since it is easy (we will see) to design a finite state machine for a given regular grammar, the theory says we can implement a tokenizer by implementing a finite state machine.

Another application of theory comes from the fact that a recognizer for a context free grammar can always be implemented as a set of mutually recursive methods. Again, the process of implementation follows a known pattern and derives directly from the structure of the grammar rules.

**Clarity First** – *Use the clearest possible algorithm to accomplish a task.*

There is a feeling amongst many programmers that programming is a creative act and that it is in complex algorithmic structures where the art is seen at its finest. While a common feeling, it is one that must be fought.

The feeling that complexity is better often derives from a perception that a particular bit of code is “inefficient,” usually from a point of view of time. If the complexity of the code is raised then the efficiency will go up. There is some logic in this reasoning. But a quest for complexity is misguided because it is in complex code where bugs hide and even multiply. The quest for efficiency is also misguided. If a simple algorithm is truly inefficient, it can always be recoded later, even in assembler language if necessary.

**Energize Design** – *Put your creative energies into designing rather than coding.*

This is really a principle derived from the first two. If one puts lots of effort into design, then clear and correct algorithms will result, sometimes from the clever application of theoretical results.

### 11.2.2 Java Features

There are two important features of Java which are used consistently in the tutorials, the Java package structure and Java’s exception handling mechanism. While this section is not meant as a review of these features, it will show how those features will be used in the tutorials.

## Packages

The Java package structure will be used throughout the tutorials as a mechanism for controlling the complexity of the software structure. At first they may seem not so useful, but by the end you will be glad to have used them. Each major component of the translator will be represented by a package, even if the resulting package has only one class in it. As an example, the class for the parser will be in a one-class package, while the syntax tree's package will have several classes. You might say this as an application of the "Clarity First" principle described above.

You should review the naming implications associated with packages if you have seen them before. If you are not familiar with packages you will find that their use is straightforward and the descriptive material in the tutorials will be enough to get you started.

## Exception Handling

Java's exception handling structure will be used throughout the tutorials that follow. You should be familiar with exceptions because Java requires their use in the context of opening file streams and reading input streams. In these situations, however, it is only the `try-catch` block that you must use; the exception classes are part of the Java class hierarchy and they are thrown from within Java class methods.

In the tutorials you will be using exceptions to signal when syntax and semantic errors are found. In this case you will have to not only catch and handle an exception, but will have to implement an exception class and instantiate and throw the exceptions as well. We will be evolving an exception hierarchy for the translator starting with the parser tutorial. As with packages, the Java requirements for exception handling will be explained as necessary in the tutorials. If you studied exceptions already in a programming languages course then adapting to the Java exceptions will be easy. If you haven't studied them yet, the introduction you get in the tutorials should provide the background necessary.

### 11.2.3 Useful Java Classes

When using an object oriented language such as Java there are certain kinds of activities that must be investigated initially because they are not handled in a uniform way by all languages. The two important examples for the tutorials are input/output and data structure processing. In this section we will take a look at the Java library classes, which will be central to our work. It will be useful for you to have access to the definitions in the Java class library, called the API, and Sun Microsystems has conveniently provided them at the following URL.

[docs.oracle.com/javase/6/docs/api/](http://docs.oracle.com/javase/6/docs/api/)<sup>1</sup>

The site has a convenient interface which makes finding any particular class quick and easy. It also gives good descriptions of the semantics of methods and details of required parameters.

---

<sup>1</sup>The URL was accurate as of August 2014. Subsequent versions of the library may cause a change to the URL.

## IO Classes

There are three IO concerns for the tutorial: opening and closing files, reading data from a file one character at a time, and putting back on the input stream a character that has already been read. This third concern may sound a bit strange but is crucial to the implementation of the tokenizer. If you were starting a translator project from scratch on your own you would have to look carefully through the class hierarchy for the particular class or classes that provide the necessary functionality. Fortunately, this has already been done. The classes mentioned below all belong to the Java `io` package. Here is a listing of the classes you will use and the relevant methods you will call.

### FileReader

The `FileReader` class makes it possible to create an object associated with a particular file and to read characters from that file. The `FileReader` constructor can be passed a string containing the name of a file. If the constructor has trouble opening the file for any reason it will throw the `FileNotFoundException` exception. This means that the call to the constructor must occur within a `try-catch` block. The `FileReader` class will only be used to create one of its objects – we will never read directly from the object. This is because the `FileReader` class doesn't provide a sophisticated enough reading mechanism. Instead we will pass an instantiated `FileReader` object to the constructor of the `BufferedReader` class.

### BufferedReader

The `BufferedReader` class provides buffered access to a character-input stream. More importantly, this class makes it possible to read a character and then put it back on the stream. This functionality is important to the proper functioning of the tokenizer's finite state machine. There are three methods you will use from this class.

The `read` method reads the next character from the input stream and returns it as an integer value. If the value is `-1` then the end-of-file has been reached. The return value can be converted to the `char` type by casting.

```
int v = fs.read(); // fs is the BufferedReader object
                // (fs stands for 'file stream')
char ch = (char) v;
```

The `mark` method marks the current position in the input buffer so that if subsequent characters need to be put back on the stream you can return to the marked position. The input stream won't know that anything has happened. So characters are not really put back, but rather the stream is returned to its marked state. The method takes one argument, the maximum number of characters which would be read before returning to the mark. For our situation that will always be 1 (one). Reading from the input stream will always be done as follows.

```
fs.mark(1);
int v = fs.read();
```

The third method we will use is the one for returning to the marked position in the input stream; it is named `reset` and takes no arguments. When called, `reset` causes the current position in the input stream to be reset to the previously set mark.

Because of the layered design we employ, it is only in the tokenizer that input is carried out. The `BufferedReader` object is created in the driver program class and passed to the tokenizer object. So the use of these classes and methods will be concentrated in the first tutorial of Tutorial 1.

## Data Structures and iterators

One of the primary data structures in the translator is the symbol table which is implemented as a Java linked list. Other data structures, such as the syntax tree, will be defined from scratch, so their functionality won't hinge on the Java class library. There are two things to investigate here: the linked list class and the Java version of iterators. If you haven't encountered iterators before, they are a transparent mechanism for traversing a data structure one element at a time, just as you might normally do in an array or linked list. It is up to the data structure class to provide the its own iterator, since it is only that class that knows about its internal structure. The classes in which we are interested are all members of the Java `util` package.

### LinkedList

The class `LinkedList` defines a list structure that can be used as a list, stack, queue, or double ended queue – so there is considerable flexibility here. We will use it as a simple sequential list, with elements being put on but never removed. We will make use of the *generic* feature for data structures. The idea is that if you want to put objects from a particular class on the list then the type of the list's elements should be part of the declaration. If `Entry` is the name of a class, then the name of a class for a linked list of such objects would be declared as follows.

```
LinkedList<Entry> list;
```

Java will make sure that the only kinds of elements added to `list` are those instantiated from the `Entry` class.

There are two methods we will use from the class. The `addFirst` method will add an element to the beginning of the list.

The `listIterator` method will construct and return a `ListIterator` object which gives access to the elements of the list from first to last. All access to the list will be via the iterator.

### ListIterator

The idea of an iterator has already been described. The class `ListIterator` provides a transparent mechanism for accessing in order the elements of a list. When instantiated, the iterator creates a list of references to the elements on the data structure. Then when requested it will return a reference to the next element on the list. There are two methods from the class that we will use.

The method `hasNext` not surprisingly returns a boolean value indicating whether there is another element on the iterator to be processed. When the list has been completely traversed the answer will be false.

The method `next` returns a reference to the next element in the iterator.

Here is an example to illustrate the use of the `LinkedList` and `ListIterator` classes. In this example we constructed a `LinkedList` object called `names` which contains objects of the class `Entry`. Here is code for doing a sequential search for a target `String` value called `name`.

```

        LinkedList<Entry> names = new LinkedList<Entry>();
        ..... // code for filling 'names'
        // 'name' has a target value of the class String
1   ListIterator<Entry> itr = names.listIterator();
2   Entry entry = null;
3   boolean found = false;
4   while (!found && itr.hasNext()) {
5       Entry ste = itr.next();
6       if ( name.equals(ste.getName() ) {
7           found = true;
8           entry = ste;
9       }
10  }
11  // entry == null OR entry carrying name

```

There are things to notice in this example. In line 1 an iterator object is instantiated from the list `names`; notice that the iterator's declaration includes reference to the type of element on the list. We then have a standard sequential search algorithm in which `itr.hasNext()` indicates if there are more elements to be processed (line 3) and if there are, then `itr.next()` gets the next reference (line 4).

#### 11.2.4 Debugging Framework

A compiler is a complex software system whose development is fraught with problems. There are times when it is useful to get feedback from the tokenizer or from the parser to understand exactly why the current version is not behaving properly. If debugging isn't planned from the beginning it will never be added on, and a debugger will be the only alternative. A debugging framework, on the other hand, requires little effort and can provide flexible, command-line driven access to execution time information. In this section we describe the architecture of an object-oriented debugging framework that will prove useful in these tutorials and in other projects described in this book.

The design of the system hinges on the following strategy. When the system is executed the user can enter a command line flag, say the character 'x', which has been designated to signal a particular component of the system to turn on the x debugging flag. Debugging for each component is controlled by a class `Component2Debug`. In each class associated with the component there will be a declaration as follows.

```

Component2Debug debug = Component2Debug()

```

The constructor for `Component2Debug` will have the character 'x' built in, so there is a clear connection between the component and the command line flag. This debugging class also has a method `show` that can be used as follows at strategic places in the component.

```
debug.show("some message");
```

The idea is that this method call should only produce output when the character ‘x’ actually appears on the command line – if it does not, these messages should not be printed. This functionality is controlled by the class `Debug`, which is the super class of `Component2Debug`.

The `Debug` class controls the execution of the component `show` methods. To do this `Debug` has three static data members to keep track of what commandline flags are detected and what flag codes (characters) are assigned to each system component. Here is the `Debug` class definition.

```
public abstract class Debug {
    private static String commandLine = "";
    private static int flagNum = 0;
    private static ArrayList<Boolean> flags = new ArrayList<Boolean>();

    public static void registerFlag(char ch)
    // Pre:  ch is a commandline character to be registered --
    // Post: commandLine.indexOf(ch) != -1
    {
        commandLine += ch;
    }
    public int registerObject(char ch)
    // Post: psn == (in)flagNum AND psn >= 0 AND flagNum == (in)flagNum+1
    //       flags.get(psn) == ( commandLine.indexOf(ch) != -1 ) AND
    //       return psn
    {
        int psn = flagNum++;
        flags.add(psn,  commandLine.indexOf(ch) != -1 );
        return psn;
    }
    public void show(int psn, String msg)
    // Pre:  0 <= psn < FlagNum AND
    //       psn is a registration number passed by
    //       an instance of a subclass of Debug
    {
        if (flags.get(psn) )
            System.out.println(msg);
    }
}
```

We focus on the three class methods. `registerFlag` is a static method and is called by the system’s `main` method when a commandline flag is detected – the flag character is passed as an argument. Each time `registerFlag` is called a new character is tacked onto the static string `commandLine`. Notice, this method is called independently of any of the component debug classes.

Rather than discuss the other two methods now, we will discuss them in the context of the class definition for `Component2Debug`, where they are called.

```
public class Component2Debug extends Debug {
    private int regPsn;
    public Component2Debug() {
        regPsn = registerObject('x');
    }

    public void show(String msg) {
        String str = msg;
        show(regPsn, str);
    }
}
```

Notice that when the class constructor is called the inherited method `registerObject` is called with the character 'x' as argument – this is an indication that 'x' has been assigned explicitly to this component. It is possible that 'x' could be assigned to more than one component. Look at the code for `registerObject` in `Debug` and notice that there is a boolean array list to hold the value of each debug flag. There are two things that happen when `registerObject` is called. First, an array position is allocated to the registered character (in this case 'x'); then, the flag in the allocated position is turned on or off based on whether the character 'x' appears in `commandLine` – this requires all commandline arguments to be registered before any component debug class is instantiated. The result of `Component2Debug`'s registration is that `Debug` returns the assigned registration number to the constructor – it is saved locally in `regPsn`. Now notice that if `Component2Debug`'s method `show` is called, it will call `Debug`'s `show` method (inherited) with the value of `regPsn` as an argument. In this way `Debug.show` will actually display something only if the flag at `regPsn` is set.

This registration strategy has the advantage that if a new component is added and needs to provide debugging information, a new commandline character can be assigned and a component debug class implemented (like `Component2Debug`) without modifying any existing code – this is an important benefit of object oriented programming.

This debugging framework will be used in the components of the PDef recognizer to be implemented in these tutorials. At a minimum you should add entry and exit debug statements to each method of a component with the debugging capability. Use the following model.

```
public AType myMethod(parameters) {
    debug.show("--->>> Entering Component2.myMethod");
    .
    .
    debug.show("<<<--- Leaving Component2.myMethod");
}
```



## Chapter 12

# PDef Tokenizer Tutorial

**Goal:** To complete the design, implementation and testing of a tokenizer for PDef.

**Code Base:** See Figure 12.1.

**Assumptions:** The tutorial assumes that you have read Chapters 1, 2, 5 and 11.

**Strategy:** In this tutorial you will start with a skeleton of a tokenizer and proceed in two steps. The tutorial will direct you to complete a tokenizer for PDef-*lite* and then, via the exercises at the end, to extend it to a tokenizer for PDef.

### The Tutorial

In this tutorial you will complete the implementation of a tokenizer for the language PDef. The tutorial will proceed by discussing a bare bones tokenizer which you will be guided to transform into a tokenizer for PDef-*lite*. At the end of the tutorial the Guided Development will guide you to complete a tokenizer for PDef.

We talked extensively in Chapter 1, especially in Sections 1.4-1.5, about the structure of a translator, and the functionality of the tokenizer. The boxed region of the UML class diagram in Figure 12.2, shows the class structure of a tokenizer.

Recall that the purpose of the tokenizer is to transform the character input stream into a token input stream that can be conveniently used by the parser. The input capability of the tokenizer is represented in the class `Tokenizer` by the public method `getNextToken` and it is this method that the parser calls in order to read a new token. The method `getNextToken` provides its functionality by reading a sequence of characters from a specified file stream object, creating an appropriate new `Token` object, and returning that token to the caller.

Code Base	
File	Implements
<code>tokenizer/Token.java</code>	This file in the <code>tokenizer</code> package implements the <code>Token</code> object depicted in the UML class diagram in Figure 12.2.
<code>tokenizer/Tokenizer.java</code>	This file in the <code>tokenizer</code> package implements the <code>Tokenizer</code> class depicted in the UML class diagram in Figure 12.2. It contains a static method <code>main</code> , which is intended for testing of the tokenizer implementation.
<code>test</code>	This is an input file which can be used for testing the tokenizer.
package	
<code>debug</code>	This package contains the definition of the class <code>Debug</code> and the definition of its one subclass <code>TokenizerDebug</code> . Together they provide the basic debug framework for the tokenizer.
<code>tokenizer</code>	This package contains the complete code for a working version of the PDef tokenizer.

Figure 12.1: Java Tokenizer Components

## 12.1 Tokenizer Structure

Before going too far we will examine more closely the structure implied by the UML class diagram above, so as to extract a bit more detail about the implementation of the tokenizer components.

### 12.1.1 Token Object Structure

The token object is the simplest kind of object; that is, it encapsulates other data values in order to provide a layer of abstraction, and its state is set once and then remains constant. Such data objects are common in object-oriented programming and have a predictable form: a set of private data components (the encapsulated data) and public accessor methods for each of the data components. The initial state is set when an object is instantiated.

What can we deduce about their required structure based on our knowledge of how the token objects will be used? Remember that the tokens of a language actually represent token classes, that is, each token represents a collection of token values that are interchangeable at the grammar level. From the parser's point of view, knowing the name of the token class is sufficient for checking

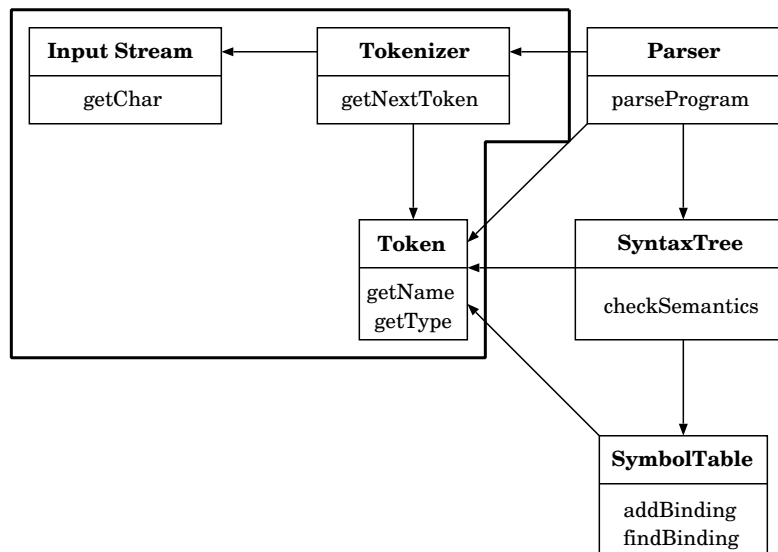


Figure 12.2: UML Class Diagram for a Translator Front-end

syntax correctness. But from the point of view of the semantic checker or code generator, two different identifier names, for example, cannot be treated in the same way; the specific token value, i.e., the identifier name, must also be available. So, the state of a token object should include at least the name of the token class, which we will call `type`, and the character string read from the input character stream for the token, which we will call `name`. Adding accessor methods for each of these data components gives us the structure summarized in the UML class diagram in Figure 12.3.

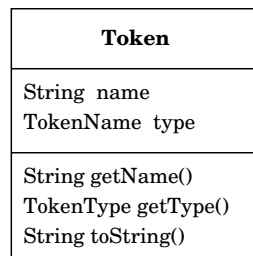


Figure 12.3: Token Object Structure

### 12.1.2 Tokenizer Object Structure

As indicated earlier the purpose of a `Tokenizer` object is to allow easy access by the parser to the tokens on the input stream. This is an example of IO layering, where layers of abstraction are implemented to facilitate high-level IO operations. In the case of the PDef front-end it is the method `getNextToken` which is the highest level of IO abstraction, where the input stream is seen as a stream of token values. A lower layer of abstraction is provided by the input class `BufferedReader` and its method `read`, which sees the input stream as a stream of ASCII characters. This layering is accomplished in the class `Tokenizer` which contains a reference to a `BufferedReader` (the input

stream) and the public method `getNextToken`. The instance variable `debug` provides access to a `TokenizerDebug` object, which completes the basic structure of `Tokenizer`. The UML class diagram in Figure 12.4 shows these structures in the context of the front-end.

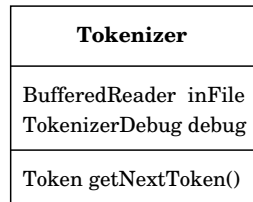


Figure 12.4: Tokenizer Object Structure

### The hidden part of `Tokenizer`

The tokenizer is implemented in order to simplify the job of the parser, which is only concerned with the sequence of tokens on the input stream provided via calls to `getNextToken`. So `getNextToken` is an abstract operation which provides the parser the tokens it needs, but hides from the parser details of how the tokens appear on the input stream. But there's another abstraction that can fit between the method `getNextToken` and the input stream named `inFile`. It's important to recognize that `getNextToken` is interested in most but not all the characters that are on the input stream: for example, it doesn't care about tab and end-of-line characters and it doesn't really care about the end-of-file either. To hide these details from `getNextToken` another layer of abstraction is added in the form of a private `Tokenizer` method named `getChar`. The idea is for `getNextToken` to call `getChar` when it needs the next character.

The responsibility of `getChar` is to hide the input stream details and return the next interesting character when asked. For example, since the tab character and the end-of-line character are not relevant to the next token value, `getChar` can watch for these characters and return a blank in their place. The method `getChar` can also hide details of detecting the end-of-file by returning a special character when the end-of-file condition is detected; doing this means that `getNextToken` never has to worry about reading past the end-of-file. In this way, the `getNextToken` method never has to worry about anything but interesting characters – never about the particularities of the input stream implementation.

Even though the initial configuration of the method `getChar` may seem to be rather simple, having it in place provides a convenient place to implement character-level information or processing that we would rather hide from `getNextToken`. For example, we could enhance `getChar` to keep track of line numbers and character positions on a line. In this way we could provide with each token an indication of the line on which it occurs and the character position on the line where its first character appears.

The diagram in Figure 12.5 illustrates how the addition of the `getChar` method adds an additional layer of input abstraction.

Based on the structures we have discussed for the tokenizer and token objects, we display in Figure 12.6 an updated UML view of the tokenizer structure. Again the parser object is included to emphasize that it is the client of the tokenizer object.

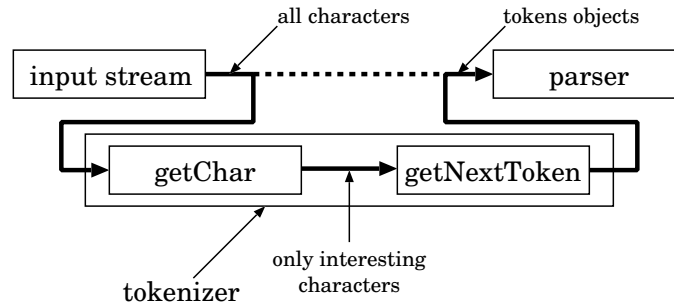


Figure 12.5: Data Flow from Tokenizer to Parser

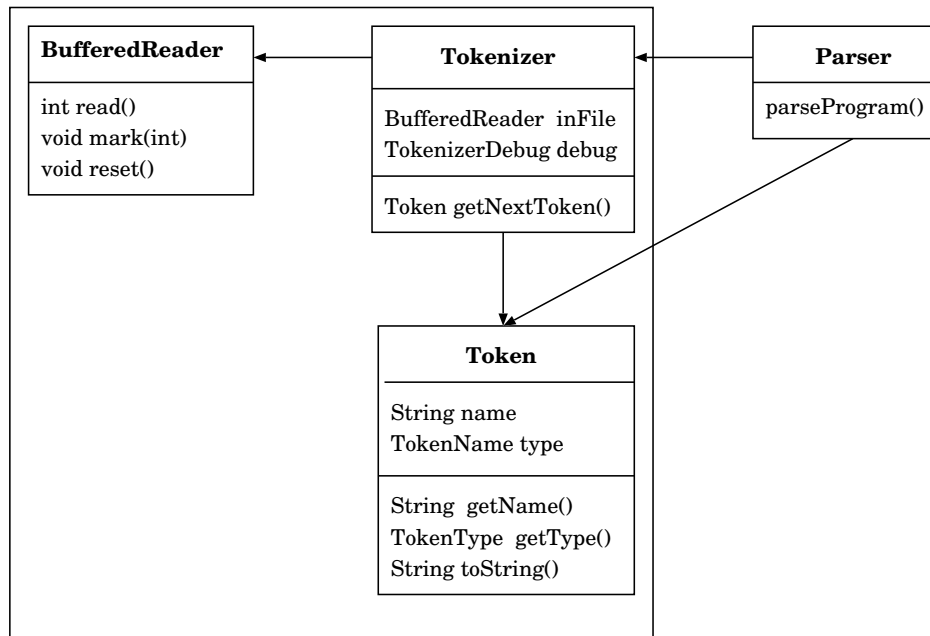
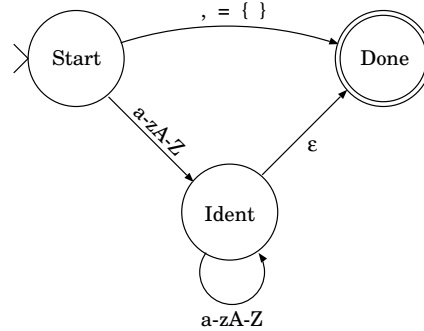


Figure 12.6: Expanded Object Structure of the Tokenizer

### 12.1.3 Theory Versus Practice

The finite-state machine on the right of Figure 5.6 (see page 70) accepts the tokens of the language PDef. It is easy, then, to reduce that machine to a finite-state machine for PDef-*lite*, and that reduction is represented in Figure 12.7. While this machine is fine from a theoretical point of view, there are other issues to discuss to make a PDef-*lite* finite-state machine that can drive an implementation.

One situation is the following. In talking about a finite-state machine we consider a token to be recognized if we end in the Done state having consumed all the input characters. But in a translator we have a long string of characters in which many tokens are embedded. So we need another criterion for what it means to recognize a token. For tokens of a fixed length there is no problem. When we reach the Done state it is because a particular token was seen, and whatever character is next on the input stream will be the first character considered in reading the next

Figure 12.7: PDef-*lite* Finite-state Machine

token. But for an identifier, where the length is variable, what will be the criterion? Here the peek-ahead capability is critical. Basically, if we peek ahead and find anything other than a letter we will transition to the `Done` state and that mysterious other character we peeked at will be the first character of the next token.

There are other details of implementation that are also important to consider before continuing and we discuss them in the following list.

**White-space Characters** In a text file, it is necessary to separate interesting stuff (i.e, the characters we are interested in) with what are called “white-space” characters – blank, tab, new line. These three characters are in fact of no interest to us and we want to skip over them when we encounter them.

The strategy followed in translators is two fold. First, the function `getChar`, which is responsible for reading the next character from the input stream, can return a blank whenever it sees any white-space character. Since we aren’t interested in distinguishing them there’s no need to worry the finite-state machine with them. In this way the finite-state machine only has to worry about blanks.

Second, we often find sequences of white space characters; programmers use them in this way to give the text a more pleasant appearance when viewed. Since these characters are in the input we have to deal with them in the finite-state machine. The easiest way is to use the start state as a white-space skipper — in other words, if a blank is encountered then read another character and return to the start state. It is important to realize that this is the only processing we need to do for white-space. You don’t have to check for it after an identifier, for example, because we are really letting anything other than a letter terminate an identifier. If there are a bunch of white space characters after an identifier, they will be skipped at the beginning of the next token.

## Keywords

We have been ignoring the problem of recognizing keywords. There are two reasons. First, to handle keywords directly in the finite state machine is very messy – it’s possible, but you don’t want to do it. To illustrate, the finite-state machine in Figure 12.8 is simply the diagram of Figure 12.7 modified to include the direct identification of the keyword `int`. This doesn’t look too bad, but imagine adding

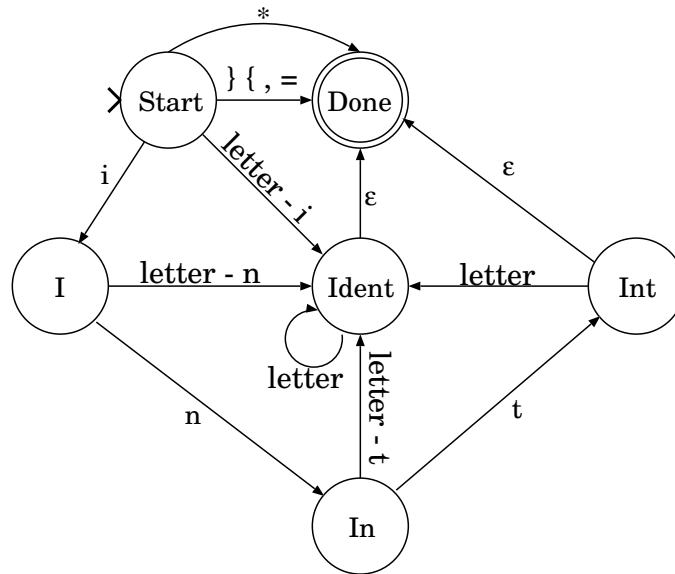


Figure 12.8: PDef-lite Finite-state Machine – Identifying the Keyword `int`

new states and transitions to directly identify `char` and `float`.

The second reason is that there is a much easier solution at hand. The solution is to check for keywords once an identifier has been found. In other words, in the state `IDENT_S` when the  $\epsilon$ -transition is taken, call a method and pass the `name` object as a parameter. The method will return the token type of the name. If `name` contains one of the two type names, `int`, `float`, then `TYPE_T` will be returned, otherwise `IDENT_T` will be returned.

### $\epsilon$ -transition

Implementing an  $\epsilon$ -transition presents a fundamental problem. When that transition is taken the machine has to have seen the next character on the input stream. If we simply read that next character to determine what to do next (in the state `Id`) we will miss that character when `getNextToken` is called the next time. So it is necessary to take some action so that the character can be read again.

Here's an example.

```
sum=total
```

We start the finite-state machine in the state `Start` at the beginning of the string, i.e., with the character `'s'`. In reading the characters `'u'` and `'m'`, we will transition to the state `Id` and then stay in that state. When we see the character `'='` we know that we have seen the identifier `'sum'` and transition to the state `Done`. But we have also read the character `'='`. When the finite-state machine is activated next it will begin with the character `'t'` – the character `'='` has been lost.

So, in the state `Id` we have to be able to look at the value of the next character without disturbing the input stream too much. The standard operations for doing this are the `'peek'` operation and the `'read/put back'` combination. By doing one of these we will implement the  $\epsilon$ -transition properly.

## Handling Errors

One principle that will emerge in the remainder of the tutorial is that the tokenizer, via its method `getNextToken`, should always return a token, even if bad characters have been seen. So our solution for the bad character problem is to define an error token type and return that for each bad character encountered. To be more precise, we will adopt the following as the second token identification rule.

**Error Tokens:** Sequences which cannot be identified as legal tokens are classified as error tokens. A single character not in the alphabet is considered a fixed length error token. In this way it can be seen as a terminator for variable length tokens. Non-token sequences of legal characters are treated as variable length tokens.

## At the End-Of-File

Remember that the `Tokenizer` is supposed to make the input stream appear to be a stream of tokens. But what should happen when we get to the end of the file? There are two issues here. First, what should `getChar` do and second what should `getNextToken` do?

Remember that in `getChar` the end-of-file indicator is the integer value -1. This can't be easily turned into a character. But `getChar` has to return some character. We will choose to return the character whose ASCII code is 0. To make things easy we define a `char` constant whose name is `eofChar` and whose value is `(char)0` – that is, the integer value 0 cast to type `char`.

Now, what should `getNextToken` do when it sees the `eofChar` character? It has to return some kind of token. If it tries to ignore this character, as it does the blanks, then on the next call to `getChar` it will just get the same character again, since we are at the end-of-the-file. The answer is to generate a special end-of-file token, `EOT.T` – the token name is meant to denote to the parser “the end of the token stream.” In this way the parser (yet to be discussed) can ask for tokens even if the end-of-file has been reached – the same token will be given each time.

In Figure 12.9, a diagram of the augmented finite-state machine we have just been discussing is shown. Notice that the white-space-skipping has been added to the start state in the form of the ‘blank’ transition and also that the `eofChar` will drive the machine from the start state to the done state. Finally, the asterisk ‘\*’ on the middle transition from `Start` to `Done` represents the recognition of error tokens – the asterisk is interpreted to mean “any character not represented on another link from this state.” This is a convenient notation generally in drawing finite-state machines.

The one thing we have been resisting is a discussion of actions to be taken with state transitions. We have chosen to wait to discuss transition actions until we completely understand the simple recognition problem. We will discuss the transition actions in detail when we deal with the tokenizer implementation.

### 12.1.4 Implementation Strategy

For convenience we display again, in Figure 12.10, the algorithmic structure discussed in Section 5.3 for a finite-state machine implementation.



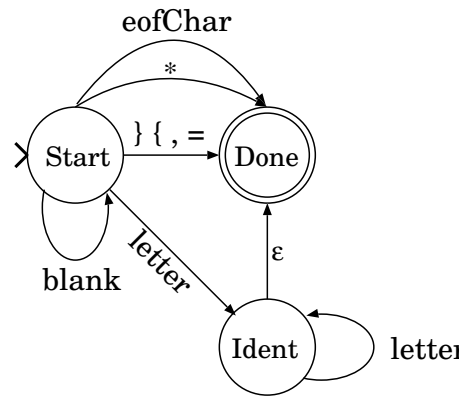


Figure 12.9: The PDef-lite finite-state machine

```

state = Start;
while (state != Done) {
    ch = input.getSymbol();
    switch (state) {
    case Start: // select next state based on current input symbol
    case S1:    // select next state based on current input symbol
        ...
    case Sn:   // select next state based on current input symbol
    case Done: // should never hit this case!
    }
}

```

Figure 12.10: FSM Algorithm

### A cautionary note

When you look at the finite-state diagram in Figure 12.9, you see a transition from the state IDENT\_S back to itself if a letter is seen. This may lead you to think that it would be appropriate to have a loop as the action in the implementation of the state IDENT\_S, perhaps as follows.

```

case IDENT_S:
    while (Character.isLetter(ch)) {
        ch = getChar();
    }
    // put the last character back (for epsilon-transitions)
    state = DONE_S;
    break;

```

In fact this is a bad idea on two scores. First it introduces a nested loop, which always increases the perceived complexity of the code. This kind of complexity should not be introduced unless absolutely necessary. Second, adding the nested loop reflects a drifting away from our use of the finite-state machine as theoretical basis for the implementation. There is already a looping mechanism in the

algorithm so embedding another in the state `IDENT_S` is simply not necessary. You should recognize this as an application of the “Theory is Best” principle from Section 11.2.1. The more appropriate code would have the following form.

```
case IDENT_S:
    if (Character.isLetter(ch)) { state = IDENT_S; }
    else { // put the last character back (for epsilon-transition)
        state = DONE_S;
    }
    break;
```

Notice that if a letter is seen we will loop back to the top of the `while` structure, read the next character, and then return to this same state `IDENT_S`.

It has been said before, but it is important to emphasize that each time we invoke `getNextToken` we start again in the state `START_S` and traverse a sequence of states, based on the characters on the input stream, finally reaching the state `DONE_S`.

## 12.2 The Supplied Java Code

We are now ready to embark on our implementation of the *PDef-lite* tokenizer. The first step is to review the supplied code base. What you will find is that, except for the methods `getChar` and `getNextToken`, the code base is relatively straightforward. It is all reviewed in this section. At this time you should review the files contained in the code base – they are listed in Figure 12.1.

### ☛ Activity 30 –

Before looking at the supplied code you will execute it. In this way you can compare the output from the executions with the code as it is discussed.

Compile and execute the driver program in `PDef.java` using the following sequence of commands:

```
% javac PDef.java
% java PDef test
% java PDef test t
```

In following this sequence you will actually run the tokenizer twice, once in normal mode and second in debug mode, thanks to the command-line option ‘`t`’. Notice in the second run you get a listing of the sequence of states of the finite-state machine visited in processing the data file.

Based on the output from the program draw a finite-state machine that seems to describe the tokens being recognized. Don’t make this too difficult – the finite-state machine is extremely simple.

A look at the class definitions contained in these files will reveal a bit more structure. We will now take a look at the structures one component at a time including and look at the Java code supplied for this tutorial.

### 12.2.1 class Token

Figure 12.11 displays the code for the class definition in `Token.java`. There are a couple of things to highlight in the code. First, the code contains a definition of an enumerated type called `TokenType` that defines the collection of token class names. In addition, there is a method `getName` in the class which returns the value of the name string stored in the token. The second thing to highlight is the additional method, `toString`, which hasn't been mentioned before. We will include a method with this name in each class we define; it is used to provide a `String` representation of the value of a particular object of the class. In particular, the method returns a string containing the name of the type of token followed by the character string containing the token name from the input stream — this name is contained in parentheses. For example, if the identifier `"test"` appears on the stream, the corresponding token will be of type `IDENT_T` and `toString()` would return `"IDENT_T( test )"`.

```
public class Token {

    public enum TokenType { IDENT_T, TYPE_T, ERROR_T, ASSIGN_T, RCB_T, LCB_T,
                           COMMA_T, EOT_T };

    private TokenType type;
    private String name;

    public Token(TokenType t, String s) {...}
    public TokenType getType() {...}
        // Pre: type has a value
        // Post: return type
    public String getName() {...}
        // Pre: name has a value
        // Post: return name
    public String toString() {...}
        // Pre: type and name have values
        // Post: return string containing character form of
        //       "type(name)"
}
```

Figure 12.11: Token Class Java Code

### 12.2.2 class Tokenizer

We have already given a good high-level description of the tokenizer object and how it functions. A look at the code supplied for this tutorial shows an implementation that matches the structure indicated in Figure 12.6. The skeleton of the Java definition for the class `Tokenizer` appears in Figure 12.12.

The important and interesting methods of the class are the methods we have discussed quite a bit already, `getChar` and `getNextToken`. We will now look at the code for these methods as it appears in the supplied Java code base.

```

public class Tokenizer{

    private static final char eofChar = (char)0;

    private enum StateName { START_S, IDENT_S, DONE_S };

    private BufferedReader inFile;
    private boolean      echo = false;
    private TokenizerDebug debug;

    public Tokenizer (BufferedReader in, boolean echo) {...}
    // Post: inFile == in, this.echo == echo, debug == new TokenizerDebug()
    private char getChar() {...}
    // Pre:  ch is the character at the head of inFile
    // Post: inFile is original inFile with ch removed AND
    //       return ch -- Except
    //       if inFile.eof is true return eofChar
    //       if ch is eol return blank character
    public void putBackChar(char ch) {...}
    // Pre:  inFile has a value
    // Post: inFile is the original inFile with ch added as its first character
    public Token getNextToken() {...}
    // Pre:  inFile has a value
    // Post: inFile has initial blanks removed as well as the characters
    //       of the next token on inFile.  The tokens are determined by the
    //       finite-state machine for the following regular expression
    //       , | = | { | } | [a-zA-Z]+
    //       NOTE: 'int' and 'float' are key words and are not allowed in [a-zA-Z]+
}

```

Figure 12.12: Tokenizer Class Skeleton

### getChar

The first method we look at is `getChar`, whose code is displayed in Figure 12.13. The purpose of the method is to hide details not only of the input stream but also of how the input stream is manipulated. There are three important things to know about the code. First, since we are doing input there is the possibility of an exception being thrown. So we surround the code that manipulates `inFile` with a `try...catch` block. If an `IOException` is thrown, then the program will terminate with a message.

The second thing to know is how character input works at this level. We want to make sure that we get access to every character appearing on the input stream, so we use the `BufferedReader` method `read`, which returns an integer value representing the character that was read. Because our two PDef languages are free form, we don't really care about where the end-of-line characters occur; similarly, tab characters are of no importance to the languages. So we use `getChar` to filter these, which means that we replace each occurrence by the space character; this filtering is handled by the selection statement in the `else`-clause. Another input detail is the fact that when the end-of-file

is reached `read` will return the value `-1`. This is convenient since no input error will occur due to reading past the end-of-file. Since `-1` isn't a character value, we will have `getChar` return ASCII 0 in this case – we give this value the name `eofChar`.

The third thing of which we must be aware is the fact that there will be times when the tokenizer, via the method `getNextToken`, will need to return a value to the input stream – i.e., when there is an  $\epsilon$ -transition. This is facilitated by the call in `getChar` to `mark(1)`, which tells the object `inFile` to retain one character when it is read. This saved character will be the next character read only if another action is taken in the `putBackChar` method.

```
private char getChar()
// Pre:  ch is the character at the head of inFile
// Post: inFile is original inFile with ch removed AND
//       return ch -- Except
//       if inFile.eof is true return eofChar character
//       if ch is tab or eol return blank character
{
    char ch;
    int v = 0;

    try { inFile.mark(1); v = inFile.read(); }
    catch (IOException e) {
        System.out.println("Input problem!");
        System.exit(0);
    }

    if (v == -1) ch = eofChar;
    else {
        ch = (char)v;
        if (ch == '\n' || ch == '\t') ch = ' ';
    }

    return ch;
}
```

Figure 12.13: `getChar` Code

### putBackChar

The `putBackChar` method is a necessity for the case where an  $\epsilon$ -transition is specified in a finite-state machine. As is the case with `getChar`, the method `putBackChar` is meant to be used by `getNextToken` and to hide the specific characteristics of the input stream. In the case of Java, the class `BufferedReader` provides a mechanism for putting a character back on the input stream. This mechanism involves the two methods `mark` and `reset`. The call `inFile.reset()` tells the object `inFile` to reset itself to its condition before the last call to `mark`. Since in `getChar` the call to `mark` was with a parameter 1, only the last character read will be returned to the input stream.

You will notice in the code for `putBackChar`, displayed in Figure 12.14, that `reset` is called only

if the end-of-file has not been reached. That's because `putBackChar` is only called by `getNextToken`, and the only way that `getNextToken` can have the `eofChar` character is if `getChar` detected the end-of-file. Since the purpose of `putBackChar` is to return the input stream to its status before the last read operation, a request to put back the `eofChar` character should result in no action – this will guarantee the input stream's status is still end-of-file.

```
public void putBackChar(char ch)
// Pre: inFile has a value
// Post: inFile is the original inFile with ch added
//       as its first character
{
    debug.show("--->>> Entering putBackChar");

    try { if (ch != eofChar) { inFile.reset(); } }
    catch(IOException e) { System.exit(0); }

    debug.show("<<<--- Leaving putBackChar");
}
```

Figure 12.14: `putBackChar` Code

### getNextToken

The heart of the tokenizer is the method `getNextToken`. This method is called by the parser each time it needs the next token from the input stream. The code for this method that accompanies this tutorial can be seen in Figure 12.16. It implements the very simple finite-state machine depicted in Figure 12.15.

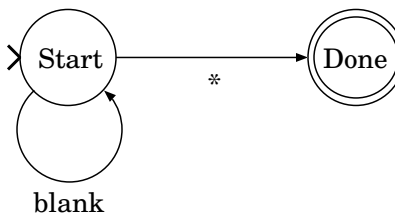


Figure 12.15: Finite-state machine for the supplied `getNextToken`

Obviously, the heart of the method `getNextToken` is the `while`-loop and the `switch`-statement it contains. Each `case` of the `switch` implements a particular state of the finite-state machine and the loop drives the machine's operation. Notice that the variable `state` is initialized at the start state, `START_S`, and the loop continues to function until the final state, named `DONE_S`, is reached. Each pass through the loop is meant to simulate the visit to a particular state, which means some action is taken and a new state is determined. The details of this structure will occupy the bulk of this tutorial.

Another important detail in the `getNextToken` code is that there is only one input statement, the very first statement in the `while`-loop. Since the next state to be visited is determined by the

current state and the next input character, this is the input where that next character is obtained. This is the only place in the finite-state machine implementation where input should occur. The only other interaction with the input stream occurs as an action in a particular state when a character must be put back on the input stream via a call to `putBackChar`.

Another detail should be mentioned at this point. The last statement of the method returns a reference to the token object representing the token read from the input stream. Each of the two parameters to the `Token` class constructor must have a value. The value of `type` is determined in a particular state just before going to the `DONE_S` state. The `name` variable is meant to hold the string of characters making up the token just read. The value of `name` is accumulated one character at a time as the states are traversed. In each state one of the actions will be to appropriately add a character to `name`. But be warned, this accumulation can only occur if the character is not going to be put back on the input stream. You can see in the final `else`-clause in the `START_S` state's code that new values are assigned to both `type` and `name`. A more detailed description of how to deal with these variables will be given in the remainder of the tutorial.

### 12.2.3 Debug and TokenizerDebug – debug classes

In the package `debug` you will find the class definitions for `Debug` and `TokenizerDebug`, as described in Section 11.2.4. The class `TokenizerDebug` code is here.

```
public class TokenizerDebug extends Debug {
    private int regPsn;
    public TokenizerDebug()
        // Post: regPsn == registration value returned by registerObject AND
        //         regPsn >= 0
        { regPsn = registerObject('t'); }
    public void show(String msg) { show(regPsn, msg); }
    public void show(String msg, char ch) { show(regPsn, msg + "(" + ch + ")"); }
}
```

Notice in this class that the method `show` takes a second parameter which, presumably, is the current input character. This means that when the `show` method is called, a message plus the character will be displayed. This can be useful during the debugging process, as illustrated in the implementation of `getNextToken`, which was discussed above.

### 12.2.4 PDef.main – test driver

The method `main` is a driver to test the tokenizer implementation and illustrates how test drivers will be structured for each component as we go along. The method takes one required command-line argument and can take an optional argument. The code for `main` is shown in Figure 12.17.

The first thing to notice in `main` is that there are two phases of operation in the method. The first segment of code processes any command-line arguments and instantiates the input stream. The name of the file containing the input data is given as the first command-line argument. The second command-line argument, if it is present, signals the program that debugging output should

```

public Token getNextToken()
{
    debug.show("--->>> Entering getNextToken");

    StateName      state = StateName.START_S;
    Token.TokenType type = Token.TokenType.ERROR_T;
    String          name  = "";
    while (state != StateName.DONE_S) {
        char ch = getChar();
        switch (state) {
        case START_S:
            debug.show("\t==>> Entering state -- START_S: '", ch);
            if (ch == ' ') {
                state = StateName.START_S;
            }
            else if (ch == eofChar) {
                type = Token.TokenType.EOT_T;
                state = StateName.DONE_S;
            }
            else {
                name += ch;
                type = Token.TokenType.IDENT_T;
                state = StateName.DONE_S;
            }
            debug.show("\t<<== Leaving state -- START_S: '", ch);
            break;
        case IDENT_S:
            debug.show("\t==>> Entering state -- IDENT_S: '", ch);
            // new code goes here!
            debug.show("\t<<== Leaving state -- IDENT_S: '", ch);
            break;
        case DONE_S: // Should never get here! For completeness.
            break;
        }
    }

    Token token = new Token(type, name);

    debug.show("<<<--- Leaving getNextToken");
    return token;
}

```

Figure 12.16: getNextToken Code

be activated for the corresponding component – in this case the class `Tokenizer`. Actually what happens is that `main` asks `Debug` to register the flag ‘t’ as having been seen.



Look closely at the `for` loop inside the `try...catch` block in `main`. This loop pulls apart the second command-line argument and uses the letters in that argument to set either the `echo` flag or to call the registration method from `Debug`. This segment of code will expand with each new component, allowing the user to set or not set debugging flags for each component via the second command-line argument.

The second phase of operation starts with instantiating the `Tokenizer` object; notice that the constructor takes the input stream object reference and the flag `echo` as parameters. After instantiating the `Tokenizer` object the program begins reading the sequence of token values from the input file, continuing until the token denoting the end of the token stream, `EOT.T`, is read.

It is very important that you study and understand this `PDef` class because it will be the basis for each phase of the `PDef` translator development. Each of the tutorials will use a modified version of this class and you will have to make adjustments to it from time to time. **STUDY IT!!**

## 12.3 Completing the *PDef-lite* Tokenizer

At this point we have reviewed the structure of finite-state machines and a strategy for their implementation. The tutorial now begins in earnest. This section consists of a sequence of activities that, when completed, will result first in a working tokenizer for *PDef-lite* and finally in a working tokenizer for the full `PDef` language. The tokenizer for the full `PDef` language will become an integral component of the code base for the parser tutorial that follows.

### 12.3.1 Implementing the simple *PDef-lite* tokenizer

Having established the structure of the finite-state machine algorithm, we should be able to write out an implementation for the transition structure for the *PDef-lite* finite-state machine. We basically apply the information contained in Figure 12.9 directly to the template laid out in Figure 5.7, creating new state names as needed. The resulting partial Java implementation for `getNextToken` is in Figure 12.19. Notice that for each state and transition from that state there is a selection block in the switch statement that sets the next state value. To make life easier the *PDef-lite* finite-state machine is displayed again in Figure 12.18.

In the state `START_S` the blank character is handled separately. The method `special` is meant to return `true` when the value of `ch` is one of the four characters `' = { } , '`.

Also notice that in the state `IDENT_S` we have applied the rule for dealing with an  $\epsilon$ -transition, discussed in Section 12.1.4. It is, of course, the call to `putBackChar` that facilitates handling this situation.

#### ☛ Activity 31 –

Edit the Java file `Tokenizer.java` and modify the code in the `getNextToken` method so that it matches the code in Figure 12.19. Notice that very little actually changes – only the code for the state `START_S`. DON'T compile this code since it is not complete; all you are doing at this point is setting up a framework for completing the *PDef-lite* version of `getNextToken`.

---

## Transition Actions

As indicated earlier there are two questions to answer in designing a finite-state machine for a tokenizer:

1. What are the states and transitions?
2. What action should be taken for each transition?

The first addresses the structure of the finite-state machine while the second addresses its implementation. We have talked about the first problem in the previous section and in fact have laid out the structure of the finite-state machine implementation. The second problem may also be familiar, but it is more complex and is key to implementing a tokenizer.

In this section we will examine each state and determine the actions for each transition. In determining the transition actions we will make a table for each state – this process should insure that no transitions are omitted.

### Transitions for the state `START_S`

If you look at the code in Figure 12.19 you will notice that the transition structure for each state is represented by a selection structure, with each selection condition representing a different transition. For the start state we can see the separate cases and will use a table-like layout to specify the action for each transition. The action specified should be placed before the assignment statement, which sets the state.

`START_S: ch == ' '`

We have seen a blank. Ignore it and return to this state.

`START_S: ch == eofChar`


When the `eofChar` character is seen in the start state we know the end-of-file has been encountered. The action, then, is clear: set the token type to `EOT_T` and the next state to `DONE_S`. In this case the value of `name` will be the null string. [Remember that you must use `Token.TokenType.EOT_T` rather than simply `EOT_T`.

`START_S: ch is a letter`

We have seen a letter, so tack it onto `name` and change to the state `IDENT_S`.

```
name += ch;
state = StateName.IDENT_S;
```

`START_S: ch == '}' '{' ',' '=' or an invalid character`

You should see that this case actually combines two transitions for the finite-state machine, the transition marked `} { , =` and the one marked `*`. If you look at the code in Figure 12.19, you will see that these two cases are handled separately. In fact, both these cases can be handled as the default `else`-clause. That means that the final transition would check for any single character that hasn't been already filtered out by the other transitions. But among the characters there are only the four listed above that need real token names, any other character will be an error character that should result in a token type of `ERROR_T`.  **Activity 32** –

Implement these five cases in a method called `char2Token` and having the following signature.

```
Token.TokenType char2Token(char ch)
```

---

Making use of this method, then, means that the action for this combined transition should add the character to `name`, set the value of `type` by calling `char2Token` and then set the next state to `DONE_S`.

### ☛ Activity 33 –

Turn again to the code for the method `getNextToken`. Complete the code for the state `START_S` by adding the actions described in the section just above titled **Transitions for the state `START_S`**. Be sure to add the method `char2Token`. Compile and test the resulting tokenizer and notice that you get proper token recognition for all single-character tokens, but that identifiers are still not handled properly.

---

### Transitions for the state `IDENT_S`

Now we turn to the state `IDENT_S` and its two transitions. The actions here will establish a pattern that you will use in other tokenizers that you write during these tutorials.

#### `IDENT_S: ch is a letter`

We have seen another letter, so tack it onto `name`

```
name += ch;
```

and then return to state `IDENT_S`. IT IS IMPORTANT that you be sure to actually include the following line of code.

```
state = StateName.IDENT_S;
```

Obviously this assignment doesn't really change the value of `state`, since its former value was already `IDENT_S`. But actually setting the next state, even when it is the same, will insure that you don't forget to set one that is actually different. It is just good defensive programming practice.

#### `IDENT_S: ch is not a character`

We have seen a non-letter, so the value of `name` already contains the identifier and we shouldn't tack on the current value of `ch`. However, since this signals an  $\epsilon$ -transition, we must remember to put `ch` back on the input stream.

Next we must check the value of `name` to see if it is an identifier or one of the reserved type names (`int`, `float`) and then assign an appropriate value to the local variable `type`.

`string2Token` is a helper method specifically meant to facilitate distinguishing identifier tokens from keyword tokens. In any tokenizer this method would be easily adapted by identifying the keyword strings and returning for each the appropriate token name. If a string is

not a keyword, then the name `IDENT_T` is returned. Here's a description of the interface for `string2Token`.

```
private Token.TokenType string2Token(String str)
// Pre:  str is a string of alphabetic characters
// Post: return TYPE_T if str == "int" or "float"
//       otherwise return IDENT_T
```

Making use of `string2Token`, here is the segment of code for this transition.

```
putBackChar(ch);
type = string2Token(name);
state = StateName.DONE_S;
```

### ☛ Activity 34 –

- Implement the method `string2Token` as described above.
- Edit the file `Tokenizer.java` and modify the method `getNextToken` by adding to the code for the state `IDENT_S` the actions described in Transitions for the state `IDENT_S`.

Compile and test the resulting tokenizer and notice that you get proper token recognition for all tokens. Be sure to run the tokenizer with the debug set so that you can see the sequence of visited states.

---

That completes the design and implementation of a simple tokenizer – namely a tokenizer for the language *PDef-lite*. It is important to remember that the process of design was completely driven by the structure of the finite-state machine. The details require a bit of work, but the work is straightforward because of the finite-state machine structure. Also, the few tricky things, like  $\epsilon$ -transitions, once learned are easily applied for similar finite machine structures.

### A sample execution of the *PDef-lite* tokenizer

It is important to have a sense for what is expected from your completed tokenizer. Assuming that you have (following the instructions) produced a correct method `getNextToken`, an execution of the executable `PDef` should work as follows.

```
input from file 'test'

A word
123
,={ }
{int a,a=b,float x}
done*done$!done.
```

output from ‘java PDef test’

Tokens appearing in input file ‘test’

```

IDENT_T( A )
IDENT_T( word )
ERROR_T( 1 )
ERROR_T( 2 )
ERROR_T( 3 )
COMMA_T( , )
ASSIGN_T( = )
LCB_T( { )
RCB_T( } )
LCB_T( { )
TYPE_T( int )
IDENT_T( a )
COMMA_T( , )
IDENT_T( a )
ASSIGN_T( = )
IDENT_T( b )
COMMA_T( , )
TYPE_T( float )
IDENT_T( x )
RCB_T( } )
IDENT_T( done )
ERROR_T( * )
IDENT_T( done )
ERROR_T( $ )
ERROR_T( ! )
IDENT_T( done )
ERROR_T( . )
EOT_T( )

```

All done!

## 12.4 Guided Development – PDef Tokenizer

We now turn to the implementation of the PDef tokenizer, whose design is based, of course, on the regular expression for PDef tokens. The regular expression which follows is the same as appears in Figure 4.1 (see page 59).

Token Class	Regular Expression	Termination Characters
addT	+	any character
subT	-	"
multT	*	"
divT	/	"
modT	%	"
commaT	,	"
assignT	=	"
lpT	(	"
rpT	)	"
lcbT	{	"
rcbT	}	"
typeT	int   float	non-letter
intT	0   [1 - 9][0 - 9]*	non-digit
fltT	(0   [1 - 9][0 - 9]*) . [0 - 9]+	non-digit
identT	[a - zA - Z]+	non-letter

### A word on software testing

Before embarking on the development activities below, you should think about what you will be doing. In each step you will be learning of a new feature of the PDef token structure and have to add code to the evolving tokenizer (typically in the method `getNextToken`). When approaching a particular activity, you will first read about the new feature, but then, before diving into the coding, it is very instructive to think about the data you will use to test the new feature. In doing so you not only have to understand what data is expected, but what data should cause errors.

While this may seem like one of those activities teachers always want you to do (i.e., neither interesting nor useful), thinking about testing before coding can give you a better understanding of the coding problem at hand. If you make this part of your work habit, then you will find when the project is finished and must be tested, you already have a very complete test suite.

### ☛ Activity 35 –

#### 1. Adding the simple tokens

The first thing to do is identify what is new. There are seven new single character tokens: five for the arithmetic operators and two for the left and right parentheses – each of these will have its own token class. In addition there are two new token classes, one defining integer literals and the other defining floating point literals.

At this time you should add nine names to the `TokenType` enumerated type: `LP_T`, `RP_T`, `ADD_T`, `SUB_T`, `MUL_T`, `DIV_T`, `MOD_T`, `INT_T`, `FLOAT_T`. In addition, add code to your implementation of the `Tokenizer` class to properly handle the new singleton token classes (operators and parentheses). In the next few sections you will add the integer and floating point literal token classes to your tokenizer, yielding a completed tokenizer for the PDef language.

You should test your new tokenizer to make sure it can properly recognize the new arithmetic operators and parentheses, but the integer and floating point literals will have to wait.

## 2. Adding integer literals

In this section you will add to your tokenizer the ability to recognize integer literals. The integer token values are described by the following regular expressions.

$$[0 - 9][0 - 9]^*$$

Note that the integer token is terminated by any character other than a digit. This is analogous to the way the identifier tokens work. As with identifiers we cannot know a complete integer token has been seen until we see one of the possible termination characters.

Do the following:

- (a) Starting with the finite-state machine diagram in the tutorial, draw a new finite-state machine that will identify integer tokens.
- (b) Have your instructor OK this finite-state diagram and the data you will use to test your finite-state machine and the resulting tokenizer.
- (c) Make the appropriate changes to `getNextToken` so that these new tokens will be recognized. Use the `IDENT_S` state as a guide.  
Be sure to make any extensions necessary to the debugging structure of the `getNextToken` – and remember to use the ‘t’ option when testing.
- (d) Don’t forget to look back at the start state to see if it needs any adjustment for this new token class.

You should now be able to test your tokenizer against a file containing integer literals, arithmetic operators, and examples of the new error tokens.

## 3. Correcting an oversight

The grammar rule we used in the last section is a bit sloppy in that it allows integer literals to begin with a zero. But if you look carefully at the PDef token grammar you will see that it disallows such values; the following tokens should be errors: ‘00’, ‘0101’, ‘0201’; these tokens, on the other hand would be accepted: ‘0’ and ‘20300’.

If a zero is encountered followed by another digit it is required that an error token be generated. An error token must include all consecutive digits starting with the initial 0. Can you use the `ERROR_S` state to consume digits that follow a zero?

A better idea will emerge if you consider using a new state – `ZERO_S`: go to this state from the start state if a zero is encountered. Then have appropriate transitions from that state. [Watch again for  $\epsilon$ -transitions!]

- (a) Modify your finite-state diagram to handle this new requirement.
- (b) Have your finite-state machine and test data approved.
- (c) Make the modifications to the code, being sure to include new debugging structure for new states.
- (d) Test this code on your test data

## 4. Adding floating point literals

The following regular expression gives the complete description of the floating point literals that are legal in PDef.

$$(0 \mid [1 - 9][0 - 9]^*) \cdot [0 - 9]^+$$

Now you will enhance the tokenizer to recognize floating point literals as described above.

- (a) As before, begin by drawing a finite-state machine fragment to show how this can be recognized. You should work from the integer component of your finite-state machine as a start. Notice that any one of these literals must have an integer value to the left of the decimal point and then any sequence of digits following.
- (b) Get your finite-state machine approved.
- (c) Make appropriate modifications to the tokenizer from the previous section. Remember to include the debugging structure in your enhancement of `getNextToken`.
- (d) Test your completed PDef tokenizer.

There is a curious thing that emerges from these new tokens. The period is a required character in a floating point token, but if a period is seen anywhere else in the input stream it will be marked as an error!

#### 5. Adding line numbers to `Token` values

Now for something a bit more challenging. You are to enhance the tokenizer so that it records for each token the line number and horizontal position of its first character. This is not as easy as it may seem at first because of the possibility of putting back a character. You will have to add new data members to `Token` and `Tokenizer`; add accessor and setter methods for these two data members to the class `Token`, such methods are not necessary for `Tokenizer`. Start by keeping track of the line number and position number in `getChar` – be sure to appropriately initialize the values in the `Tokenizer` constructor. Modify `toString` in `Token` so that it prints position information along with the token name and type. To fine tune your solution test it on data where a put back is necessary and think carefully about how the `Tokenizer` works.

---



```

public class PDef {
    public static void main(String[] args) {
        // local variables
        BufferedReader in = null;    // the input character stream
        boolean echo      = false;  // if true the input is echoed --
                                    // value is 'false' by default
                                    // and set to true if 'e' appears
                                    // as a command-line argument
        int numArgs = args.length;  // number of command-line arguments
                                    // (doesn't include command name)

        if (numArgs < 1) {
            // There must be a file name!
            System.out.println("Not enough arguments!\n");
            System.exit(0);
        }
        else {
            // args[0] is the data file name
            try {
                in = new BufferedReader(new FileReader(args[0]));
                if (numArgs > 1) // args[1] holds debug flags
                    for (int i = 0; i != args[1].length(); i++) {
                        switch (args[1].charAt(i)) {
                            case 'e': echo      = true; break;
                            case 't': Debug.registerFlag ('t'); break;
                        }
                    } // ignore invalid flag names
            }
            catch (FileNotFoundException e) {
                System.out.printf("Could not open file '%s'\n", args[0]);
                System.exit(0);
            }
        }

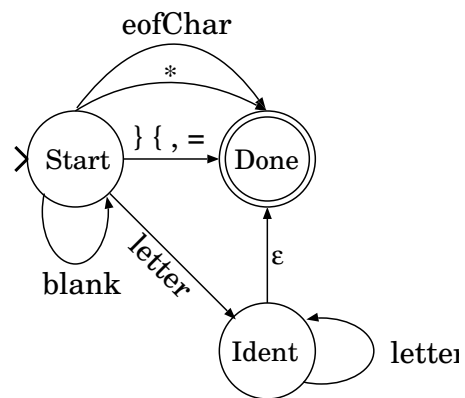
        Tokenizer tins = new Tokenizer(in, echo);

        System.out.println("\nTokens appearing in input file '" + args[0] + "'");
        System.out.println();

        Token t = tins.getNextToken(); // get the first token
        while (t.getType() != Token.TokenType.EOT_T) {
            // there's another interesting token -- print it
            System.out.println(t);
            t = tins.getNextToken(); // make progress toward termination
        }
        System.out.println(t);
        System.out.println( "\nAll done!\n");
    }
}

```

Figure 12.17: Code for main()

Figure 12.18: The PDef-*lite* finite-state machine

```

Token.TokenType type = Token.TokenType.ERROR_T;
StateName state = StateName.START_S;
String name = "";
while (state != StateName.DONE_S) {
    char ch = getChar();
    switch (state) {
    case START_S:
        debug.show("\t==> Entering state -- START_S: '", ch);
        if (ch == ' ') {
            state = StateName.START_S;
        }
        else if (ch == eofChar) {
            type = Token.TokenType.EOT_T;
            state = StateName.DONE_S;
        }
        else if (Character.isLetter(ch)) {
            name += ch;
            state = StateName.IDENT_S;
        }
        else if (special(ch)) {
            name += ch;
            state = StateName.DONE_S;
        }
        else { // Assert: illegal character
            name += ch;
            type = Token.TokenType.ERROR_T;
            state = StateName.DONE_S;
        }
        debug.show("\t<<== Leaving state -- START_S: '", ch);
        break;
    case IDENT_S:
        debug.show("\t==> Entering state -- IDENT_S: '", ch);
        if (Character.isLetter(ch)) { state = StateName.IDENT_S; }
        else {
            putBackChar(ch); // since an epsilon-transition
            state = StateName.DONE_S;
        }
        debug.show("\t<<== Leaving state -- IDENT_S: '", ch);
        break;
    case DONE_S:
        // should never reach this point
        // but good for completeness
        break;
    }
}

```

Figure 12.19: A Partial-implementation of getNextToken



## Chapter 13

# PDef Parser Tutorial

**Goal:** To complete the design, implementation and testing of a parser for the language PDef.

**Strategy:** This tutorial will begin with a detailed discussion of the relationship between the rules of a CFG and a recursive descent parser. You will start with a working parser for PDef-*lite* and through the course of the tutorial will extend this implementation to allow for general arithmetic expressions on the right-hand side of assignment entries.

**Assumptions:** The tutorial assumes that you have a basic knowledge of context free grammars and how they can be employed to describe programming language syntax.

**Java Features:** In this tutorial you will use the exception handling features of Java to implement an exception class that will be used to implement the parse-error handling.

**Code Base:** You will work with a completed Java implementation of a parser for PDef-*lite*. A table describing the components of the implementation is displayed in Figure 13.1. It is important that you look over this code base before you embark on the tutorial. Also, you will want to have the code base available (online or in printed form) for reference as you progress through the tutorial.

### The Tutorial

In this tutorial we will discuss the implementation of a *recursive descent* parser for PDef-*lite* and then, in the Guided Development at the end of the tutorial, extend it to a parser for PDef. Figure 13.2 shows how the parser interacts with the tokenizer produced in the previous tutorial. As was discussed in Section 6.2, a recursive descent parser consists of a set of mutually recursive methods that cooperate to check the syntactic correctness of a string of tokens. We will apply the methodology of Section 6.2 to the PDef-*lite* grammar to generate an appropriate set of mutually recursive parse methods. But implementing a recursive descent parser involves more than simply designing the parse methods. In this tutorial we will expose the internal structure of the parser class (see Figure 13.2) and discuss parse error identification and reporting as well as parse error recovery

Code Base	
File	Implements
PDef.java	This file contains the class <code>PDef</code> and its method <code>main</code> which is the driver for the PDef parser.
parser/Parser.java	This file implements the class <code>Parser</code> for PDef- <i>lite</i> .
exception/PDefException.java	This file implements the class <code>PDefException</code> that facilitates exception handling for the parser.
package	
debug	This package is the same as for the last tutorial but has added the class definition <code>ParserDebug</code> , which facilitates debugging of the parser code.
exceptions	This package contains the classes defining various exceptions that are thrown in this system.
tokenizer	This package contains the complete code for a working version of the PDef tokenizer – including the testing method <code>main</code> in the class <code>Tokenizer</code> .
Test-PDefL	This directory contains text files that can be used for testing the PDef- <i>lite</i> parser driver program.
Test-PDef	This directory contains text files that can be used for testing the PDef parser driver program.
Test-errors	This directory contains text files that can be used for testing the PDef parser error recovery.

Figure 13.1: Parser Java Components

techniques.

In Chapter 6 we saw a more theoretical discussion of parsing, especially recursive descent parsing. In this tutorial we will take advantage of the techniques discussed there but will also look into the more practical problems of implementing a recursive descent parser. We will be interested in the implementation of the method `consume`, used heavily in the parse methods discussed in Forms 1-6, and how parse methods can deal, in an effective way, with syntax errors. The handling of syntax errors will also bring in the use of exception handling as a way of reporting errors and of recovering from errors so that parsing can continue.

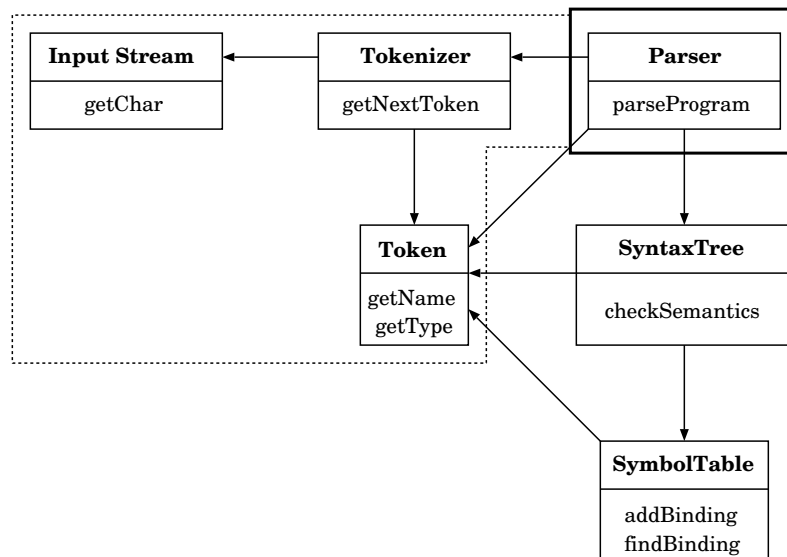


Figure 13.2: Adding the Parser Component

## 13.1 Parser Structure

In this tutorial you will expand on the system started in the Tokenizer Tutorial by integrating into it a recursive descent parser for PDef. Our focus will be the class `Parser` which is the implementation of a recursive descent parser for PDef, though our initial focus will be a parser for PDef-*lite*.

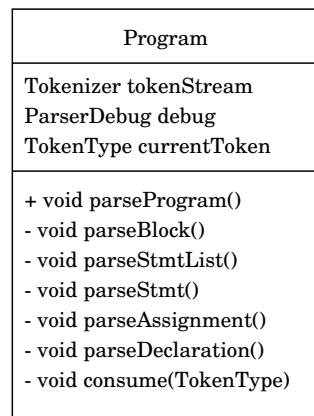
In this section we will discuss the structure of `Parser` and how it interacts with the tokenizer. While the discussion in Section 6.2.3 focused on a more formal approach to parse method design, one which we will take full advantage of, there are several other implementation issues that must be dealt with in the tutorial. In the two sections that follow we will look at the basic structure of the parser class and discuss the method `consume` that was so instrumental in defining parse methods in Section 6.2.3.

### 13.1.1 The Class Parser

The class `Parser` according to the diagram in Figure 13.3 has a single public parse method, `parseProgram`, and also makes reference to the tokenizer subsystem. The UML class diagram in Figure 13.3 gives a high-level description of the class `Parser` appropriate for PDef-*lite*.

Notice in the UML class diagram that, besides the public method `parseProgram`, there are five other private parse methods, each corresponding to a PDef-*lite* non-terminal. There is also the private method `consume`. The data members are also worth discussing. When a parse method is called it must have access to the token stream, represented by `tokenStream`, but also to the most recently read token – since this token may be referenced by more than one parse method there is a data member `currentToken` to hold that value.

An important consideration in designing `Parser`, based on the UML class diagram, is to devise a protocol for dealing with its central data member `currentToken`. In fact, we did this in Section 6.2 when we defined the Parser Rule and the Parser Invariant. We repeat the definitions here for

Figure 13.3: UML Class Diagram for a PDef-*lite* Parser

convenience.

**Parser Invariant:** We will assume an object called `currentToken` exists and that at any point during the parsing process the value of `currentToken` is the next token to be processed.

**Parser Rule:** Read the next token immediately after verifying that a target token's type matches the type of `currentToken`.

While the Parser Rule specifies when tokens are read from the token stream, the Parser Invariant indicates a condition that must be true at the time a parse method is called and at the time a parse method returns – i.e., it is the responsibility of the parse method to guarantee the Parser Invariant.

### 13.1.2 Consuming Tokens

There is one private method of this class that is not a parse method. The method `consume`, which we discussed in Section 6.2 (see page 84), has the signature

```
private void consume(TokenType ttype)
```

and the following functionality:

1. verify that the current token type is the same as `ttype` and
2. if the match succeeds then read the next token and  
if the match fails display an error message and terminate the program.



This method can be implemented as follows – notice how it implements the Parser Rule described above.

```
private void consume(Token.TokenType ttype) {
    if (currentToken.getType() != ttype)
        System.out.println("Expected to see token " + ttype +
            " but saw token " + currentToken.getType());
        System.exit(0);
    }
    currentToken = tokenStream.getNextToken();
}
```

[Notice that since there is a call to `exit` in the `if`-block, no `else` is needed.]

This is a reasonable solution to a common error handling situation, though calling `System.exit(0)` is a bit clumsy. We will see later, however, that exception handling can be integrated into the parse methods and `consume` in such a way to allow a degree of error recovery.

## 13.2 The Supplied Java Code

A parser for the language *PDef-lite*, written in Java, accompanies this tutorial. The included files are described in the table displayed in Figure 13.1. Our interest initially will be the classes `Parser` and `PDef`, which implement the parser for *PDef-lite*. In the remainder of this section we will give a brief overview of these new components and then discuss at length the definition of the class `Parser` – in particular we will focus on the implementation of its parse methods.

### 13.2.1 class `Parser`

The structure of the class `Parser`, which appears in the UML class diagram in Figure 13.3, is partially implemented in the file `Parser.java`. The extent of the implementation will be discussed in the next section.

```
public class Parser {

    // Class Invariant:
    //     The data member currentToken is a
    //     reference to the next token of interest.

    private ParserDebug debug;
    private Token currentToken;
    private Tokenizer tokenStream;

    public Parser(Tokenizer toks) {...}
    // Pre: tokenStream has a value
    // Post: debug == new ParserDebug() AND
    //       this.tokenStream == tokenStream AND
```

```

//      class invariant is true
public void parseProgram() {...}
    // Grammar Rule:  Program --> Block eotT
private void parseBlock() {...}
    // Grammar Rule:  Block --> lcbT StmtList rcbT
private void parseStmtList() {...}
    // Grammar Rule:  StmtList --> Stmt [ commaT StmtList ]
private void parseStmt() {...}
    // Grammar Rule:  Stmt --> Assignment |
    //                  Declaration |
    //                  Block
private void parseAssignment() {...}
    // Grammar Rule:  Assignment --> identT assignT identT
private void parseDeclaration() {...}
    // Grammar Rule:  Declaration --> typeT identT

private void error() {...}
private void consume(TokenType ttype) {...}
}

```

The one data member not discussed to this point is `debug`, which has a similar purpose to the data member `debug` in the class `Tokenizer`. There is an additional class in the `debug` package, as described in Figure 13.1.

Before moving on it is useful to focus on the class invariant and its relationship to the methods of the class. An invariant is a logical statement that must be true before and after each call to any method of the class. Since the first parse method called will be `parseProgram`, the class invariant must hold before that call. But this means that `currentToken` must be initialized in the constructor for the class – notice this is implied in the postcondition for the constructor. The implementation of the constructor, therefore, has the following form.

```

public Parser(Tokenizer tokenStream)
// Pre:  tokenStream has a value
// Post: debug == new ParserDebug() AND
//       this.tokenStream == tokenStream AND
//       class invariant is true
{
    this.debug = new ParserDebug();
    this.tokenStream = tokenStream;
    currentToken = tokenStream.getNextToken();
    // This makes the class invariant true
    // for the call to parseProgram
}

```

### 13.2.2 The Class PDef

The class `PDef` contains a single driver method `main`. It is a modified version of the corresponding method in the class `Tokenizer`. The changes to the tokenizer driver are simple: an additional

command-line flag is provided for initializing `Parser`'s internal debug flag. In addition, the actual testing code has been changed to adapt to the testing of the parser rather than the tokenizer. The relevant code is reproduced here.

```
public static void main(String[] args) {

    // code for processing command line arguments

    Tokenizer tokenStream = new Tokenizer(in, echo);
    Parser parse = new Parser(tokenStream);

    parse.parseProgram();
    System.out.println("Program parsed!");
}
```

Notice that after the `Tokenizer` object is instantiated, the reference to it is passed as a parameter to the constructor for `Parser`. The parsing of the input data is then handled via the call to the `parse` method `parseProgram`. The code here assumes that if a parse error is encountered the program will fail in a relatively ungraceful way, meaning it will stop and perhaps print an indication of what caused the problem. At the end of the tutorial we will discuss how we will use the Java exception handling mechanism and the class `Exception` to provide for graceful error handling.

The fact that the file `PDef.java` is in the directory containing the java packages means that the compilation and execution will take a slightly different form. First, we will want the test files, or the directory holding the test file, to be contained in the same directory with the file `PDef.java`. The compiling and testing of the parser system is illustrated in the following transcript.

```
% javac PDef.java */*.java
% java PDef test
Program parsed!
%
```

### 13.2.3 Parse Methods in Parser

In Section 6.4 we discussed two parse methods for `PDef`, `parseBlock` and `parseStmtList`, as examples of the application of the parse method design strategy. What follows are the implementations for these same parse methods as they appear in the supplied code in `Parser.java`.

```
private void parseBlock()
// Grammar Rule:  Block --> lcbT StmtList rcbT
{
    debug.show(">>> Entering parseBlock");

    consume(Token.TokenType.LCB_T);
    parseStmtList();
    consume(Token.TokenType.RCB_T);
}
```

```

    debug.show("<<< Leaving  parseBlock");
}

private void parseStmtList()
// Grammar Rule:  StmtList --> Stmt { commaT Stmt }
{
    debug.show(">>> Entering parseStmtList");
    parseStmt();
    while (currentToken.getType() == Token.TokenType.COMMA_T) {
        consume(Token.TokenType.COMMA_T);
        parseStmt();
    }
    debug.show("<<< Leaving  parseStmtList");
}

```

### ☛ Activity 36 –

It is an interesting exercise to see the impact on the code design if we had chosen to use the right-recursive form for the `StmtList` grammar rule— i.e., the following.

```

    StmtList --> Stmt [ commaT StmtList ]

```

Assuming this grammar rule, design and implement the parse method according to the strategy given for Form 5 (clearly the form for this new rule).

---

### parseProgram

The method `parseProgram` is a curiosity. Earlier this method’s presence was attributed to the need for a `public` method to get things rolling. But wouldn’t it seem that the method `parseBlock` could simply be declared `public` rather than `private`? Actually this wouldn’t work very well because the first call to `parseBlock` wouldn’t be exactly the same as other calls to it.

To understand why `parseProgram` is a good idea think of the structure of an entire “program,” i.e., the entire input string. We know that such a program consists of a list of entries as described by `Block` for sure, but, if you remember how the tokenizer works, the list should be followed at the end by a `eotT` token. This means that the grammar should have the following rule.

```

    Program → Block eotT

```

This rule has Form 3 so the parse method is easy to write. Actually, since this is meant to be the very end of the parsing, there is really no reason to input another token; if the token `eotT` is found then we know that the next input token will also be `eotT`. There is one other line in this method that doesn’t follow the proscribed pattern, namely the initialization of the variable `currentToken`. But this initialization is necessary so that when we call `parseBlock` this first time the Parser Invariant is satisfied.

```

void parseProgram()
// GrammarRule: Program --> Block eotT
{
    debug.show(">>> Entering parseProgram");

    parseBlock();
    consume(Token.TokenType.EOT_T);

    debug.show("<<< Leaving parseProgram");
}

```

There is one final point raised by this method. When the method is called, what will be the initial value of `currentToken`? Since the first thing we do is to call `parseBlock` it would seem that `currentToken` had better already have been initialized with the first token from the input stream. This is a task that we relegate to the constructors of the class `Parser`.

### ☛ Activity 37 –

There are three parse methods that remain unimplemented: `parseDeclaration`, `parseAssignment`, and `parseStmt`. The first two are straightforward but the second takes some study. Design and implement each of these three parse methods. For `parseStmt` return to page 87 (in Section 6.2.3) to study how a parse method for a grammar rule with options is structured. These three parse methods should complete the parse for *PDef-lite*.

At this point test the *PDef-lite* parser on a test file found in the distribution directory. When you test the parser run the following tests, remembering that the option ‘e’ turns on echoing of input and ‘p’ turns on the `Debug` object in the parser.

```

% javac PDef.java
% java PDef test
% java PDef test p
% java PDef test ep

```

## 13.3 Dealing with Parse Errors

There are two kinds of errors to worry about in any program: the expected and the unexpected. If an unexpected error (i.e., a bug) occurs while your program is running then the results will be unpredictable. An expected error, on the other hand, is one that you can detect before it actually happens, and in this way either recover from the error or gracefully shut down the program. In a program such as a text editor this is an important consideration since an edited file can be saved before causing the program to shut down.

A parser is a good example of a program for which there are expected errors. In a call to the parse method `parseBlock`, for example, if when `parseStmtList` returns no `rcbT` token is detected

an error has occurred in the sense that the parsing process cannot continue normally. In this situation we have designed the code of the method `consume` to print a simple message and then call `System.exit`. While this way of handling expected errors works fine, it is clumsy because it can generate many exit points from the program. It would be nicer if we could have a consistent but flexible way of dealing with expected errors, have a single exit point, and not have to rely on the clumsy `exit` call to terminate the program.

In this section we will discuss the technique of exception handling and we will see how this technique can be adapted to deal with processing of syntax errors. The technique will satisfy our three wishes above (consistent and flexible error handling, single exit point, no use of the `exit` call) and in particular be flexible enough to also work for other classes of expected errors that will arise in later tutorials.

### 13.3.1 Exception Handling

There are two aspects or components of exception handling that must be discussed. When an error is detected (1) an exception must be signaled and then (2) the exception must be handled. It is these two actions, signaling and handling, which are the focus of this section.

#### How exceptions are handled

“Exception handling” is a good alternative to the “bail out” strategy we implemented in the method `consume`. The idea behind exception handling is to provide a smooth and predictable mechanism for responding to conditions in a program that are exceptional in some sense. In Java exception handling involves two separate syntactic components, one for signaling the presence of an exception and the other trapping the signal. We will look at an example using code from the method `main` in `PDef.java`. Here is the original code.

```
public static void main(String[] args) {  
  
    // code for processing command line arguments  
  
    Tokenizer tokenStream = new Tokenizer(in, echo);  
    Parser parse = new Parser(tokenStream);  
  
    parse.parseProgram();  
    System.out.println("Program parsed!");  
}
```

If a parse error (an exception) is detected it will occur during the call to `parseProgram` or one of the parse methods it calls. Since an exceptional condition is possible in this call, we provide a mechanism for trapping it as follows.

```

public static void main(String[] args) {

    // code for processing command line arguments

    Tokenizer tokenStream = new Tokenizer(in, echo);
    Parser parse = new Parser(tokenStream);

    try {
        parse.parseProgram();
        System.out.println("Program parsed!");
    }
    catch (PDefException exc) {
        exc.print();
    }
}

```

What has been inserted is called a `try...catch` block. You can understand the code to read as follows:

Try to execute the following block of code.

```

parse.parseProgram();
System.out.println("Program Parsed!!");

```

If the execution succeeds continue after the `catch` block. But if an exception is signaled during the execution, transfer control immediately to the `catch` block of code.

The `try` block indicates where an exception may occur. The `catch` block specifies (almost like a function call) what to do if an exception occurs. If an exception occurs during the call to `parseProgram` then the code in the `catch` block is executed, with the thrown exception object being the argument to the `catch` block. In either case the execution continues after the `try...catch` block.

It is important to realize from this explanation that if an exception is signaled during the call to `parseProgram`, the output statement that follows will not be executed. So the `try...catch` provides a structured mechanism for transferring control from one part of a program to another syntactically unrelated part of the program.

### ☛ Activity 38 –

Add the `try-catch` block to the `PDef.java` code. You can't test these changes reasonably until you have some exceptions to throw – that is the topic of the sections that follow.

---

### How exceptions are signaled

The other half of the exception handling mechanism is signaling that an exception has occurred. If we are to signal an exception we must find places in the code where errors, parse errors in our

current situation, can be detected. But we already know where these points are because when an error is detected we eventually call `System.exit` – if we identify these calls we will know where exceptions must be thrown. The most obvious place where this happens is in our method `consume`. We redisplay this method’s code here for convenience.

```
private void consume(Token.TokenType ttype) {
    if (currentToken.getType() != ttype)
        System.out.println("Expected to see token " + ttype +
                           " but saw token " + currentToken.getType());
    System.exit(0);
}
currentToken = tokenStream.getNextToken();
}
```

We detect an error by comparing the expected token (passed as a parameter) with the current token value – if they are different we display an error message and exit. What we need to do is to signal an exception instead.

Signaling an exception takes place as follows.

1. Detect that an exception has occurred.  
We do this in the `consume` method by comparing token types.
2. Construct an object that contains information about the exception.
3. Throw the exception object so that it can be caught in a “surrounding” `catch` block.

The Java implementation of exception throwing as applied to `consume` is shown in Figure 13.4. The `throw` statement causes control to be passed to a `try...catch` block that contains, perhaps

```
private void consume(Token.TokenType type) throws PDefException
{
    if (currentToken.getType() != type) {
        String msg = "Expected to see token " + type +
                    " but saw token " + currentToken.getType() + "\n";
        throw new PDefException(msg);
    }
    currentToken = tokenStream.getNextToken();
}
```

Figure 13.4: Exception Throwing in `consume`

via a method call, a call to `consume`. The corresponding `catch` block is executed with the object, instantiated in the `throw` statement, passed to the `catch` as parameter.

Also notice that the method signature for `consume` has a bit of additional syntax at the end, namely indicating that this method throws a particular kind of exception object. We call this a `throws`-clause. In the next section we will discuss an appropriate parse exception class.



### 13.3.2 A Class for Basic Exceptions

Java has stringent requirements on the kinds of “objects” that can be thrown when an exception is detected – in particular, the class for a thrown object **MUST** be a subclass of the Java class `Exception`. When a parse error occurs, however, the exception handling structure does provide the opportunity to send supplementary data in an exception object. Thus, the strategy for exception handling is to define a class that can contain (as data members) particular data values relevant to the particular condition. In the simplest situation it may be sufficient to send a string value that contains a description of the exceptional condition. Here is a class definition for basic exception handling.

```
public class PDefException extends Exception {

    public PDefException(String msg) { super(msg); }
    public void print() { System.out.println(super.getMessage()); }
}
```

If we look back at the code in Figure 13.4 we see this class used for instantiating objects to be thrown. Notice that the appropriate error message is passed as a `String` parameter to the constructor for `PDefException` and then passed by that constructor to the constructor of the super class `Exception`. The fact that the message is defined as `private` in the super class means that we must use `Exception` method `getMessage` to retrieve this message value. The class `PDefException` is implemented in the distributed file named `PDefException.java`.

### 13.3.3 A Class for Parse Exceptions

While the class `PDefException` is useful in certain basic situations for exception handling, our situation with parse errors seems to call for a bit more structure. When a parse error is detected, it would seem to be useful to pass with the exception object the name of the token class that was expected as well as the value of `currentToken` (the token actually seen) when the error is detected. We thus arrive at the class definition that follows. Notice that we use the constructor for the super class to set a value for the exception message to be printed when the inherited method `PDefException.print` is called; also note that the value of the token parameter is included in the message passed via `super`.

```
public class ParseException extends PDefException {

    Token token;

    public ParseException(String msg, Token t) {
        super( msg + ", but saw the token " + t );
        token = t;
    }

    public Token getToken() { return token; }
}
```

### ☛ Activity 39 –

Add the class definition `ParseException` to the package `exception` as defined above. Remember that since this new exception class contains a data member of type `Token`, that the package `tokenizer` will have to be imported.

Also remember to include “`package exceptions;`” as the first line of the file containing the class `ParseException`.

---

### 13.3.4 Adapting the class `Parser` for Exception Handling

How do we fold these exception handling mechanisms into our parser? We know that we must replace blocks of code containing calls to `exit` with an exception throw. But having modified our parser to use the method `consume`, there is only one other location where `exit` is called, in the default case in `parseStmt`.

In the `parseStmt` method we can replace the code block in the `default` case with the following exception throw. Notice carefully the message that is passed as the first parameter to the exception constructor.

```
throw new ParseException("Expected to see type, identifier, or left brace!",
                        currentToken);
```

### ☛ Activity 40 –

Make appropriate modifications to each parse method and to the method `consume`. Remember that for every parse method you must add the `throws` clause

```
throws ParseException
```

to the end of the signature of the method. For the method `consume` you need to replace `PDefException` with `ParseException`.

The basic rule concerning the `throw`-clause in method signatures is that if a method throws an exception or calls a method that throws an exception, then that exception’s class name must be included in a `throws` clause for that method. Since every method either calls `consume` or throws a `ParseException` exception itself, every parse method must have this `throws`-clause added.

---

### ☛ Activity 41 –

At this point your *PDef-lite* should be complete. Test your parser using the files in the supplied directory `Test-PDefL`.

---

## 13.4 Recovering from Parse Errors

The strategy for parse error handling to this point has been to identify the first parse error and then to terminate the parsing, giving relevant information about the error. In this section we will discuss a strategy which allows the parser to recover from a parse error and continue the parsing process. This error recovery has the clear advantage of giving more error information with each parser execution.

### 13.4.1 An Example

As an example of other possibilities, consider the following alternative implementation for `parseAssignment`.

```
public void parseAssignment() throws ParseException {
    // Grammar rule: Assignment --> identT assignT identT
    consume(Token.TokenType.IDENT_T);

    try { consume(Token.TokenType.ASSIGN_T) }
    catch (ParseException exc) {
        // assume the ASSIGN_T is simply missing
        exc.print();
    }

    consume(Token.TokenType.IDENT_T) }
}
```

What will be the effect of using this method in place of our old version? Apparently, the method will behave as it did on valid data. If the third `consume` call generates an exception then that exception will be caught in `PDef.main` as usual. But what if the second call to `consume` throws an exception? In this case the exception will be caught right away, an error message will be displayed, and execution will continue after the `catch` block. So if we type input and simply leave out the assign operator, the error will be reported and parsing will continue.

This is an example of very fine-grained error recovery, meaning that the parser assumes the expected token was left out and continues. The problem with this approach is that each possible exception throw must be caught and handled essentially at the point where it is thrown. If the language being implemented is complex this could mean inserting a lot of `try-catch` statements, a new source of coding errors. Of course our original error-handling strategy is so large-grained that we only learn of one error when running the parser. What would be more useful is a strategy somewhere in the middle.

If we would like to avoid having our error recovery code overwhelm our parsing code, then where else could we catch the exception thrown when the assign operator is missing? Since `parseAssignment` is called by `parseStmt`, we could catch it there. The call in `parseStmt` would look as follows.

```

case IDENT_T:
    try { parseAssignment(); }
    catch (ParseException exc) {
        exc.print();
    }
case TYPE_T:
    .....

```

The problem with this solution is that, since we can't tell which of the `consume` calls in `parseAssignment` actually threw the exception, we need to take some explicit but more general recovery action. If the assignment statement is in error then we could simply skip the rest of it and continue with the next statement. This would mean actually reading enough tokens to know we have reached the end of the assignment statement.

How would the parser know where the end of the statement is? According to the *PDef-lite* grammar, a statement can be followed by either a comma or a right brace. So if we write a method to skip tokens until we find a comma or right brace, then the parser would be in a position to resume parsing. Actually, in the extreme case where there are no commas or right braces left on the input stream, it would be prudent to also check for the end-of-file token as well. Assuming we have a method to do the skipping, we would rewrite the code above as follows.

```

case IDENT_T:
    try { parseAssignment(); }
    catch (ParseException exc) {
        exc.print();
        skip2StatementEnd();
        // Assert: currentToken == COMMA_T or RCB_T or EOT_T
    }
case TYPE_T:
    .....

```

You may have anticipated the next step in the evolution of our error recovery strategy. This recovery from an exception in `parseAssignment` would work just as well for exceptions thrown in `parseDeclaration` or `parseBlock`. So a better modification to the code in `parseStmt` is as follows.

```

private void parseStmt()
// Grammar Rule: Stmt --> Assignment |
//                Declaration |
//                Block
{
    debug.show(">>> Entering parseStmt");
    try {
        switch (currentToken.getType()) {
            case Token.TokenType.IDENT_T: // parseAssignment also checks for IDENT_T
                parseAssignment();
                break;
            case Token.TokenType.TYPE_T: // parseDeclaration also checks for TYPE_T
                parseDeclaration();

```

```

        break;
    case Token.TokenType.LCB_T:    // parseBlock also checks for LCB_T
        parseBlock();
        break;
    default:
        throw new ParseException("Expected to see token IDENT_T, TYPE_T, or LCB_T"
                                + currentToken);
    }
    catch (ParseException exc) {
        exc.print();
        consume2StatementEnd();
        // Assert: currentToken == COMMA_T or RCB_T or EOT_T
    }
}
debug.show("<<< Leaving parseStmt");
}

```

### 13.4.2 Designing an Error Recovery Strategy

What we have done in evolving the solution above is to raise the error recovery problem from the individual statement level up to the general statement level. The solution is less fine-grained, but we have just one `try-catch` statement. This development, besides giving us an error recovery strategy for *PDef-lite*, has highlighted the three critical questions which must be answered when designing an error recovery strategy:

1. What errors can be trapped (and signaled via a thrown exception)?
2. Where should the exceptions be caught?
3. What recovery actions should be taken when an exception is caught?

The answers to the questions reside in the target grammar. The level of granularity for a design is determined by the rule in the grammar where the exceptions are trapped. Here is the *PDef-lite* grammar again for convenience.

```

1 Program    → Block
2 Block      → lcbT StmtList rcbT
3 StmtList   → Stmt { commaT Stmt }
4 Stmt       → Declaration | Assignment | Block
5 Declaration → typeT identT
6 Assignment → identT assignT identT

```

Our original large-grained solution is to trap the exceptions at the level of rule 1, i.e., by trapping exceptions during the parsing of `Program`. The second level would seem to be rule 2, but since `Block` is an option for `Statement` this isn't appropriate. So the next level down would be to trap exceptions at the `Block` level, but this would require recovery to skip the remainder of a list, rather than the remainder of a statement. Stepping down to the next level of granularity brings us to rule 4 which we implemented above.

The level of error recovery dictates the recovery algorithm for each recovery point. If there is no recovery then the only problem is to guarantee that all errors will eventually be caught. In the *PDef-lite* there is a subtle error problem which doesn't become apparent until we start thinking about error recovery. At this point the strategy we have determined is to do error recovery at the statement level, that is, if an error is detected at the statement level then skip to the end of the current statement and resume parsing. Now consider the following erroneous input.

```
{ int a  a = b, float b }
```

How will the missing comma after the first declaration statement be handled? If we look at the parse methods involved, we can see that `parseStmtList` will call `parseStmt` which will call `parseDeclaration`, which will return to `parseStmt` having determined that 'int a' is a valid declaration statement. `parseStmt` will return to `parseStmtList` and then see the token `IDENT_T` rather than another comma (which we know to be simply missing). `parseStmtList` doesn't check for this possibly missing comma so returns to `parseBlock` which throws a missing right brace exception. Now in our original no-recovery error handling strategy, this is fine. The fact that the declaration statement is incorrectly terminated is finally caught, but the catch-point is so far from the actual problem (missing comma) that a recovery will undoubtedly be awkward.

In fact, we can trap this error earlier by checking in `parseStmt`, after the `switch`, to see if the current token can follow a statement. Notice that it is this same set of tokens which attracted our attention when we needed to skip to the end of a statement. To solve the missing comma problem, then, it will suffice to insert a `try-catch` statement at the end of the `try` block in `parseStmt`. At that point we have two choices as to how to handle the exception. We can assume the comma terminating token is missing and continue or skip until we find such a symbol. For consistency, we will adopt the second approach, giving us the following `try-catch` statement for `parseStmt`.

```
try {
    switch (currentToken.getType()) {
        case IDENT_T:
            .....
            .
            .
    }
    if ( !canFollowStatement(currentToken) )
        throw new ParseException("Expected a statement terminator.", currentToken);
} // end of try
catch (ParseException exc) {
    exc.print();
    consume2StatementEnd();
    // Assert: currentToken == COMMA_T or RCB_T or EOT_T
}
```

The method `canFollowStatement` returns true if the current token is a comma, right brace or end-of-file. Notice that since our recovery action is to be the same as for an error inside a statement, we can simply throw the exception if the current token can't follow a statement. If we were to take the other approach, i.e., assume the comma is missing, then we would have to insert a `try-catch` around the new selection statement.

### 13.4.3 Error Recovery in PDef-*lite*

Error recovery in PDef-*lite* is based on answers to the 3 design questions which we have discussed above. Here are the questions and answers.

1. What errors can be trapped?

The errors to be trapped in the parser are those involving mismatches between the current token and the token predicted by the parser.

- A missing assign operator or a missing right-hand identifier will be reported in `parseAssignment`.
- A missing identifier will be reported in `parseDeclaration`.
- A missing right brace will be reported in `parseBlock`.
- An improper starting token for a statement will be reported in `parseStmt`.
- A missing comma can be reported in `parseStmt` by checking that the token following a statement is legal.

2. Where should the exceptions be caught?

All exceptions save one will be caught after the completion of the switch statement in `parseStmt`. If the final closing right brace (of the outermost list) is missing that exception will be caught in `PDef.main`.

3. What recovery actions should be taken when an exception is caught?

No recovery is necessary when a missing final right brace is reported. For all other exceptions the parser will skip until it finds the next statement terminator – either a comma, right brace, or end-of-file.

The error recovery just described is implemented in the PDef-*lite* parser by modifying the method `parseStmt` as follows.

```
private void parseStmt()
// Grammar Rule: Stmt --> Assignment |
//                Declaration |
//                Block
{
    debug.show(">>> Entering parseStmt");
    try {
        switch (currentToken.getType()) {
            case Token.TokenType.IDENT_T: // parseAssignment also checks for IDENT_T
                parseAssignment();
                break;
            case Token.TokenType.TYPE_T: // parseDeclaration also checks for TYPE_T
                parseDeclaration();
                break;
            case Token.TokenType.LCB_T: // parseBlock also checks for LCB_T
                parseBlock();
                break;
        }
    }
}
```

```

    default:
        throw new ParseException("Expected to see token IDENT_T, TYPE_T, or LCB_T"
                                currentToken);
    }
    if ( !canFollowStatement(currentToken) )
        // Assert: currentToken != COMMA_T or RCB_T or EOT_T
        throw new ParseException("Expected a comma, right brace or end of file token"
                                currentToken);
}
catch (ParseException exc) {
    exc.print();
    consume2StatementEnd();
    // Assert: currentToken == COMMA_T or RCB_T or EOT_T
}
debug.show("<<< Leaving parseStmt");
}

```

This code expects two other private methods be implemented, one to identify tokens which can follow a statement and the other to skip to the next such token. The appropriate code follows.

```

private void consume2StatementEnd() {
// Post: currentToken == COMMA_T or RCB_T or EOT_T
    Token.TokenType type = currentToken.getType();
    while ( !canFollowStmt(currentToken) ) {
        currentToken = tokenStream.getNextToken();
        type = currentToken.getType();
    }
    // Assert: currentToken == COMMA_T or RCB_T or EOT_T
}

private boolean canFollowStatement(Token t) {
// Post: return true if t == COMMA_T or RCB_T or EOT_T
    Token.TokenType type = t.getType();
    return ( type == Token.TokenType.COMMA_T ||
            type == Token.TokenType.RCB_T ||
            type == Token.TokenType.EOT_T );
}

```

#### ☛ Activity 42 –

Add the error recovery code described above to your PDef-*lite* parser and test to see how multiple errors can be found in a single execution of the parser.

---



## 13.5 Guided Development – PDef Parser

### Activity 43 –

1. At this point you have a parser that correctly parses the simple PDef-*lite* language as described by the following grammar.

```

Program    → Block
Block      → lcbT StmtList rcbT
StmtList   → Stmt { commaT Stmt }
Stmt       → Declaration | Assignment | Block
Declaration → typeT identT
Assignment → identT assignT identT

```

Now you are in a position to extend your parser to one for the language PDef, which permits arithmetic expressions to occur on the right-hand sides of assignment entries. Here we re-display the grammar for PDef. Most of this grammar is already implemented in the PDef-*lite* parser. It will be a matter of adding parse methods for the new parse rules. Notice that the rule for **Assignment** has changed and so its parse method will have to be modified.

```

Program    → Block
Block      → lcbT StmtList rcbT
StmtList   → Stmt { commaT Stmt }
Stmt       → Declaration | Assignment | Block
Declaration → typeT identT
* Assignment → identT assignT Exp
* Exp        → Exp (addT | subT) Term
*           → Term
* Term       → Term (multT | divT | modT) Factor
*           → Factor
* Factor     → intT | floatT | identT | lpT Exp rpT

```

To implement the new parser do the following:

- (a) The rules for **Exp** and **Term** are left-recursive. Convert them to a new form by applying left-recursion elimination (see Section 2.3.4, page 39). Following the lead established in Section 6.3 (page 93), implement parse methods for these revised rules.
  - (b) Consider the grammar rule above for **Factor**. Identify the appropriate parse form for this rule. Following that form, implement the parse method `parseFactor`, being sure to make appropriate use of `consume` and exception throwing.
  - (c) Complete the parser for PDef, making sure that `parseAssignment` is correct for the PDef grammar.
  - (d) At this point you can test your PDef parser on the test files in the directory `Test-PDef`.
2. At this point the PDef parser works correctly on syntactically correct PDef files and generates a single parse error on files with one or more errors. Section 13.4 discusses a way to alter the exception handling structure so that more than one parse error

can be flagged on a single execution of the parser. Following the strategy described in Section 13.4, modify the PDef parser so that it can recover from errors during parsing.

Make the assumption that if an error occurs during an expression then recovering should resume at the end of the expression. The supplied directory **Test-errors** contains a test file for verifying that your error recovery works.

---

## Chapter 14

# PDef Syntax Tree Tutorial

**Goal:** To complete the design, implementation and testing of a parser for PDef that generates a syntax tree object. The completed system will be able to generate and display the syntax tree for the specified input file.

**Strategy:** The tutorial will begin with a discussion of strategies for designing a syntax tree based on a CFG. Starting with a partial syntax tree generator for PDef-*lite*, you will apply the design principles to complete the syntax tree for PDef-*lite* and then extend that to a syntax tree generator for PDef.

**Assumptions:** The tutorial requires no specific theoretical background other than familiarity with context free grammars.

**Java Features:** In this tutorial you will use the features of Java related to subclasses, including inheritance and late binding. You will also make use of the generic class `LinkedList`.

**Code Base:** You will work with a completed Java implementation of the parser for PDef. The parser, however, has been extended to include an partial set of syntax tree class definitions for PDef-*lite*. A table describing the components of the implementation is displayed in Figure 14.1. It is important that you look over this code base to understand what is there before you embark on the tutorial. Also, you will want to have the code base available (on line or in printed form) for reference as you progress through the tutorial.

### The Tutorial

We are not finished with the parser. As currently constructed it is fine for checking syntactic correctness, but we are also interested in a certain level of semantic correctness in our PDef strings and that requires further processing. A syntax tree is meant to facilitate any processing that

Code Base	
File	Implements
PDef.java	This file contains the class PDef and its method main which is the driver for the PDef syntax tree generator.
package	
debug	This package is the same as for the last tutorial but has added <code>SyntaxTreeDebug</code> , which facilitates debugging of the code for the syntax tree.
exceptions	This package contains the classes defining various exceptions that are thrown in this system.
tokenizer	This package contains the complete code for a working version of the PDef tokenizer.
parser	This package contains the complete code for a working version of the PDef parser.
syntaxTree	This package contains the files that implement a partial syntax tree class hierarchy for PDef- <i>lite</i> . The class files included are <code>StmtList</code> , <code>BlockST</code> , <code>StmtST</code> .
Tests	This directory contains text files that can be used for testing the syntax tree driver program.

Figure 14.1: Syntax Tree Java Components

involves language characteristics that are context sensitive. For example, from a syntactic point of view the assignment entry ‘`a = x`’ is parsed in the same way regardless of where it appears in a PDef string. But this same assignment entry will be interpreted differently depending on whether `a` has the type `int` or the type `float`. The rationale for generating a syntax tree is to have an internal (in memory) representation of the input data so that these context sensitive characteristics can be checked. Figure 14.2 shows how the parser and syntax tree structures fit into the system built so far. There are three issues to be confronted in this tutorial: how to structure the syntax tree for PDef, how to modify the PDef parse methods so that they can generate an appropriate syntax tree, and how to display a syntax tree.

## 14.1 Supplied Java Code

The code distributed with this tutorial is based on a working version of the PDef parser resulting from the parser tutorial of the previous chapter. There are several components in the distribution

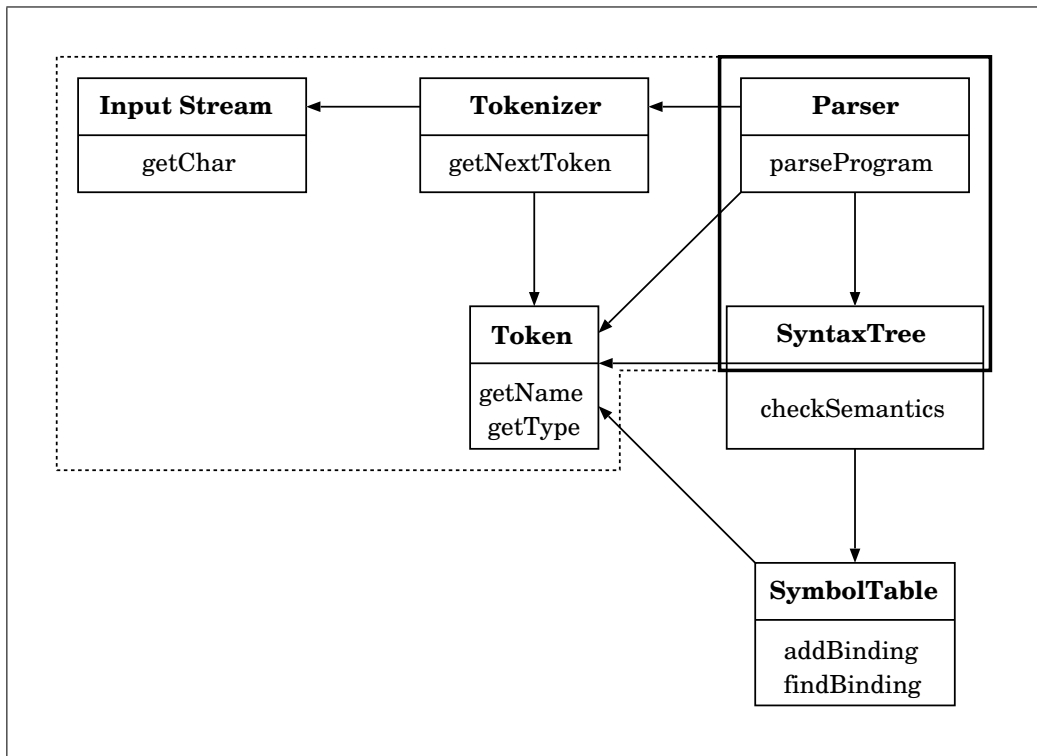


Figure 14.2: Adding Syntax Tree Generation to the Parser

that require some comment. You should be sure to view the distribution as you read this section – and of course will need access to it during the entire tutorial. In the brief sections that follow we will review the new and/or modified components in the distribution, beginning with the form of the driver class `PDef`. You should be able to parse the distributed code and execute it on the distributed test file. The execution should proceed as follows.

```

java PDef test
Program parsed!
Here's the Syntax Tree
BlockST
  
```

### The Driver – `PDef.java`

The method `main` in the class `PDef` is the driver program for testing the parser's construction of the syntax tree. This is simply an extended version of the driver for the parser tutorial. The function `main` differs from the previous one in that it adds one new debug object for syntax tree debugging and it tests not only the parser but also the syntax tree structures – with a quick look at `PDef.java` you should see the letter code to turn on syntax tree debugging. Here is the testing part of the function.

```

Tokenizer tokenStream = new Tokenizer(in, echo);
Parser parse = new Parser(tokenStream);

try {
    BlockST syntaxTree = parse.parseProgram();
    System.out.println("Program parsed!");
    System.out.println("Here's the Syntax Tree:");
    syntaxTree.traverseST();
}
catch (PDefException exc) {
    exc.print();
}

```

You can see that the `try` block is longer now. There is still a message to indicate that parsing was successful, which will be displayed if no exception is thrown, but there is also a message to display in advance of printing the syntax tree. Remember that if an exception is thrown during the execution of the method `parseProgram`, control transfers directly to the `catch`-clause, and the two output statements in the `try`-clause are not executed!

When testing this program you should be aware of one problem that can arise because of changes to be made to the parser. You might think that since the parser worked in the last tutorial that this one works as well. It is always best to make sure. Here is one approach. Once you have completed the modifications to the parse methods, you can comment out the line displaying the syntax tree, recompile, and then run the resulting program on the tests from the previous tutorial (the directory `Test`). This is a form of regression testing. What it will demonstrate to you is that you haven't broken anything in the process of modifying the parser.

### The Debug Environment

The syntax tree debugging environment is similar to those for the tokenizer and parser packages. A new class `SyntaxTreeDebug` has been added to the package `debug` and it is this class which is used as the type of the debug instance variable in the abstract class `SyntaxTree`, facilitating the debugging framework within all classes of the syntax tree hierarchy.

### The Syntax Tree

The package `syntaxTree` contains the Java definitions of the two syntax tree classes described in Section 7.3.3, `BlockST` and `StmtST`, in addition to the abstract class `SyntaxTree`. [The class `LinkedList` is built into Java so its implementation isn't included.] During the tutorial you will complete this package first for *PDef-lite* and later, in the Guided Development, for *PDef*. Remember that, since the class `SyntaxTree` is a super class of each class in the package, each class in the package inherits its own `debug` instance variable. This variable will be instantiated when the syntax tree node is created and, since `debug` is declared to be `protected`, the class code for each node has direct access to it.

There is an additional component present in the three syntax tree classes included in the distribution – namely, the method `traverseST`. The method is declared abstract in `StmtList`; since `StmtST` is an abstract subclass of `StmtList` it doesn't have to be defined. But in `BlockST` the implementation of `traverseST` described in Section 7.5 must be supplied – this is because `BlockST` is a subclass of `StmtST`. You will supply appropriate implementations for `traverseST` for the other syntax tree classes later in the tutorial. The traversal methods will provide a primitive way of

testing the structure of a generated syntax tree.

### The Parser

This file contains the PDef parser class resulting from the parser tutorial, but the class has been augmented in two ways. First, the parse methods `parseStmtList`, `parseBlock`, and `parseStmt` have been augmented, as described earlier in Section 7.4 and 7.4.3, to generate syntax tree nodes, though they may not be complete. The other methods for PDef-*lite* and for the arithmetic expressions have the form they had after the parser tutorial. The final point of interest is the method `parseAssignment`, whose code is displayed here.

```
public void parseAssignment() throws ParseException
// Grammar Rule: Assignment --> identT assignT identT
{
    debug.show(">>> Entering parseAssignment");

    consume(Token.TokenType.IDENT_T);
    consume(Token.TokenType.ASSIGN_T);
    consume(Token.TokenType.IDENT_T);
    // parseExp();

    debug.show("<<< Leaving parseAssignment");
}
```

Notice that the method call to parse an expression on the right-hand side has been commented out and replaced by a second call to `consume` – this is the `parseAssignment` appropriate for PDef-*lite*.

## 14.2 Syntax Tree Structure

In Section 7.3 we talked generally about the structure of a syntax tree, with examples drawn from PDef. With that analysis in hand we can address the specific structure of the syntax tree class hierarchy for the PDef-*lite* grammar. As in the case of the parser design, we will derive the syntax tree structures by focusing on the PDef-*lite* grammar and the six grammar rule Forms. Here we have the PDef-*lite* grammar for convenience.

```
Program      → Block eotT
Block        → lcbT StmtList rcbT
StmtList     → Stmt { commaT Stmt }
Stmt         → Declaration | Assignment | Block
Declaration → typeT identT
Assignment   → identT assignT identT
```

Remembering the six forms for syntax tree classes (Section 7.3.1), we can get an idea of a high-level view of the PDef-*lite* syntax tree hierarchy. In what follows we will also adopt a general naming strategy for syntax tree classes: other than the abstract class `SyntaxTree`, we will use the strategy of ending syntax tree class names with `ST`.

1. First, there will be an abstract class `SyntaxTree` as the top of the hierarchy.

2. When we examine the first rule we see an example of the special case for Form 2, when there is a single non-terminal and only punctuational terminals on the right. According to the special case the class associated with the non-terminal `Program` will be the class `BlockST`, corresponding to the non-terminal on the right side.
3. Remember that we examined the rules for `Block` and `StmtList` in Section 7.3.3 and deduced that the class for `StmtList` would be the Java class `LinkedList<StmtST>` and that the class for `Block` would be a class named `BlockST`. While the special case for Form 2 (just applied the `Program` grammar rule) would seem to apply to the `Block` rule, it was pointed out in Section 7.3.3 that the future need to associate the symbol table with the statement list means we apply Form 2 directly.
4. There are three rules for `Stmt` and so, by Form 4, we will have an abstract class called `StmtST` and then subclasses for each option. We have already discussed the class `BlockST` and now we see it must be made a subclass of `StmtST`. The other two subclasses follow from the last two rules: `Declaration` and `Assignment`.
5. The rule for `Declaration` is of Form 1 and so we will have a class named `DeclarationST` that will be a subclass of `StmtST`.
6. The rule for `Assignment` is similarly of Form 1 implying a class named `AssignmentST` that will be a subclass of `StmtST`.

According to this analysis we have identified `SyntaxTree` as the root of the hierarchy, below it will be another abstract class `StmtST` and below it will be three subclasses, `BlockST`, `DeclarationST`, and `AssignmentST`. The final element is the class `LinkedList<StmtST>` which will be present in the hierarchy as a data member of `BlockST`. The UML class diagram for this PDef-*lite* hierarchy appears in Figure 14.3. Of course, each of these classes has an internal structure which has yet to be determined, but our initial analysis provides a valuable first view.

### 14.2.1 A Syntax Tree Example

Before finishing this section, it will be helpful to recap what we have done by taking a new syntax tree diagram and see if we can reconstitute the PDef-*lite* string from which it was derived. Our syntax tree is displayed in Figure 14.4.

The easiest thing to do is to start from the left, which is the highest, most abstract level, and work our way to the right. To start, then, we see that, as required by the syntax of PDef-*lite*, the leftmost component is a block labeled `BlockST`. A `Block` is defined to be a list enclosed in matched curly braces – in this case, by looking to the right, we see the list has two elements. The string must have the following structure

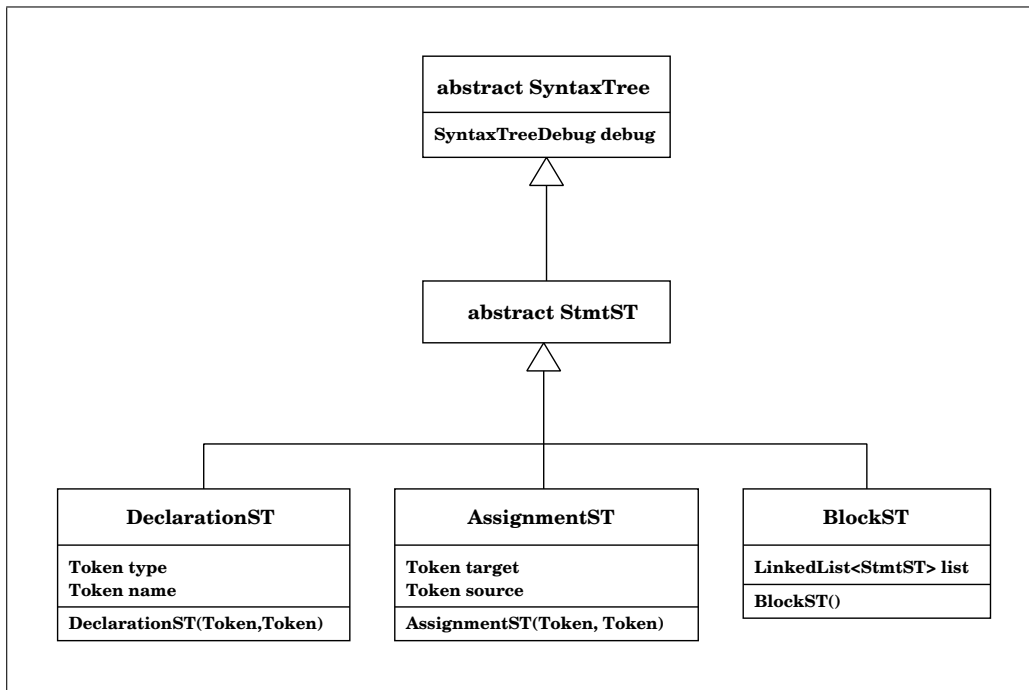
$$\{ X, Y \}$$

with `X` and `Y` being the elements in the list. Focusing on the first element, `X`, of the list we see that it is also a `BlockST`, i.e., it is also, in this case, a list of three elements. So our string has the following form.

$$\{ \{ X1, X2, X3 \}, Y \}$$

Returning the other element `Y` we see it is also a `BlockST` which, in this case, is a list of one element. So we expand our representation once more and get the following.



Figure 14.3: UML Class Diagram of the Syntax Tree for PDef-*lite*

```
{ { X1, X2, X3 }, { Y1 } }
```

Now we can visit these four remaining abstract elements one at a time. We see that none of them is a list, so each is either a declaration or an assignment and we glean the information easily from each boxed representation. We conclude that the original string had to be the following.

```
{ { float r, r = s, int s }, { int b } }
```

Why is this little exercise important? Because it shows that this syntax tree *is* an abstract representation of the structure of the PDef-*lite* string we started with. All the important data is there also, including the names of identifiers and the particular type keywords. So, if we want to check the original string for some property, we need only traverse the syntax tree and do the appropriate checking as we go.

### 14.2.2 Syntax Tree Classes

There are three class definitions which are provided in the distribution for this tutorial in the package `syntaxTree`: `SyntaxTree.java`, `StmtST.java`, and `BlockST.java`. The first, of course, is for the abstract class at the top of the hierarchy – the class definition in the file is as follows.

```
public abstract class SyntaxTree {
    protected SyntaxTreeDebug debug = new SyntaxTreeDebug();
}
```

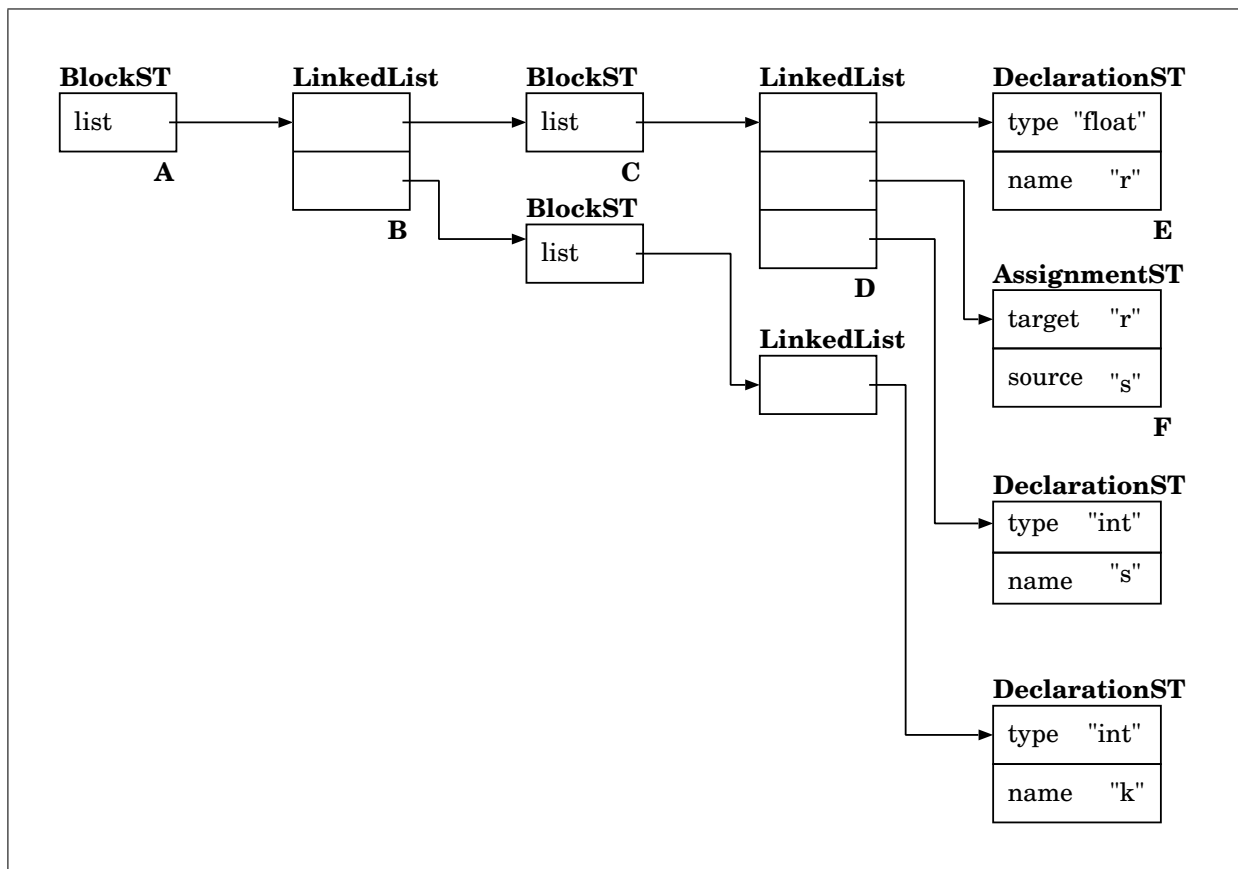


Figure 14.4: PDef-lite Syntax Tree

The second is for the abstract class at the top of the statement hierarchy.

```
public abstract class StmtST extends SyntaxTree { }
```

There are no constructors in these two classes because they are abstract. The protected data member `debug` in `SyntaxTree` is defined here so that all members of the hierarchy inherit it. The following code is from the third file and defines the class `BlockST`, as described in Section 7.3.3.

```
public class BlockST extends SyntaxTree {
    private LinkedList<StmtST> list;
    public BlockST(LinkedList<StmtST> l)
    { list = l; }
    public void traverseST() {
        // for (StmtST st : list)
        //     st.traverseST();
        System.out.println("BlockST");
    }
}
```

The data member `list` in `BlockST` represents the non-terminal `StmtList` on the right side of the grammar rule for `Block`. Of course there are two other classes in the hierarchy that must be

defined and implemented before the PDef-*lite* syntax tree is complete. Also note that the lines in `traverseST` are commented out to make the distributed system compile and execute. The system in its incomplete form generates an empty list, which causes a runtime error.

#### ☛ Activity 44 –

Following the strategies described in Section 7.3.1, implement class definitions for the remaining syntax tree class: `DeclarationST` and `AssignmentST`. Be sure to document the definitions appropriately, as illustrated in the supplied class definitions and the recommendations made in Section 7.3.1 – being sure to use class invariants and pre and postconditions as illustrated and where appropriate.

It is a good idea for classes of this form to include accessor methods for each data member. While these may not be used, having them present provides more flexibility.

These files cannot be tested at this point – we must wait for the parser modifications, which will facilitate generation of the syntax tree.

## 14.3 Traversing the Syntax Tree

In Section 7.5 we discussed a simple mechanism of tracing a depth-first traversal of a syntax tree, using a collection of methods named `traverseST`, one for each class in the syntax tree hierarchy. On page 117 we discussed the implementations for `traverseST` for the classes `BlockST` and `DeclarationST`. The provided code for these two classes contains appropriate implementations of `traverseST` method. The methods are shown here.

Notice that since the method `traverseST` is the same in all of the files, we should also find an abstract method in `SyntaxTree`. Notice that it is the method `BlockST.traverseST` which is responsible for forcing calls to the traversal method down the syntax tree.

```
// code for DeclarationST.traverseST
public void traverseST() {
    System.out.println("DeclarationST");
}

// code for BlockST.traverseST
public void traverseST() {
    for (StmtST st : list)
        st.traverseST();
    System.out.println("BlockST");
}
```

#### ☛ Activity 45 –

Following the strategies described in Section 7.5, implement a method `traverseST` in each of the remaining syntax tree classes: `StmtST`, `DeclarationST`, and `AssignmentST`. To help, the design table for `traverseST` for the language PDef-*lite* is in Figure 14.5.

class of node	layout description
<code>SyntaxTree</code>	The method is declared <b>abstract</b> .
<code>StmtST</code>	Inherits the method from <code>SyntaxTree</code> .
<code>DeclarationST</code>	This is a leaf node so display the name <code>DeclarationST</code> .
<code>AssignmentST</code>	This is a leaf node so display the name <code>AssignmentST</code> .
<code>BlockST</code>	This is an internal node and all links to subtrees are stored in the data member <code>list</code> . The first thing we do is to step through <code>list</code> and call <code>traverseST</code> on each of its elements, thus displaying each subtree referenced in the list. Then we display the name <code>BlockST</code> .

Figure 14.5: Simple Traversal Design Table

Notice that the definitions for the method have already been added to the classes `SyntaxTree` and `BlockST` – you will need to add the definition above to the class `DeclarationST`. Why is it not necessary to add such an abstract entry in `StmtST`?

This work cannot be tested at this point – we must wait for the parser modifications which will facilitate generation of the syntax tree.

## 14.4 Enhancements to the Class Parser

In Section 7.4.1 we discussed the general strategy, based on Forms 1-6, for defining parse methods that generate appropriate syntax tree nodes. In Section 7.4.3 we determined the appropriate modifications for the parse methods `parseBlock` and `parseStmtList` from `PDef`. We review the modifications for these methods here, once again.

```
private BlockST parseBlock () throws ParseException
// Grammar Rule:  Block --> lcbT StmtList rcbT
{
    debug.show(">>> Entering parseBlock");

    consume(Token.TokenType.LCB_T);
```

```

    LinkedList<StmtST> list = parseStmtList();
    consume(Token.TokenType.RCB_T);

    debug.show("<<< Leaving parseBlock");
    return new BlockST(list);
}

```

This method is quite simple and illustrates the synergy between the parser and the structure of the syntax tree. During its execution, `parseBlock` obtains a list of statements returned by the call to `parseStmtList`. The list of statements is passed to the constructor for `BlockST` as an argument and the constructed object returned as `parseBlock`'s return value. Our method `parseBlock` simply returns the object returned by `parseStmtList`.

The following method, on the other hand, is more complex. The method `parseStmtList` needs to construct a `LinkedList<StmtST>` object, fill it with the syntax tree objects corresponding to the statements in the list, and then return the filled list as the method return value. Notice below, that each time `parseStmt` is called, the return value is saved and added to the end of `list`. When the loop is complete (no `COMMA_T` token seen) the filled list is returned as the methods return value.

```

private LinkedList<StmtST> parseStmtList()
// Grammar Rule:  StmtList --> Stmt { commaT Stmt }
{
    debug.show(">>> Entering parseStmtList");

    LinkedList<StmtST> list = new LinkedList<StmtST>(); // this list is empty
    StmtST element = parseStmt();
    list.addLast(element);
    while (currentToken.getType() == Token.TokenType.COMMA_T) {
        consume(Token.TokenType.COMMA_T);
        element = parseStmt();
        list.addLast(element);
    }
    debug.show("<<< Leaving parseStmtList");
    return list;
}

```

#### Activity 46 –

Following the examples above and the design process discussed in Section 7.4, complete the modification of each of the other parse methods so that the resulting parser will generate appropriate syntax trees.

At this point you can test your parser/syntax tree fusion. First, compile and execute the file `PDef.java` (as described earlier) and execute it on files you used to test your parser in the last lab — you should be able to enter the directory `Test` and run the supplied script.

Now test the syntax tree structure. Uncomment the line

```
    syntaxTree.traverseST();
```

at the end of the `try` block in `PDef.main`. Compile this and test it on the file suggested in Section 7.5 (page 118). Compare your results to those expected.

---

## 14.5 Pretty-printing for the Syntax Tree

While not part of the design of the syntax tree structure, it is important to consider how to display a printed form for our syntax tree – what is commonly referred to as “pretty-printing”. The DFS `traverseST`, developed earlier, helps a little by showing what kinds of nodes are visited in traversing a tree. But it would be more useful if we had the ability to display the data contained in the tree as well as the structure of the tree. This capability would be critical for testing and debugging of the syntax tree generator. Because of the nested nature of the *PDef-lite* language, we should have a display mechanism that will show the nesting levels along with the structure of each level. This is a standard algorithmic problem and in this section we will look at the solution.

### 14.5.1 Analyzing the Syntax Tree Pretty-printing Problem

The best way to begin the design process for a display algorithm is to take an example input and sketch out what output is desired. In the process one can determine details of indentation, line spacing, and where new lines should start. So we begin our process in that way with the following string.

```
{ int a, { float r, r = s, int s, { int b } }, a = a }
```

We will require this string to be displayed in the following form.

```
List:
  Declaration: [ type int, name a ]
  List:
    Declaration: [ type float, name r ]
    Assignment: [ target r, source s ]
    Declaration: [ type int, name s ]
    List:
      Declaration: [ type char, name k ]
  Assignment: [ target a, source a ]
```

Every list encountered has its elements listed, each starting on a new line. The declaration and assignment entries are described on a single line. When a list is encountered each of its elements is listed with an indentation. The first list begins “List:” with no indentation, but each of its elements has a four-space indent; each nested list seems to be handled the same way, with indentations of four spaces build up for each subsequent level.

From the Java perspective, since we have defined `SyntaxTree` as the head of the syntax tree hierarchy, it is appropriate for the traditional method `toString` in `SyntaxTree` to return a string containing a representation of the syntax tree. This isn’t quite enough. Imagine constructing the

string using only versions of `toString` at each node; it isn't clear how the padding is taken care of. If, however, we define a second, overloaded version `toString(String pad)`, then we can use the argument `pad` to instruct recursive calls to `toString` how much padding to use.

### 14.5.2 Designing the Pretty-printing Algorithm

Our display algorithm will be implemented as a traversal algorithm and we design and implement it by following the three-step procedure described in Section 7.5. But remember, we already have a structure for our traversal algorithm; here we need only focus on the problem of constructing a string that will carry the structure of our syntax tree.

#### Step 1

We want to design a method with signature `String toString(String pad)`.

Since statement objects require indentation at the beginning of their string results, each statement class version of `toString` needs a `pad` passed in to specify the indentation. Since this includes all of the non-abstract classes, we can place an abstract version of the method `toString` in `SyntaxTree` – it will thus be inherited in `StmtST` and will have to be implemented in each of the three statement classes.

One general assumption about the solution. The method `toString` is responsible for adding an end-of-line character at the end and the `pad` at the beginning of its return value.

#### Step 2

We must produce a table listing, for each node-class, the processing that must take place when the method is called. In the case of this algorithm, the processing is simply a description of the display form for each node type. The node types of course are `BlockST`, `DeclarationST` and `AssignmentST`. We have already discussed the layout and we summarize those requirements in the table displayed in Figure 14.6.

#### Step 3

From the table mentioned in Step 2 we see that there are only three classes needing implementations of the traversal method. One will be illustrated here, for `AssignmentST` and you will be responsible for the others. Because the data members of `AssignmentST` are tokens, we will have to extract the name by calling `getName()` on the token. The implementation is as follows.

```
// for AssignmentST.toString
public String toString(String pad)
// Pre: pad is a string consisting of a multiple of 4 blanks
{
    debug.show(">>> Entering AssignmentST.toString(String)");
    String val = pad;
    val += "Assignment: [ target " + target.getName();
    val += ", source " + source.getName() + " ]\n";

    debug.show("<<< Leaving AssignmentST.toString(String)");
    return val;
}
```

class of node	layout description
SyntaxTree	This class is abstract and the method is declared abstract here.
StmtST	This class inherits its declaration from SyntaxTree.
DeclarationST	Declaration: [ type <t>, name <n> ] This string value is preceded by the pad. The values for <t> and <n> are the string values for the data members <code>type</code> and <code>name</code> , respectively.
AssignmentST	Assignment: [ target <t>, source <s> ] This string value is preceded by the pad. The values for <t> and <s> are the string values for the data members <code>target</code> and <code>source</code> , respectively.
BlockST	List: <s1> ... <sn> "List:" will be preceded by the pad. In this display each <si> is the string that represents the <i>i</i> th StmtST reference in the list. The display for each list entry has a four-blank pad, in addition, of course, to the padding already required of the list itself.

Figure 14.6: Syntax Tree Pretty-printing: Traversal Design Table for PDef-*lite*

Using `toString(String)` we can generate an appropriate string for a complete syntax tree (where the first list has no indent) by calling `toString("")`. But at the highest level, it seems unreasonable to expect the user of the syntax tree hierarchy to realize the need for the parameter to `toString`. For this reason, we include the following method in the class `SyntaxTree`.

```
// for SyntaxTree.toString
public String toString()
{
    debug.show(">>> Entering SyntaxTree.toString(String)");
    String val = this.toString("");
    debug.show("<<< Leaving SyntaxTree.toString(String)");

    return val;
}
```



### Activity 47 –

Following the example and the design process discussed above, complete the methods `toString` in `DeclarationST` and `BlockST`. A hint for the `BlockST` version of `toString`: since `toString` is a traversal, its version in `BlockST` should have the same structure as the basic traversal method `Block.traversalST`. Assuming that structure (a `for` loop with recursive calls of the form `entry.traverseST()`) we can see the return value for `toString` constructed by concatenating the strings returned by the recursive calls to `toString`. Also, since statement displays have an additional four blanks of pad from the string "List:", the argument to the recursive calls to `toString` must be the pad plus an additional four blanks, as follows.

```
entry.toString(pad + "    ")
```

This is how we drive an increased indentation down the hierarchy.

Having implemented the `toString` methods you should test your work by replacing the call to `traverseST` in the `PDef.main` by a call to `toString()`. The output from the program should “match” the form of the input string.

## 14.6 Guided Development – PDef Syntax Tree

Our goal is to extend the syntax tree structure and modify the parser so that it generates a syntax tree for `PDef`. Mimicking the process discussed earlier in this tutorial, you will design and implement syntax tree classes for the `PDef` expression grammar and also modify the expression parse methods to generate the appropriate syntax tree structures. You will be guided by the following activity.

In Section 2.2 two grammars were presented for describing arithmetic expressions. The first grammar has the following rules and has the unfortunate characteristic of being ambiguous.

```
Exp → Exp (addT | subT | multT | divT | modT) Exp
    → intT
    → floatT
    → identT
    → lpT Exp rpT
```

The second grammar is the one adopted for the `PDef` grammar.

```
Exp → Exp (addT | subT) Term
    → Term
Term → Term (multT | divT | modT) Factor
    → Factor
Factor → intT |
        → floatT |
        → identT |
        → lpT Exp rpT
```

Even though the two grammars describe the same set of strings, since the first is ambiguous, it is inappropriate for parsing. However, the fact that the first grammar is ambiguous is not a problem in the context of designing a syntax tree – that is because the construction of the syntax tree is done from within the parse methods which have the structure of the second grammar. The crucial facts are that the two grammars describe the same strings and that the first is simpler than the second.

There is one simplification we can make to the first grammar – again it is possible because the syntax tree structures are generated from within the parse methods. When we look at the last rule of the first grammar we see on the right-hand side two terminals that carry no actual data – the `lpT` and `rpt`. Because they don't represent data, the tokens will not be represented in the syntax tree. So we could modify the grammar by simply removing the tokens. But then our last rule is simply `Exp → Exp` and it can be eliminated without losing any information. Notice that any nested structure can be described simply with the first rule, which has `Exp` appearing twice on the right-hand side. So our PDef syntax tree will have its structure defined based on the following grammar.

```
Exp → Exp (addT | subT | multT | divT | modT) Exp
    → intT
    → floatT
    → identT
```

We could arrive at this grammar by a more analytical strategy. Consider the original non-ambiguous PDef grammar above. The second rule says `Exp → Term` – i.e., the non-terminals `Exp` and `Term` are equivalent. So we can substitute `Exp` for `Term` in the rules. But the same can be said for `Factor` (see the the fourth rule), so we can replace `Factor` by `Exp` as well. This gives us the following grammar.

```
Exp    → Exp (addT | subT) Exp
       → Exp
Term   → Exp (multT | divT | modT) Exp
       → Exp
Factor → intT |
       → floatT |
       → identT |
       → lpT Exp rpT
```

Obviously rules two and four can be eliminated and rules one and three can be combined. The result is the ambiguous grammar we adopted earlier for our syntax tree structure. We will use the verb *flatten* to describe the process of converting a non-ambiguous grammar to a more compact grammar appropriate for syntax tree design. The flattening process is what we just went through. So when designing a syntax tree for a translator one should start with the grammar used to structure the parser and flatten the grammar to the more compact form.

#### ☛ Activity 48 –

1. Design and implement appropriate classes for the syntax tree based on the first expression grammar above. As a hint, there should be one abstract class, call it

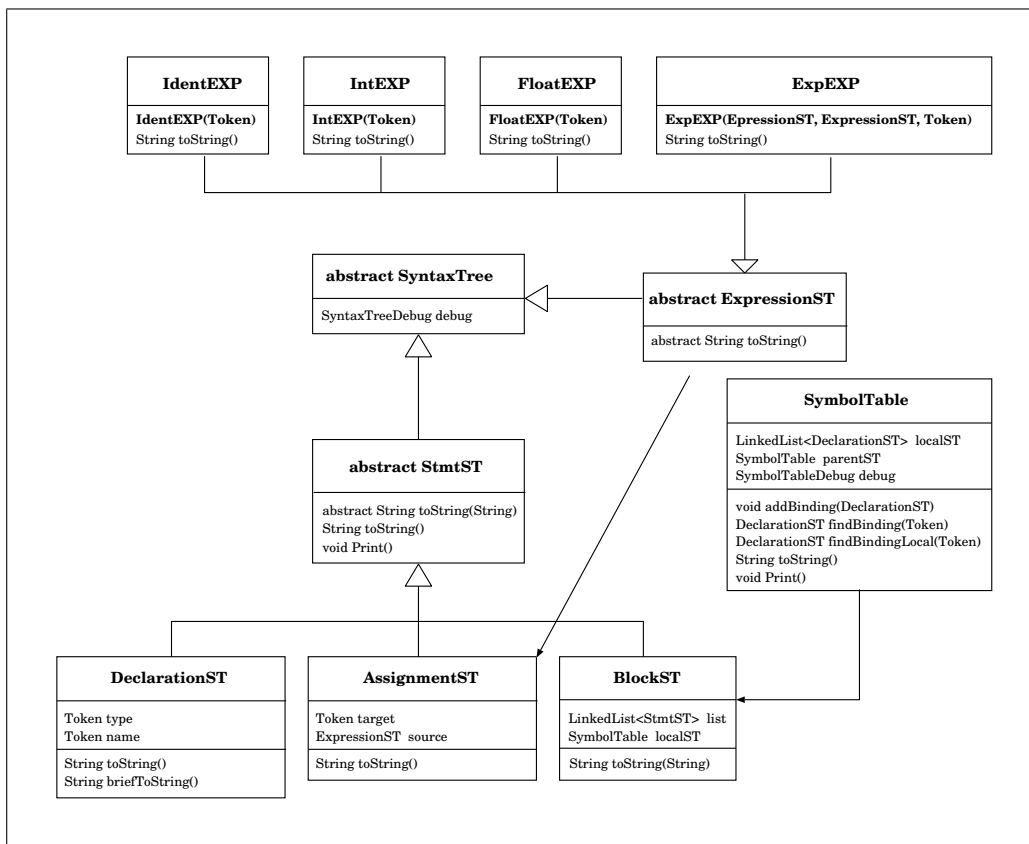


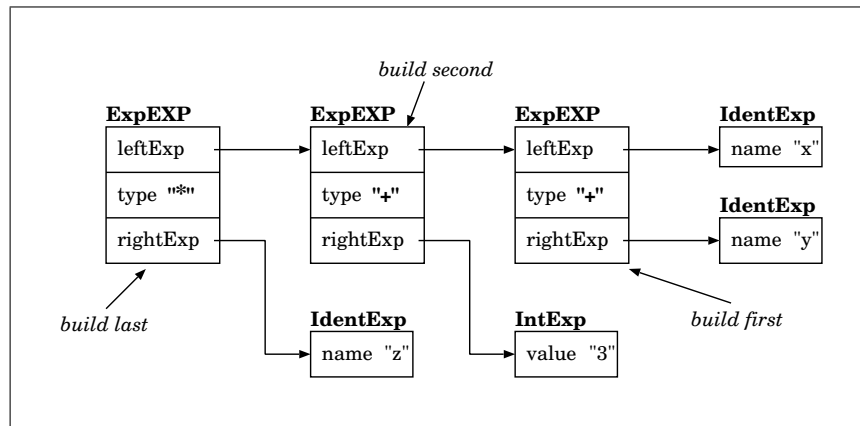
Figure 14.7: UML Class Diagram of the Syntax Tree for PDef

`ExpressionST`, and it should have four subclasses, one for each of the first four lines of the first grammar – the new hierarchy is given in Figure 14.7. Notice in this diagram that there is a naming adjustment: since it is possible for the token `identT`, for example, to appear on the right side of more than one rule of Form 4 (i.e., with options), we use an abbreviation of the name of the super class as a suffix on the subclass names in place of the suffix `ST`. Review the strategy described in Section 7.3.1. At the same time add appropriate implementations in each new class of the method `traverseST`.

2. Modify the expression parse methods in the supplied parser class so that they generate appropriate syntax tree objects. It is important that you review the strategies discussed in Section 7.4 and, for `parseExp` and `parseTerm`, review Section 7.4.2 (the case for left-recursive rules). As an example, the small syntax tree for ‘`(x + y + 3) * z`’ will have the structure shown in Figure 14.8.

When the parse methods have been completed run and test the program twice. The first test is to insure that the parsing methods aren’t broken. The second test is to test the syntax tree structure, which can be done by traversing the syntax tree using the `traverseST` method. Given the input

```
{ int x, float y, y = x + 3, { int a, a = x + y }, y = x }
```

Figure 14.8: Syntax Tree for  $(x + y + 3) * z$ 

the program should print the following list of class names (comments on the right are not part of the output) – remember that the order of traversal is depth-first.

DeclarationST	for int x
DeclarationST	for float y
IdentEXP	for x
IntEXP	for 3
ExpEXP	for x + 3
AssignmentST	for y = x + 3
DeclarationST	for int a
IdentEXP	for x
IdentEXP	for y
ExpEXP	for x + y
AssignmentST	for a = x + y
BlockST	for the inner list
IdentEXP	for x
AssignmentST	for y = x
BlockST	for the outer list

- Having completed the implementation for the syntax tree for PDef, the final task is to extend the definition of the method `toString`, defined for `PDef-lite` in Section 14.5, to the classes of the expression hierarchy. You are to provide appropriate implementations for `toString()` in each new class so that expressions will be displayed in place of the source identifier. Assume that the entire expression is to be displayed on a single line (don't add a final line-break character) and that the PDef string

```
{ int a, float b, { int x, a = x + b - a - b }, b = (a + b) * b
- a }
```

will be displayed as follows.

List:

Declaration: [ type int, name a ]

Declaration: [ type float, name b ]

List:

Assignment: [ target a, source ( ( x + b ) - a ) - b ) ]

Assignment: [ target b, source ( ( a + b ) \* b ) - a ) ]

Notice that you will have to recover the particular operation to display from the value of the operator token; the particular operation is there in the form of the **name** component in the **Token** object. Finally, notice that since the expression display occurs at the end of the line, there is no need for padding. Thus, in the expression syntax tree classes the display method should be a parameterless version of `toString`.

class of node	layout description
ExpressionST	This is an abstract class – the method is abstract in this class.
ExpEXP	( <v> <op> <w> ) The values <v> and <w> are determined by recursively traversing the data members <code>leftExp</code> and <code>rightExp</code> . The value of <op> is the string value of the data member <code>op</code> .
IdentEXP	<v> The <v> represents the string value of the data member <code>value</code> .
IntEXP	<v> The <v> represents the string value of the data member <code>value</code> .
FloatEXP	<v> The <v> represents the string value of the data member <code>value</code> .
AssignmentST	Assignment: [ target <t>, source <s> ] This string value is preceded by the pad. The value of <t> is the value of data member <code>target</code> and <s> is determined by recursively traversing <code>source</code> .

Add the methods `toString` described above to each class. Compile and test the implementation as in Activity 47.



## Chapter 15

# PDef Semantics Tutorial

**Goal:** To add semantic checking to the existing PDef implementation. This is done in two steps: first integrate a symbol table structure into the syntax tree; second carry out semantic checking for PDef in accordance with the semantic definition for the language.

**Strategy:** The tutorial is in two parts. The first part begins with a discussion of identifier semantics in statically typed languages and the basic structure of a symbol table. The strategy for integrating the symbol table into the PDef syntax tree is laid out and a DFS algorithm is designed for instantiating, filling and printing the symbol table. The second part discusses the design and implementation of semantic checking first for PDef-*lite* and then for PDef.

**Assumptions:** The tutorial assumes that you have been introduced to the basic concepts surrounding the static semantics of identifiers, including concepts for block structured languages such as the scope of a declaration and the implications of nested scopes for the same identifier. These ideas will be briefly reviewed.

**Code Base:** You will work with a completed Java implementation for the previous tutorial. There is also a new package containing the implementation of a basic symbol table – the code is complete except for three methods which are present but not complete. The exception class hierarchy is significantly extended for this tutorial. A table describing the components of the implementation and what they implement is displayed in Figure 15.1. The code base is reviewed later in this tutorial. You will want to have the code base available (online or in printed form) for reference as you progress through this tutorial. The code distributed for the tutorial can be compiled and executed on a PDef-*lite* input file but there is no semantic checking or symbol tree building.

Code Base	
File	Implements
PDef.java	This file contains the class PDef that contains the driver for testing the symbol table implementation.
package	
debug	This package is the same as for the last tutorial but has added the class definition <code>SymbolTableDebug</code> , which facilitates debugging of the code for the symbol table.
exceptions	This package contains the classes defining various exceptions that are thrown in this system. New exception class definitions include <code>AlreadyDeclaredException</code> , <code>AssignCompatException</code> , <code>DeclarationException</code> , <code>NotDeclaredException</code> , <code>OperationException</code> , and <code>UseException</code> .
tokenizer	This package contains the complete code for a working version of the PDef tokenizer.
parser	This package contains the complete code for a working version of the PDef parser that generates the syntax tree appropriate for the provided input.
syntaxTree	This package contains the files for the implementation for the syntax tree class hierarchy for PDef.
symbolTable	This package contains the complete framework for a working version of the PDef symbol table – it mostly is complete.
Test	This directory contains text files that can be used for testing the symbol table driver program.

Figure 15.1: Symbol Table Java Components

## The Tutorial

The first three tutorials have resulted in a PDef recognizer that recognizes syntactically correct PDef strings and also produces a syntax tree whose structure and content matches that of the input string. In this tutorial you will integrate into the syntax tree structure symbol tables that will



contain the static semantic data associated with each identifier in the syntax tree. The enhanced syntax tree will make it possible to check that an input string is not only syntactically correct, but also semantically correct. Figure 15.2 shows the structure of the symbol table and semantic checking within the structure of the entire system.

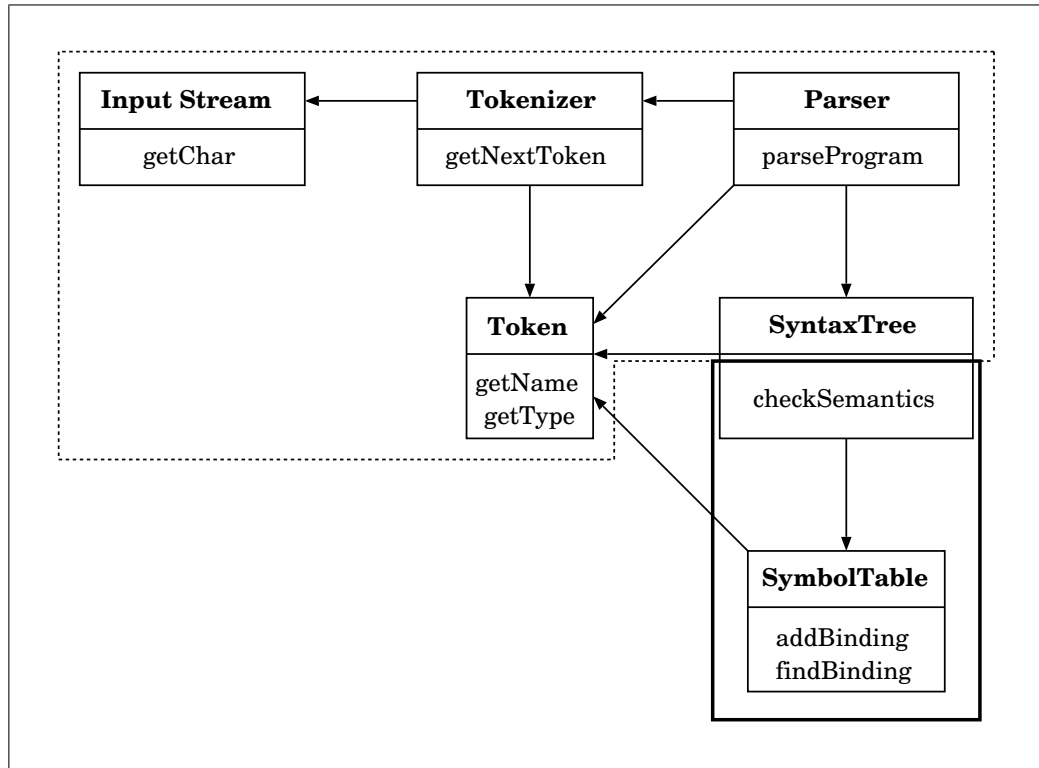


Figure 15.2: Adding Symbol Table and Semantic Checking to the Syntax Tree

In a language processor the *symbol table* is used to store bindings between identifiers and their attributes. In statically typed languages some attribute bindings are specified explicitly in declaration statements (the identifier is declared to be of type `int`, for example) or implicitly (the literal `3.4` is implicitly typed `float`). The symbol table is used in checking static semantics and, where appropriate, in generating code. In this tutorial we will focus on the structure of the symbol table, its integration into the syntax tree, and the semantic checking this integration facilitates.

## 15.1 Symbol Table Structure

In Figure 8.5 we see the form we would like the symbol table to have in an implementation; the crucial ideas are the local symbol table linked to its parent symbol table, and the relationship between the syntax tree and the symbol table. Figures 8.4 and 8.5 clearly show that these two ideas are independent, so we can approach their design and implementation separately, which we do starting in the next section.

### 15.1.1 Symbol Table Design

The structure of the entire symbol table is that of an inverted tree of local symbol tables linked together, in the upwards direction, via their respective parent references. So the basic element to design is that of the local symbol table that must have a list of bindings, a parent reference, and the ever present `debug` reference.

There are really just two decisions to make in designing the local symbol table structure: how should the list of bindings be structured and how should the bindings themselves be represented. As in the syntax tree, we can use the Java generic class `LinkedList` to implement our list structure. You should review the discussion of this generic class in Section 11.2.

Now, how should the bindings be represented? An identifier can have only one attribute bound to it in PDef-*lite*— a type attribute. We require a structure whose data members are the identifier name and the bound type. But we already have a class with this exact structure: the class `DeclarationST`. So we will use the `LinkedList` and the `DeclarationST` classes as our basic building blocks of the symbol table structure, along with the class `SymbolTableDebug`, of course. The following skeleton class structure shows the public interface to a `SymbolTable` object.

```
public class SymbolTable {

    private LinkedList<DeclarationST> localST;
    private SymbolTable          parentST;
    private SymbolTableDebug     debug;

    public SymbolTable (SymbolTable up) {...}
    // Post: parentST == up AND localST != null AND
    //       debug == new SymbolTableDebug()

    public void addBinding(DeclarationST entry)
        throws AlreadyDeclaredException {...}
    // Pre:  entry != null
    // Post: if an entry already exists in localST for entry.getName()
    //       then an ALreadyDeclaredException exception is thrown
    //       otherwise entry is added to localST

    public DeclarationST findBinding(Token name) {...}
    // Pre:  name != null AND name.getType() == IDENT_T
    // Post: if localST.findBindingLocal(name) != null then return decl
    //       else if parentST != null return parentST.findBindingLocal(name)
    //       else return null

    public DeclarationST findBindingLocal(Token name) {...}
    // Pre:  name != null AND name.getType() == IDENT_T
    // Post: if decl in localST AND decl.getName() == name then return decl
    //       otherwise return null

    public String toString() {...}
}
```

```

// Post: return String value which is the concatenation of
//       entry.toString() for each reference entry on localST

public void print() {...}
// Post: the value this.toString() is displayed on standard out
}

```

The constructor for this class takes an extra parameter of type `SymbolTable`, obviously a reference to the parent symbol table. For the outermost list the parent symbol table will be indicated by the `null` reference. The method `addBinding` is the crucial method here, since there must be a way to populate the symbol table. Study the `pre` and `postconditions` for this method. When this method is called to add an item to the symbol table, if it finds an entry for the particular identifier already existing in `localST`, then it throws an exception. The exception `AlreadyDeclaredException` is discussed in Section 15.2.

### 15.1.2 Integrating the Symbol Table into the Syntax Tree

Integrating this symbol table structure into the syntax tree is straightforward with the diagram in Figure 8.5 giving a good indication of how to proceed. We provide each local list of entries, that is instances of `BlockST` objects, a reference to a local symbol table. This requires the addition to the class `BlockST` of a private data member, a reference to a `SymbolTable` object. This value of `localST` will be initially set to `null` and will only be instantiated when the symbol table initialization is carried out during semantic checking.

```

public class BlockST extends StmtST {

    private LinkedList<StmtST> list;
    private SymbolTable      localST;

    public BlockST(LinkedList<StmtST> s1) {
        super(db);
        list    = s1;
        localST = null;
    }
    ...
}

```

Figure 15.3 shows the new UML class diagram for our extended PDef-*lite* syntax tree class hierarchy. Notice that a diagram for `SymbolTable` has been integrated into the original hierarchy – also note the addition of an arrow from `SymbolTable` to `BlockST`, reflecting `BlockST`'s data member `localST`. Remember that in such diagrams, the open triangle indicate a subclass relationship with the class `StmtST`.

### 15.1.3 A Final Look at the Structure

The structure we have created is an interesting one. It has been derived from our intuitive understanding of the syntax tree and the symbol table as separate entities but also from our requirement

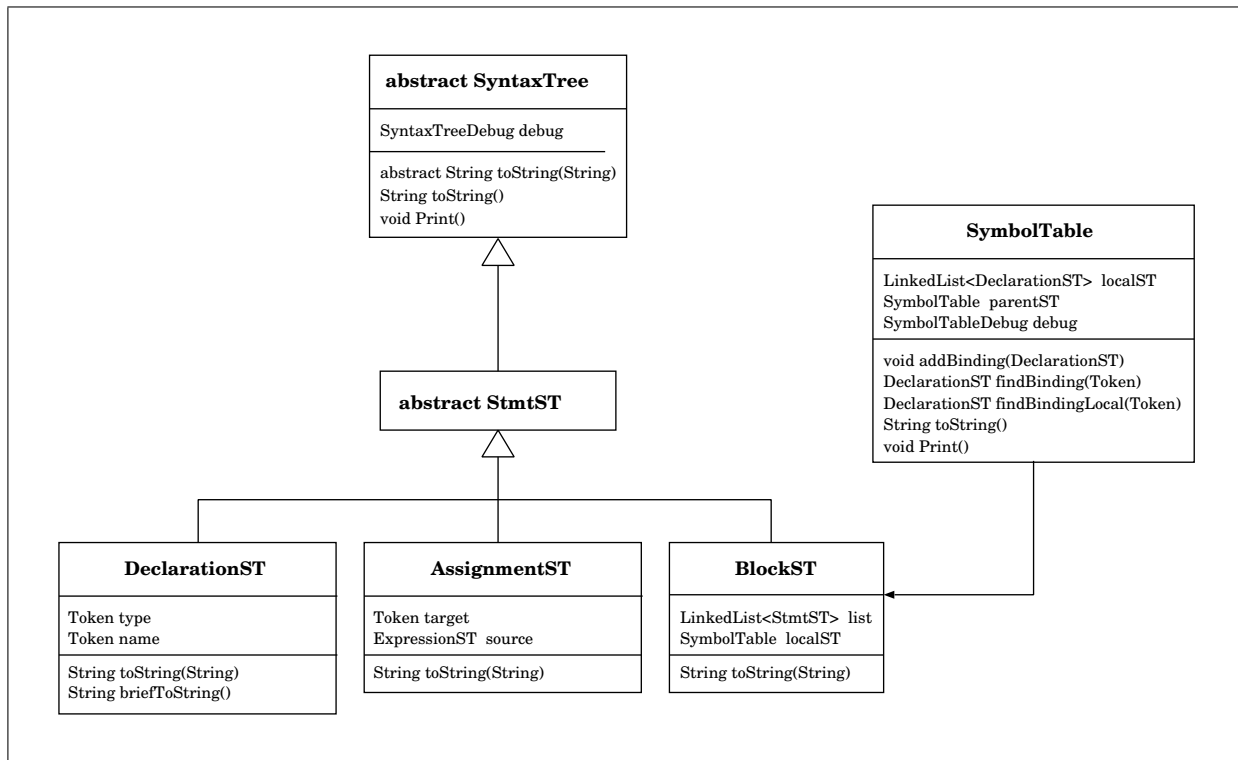


Figure 15.3: UML Class Diagram of PDef-lite Syntax Tree with Symbol Table

that they be linked together in a particular way. That linkage was described in Figure 8.5.

Figure 15.4 is a representation of the newly described structure that integrates the symbol table into the syntax tree structure. The similarity to the diagram in Figure 8.5 are unmistakable. But there is one interesting difference: both the syntax tree and symbol table share the `DeclarationST` objects created for the syntax tree structure.

One of the things our symbol table structure is supposed to implement is the scoping rules for PDef. We can see that this is accomplished by looking at the example that appears in Figure 8.4 and is represented in the syntax tree form in Figure 15.4. If we consider the identifier ‘a’ as it occurs in the assignment entry in the inner list, we can check to see if the reference is a valid one. First we check the local symbol table and see that there is no declaration entry for ‘a’, but if we follow the `parent` reference we see that the parent symbol table *does* contain a declaration for ‘int a’, which becomes the declaration for the ‘a’ in the inner list. While this is not a proof, it certainly makes it clear how the appropriate binding, if there is one, will be found for a particular identifier.

## 15.2 Extending the Exception Structure

We commented above in Section 15.1.1 that the method `addBinding` throws an exception, but an exception of a class different from `ParseException`. The idea in `inaddBinding` is that an exception should be thrown if the program tries to add a declaration for a name that is already declared in the local symbol table. This exceptional condition is not a parsing problem; rather it is a semantic checking problem. So we should have a new kind of exception to throw.

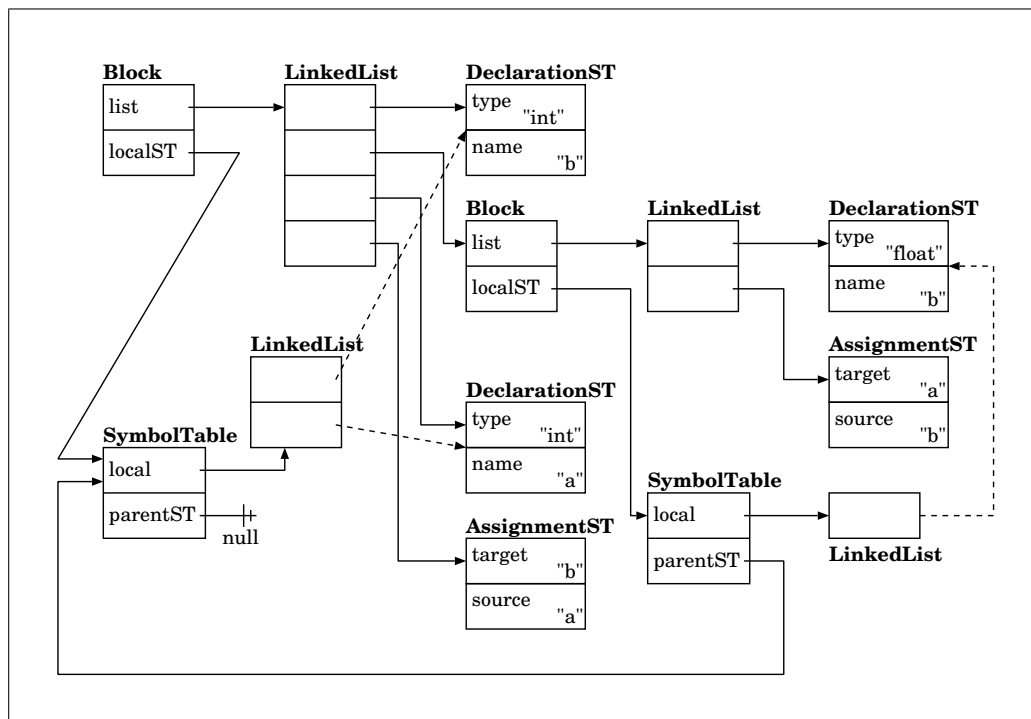


Figure 15.4: PDef-lite Syntax Tree with Symbol Table

When it comes to semantic problems, however, there seem to be several exceptional situations that can occur, and consequently, several exception classes:

**AlreadyDeclaredException:** An identifier is declared more than once.

**NotDeclaredException:** An identifier use falls outside the scope of any declaration for the identifier.

**AssignCompatException:** In an assignment statement the source type can be assignment incompatible with the target type

**OperationException:** In PDef, there can be two arguments for an arithmetic operator that are incompatible (in an expression).

These four situations are associated with four exception classes fall naturally into two categories: declaration exceptions and use exceptions. Our approach is to create a class hierarchy, with two classes (**DeclarationException** and **UseException**), one for each of the two categories – the structure is shown in Figure 15.5. Besides reflecting the natural structure of the exceptional conditions, the hierarchy provides the chance to construct error messages that have consistent structures across the five forms of exception condition.

One of the issues to be considered in expanding the details of the exception class hierarchy is to determine what kind of data is provided to the constructor for each of the four exception classes. In each case we will supply a message, just as in the case for **ParseException**. For the declaration exceptions we can supply the declaration found in the symbol table for the already-declared exception and the identifier's token for the not-declared exception. For the assignment compatibility exception we should supply the type tokens for the target and source of the assignment

statement. In the case of an operation exception, we should supply the type tokens of the left and right arguments (expressions) and, in addition, the operation for which the argument types are improper. Notice that these data requirements are reflected in the class descriptions in Figure 15.5.

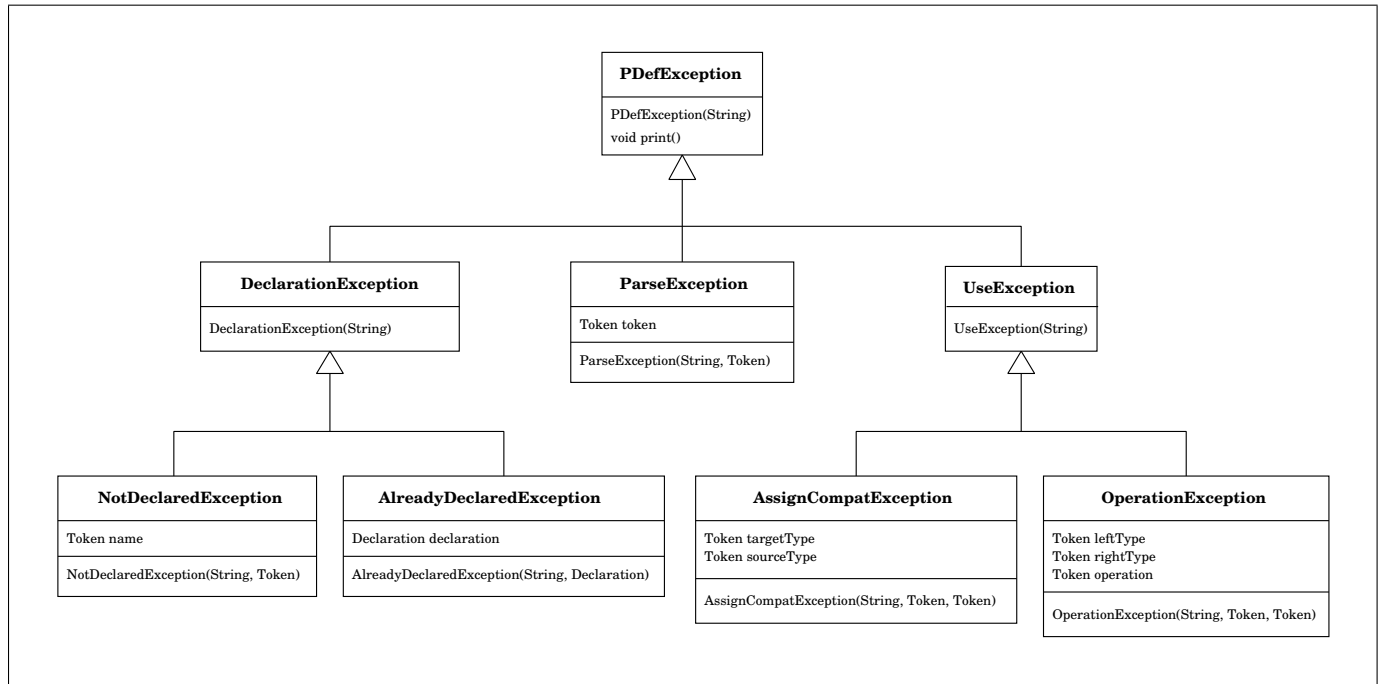


Figure 15.5: Exception Class Hierarchy

## 15.3 The Supplied Java Code

As with the earlier tutorials there is a partial implementation of the PDef recognizer including the symbol table that you will reference and modify in the rest of the tutorial. The supplied code is based on the final version of the recognizer from the syntax tree tutorial. That code is augmented by the symbol table implementation and a new exception hierarchy that reflects the conclusions of the previous section. In this section we will look at those parts of the code base that are new or altered from the syntax tree final implementation.

### 15.3.1 PDef.java

The method `main` in the class `PDef` is the driver program for testing the integration of the symbol table into the syntax tree and for evaluating correctness of PDef-*lite* strings; it is derived directly from the code for the syntax tree driver. The only modifications deal with establishing a new debugging flag for the symbol table object and the additional code necessary to check semantics and display the syntax tree, complete with symbol table.

For symbol table debugging the class `SymbolTableDebug` has been added to the `debug` package. In addition, the command line flag `'b'` has been assigned to trigger the symbol table debugging capability. Also notice that a new `boolean` flag has been added, `printSTree`, to specify that the

syntax tree should be printed. This flag makes it possible to print the syntax tree without getting all the other syntax tree debugging messages – this flag is triggered by the command line code ‘x’.

We will take a quick look at the testing half of the driver program.

```

Tokenizer tokenStream = new Tokenizer(in, echo);
Parser parse = new Parser(tokenStream);

try {
    BlockST syntaxTree = parse.parseProgram();
    System.out.println();
    System.out.println("Program parsed!");
    syntaxTree.checkSemantics();
    if (printSTree) syntaxTree.printST();
    System.out.println();
    System.out.println("Program passed semantic checking!");
}
catch (PDefException str) {
    str.print();
}

```

The call to `checkSemantics` handles two activities: building the symbol table and checking the *PDef-lite* semantic rules. The implementation of the semantic checking will be done in two phases – first the symbol table construction and second the semantic checking. `Pdef.main` above will facilitate the testing of both phases.

### 15.3.2 SymbolTable.java

In Section 15.1.1 we looked at the basic structure of the class `SymbolTable`, but we did not discuss any of the implementation details. In this section we will take a look at the primary `SymbolTable` methods, the implementations that are provided and the implementations you will be responsible for.

The responsibility of `addBinding` is to add its parameter to the symbol table. But if we remember the scoping rules from Section 8.1.3, we shouldn’t add a binding to the symbol table if its identifier already has a binding in the symbol table at the current level. This means that before adding something to the symbol table we should scan the table for the identifier in the parameter. If we find that the identifier is already present, then there is an error and an exception should be thrown. If the identifier is not present then we can go ahead and add the new binding to the symbol table. This, of course, can be implemented in a single method, but there is a benefit in clarity and modularization if we implement the method using a helper method, `isAtThisLevel`. Here’s the supplied code for `addBinding`.

```

public void addBinding(DeclarationST entry)
    throws AlreadyDeclaredException
// Pre:  entry != null
// Post: if an entry already exists in localST for entry.getName()
//       then an AlreadyDeclaredException exception is thrown

```

```

//      else entry is added to localST
{
    debug.show("-->>> Entering addBinding" + entry);

    if ( isAtThisLevel( (entry.getName()).getName() ) )
        throw new AlreadyDeclaredException
            ("Identifier already defined!", entry.getName());

    // Assert: n.getName() not yet defined at this level
    localST.addLast(entry);

    debug.show( "<<<--- Leaving addBinding", localST.size() );
}

```

The helper method, `isAtThisLevel`, reports if there is already a declaration entry for the parameter in the local symbol table. This new method is a `private` method that does a sequential search through the symbol table `localST`. Here is the code.

```

private boolean isAtThisLevel(String n) {
    debug.show("-->>> Entering SymbolTable.isAtThisLevel");

    ListIterator<DeclarationST> itr = localST.listIterator();
    boolean found = false;
    while (!found && itr.hasNext()) {
        DeclarationST ste = itr.next();
        if ( n.equals(ste.getName().getName()) )
            found = true;
    }
    debug.show( "<<<--- Leaving SymbolTable.isAtThisLevel" );
    return found;
}

```

This code should look familiar and, if not, refer back to Section 11.2.2 where we discussed it and gave an example of the use of the Java iterator concept. Study and understand this example since the technique using the iterator will be repeated in other classes.

Finally, there are three stubbed methods in the `SymbolTable` class definition: `findBinding`, `findBindingLocal`, and `toString`. The implementations of these methods will be discussed later in the tutorial.

### 15.3.3 `syntaxTree` package

The package `syntaxTree` is the same as that produced in the Syntax Tree Tutorial.

### 15.3.4 `exception` package

These files implement the new extended exception structure described above in Section 15.2.



## 15.4 Implementing Semantic Checking

As described in Chapter 11, PDef-*lite* uses define-before-use scoping, thus semantic checking must be accomplished in a single traversal of the syntax tree. During the traversal the semantic checker must not only build the symbol table but also check that identifiers are used in accordance with the bindings stored (so far) in the symbol table – these responsibilities are summarized in Figure 8.6. Even though we need only build a single traversal semantic checker, it will be easiest if we break the development into two parts – building the symbol table and checking semantics. The activities in this section will lead you through the implementation of the PDef-*lite* semantic checker.

### 15.4.1 Creating and Filling the Symbol Table

Before getting started, we will address a new debugging problem that will emerge. The class `SyntaxTreeDebug` facilitates the debugging of basic syntax tree functionality already implemented. But since we are embarking on a new functionality for the syntax tree, it can be useful to have a debugging facility directed specifically at semantic checking. The following Activity sets in place this capability.

#### ☛ Activity 49 –

First, create a new debug class called `SemanticCheckDebug` and use the command line code ‘b’ as its trigger. With that class in place, modify `PDef.java` to deal appropriately with the new command line code. Finally, add to the class `SyntaxTree` a new debug data member as for `debug` – call the new data member `debugSC` (the SC is for “semantic checking”), and instantiate it in the class constructor.

As you progress through this tutorial, be sure that the entry and exit debug messages are guarded by `debugSC`, rather than `debug`.

While you are busy editing code, edit `BlockST` in the package `syntaxTree` and add a new data member named `localST` and declare it of type `SymbolTable`. It should not be given a value at this time.

---

With all the data from the input stream already in the syntax tree, we can instantiate and fill the local symbol tables in each of the `BlockST` objects by performing an appropriate traversal. The basic strategy is to do a DFS and for each `BlockST` object encountered, instantiate a `SymbolTable` object and then fill it with references to the `DeclarationST` objects that appear in the `list` data member in that `BlockST` object. Our traversal method for `DeclarationST` will require a reference to the local symbol table, so it can add itself to the table. Also, any `BlockST` object on the list will need a reference to the local symbol table so it can be used as the nested list’s parent table. Before looking at the design steps it would be good to reflect on how little there is to do here. The only classes that need very much modification are `DeclarationST` and `BlockST`; the other classes need only facilitate the traversal. When we get to semantic checking we will find the same thing. All is made possible by the context free grammar, the syntax tree, and traversal. The simplicity is revealed by the three steps of the following traversal development.

**Step 1:** We will use the name `checkSemantics` for the methods that provide the symbol table filling functionality; the signature of these methods should be as follows.

```
public void checkSemantics(SymbolTable st);
```

The parameter provides a reference to the local symbol table in the `BlockST` object “above”. Since it is the statements in *PDef-lite* that are the focus of semantic checking, we place the following signature into the class `StmtST`.

```
public abstract void checkSemantics(SymbolTable st);
```

We might have put this signature into the `SyntaxTree` class, but we will find in the Guided Development at the end of this tutorial that placement in `StmtST` is best.

**Step 2:** This step requires that we produce a table containing, for each possible syntax tree node, a description of the processing that must be carried out in each `checkSemantics` method. Here is the table.

class of node	required semantic checking
<code>SyntaxTree</code>	No functionality is needed in this class.
<code>StmtST</code>	An abstract method declaration is put here to supply the signature for the methods in its subclasses.
<code>BlockST</code>	The argument passed into this method must be a reference to the local syntax tree of the calling <code>BlockST</code> object or <code>null</code> if this is the top of the syntax tree. Instantiate the <code>SymbolTable</code> object and assign its reference to the data member <code>localST</code> – use the method argument as the argument to the <code>SymbolTable</code> constructor. Traverse <code>list</code> , calling <code>checkSemantics</code> on each reference, passing <code>localST</code> as the argument.
<code>DeclarationST</code>	Add me (i.e., use <code>this</code> ) to the symbol table passed to me as a parameter.
<code>AssignmentST</code>	There is no processing necessary at this time.

**Step 3:**

☛ **Activity 50** –

Complete Step 3 by implementing the method `checkSemantics` in each of the classes `BlockST`, `AssignmentST`, and `DeclarationST`, as described in Step 2 above. Be sure to include entering and leaving debug statements in all methods. At this time `AssignmentST.checkSemantics` can be a stub.

---

It is important to see that in each method the purpose of the parameter `st` changes slightly. When passed to a `BlockST` object the parameter is the reference to the parent symbol table for the new symbol table about to be created. But when passed to `DeclarationST.checkSemantics` the value `st` is interpreted as a reference to the local symbol table to which this declaration should be added. The perspective from the method `AssignmentST.checkSemantics` is similar in that the correctness of the assignment is interpreted in the context of `st`.

One final detail. We would like to have a way to start the semantic checking process in the `try`-block in `PDef.main` without having to reference a symbol table, since none will be accessible in the method `main`. We know that the parse method `parseProgram` returns a `BlockST` object, so we can use a standard slight of hand: if we place the following definition in `BlockST` (notice there's no argument)

```
public void checkSemantics()
    throws AlreadyDeclaredException
{
    this.checkSemantics(null);
}
```

then we can get the symbol table generation started with the following line in the `PDef.main` `try`-block.

```
syntaxTree.checkSemantics();
```

Actually, this line of code is already there in the supplied `PDef` class definition.

### ☛ Activity 51 –

Add the parameterless version of `checkSemantics` to the class `BlockST`. Compile the `PDef` system with the additions from this and the last Activity and test with your (syntactically correct) test files. To see what is happening, run at least one test with the symbol table and semantic debug flags turned on – this will allow you to see the sequence of method calls and returns.

## 15.4.2 Displaying the Symbol Table

The ability to display the symbol table is necessary for the same reason it is necessary to display the syntax tree: for debugging purposes. There are two aspects to printing the symbol table. First, we want to be able to display the bindings contained in a local symbol table on their own. This is a straightforward operation and is represented in the `SymbolTable` definition in the form of the method `toString`. With `SymbolTable.toString` in place we can easily add the display of `localST` to the syntax tree display. To do this it is sufficient to modify the `BlockST.toString` method to call `SymbolTable.toString` and tack on the appropriate padding.

We might also like to display the entire symbol table structure without all the extra information included in the display of the entire syntax tree. Such a display would more clearly show the

shadowing of declaration scopes. Since the local symbol tables reside in the syntax tree, it is in the syntax tree classes where this functionality must be added.

### ☛ Activity 52 –

For displaying a symbol table we will start with the minimalist approach, i.e., we will simply generate a list of the entries in the symbol table on one line. Since each entry is a reference to a `DeclarationST` object, it makes sense to just use the `toString` method of that class to display the entries. But for our purposes, these displays are too lengthy. So we will add to the `DeclarationST` class a method named `briefToString`, which will display the type and identifier names simply as an ordered pair. Assuming this structure, then for the *PDef-lite* string

```
{ int a, float b, b = a, int c, c = a, int d }
```

`SymbolTable.toString` would produce the following `String` return value.

```
(int, a) (float, b) (int, c) (int, d)
```

Implement the method `briefToString` in `DeclarationST` and then the method `toString` in `SymbolTable`.

---

### ☛ Activity 53 –

Because we have added a new data member, `localST`, to the class `BlockST`, we should also update the method `BlockST.toString(String)` so that the value of `localST` is displayed. If we have the *PDef-lite* string

```
{int b, {float b, a = b}, int a, b = a}
```

the syntax tree will be displayed as follows.

List:

```
Symbol Table: (int, b) (int, a)
Declaration: [ type int, name b ]
```

List:

```
Symbol Table: (float, b)
Declaration: [ type float, name b ]
Assignment: [ target a, source b ]
Declaration: [ type int, name a ]
Assignment: [ target a, source b ]
```

Modify the method `BlockST.toString(String)` to return a `String` value of this form. Compile and test this implementation. Notice this allows you to finally test the correctness of the symbol table construction in the `checkSemantics` traversal.

---

### Activity 54 –

But we would also like to be able to display only the symbol table – without all the other data in the syntax tree. If we have the PDef-*lite* string

```
{int a,{float b,int d,{float a},{int c,int x},float y},{int c,int k},int b}
```

the symbol table will be displayed as follows.

Symbol Table Hierarchy:

```
Symbol Table: (int, a) (int, b)
  Symbol Table: (float, b) (int, d) (float y)
    Symbol Table: (float, a)
      Symbol Table: (int, c) (int, x)
        Symbol Table: (int, c) (int, k)
```

Design a DFS algorithm (that means follow the design steps!) for the syntax tree classes that will display the symbol table as described above. Then implement the methods you have designed. Call these methods `symbolTable2String`. Don't forget about the padding! Finally, implement the method `printSymbolTable` in `BlockST` so that it calls `symbolTable2String("")`, rather than `toString`.

[**Hint:** Start with the design steps for the complete syntax tree display from the previous tutorial (see Section 14.5). Go through the steps and describe only what is necessary to display the symbol table component of each `BlockST` object. Use the resulting design as the basis for your implementation.]

Now test your symbol table display additions.

### 15.4.3 Verifying Semantic Rules

Our goal in this section is to complete the implementation of the `checkSemantics` traversal methods by adding the semantic checking capability. But since semantic checking focuses on the uses of identifiers, and since identifier uses in PDef-*lite* occur only in assignment statements, it is only the `checkSemantics` in `AssignmentST` which must be modified. There are two PDef-*lite* semantic rules that need verifying.

1. The use of an identifier must occur in the scope of a declaration for the same identifier.
2. In an assignment entry the type of the source identifier must be *assignment compatible* with the type of the target identifier. Assignment compatibility is defined by the following chart.

target type	compatible source types
int	int
float	int, float

So `AssignmentST.checkSemantics`, which will be called from `BlockST.checkSemantics`, must check that the data members `source` and `target` are both properly declared and that the type of `source` is assignment compatible with the type of `target`. We address these two capabilities in the following sections.

### Is That Identifier Declared?

When we encounter the use of an identifier `x` we must make sure that it occurs in the scope of a declaration for `x`. As we saw illustrated in Section 15.1.3, we can find the closest declaration for an identifier by traversing `localST` for the block in which the identifier use is found, following parent links if necessary, until a declaration for the identifier is found. If a `null` parent is reached then the identifier does not fall in the scope of an appropriate declaration and some sort of exception should be thrown.

So `AssignmentST.checkSemantics` must carry out this processing for each of the data members `target` and `source`. For each of these identifiers we can find the governing declaration (if there is one) by calling `findBinding` on the symbol table parameter `st` – it will return `null` if a declaration isn't found. The following sequence provides this functionality.

```
DeclarationST entry = st.findBinding(name);
if (entry == null)
    throw new NotDeclaredException("Identifier not declared", name);
```

#### ☛ Activity 55 –

Making use of this code segment, modify the current `AssignmentST.checkSemantics` so that it obtains a `DeclarationST` object for each of `target` and `source` and throws appropriate exceptions in the case `null` is returned.

---

#### ☛ Activity 56 –

You will have noted that in the code segment above there is a call to a new symbol table method `findBinding`. If you look in the provided file `SymbolTable.java` you will see that there are two stubbed (i.e., incomplete) methods, `findBinding` and `findBindingLocal`. Complete these stubbed methods according to the interface described by the pre and postconditions. For convenience, here are the headers for these two methods.

```
public DeclarationST findBindingLocal(Token name)
// Pre:  name.getType() == IDENT_T
// Post: ( obj is an entry in 'list' AND
//        obj.getName() == name AND return obj ) OR
//        (name not found in 'list' AND return null)
Algorithmic hint:
```

use a standard iterator-based sequential search through the list 'symbol' and return the DeclarationST object if found or null if not found

```
public DeclarationST findBinding(Token name)
// Pre: name.getType() == IDENT_T
// Post: obj == findBindingLocal(name) AND
//       ( if (obj != null) return obj
//         else if (parentST != null) return parentST.findBinding(name)
//         else return null )
```

Implement these methods – follow the algorithmic hints.

---

Now that we have designed our semantic checking traversal the next question is, how will we use it in our recognizer? First we remember that `parseProgram` returns a `BlockST` object and that the method `checkSemantics` in `BlockST` takes no parameter, we can call it directly from the main method's `try...catch` statement, as follows.

```
try {
    BlockST syntaxTree = parse.parseProgram();
    System.out.println();
    System.out.println("Program parsed!");
    syntaxTree.checkSemantics();
    if (printSTree) syntaxTree.printST();
    if (printSTable) syntaxTree.printSymbolTable();
    System.out.println();
    System.out.println("Program passed semantic checking!");
}
```

### Activity 57 –

Add this to the method `main`. Compile and test the *PDef-lite* recognizer and make sure that it throws exceptions only for those identifier uses that do not fall in the scope of an appropriate declaration.

Notice in the `try` block that there is a new `boolean` flag – `printSTable`. Add that flag to `PDef.java` – use the command line option 'y' to trigger this new flag. Follow the example of the `printSTree` flag.

---

### Assignment Compatibility

Now we must deal with assignment compatibility in assignment statements. The implementation builds directly from the implementation of `AssignmentST.checkSemantics` in the last section. Having retrieved declaration objects for each of `target` and `source`, type tokens can be retrieved

from each declaration object<sup>1</sup> and these types can be checked for assignment compatibility. To help with assignment compatibility checking we will implement a helper method `assignCompatibility` in `AssignmentST` having the following signature. Notice you have to pass the `TokenType` value to this method.

```
private boolean assignCompatibility(Token.TokenType target, Token.TokenType source)
// Pre: (target == Token.TokenType.INT_T OR Token.TokenType.FLOAT_T) AND
//      (target == Token.TokenType.INT_T OR Token.TokenType.FLOAT_T)
// Post: ???
```

### ☛ Activity 58 –

Determine a boolean expression that evaluates to `true` exactly when the target and source types satisfy the assignment compatibility definition – i.e., an expression appropriate for the postcondition of `assignCompatibility`. [See page 273.]

Now implement the method `assignCompatibility` to meet the postcondition.

---

### ☛ Activity 59 –

Using the new `assignCompatibility` method, extend the code in `checkSemantics` for `AssignmentST` to determine if assignment compatibility is satisfied. In doing this remember that if `assignCompatibility` returns `false`, an exception must be thrown. Also remember that this new exception must be added to the `throw` clause in the signature in both the class definition and the method implementation.

With these changes in place, test the resulting recognizer for *PDef-lite*, paying special attention to semantic errors involving assignment incompatibility. This completes the *PDef-lite* recognizer so it should pass any test you throw at it.

---

## 15.5 Guided Development – PDef Semantics

Now we turn our attention to adding semantic checking for arithmetic expressions to our *PDef-lite* recognizer. When this is completed we will have a completed *PDef* recognizer. Our strategy will be to extend the functionality of `checkSemantics` which we established for *PDef-lite*.

There were two semantic problems in the *PDef-lite* recognizer: make sure all identifiers are declared and make sure that all assignment statements satisfy the assignment compatibility rule. For *PDef* we still have the first two rules, but there is a third rule as well – namely that arithmetic operations are carried out on values of the right type. All three rules will be checked in `checkSemantics`. The following activity will guide the implementation of these three rules.

As a reminder and a check against your work in the Guided Development in Section 14.6, Figure 15.6 shows a UML class diagram for a class hierarchy rooted by the class `Expression`.

<sup>1</sup>In Activity 44 it was suggested that accessor methods be included in `DeclarationST` for each of the data members (see page 14.2.2).



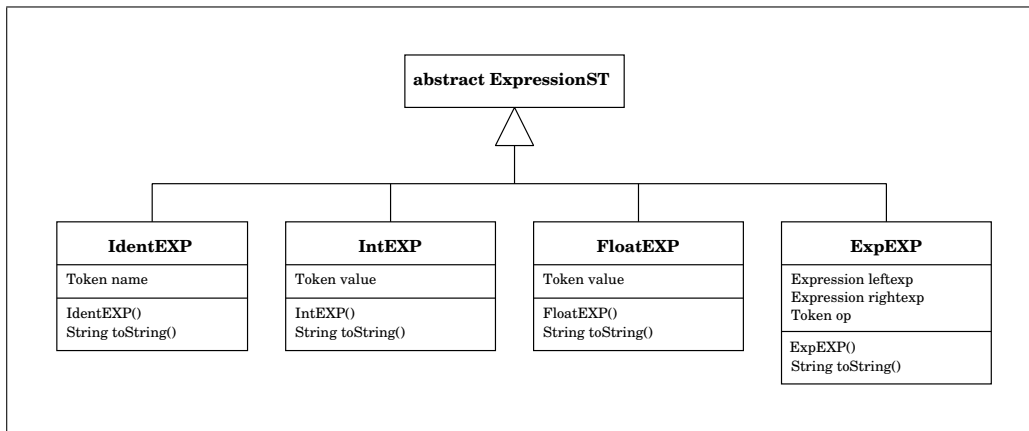


Figure 15.6: UML Class Diagram for Expression Class Hierarchy

### Activity 60 –

1. Edit the class `AssignmentST` and the parse method `parseAssignment` to return the supplied implementation to the functionality at the end of the Guided Development in Section 14.6 – i.e., `parseAssignment` should be as follows.

```

private AssignmentST parseAssignment() throws ParseException
// Grammar Rule: Assignment --> identT assignT Exp
{
    debug.show(">>> Entering parseAssignment");

    Token target = currentToken;
    consume(Token.TokenType.IDENT_T);
    consume(Token.TokenType.ASSIGN_T);
    ExpressionST source = parseExp();

    debug.show("<<< Leaving parseAssignment");
    // Assert: target.getType() == IDENT_T
    return new AssignmentST(target, source);
}
  
```

Compile and test the resulting code so that it will properly process PDef files and generate syntax trees appropriately.

2. Design a traversal of the `ExpressionST` hierarchy that will implement semantic checking for expressions – i.e., go through the three step process. There are two requirements of the method `checkSemantics` for this traversal. First, it must take a symbol table as parameter. Second, it must return a `Token.TokenType` value representing the type of the particular expression. Remember that the the methods must both check that identifiers are declared (that should be done in `IdentEXP`) and check that the arguments to arithmetic operators behave as expected. One warning. When `checkSemantics` is implemented in the leaf nodes (`IdentEXP`, `IntEXP`, and `FloatEXP`) the methods must return a type – but none are readily available in those classes. So the return value from each of the three methods will have to be explicit

– e.g., `Token.TokenType.INT_T`. But for `IdentEXP` the type will be retrieved from the symbol table entry it receives when checking the identifier is in scope.

Put the abstract signature in the class `ExpressionST` and look again at Step 1 on page 269 (in Section 15.4.1).

Implement the designed versions of `checkSemantics` in each class of the hierarchy except the class `ExpEXP`.

- To set a basis from which to implement `ExpEXP.checkSemantics`, design and implement a method with the following signature.

```
private Token combineTypes(Token.TokenType left,
                          Token.TokenType op,
                          Token.TokenType right)
    throws OperationException
// return the string representing the appropriate result type
// unless there is an error, in which case throw an exception
```

The method will either throw an exception or return the appropriate type string according to the following table.

operation	argument 1	argument 2	result type
+ - * /	int	int	int
"	float	int	float
"	int	float	float
"	float	float	float
% (mod)	int	int	int
"	float	int	error
"	int	float	error
"	float	float	error

Notice that the implementation of this method provides the checking of the third semantic rule. In addition, it provides an appropriate type for any legal combination of argument values.

- Now implement `checkSemantics` for the class `ExpEXP`. This method should call `checkSemantics` on `leftExp` and `rightExp` and recover the types for these calls. The method should return the value returned by a call to `combineTypes`; the call to `combineTypes` takes as arguments the return values of the two subexpressions and the expressions operator. The method `checkSemantics` in `ExpEXP` must have the following signature – as specified in `ExpressionST`.

```
public Token checkSemantics(SymbolTable st)
    throws NotDeclaredException, OperationException
// Pre: st is a reference to the current level's symbol table
// Post: return the type of the expression
```

- At this point you can adjust `AssignmentST` so that, in its version of `checkSemantics`, it recovers a `Token` type value from its data member `target` and uses this type to check assignment compatibility with the type returned by the call to `parseExp`. Be sure to adjust appropriate method signatures in the `StmtST` classes to reflect the fact that `AssignmentST.checkSemantics` might throw the `OperationException` exception.

This should complete the implementation of the PDef recognizer – test it on the file `intest-PDef` to convince yourself it is complete and accurate.

---



## Chapter 16

# PDef LR Parser Tutorial

In this section we will study the design and implementation of a replacement parser package for the PDef-*lite* recognizer. The grammar of PDef-*lite* was constructed originally for parsing by a recursive descent parser. But the LR parser will be easier if the grammar is shorter and, as we have seen above, if there is no right recursion. Remember that the right recursion in the original PDef fragment 3 resulted in a shift/reduce conflict in parse table while the conflict disappeared when the right recursive rule was replaced by an equivalent left-recursive rule. For these reasons we will use the following equivalent grammar for PDef-*lite*.

```
0 P  → SL
1 SL → lbT L rbT
2 L  → L commaT S
3 L  → S
4 S  → identT assignT identT
5 S  → typeT identT
6 S  → SL
```

Our presentation will be similar to that found in Chapter 13: we will demonstrate the LR parsing technique by looking at a completed parser package for PDef-*lite* and then guide you through an extension to the full PDef.

### 16.1 The Design of an LR Parser

We want to design a set of classes that will implement an LR parser for PDef-*lite* and will be able to replace the parser package in our PDef-*lite* recognizer. The structure of the parser package for the recursive descent parser is very simple. The single class `Parser` contains one public method, `parseProgram`, and a set of private methods, one for each grammar rule. An LR parser, however, has a more complex structure. The discussion in Section 9.1 makes it clear that the parser depends on a parse table, in which each parse table entry is a specific action (shift, reduce, goto, action, error), and an integer stack, where each entry identifies a particular state in the LR(0) transition graph. The algorithm used to parse an input stream has a simple loop in which an entry is retrieved from the parse table and the specified action is carried out. Here is a sketch of that algorithm as we would expect to see it in the new parser package.

```

Token currentToken = tokenStream.getNextToken();
while (!done) {
    state = stack.top();
    symbol = convertToken2Symbol(currentToken);
    ParseTableEntry pte = parseTable.getEntry(state, symbol);
    pte.doAction(parseTable, stack, currentToken);
}

```

The diagram in Figure 16.1 illustrates a single step in the algorithm above, where the current token (`commaT`) and the current state (`6`) are used to access and then carry out the next action, which is `shift 4`. The numbered arrows in the diagram specify the following sequence of steps.

1. retrieve the parse table entry – `s4`
2. carry out the action – push `4` onto the stack
3. since this is a shift operation we must set a new value for `Current Token` – `identT`.

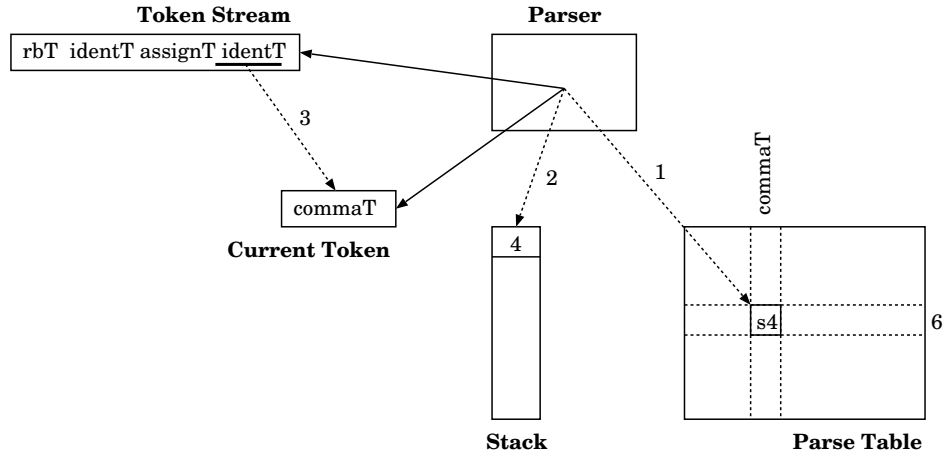


Figure 16.1: LR Parser Algorithm Illustrated

This description carries certain implications for other structures in the parser. In particular, we need to examine the structure of the parse table, in the form of the class `ParseTable`, and consequently the structure of the entries in the parse table, in the form of the class `ParseTableEntry`.

There are certain structures which have remained undefined or only partially defined. In particular, we need to discuss how to implement the five types of parse table entries and how we will represent the symbol names, which index the columns of the parse table, and the grammar rules, which are needed by reduce actions.

### `ParseTableEntry` class hierarchy

The implementation of the five types of parse table entries is easily handled. We can define the class `ParseTableEntry`, as illustrated in the UML class diagram, to be an abstract class and then define a subclass for each of the entry types. Of the five, there are three entry types needing extra

data, namely, a reduce entry needs a reference to its rule parameter while the shift and goto entries need to reference their next-state parameter. The error and accept entries have the same structure as described by `ParseTableEntry`.

### Implementing symbols and grammar rules

There are two other components to the parse package which we have not yet discussed: the symbols which index the columns of the parse table and then the grammar rules, which are used by the reduce actions. The implementation of the symbol values seems ready made for an enumerated type, and in fact that will be our choice. But we will make use of features of the `enum` class in order to represent more than just the names of the symbols. These same features will make it possible to use an `enum` class to implement the grammar rules as well.

The object oriented structure of the parser package described above is depicted in the UML class diagram in Figure 16.2. Notice that the two `enum` classes have their own state and methods. These features will be discussed in more detail in the next section.

## 16.2 The Supplied Java Code

Our goal is to produce a parser package to replace the one in the *PDef-lite* recognizer. That package was very simple and contained a single class, the class `Parser`. In our new implementation the package will be considerably more complex, but in order for the package to transparently replace the recursive descent parser package we must implement the same interface for the new class `Parser`. In this section you will investigate the supplied Java implementation for an LR parser for *PDef-lite*, the grammar rules for which we repeat here for convenience.

```

0 P  → SL
1 SL → lbT L rbT
2 L  → L commaT S
3 L  → S
4 S  → identT assignT identT
5 S  → typeT identT
6 S  → SL

```

The Java code we will review in this section matches the structure given in the UML class diagram of Figure 16.2, so it will be useful to reference that diagram from time to time.

### 16.2.1 class `Parser`

The parser class, as indicated above, must have the same public interface as the recursive descent parser class. At the same time, the state of the class is considerably different; we have already discussed that structure. The structural details can be seen in this abbreviated class description.

```

public class Parser {

    private ParserDebug debug    = new ParserDebug();

```

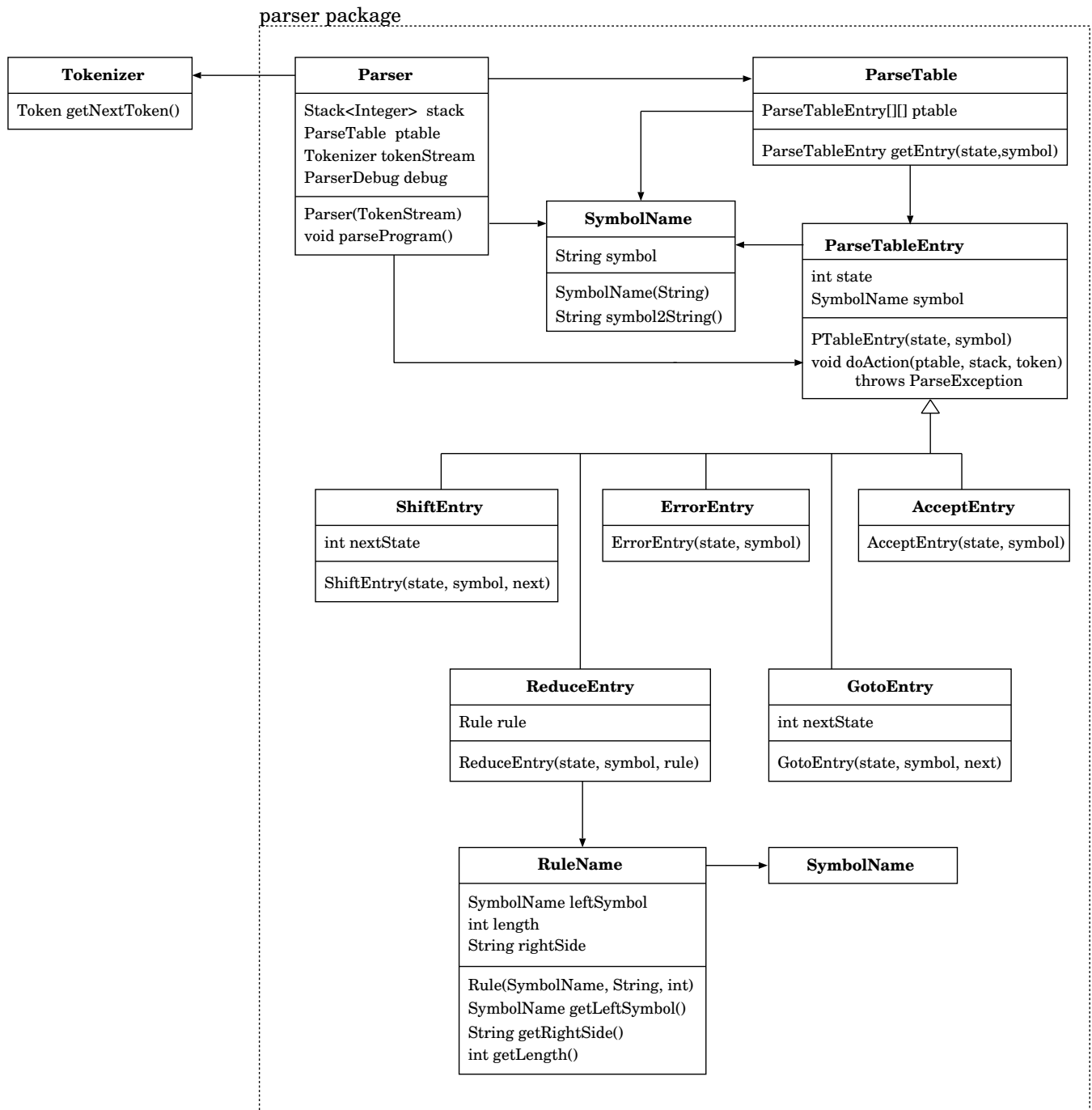


Figure 16.2: UML Class Diagram for an LR Parser - II

```

private ParseTable ptable    = new ParseTable();
private Stack<Integer> stack = new Stack<Integer>();
private Tokenizer tokenStream;
private Token    currentToken;

```



```

public Parser(Tokenizer tokenStream)
    // Pre: tokenStream has a value
    // Post: debug == new ParserDebug() AND
    //       this.tokenStream == tokenStream AND
    //       stack.empty() == true AND ptable != null
{ this.tokenStream = tokenStream;
  currentToken = tokenStream.getNextToken();
}
public void parseProgram() throws ParseException {...}
}

```

The algorithm for `parseProgram` is more complex and matches the algorithm sketched in Section 16.1. There are details, however, which need to be pointed out: here is the code for the method `parseProgram`.

```

public void parseProgram() throws ParseException {
  debug.show("===>>> Entering Parser.parseProgram");
  stack.push(1);
  SymbolName sym = SymbolName.token2Symbol(currentToken);
  ParseTableEntry pte = pTable.getEntry(1, sym);
  while (pte.getClass().getName().equals("parser.AcceptEntry")) {
    pte.doAction(pTable, stack, currentToken);
    if (pte.getClass().getName().equals("parser.ShiftEntry")) {
      currentToken = tokenStream.getNextToken();
    }
    int st = stack.peek();
    sym = SymbolName.token2Symbol(currentToken)
    pte = pTable.getEntry(st, sym);
  }
  debug.show("<<<=== Leaving Parser.parseProgram");
}

```

The parsing process begins when the state value 1 is pushed onto the stack; this indicates that the parsing begins in the first state of the transition graph for the grammar. The loop structure which follows is that of an indefinite loop, driven by the value of `pte`, which references the current parse table entry. The loop executes until either an exception is thrown by the method `doAction` or the entry referenced by `pte` is an accept entry. The one awkward point in the code is the selection statement to determine whether to read a new token value, which is to be done only when a shift action is carried out. Finally, remember that it is late binding, facilitated by the class hierarchy defined for `ParseTableEntry`, which guarantees that the correct action will be carried out in the first line of the `while` loop.

### 16.2.2 class ParseTable

The parse table is the repository for the data defined by the grammar parse table. The parsing process described in Section 9.1 makes it clear that once defined, the parse table is not modified. Consequently, there are simply three operations we need for the parse table: initialization, access,

display. The initialization of the parse table can be handled by the class constructor. The parsing process again makes it clear that when access to the parse table is needed, that access is to one entry based on the entry's row and column, i.e., a particular state (row) and a particular symbol (column). This means `ParseTable` requires a public method, call it `getEntry`, which, when called with row and column designators, will return the corresponding entry in the table. Finally, the display of the table may be useful for debugging purposes and that responsibility will be, as usual, taken by the `toString` method.

The obvious structure for holding the parse table entries is a 2-dimensional array. Since parse tables can tend to be rather sparse, an array structure may not be appropriate for larger grammars, but for our purposes the 2-dimensional array structure is quite convenient.

This analysis leads to the following structure for the class `ParseTable`.

```
public class ParseTable {

    private final int MAXSTATE = 9; // value matches the number of states in
                                   // the transition graph
    private ParseTableEntry[] [] ptable;

    public ParseTable() {...}
        // Post: ptable[state][symbol] == corresponding entry from parse table
    public ParseTableEntry getEntry(int state, SymbolName symbol)
        // 0 <= state <= MAXSTATE AND symbol has value
        { return ptable[state][symbol]; }
}
```

### 16.2.3 enum SymbolName

A new detail of structure emerges from the description of `ParseTable`, namely the references to “symbol” in its Java description. The symbol will have to stand for token values and for non-terminal names. The obvious way to implement the symbol is in terms of an enumerated class. In this way we will be able to specify the symbol dimension symbolically rather than via an arbitrarily assigned integer constant. In defining the `enum` class `SymbolName` we will use a special capability of `enum` which allows us to associate with each enumerated value the `String` the value represents. We define the class as follows.

```
public enum SymbolName {

    ID_S("id"), TY_S("ty"), EQ_S("="), LB_S("{"), RB_S("}"),
    CM_S(", "), END_S("eof"), SL_S("SL"), S_S("S"), L_S("L");

    private String symbol;

    SymbolName(String sym) { symbol = sym; }

    public String symbol2String() { return symbol; }
```

```

    public static SymbolName token2Symbol(Token t) {
        switch (t.getType()) {
            case identT:    return SymbolName.ID_S;
            case typeT:    return SymbolName.TY_S;
            case commaT:   return SymbolName.CM_S;
            case lbT:      return SymbolName.LB_S;
            case rbT:      return SymbolName.RB_S;
            case assignT:  return SymbolName.EQ_S;
            case eofT:    return SymbolName.END_S;
            default:       return SymbolName.END_S;
        }
    }
}

```

This code perhaps requires a brief explanation. First, notice that this enumerated looks like a class, rather than a simple list of names as we saw it used in the PDef tutorial. The list of value names has been enhanced, with each name, for example `ID_S`, having a parameter specifying a `String` equivalent. That parameter is actually passed to the constructor when that particular symbol is constructed. So `SymbolName` not only lists each possible value name, but also describes the state of each of these values. So `SymbolName.ID_S` as a value has an associated `String` value which can be accessed via the method `symbol2String`. Also, for convenience, there is a `static` method `token2Symbol` which will take a `Token` value and return the corresponding `SymbolName` value.

#### 16.2.4 class ParseTableEntry

This entries in the parse table can be of five different types: shift, reduce, goto, accept, and error. Hopefully it is clear by this time that the way to manage this structure, from an object oriented point of view, is to define an abstract super class and then five subclasses, one for each type of entry. From the parser's point of view, when it gets an entry from the parse table it should carry out the action associated with the entry. The obvious place to put the functionality is within each entry since it is that entry that understands its own responsibilities. So the following abstract class structure suggests itself.

```

public abstract class ParseTableEntry {
    protected ParserDebug debug;
    protected int state;
    protected SymbolName symbol;
    public ParseTableEntry(int st, SymbolName sn) {...}
    // Pre:  0 <= st <= MAXSTATES AND sn has a value
    public abstract void doAction(ParseTable pt, Stack<Integer> st, Token t)
        throws ParseException;
}

```

We will see that it is useful for a parse table entry to know where it is in the table, so we include the state (row) and symbol data members in the abstract class. Also, though it varies from one type of action to another, passing references to the parse table, stack, and current token satisfy the general needs of the five actions. The details of the five subclasses follow.

class ShiftEntry

As we see in the parse tables we have constructed, the shift operation has one parameter, the next state to be entered. The shift action simply involves pushing the parameter value (next state) onto the stack. The subclass is defined as follows.

```
public class ShiftEntry extends ParseTableEntry{
    private int nextState;
    public ShiftEntry(int st, SymbolName sn, int ns)
    // Pre:  0 <= st,ns <= MAXSTATES AND sn has a value
    { super(st, sn); nextState = ns; }
    public void doAction(ParseTable pt, Stack<Integer> st, Token t)
        throws ParseException
    {
        st.push(nextState);
    }
}
```

class ReduceEntry and enum Rule

The reduce action is the most complex of the five. It requires information about the rule being reduced and must also be able to determine the next state to be pushed onto the stack. Each reduce action must have a parameter indicating the rule to be reduced. The rule, as parameter, must be able to give up two pieces of data – the length of the right-hand side of the rule being reduced and the symbol on the left-hand side of the rule. All of this information, of course, is available in the grammar and is fixed for the parser. In order to collect the data for the grammar rules in a convenient structure, we will once again make use of the Java enumeration structure. As in the case of the `SymbolName` enumeration, we create an enumeration with a name for each rule and for each name two parameters: the rule's left-hand side and the rule's right-hand side length. Here is the definition of `Rule`.

```
public enum Rule {
    STMT_LIST_R(SymbolName.SL_S, "{ L }", 3),    // SL -> { L }
    LIST_REC_R(SymbolName.L_S, "L , S", 3),      // L  -> L , S
    LIST_END_R(SymbolName.L_S, "S", 1),          // L  -> S
    STMT_ASSIGN_R(SymbolName.S_S, "id = id", 3), // S  -> type ident
    STMT_DEC_R(SymbolName.S_S, "type id", 2);    // S  -> ident = ident
    STMT_SL_R(SymbolName.S_S, "SL", 2);         // S  -> SL

    private SymbolName leftSym;
    private String     rightSide;
    private int        length;

    Rule(SymbolName sn, String rhs, int l) {
        leftSym = sn;
        rightSide = rhs;
        length = l;
    }
}
```

```

    }

    public SymbolName getLeftSym() { return leftSym; }
    public String     getRightSide() { return rightSide; }
    public int        getLength() { return length; }
    public String     toString() {
        return leftSym + " --> " + rightSide;
    }
}

```

Notice that for convenience there is a third parameter with each name, a parameter that is used in `toString` to produce a printed version of the rule.

The definition of `Rule` is critical to the definition of `ReduceEntry` which is distinguished by the implementation of the `doAction` method, which pops the stack an appropriate number of times and then obtains and pushes the next state value onto the stack.

```

public class ReduceEntry extends ParseTableEntry{
    private Rule rule;
    public ReduceEntry(int st, SymbolName sn, Rule r)
    // Pre:  0 <= st <= MAXSTATES AND sn has a value AND r has a value
    { super(st, sn); rule = r; }
    public void doAction(ParseTable pt, Stack<Integer> st, Token t)
        throws ParseException
    {
        // pop stack by the length of rule
        int next = st.peek();
        GotoEntry goE = (GotoEntry)pt.getEntry(next, rule.getLeftSym());
        st.push(goE.getState());
    }
}

```

#### class GotoEntry

The goto entry has a similar structure to the shift entry, though the goto entry is used by a reduce action to determine the next state.

```

public class GotoEntry extends ParseTableEntry {
    private int nextState;
    public GotoEntry (int r, SymbolName sn, int st) {
        super(r, sn);
        debug.show("===>>> Entering GotoEntry");
        nextState = st;
        debug.show("<<<=== Leaving  GotoEntry");
    }
    public int getState() {
        debug.show("===>>> Entering GotoEntry.getState");
    }
}

```

```

        debug.show("<<<=== Leaving GotoEntry.getState");
        return nextState;
    }
    public void doAction(ParseTable pt, Stack< StackEntry > st, Token t)
        throws ParseException {
        debug.show("===>>> Entering GotoEntry.doAction");
        debug.show("        this method does nothing!");
        debug.show("<<<=== Leaving GotoEntry.doAction");
    }
}

```

### classes AcceptEntry and ErrorEntry

These two classes are very simple and add no data members to those inherited from `ParseTableEntry`. We can see from the implementation of the method `Parser.parseProgram` that a parse table entry of the class `AcceptEntry` will cause the shift/reduce loop to terminate; the entries act purely as flags, with no other functionality. The implementation for this class follows.

```

public class AcceptEntry extends ParseTableEntry {
    public AcceptEntry(int r, SymbolName sn) {
        super(r, sn);
        debug.show("===>>> Entering AcceptEntry");
        debug.show("<<<=== Leaving AcceptEntry");
    }
    public void doAction(ParseTable pt, Stack< StackEntry > st, Token t)
        throws ParseException {
        debug.show("===>>> Entering AcceptEntry.doAction");
        debug.show("<<<=== Leaving AcceptEntry.doAction");
    }
}

```

The class `ErrorEntry` is responsible for throwing an appropriate exception when its version of `doAction` is called. The implementation of `ErrorEntry.doAction` determines which symbols were acceptable in the state where the error occurred and then constructs an error message indicating what went wrong and what was expected. The `for`-loop in the code which follows constructs "what was expected" part of the message. Notice that the `for`-loop has a special form which allows sequencing through all the values in the enumerated type `SymbolName`.

```

public class ErrorEntry extends ParseTableEntry {
    public ErrorEntry(int r, SymbolName sn) {
        super(r, sn);
        debug.show("===>>> Entering ErrorEntry");
        debug.show("<<<=== Leaving ErrorEntry");
    }
    public void doAction(ParseTable pt, Stack< StackEntry > st, Token t)
        throws ParseException {
        debug.show("===>>> Entering ErrorEntry.doAction");
    }
}

```

```

String str = "Parse Error: ";
int tos = st.peek().getState();
str += "saw token '" + t + "' but expected one of ";
for (SymbolName sName: SymbolName.values()) {
    String name = pt.getEntry(tos, sName).getClass().getName();
    if (name.equals("parser.ShiftEntry") || name.equals("parser.ReduceEntry"))
        str += " " + sName;
}
debug.show("<<<=== Leaving ErrorEntry.doAction");
throw new ParseException( str + " [parse table row: " + tos + "]\n");
}
}

```

### 16.2.5 Initialization in ParseTable

The parser package as described to this point would serve as is regardless of the grammar being parsed. This isn't quite true since the classes `SymbolName` and `RuleName` are tied to the particular grammar. But the rest of the code, as seen in the classes `Parser`, `ParseTableEntry` and its subclasses, and `ParseTable`. The one remaining component not yet described is that for initializing the parse table in `ParseTable`. This final component is supplied in this section.

#### The PDef-lite SLR parse table

The first step in producing a parse table is to produce the transition graph for the target grammar. Remember that the *PDef-lite* grammar is the combination of the three fragments discussed in Section 9.1. The transition graph for the *PDef-lite* grammar is a mix of the transition graphs for the three fragments and is displayed in Figure 16.3.

In this transition graph we see that each state containing a terminal dotted form is a singleton state. This means there are no shift/reduce conflicts and, consequently, the grammar is LR(0). Even though we know that *PDef-lite* is LR(0), we will still produce and implement an SLR parse table, since the SLR table will have more precision. Namely, there will be error entries in place of some of the reduce entries from the LR(0) table. In fact, the LR(0) table would find the same errors, but just not quite as soon as the SLR table. The SLR parse table derives from the grammar in the way demonstrated in Section 9.3. The table resulting from the process applied to this grammar follows.

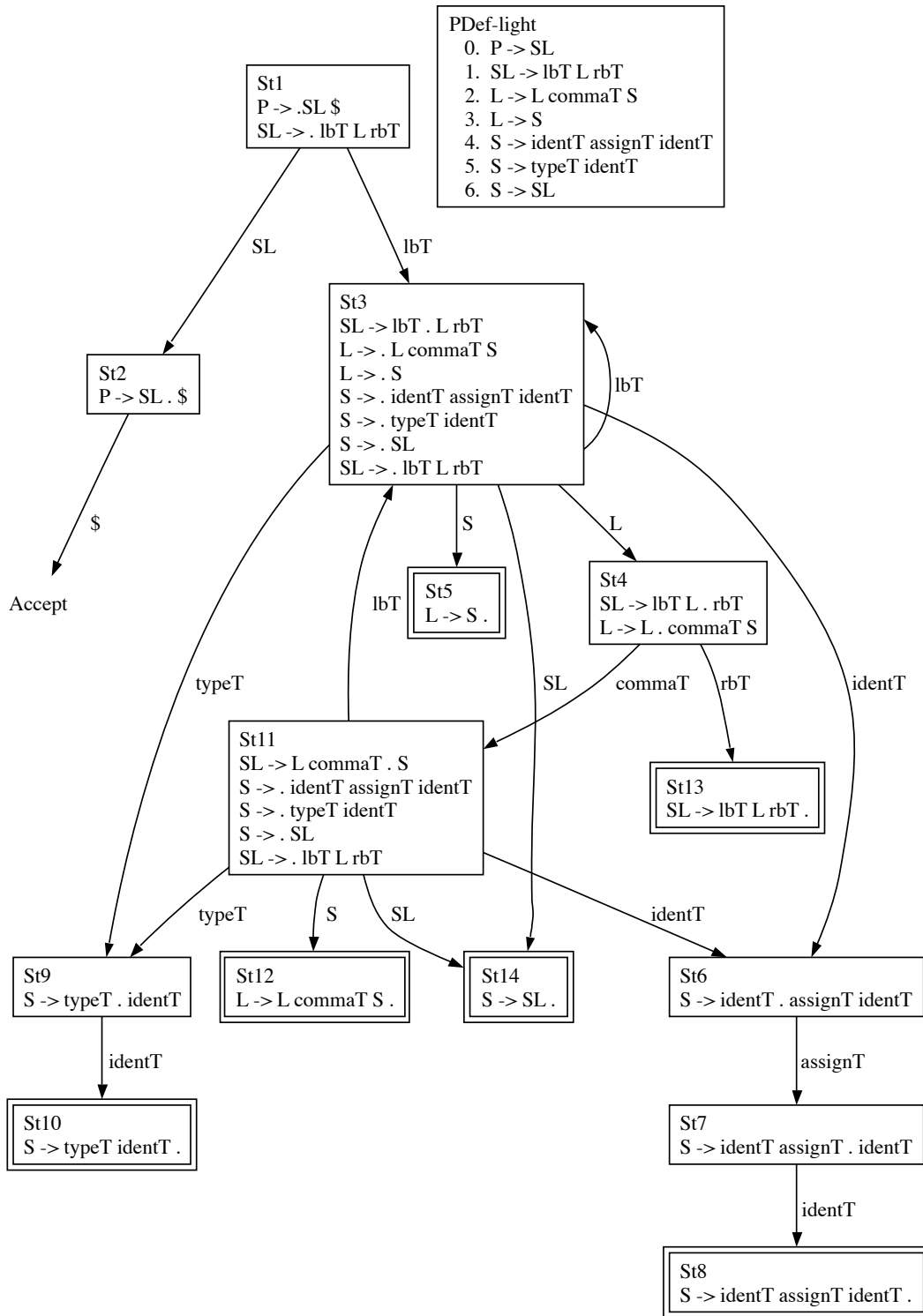


Figure 16.3: Transition Graph for PDef-lite

Input	Input						Goto			
	id	ty	=	{	}	,	\$	SL	S	L
1				s3				g(2)		
2							Acc			
3	s6	s9		s3				g(14)	g(5)	g(4)
4					s13	s11				
5				r(3)	r(3)					
6			s7							



**Initializing the parse table entries**

The initialization of the parse table is actually straight forward. It simply requires assigning the appropriate parse table entry object to each position in the parse table. The appropriate parse table entry objects can be determined by looking at the specific parse table – remember that all empty entries in the table are to be made error entry objects.

The easiest way to deal with the error objects is to start out by storing an error object in each table position. Then we can reassign shift, reduce, accept, or goto entry objects to the appropriate positions. Here is the first part of the code for the constructor method `ParseTable`.

```
public ParseTable() {
    debug = new ParserDebug();
    debug.show("===>>> Entering ParseTable");
    for (int i = 0; i < 14; i++) {
        for (SymbolName n: SymbolName.values()) {
            ptable[i][n.ordinal()] = new ErrorEntry(i,n);
        }
    }

    // Accept action
    ptable[2][SymbolName.END_S.ordinal()] = new AcceptEntry(2, SymbolName.END_S);

    // Shift actions
    ptable[1][SymbolName.LB_S.ordinal()] = new ShiftEntry(1, SymbolName.LB_S, 3);
    ptable[3][SymbolName.ID_S.ordinal()] = new ShiftEntry(3, SymbolName.ID_S, 6);
    ptable[3][SymbolName.TY_S.ordinal()] = new ShiftEntry(3, SymbolName.TY_S, 9);
    ptable[4][SymbolName.RB_S.ordinal()] = new ShiftEntry(4, SymbolName.RB_S, 13);
    ptable[4][SymbolName.CM_S.ordinal()] = new ShiftEntry(4, SymbolName.CM_S, 11);
    // there are more shift entries

    // Reduce actions

    ptable[5][SymbolName.ID_S.ordinal()]
        = new ReduceEntry(5, SymbolName.ID_S, Rule.LIST_END_R);
    ptable[5][SymbolName.TY_S.ordinal()]
        = new ReduceEntry(5, SymbolName.TY_S, Rule.LIST_END_R);
    ptable[5][SymbolName.LB_S.ordinal()]
        = new ReduceEntry(5, SymbolName.EQ_S, Rule.LIST_END_R);
    // There is one of these also for symbols RB_S, CM_S, and END_S
    ptable[8][SymbolName.ID_S.ordinal()]
        = new ReduceEntry(8, SymbolName.ID_S, Rule.STMT_ASSIGN_R);
    // There is one of these also for symbols TY_S, LB_S, RB_S, CM_S, and END_S
    // There are more reduce entries

    // Goto actions
    ptable[1][SymbolName.SL_S.ordinal()] = new GotoEntry(1, SymbolName.SL_S, 2);
```

```

    ptable[3][SymbolName.S_S.ordinal()] = new GotoEntry(3, SymbolName.S_S, 5);
    ptable[3][SymbolName.L_S.ordinal()] = new GotoEntry(3, SymbolName.L_S, 4);
    ptable[11][SymbolName.S_S.ordinal()] = new GotoEntry(11, SymbolName.S_S, 12);

    debug.show("<<<=== Leaving ParseTable");
}

```

There are several things to take away from this method. First, with a small grammar the initialization process seems manageable. Second, even though it is more to write, the trouble taken to define `SymbolName` and `RuleName` make the initialization code easier to read and consequently easier to verify. Third, if we have a large grammar then, even with definitions for `SymbolName` and `RuleName`, the initialization code will be problematic if written by hand – the chances are high that errors exist somewhere, in the original transition graph or parse table, in `SymbolName` or `RuleName`, or in the initialization code itself. Fourth, given a large grammar it would be better to spend some time implementing an automatic mechanism for taking a grammar as input and generating the parse table, and thus the initialization code.

The automatic generation of the parse table and its initialization code exists in several forms, the best known being the packages YACC and LEX.

### 16.3 Error Handling and Recovery

The strategy for error handling described above is conservative, as was our error handling in the original PDef parser. But the LR parsing strategy, in the form of error entries in the parse table, provides a convenient setting for error recovery.

### 16.4 Guided Development – LR Parser for PDef

#### ☛ Activity 61 –

Before extending the work of the previous section, insert the supplied LR parser package into the provided code base for Chapter 13. The best way to do this is to first rename the package directory `parser` in that code base and then copy in the new package directory `parser`. Then compile the recognizer and execute it on some test files. Remember that the implemented parser will generate an error if the input includes a nested list. Execute the recognizer with the ‘p’ option set to see the flow of control through the more complex structure of the parser package.

---

#### ☛ Activity 62 –

Modify the supplied parser code so that the full PDef-*lite* grammar is recognized. Make sure that not only the parse table initialization is updated, but also that the classes `SymbolName` and `RuleName` are modified if necessary. When the modifications have been made test them to verify that what should be accepted is accepted and what should

generate an error does so.

---

We have in Figure 16.1 the SLR parse table for PDef. We also have the PDef LR(0) transition graph from which the parse table is constructed. The graph is presented in two parts: Figure 16.4 displays the graph for the first component of the grammar and Figure 16.5 displays the graph for the second component of the grammar.

## 16.5 Syntax Tree Generation by an LR Parser

Syntax tree generation in the PDef recursive descent parser is very natural. Each parse method collects the information it needs from its components and then returns an appropriate structure for the syntax tree. The fact that the syntax tree nodes derive their structure from the grammar as well, the parse methods and syntax tree nodes have a tight correlation. In LR parsing, however, the grammar rules are reduced in a reversed order, so that when data can be collected it is not clear what syntax tree node will ultimately be created. This makes the connection between an LR parser and the syntax tree less obvious. In this chapter we will investigate the syntax tree generator for our PDef-*lite* LR parser and then, in the Guided Development at the end of the chapter, extend the PDef-*lite* implementation to a PDef syntax tree generator.

Our goal in this section is similar to that in the previous sections of this chapter – to modify the parser package so that it can be directly substituted into the implementation from the PDef tutorial, thus illustrating the way in which the syntax tree generating parser can be a pluggable component for a language processor.

### 16.5.1 The Strategy

Remember in Chapter 9 we studied the LR parsing algorithm by tracing the stack state on various input strings for the three PDef fragments. In those examples we pushed either a terminal or a non-terminal symbol along with the transition graph state number. Here is such a trace for input

```
int a, float c
```

using the parse table for the left-recursive version of PDef fragment 3. [Reference the parse table in Figure 9.4.] Remember that we have used single letter abbreviations for the non-terminal symbols.

State	Input											Goto								
	id	ty	=	{	}	,	int	Float	+	*	(	)	\$	SL	S	L	E	T	F	
1				s3										g(2)						
2													Acc							
3	s6	s9		s3										g(14)	g(4)	g(5)				
4					s13	s11														
5					r(3)	r(3)														
6			s7																	
7	s17						s25	s26			s18						g(8)	g(15)	g(16)	
8					r(4)	r(4)			s23											
9	s10																			
10					r(5)	r(5)														
11	s6	s9		s3										g(14)	g(12)					
12					r(2)	r(2)														
13					r(1)	r(1)								r(1)						
14					r(6)	r(6)														
15					r(8)	r(8)				r(8)	s19			r(8)						
16					r(10)	r(10)				r(10)	r(10)			r(10)						
17					r(13)	r(13)				r(13)	r(13)			r(13)						
18	s17						s25	s26			s18						g(21)	g(15)	g(16)	
19	s17						s25	s26			s18									g(20)
20					r(9)	r(9)				r(9)	r(9)			r(9)						
21									s22					s21						
22					r(14)	r(14)				r(14)	r(14)			r(14)						
23	s17						s25	s26			s18									g(24)
24					r(7)	r(7)				r(7)	s19			r(7)						
25					r(11)	r(11)				r(11)	r(11)			r(11)						
26					r(12)	r(12)				r(12)	r(12)			r(12)						

Table 16.1: SLR Parse Table for PDef

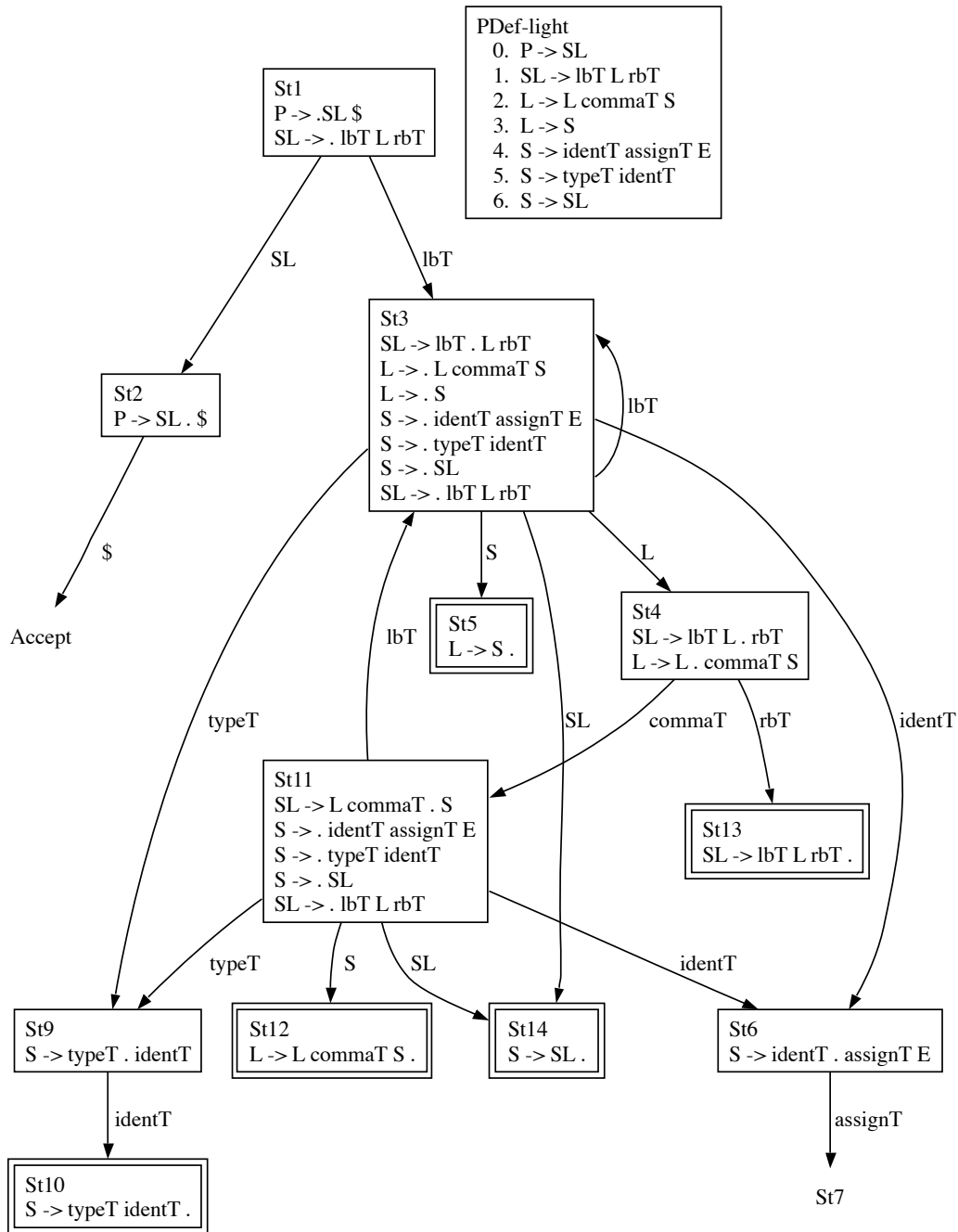


Figure 16.4: Transition Graph for the PDef List Component

Action	Stack (top at the right)	Target (after operation)
	\$ (1)	↑ int a, float c
shift typeT	\$ (1) - typeT (5)	↑ a, float c
shift identT	\$ (1) - typeT (5) - identT (6)	↑ , float c
reduce rule 4	\$ (1) - D (4)	↑ , float c
reduce rule 3	\$ (1) - S (3)	↑ , float c
reduce rule 2	\$ (1) - L (2)	↑ , float c
shift commaT	\$ (1) - L (2) - commaT (7)	↑ float c
shift typeT	\$ (1) - L (2) - commaT (7) - typeT (5)	↑ c
shift identT	\$ (1) - L (2) - commaT (7) - typeT (5) - identT (6)	↑ ε
reduce rule 4	\$ (1) - L (2) - commaT (7) - D (4)	↑ ε
reduce rule 3	\$ (1) - L (2) - commaT (7) - S (8)	↑ ε

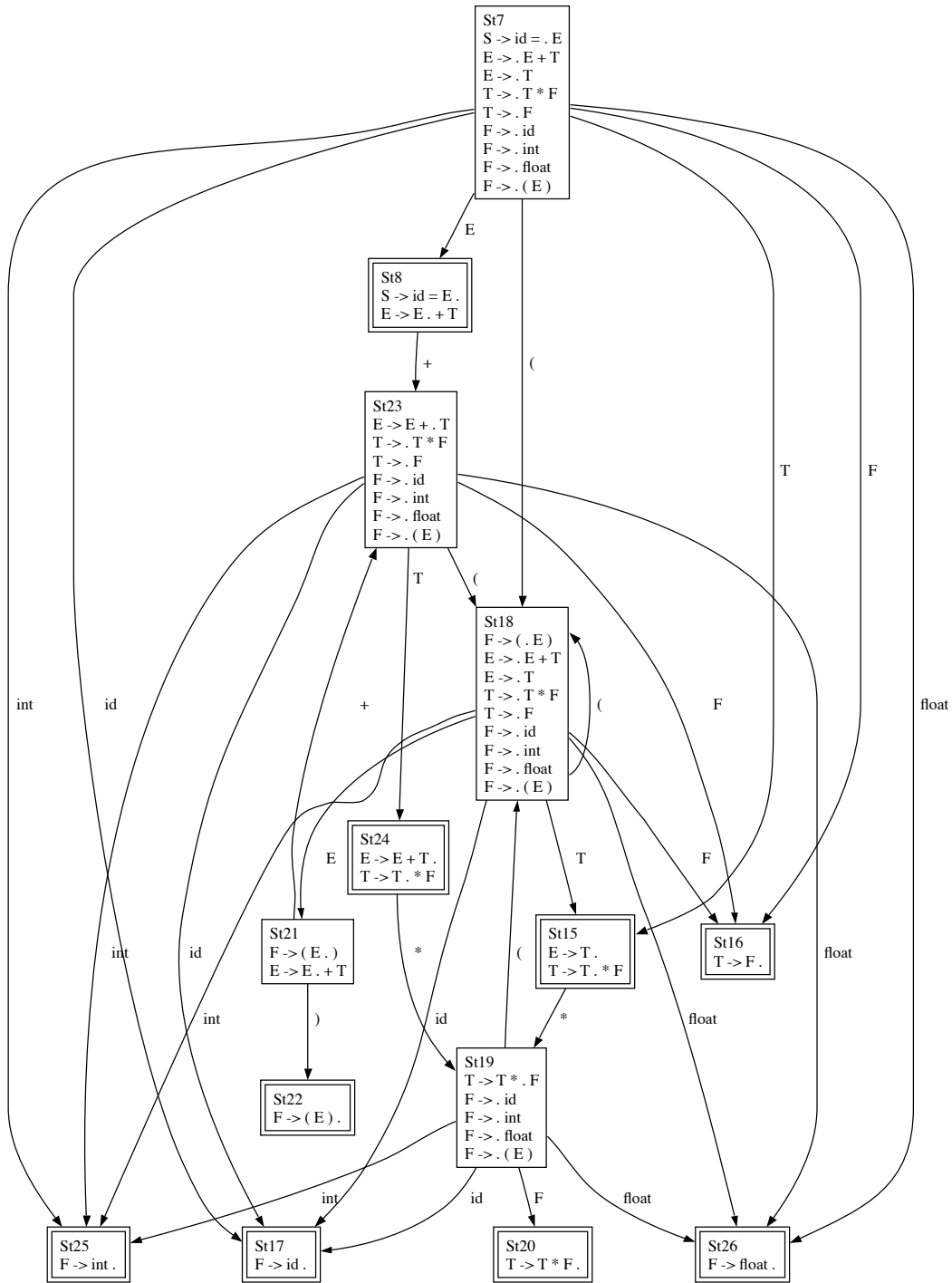


Figure 16.5: Transition Graph for the PDef Expression Component

One of the things we discovered in the discussion of the LR parsing algorithm is that if we are only interested in parsing, then only the state number is needed. So in our implementation of the LR parsing algorithm we used a stack of integers.

But if we look at this trace we see the clue to how to generate the syntax tree during the parsing process. Rather than using simply state numbers as stack entries, we use a stack entry which contains a state number along with a reference to a constructed syntax tree node. More specifically, when we perform a shift operation the stack entry will consist of the new state number along with the syntax tree object corresponding to the current token (or a null reference if the token is punctuation). When we perform a reduce operation, we will collect the relevant syntax tree nodes from the stack entries as they are popped and construct a new object, based on the rule being reduced, and use the new state number along with the constructed object as the new stack entry. Here is an annotated version of this trace, where the annotations indicate how the syntax tree can be built.

Action	Stack (top at the right)	Target (after operation)
		$\$(1)$ ↑ int a, float c
shift typeT	$\$(1) - \text{typeT}(5)$	↑ a, float c
	<i>Push 5 and the <b>TypeST</b> object for <b>int</b>.</i>	
shift identT	$\$(1) - \text{typeT}(5) - \text{identT}(6)$	↑ , float c
	<i>Push 6 and the <b>IdentST</b> object for <b>a</b>.</i>	
reduce rule 4	$\$(1) - D(4)$	↑ , float c
	<i>Recover the two objects and create a <b>Declaration</b> object from them. Push 4 along with the new object.</i>	
reduce rule 3	$\$(1) - S(3)$	↑ , float c
	<i>Recover the <b>Declaration</b> object. Push 3 along with the object.</i>	
reduce rule 2	$\$(1) - L(2)$	↑ , float c
	<i>Recover the <b>Declaration</b> object. Create a new <b>LinkedList</b> object and add the recovered object. Push 2 along with the new list object.</i>	
shift commaT	$\$(1) - L(2) - \text{commaT}(7)$	↑ float c
shift typeT	$\$(1) - L(2) - \text{commaT}(7) - \text{typeT}(5)$	↑ c
	<i>Push 5 and the <b>TypeST</b> object for <b>float</b>.</i>	
shift identT	$\$(1) - L(2) - \text{commaT}(7) - \text{typeT}(5) - \text{identT}(6)$	↑ $\epsilon$
	<i>Push 6 and the <b>IdentST</b> object for <b>c</b>.</i>	
reduce rule 4	$\$(1) - L(2) - \text{commaT}(7) - D(4)$	↑ $\epsilon$
	<i>Recover the two objects and create a <b>Declaration</b> object from them. Push 4 along with the new object.</i>	
reduce rule 3	$\$(1) - L(2) - \text{commaT}(7) - S(8)$	↑ $\epsilon$
	<i>Recover the <b>Declaration</b> object. Push 8 along with the object.</i>	
reduce rule 2	$\$(1) - L(2)$	↑ $\epsilon$
	<i>Recover the <b>Declaration</b> object and the list. Add the declaration object to the list. Push 2 and the list object.</i>	
accept	$\$(1) - L(2)$	↑ $\epsilon$
	<i>Recover the list object and return it as the syntax tree object.</i>	

This example would seem to define a general strategy for generating the syntax tree, but there are some critical details which need attention. We discuss these details in the three sections which follow.

### The stack entry

The stack can no longer be an **Integer** stack – the stack must be able to hold in each entry a state number and a syntax tree structure, so we must define a new class for stack entries. A stack entry object needs to have the state number and syntax tree reference and should be able to respond with



either of those state values on request. The class described by the UML class diagram in Figure 16.6 provides this functionality.

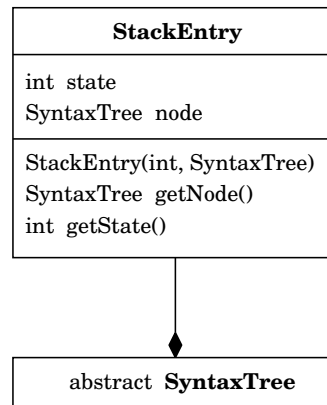


Figure 16.6: UML Class Diagram for `StackEntry`

### Implementing shift and reduce actions

In the simple LR parser the shift actions were simple – push the state number associated with the particular action. If we are to generate appropriate syntax tree nodes, however, the method `ShiftEntry.doAction` must be modified so that it pushes appropriate syntax tree structures. The strategy is to divide the token classes into two groups, the punctuation tokens and the data-containing tokens. Let's say the data-containing tokens are  $X_1\_T, \dots, X_n\_T$ . For each of these data-containing tokens there will be a syntax tree class for generating corresponding objects. If the corresponding syntax tree classes are named  $X_1ST, \dots, X_nST$  then the new version of `ShiftEntry.doAction` would have the following form.

```

public void doAction(PTable pt, Stack<StackEntry> st, Token t)
    throws ParseException {
    SyntaxTree val = null;
    switch (t.getType()) {
    case X1_T:
        val = new X1ST(t);
        break;
    ...
    case Xn_T:
        val = new XnST(t);
        break;
    default: // for all other tokens -- i.e., punctuation
        // val == null from above
    }
    // val references an appropriate syntax tree structure or null
    st.push(new StackEntry(nextState, val));
}
  
```

The simple LR parser reduction process is also simple, a matter of popping the stack a specified number of times and pushing a new state number – all reductions are handled by the same algorithm. But in generating syntax tree nodes the action taken for each reduction will be different. Since every `ReduceEntry` object contains a reference to the rule to be reduced, a call to `doAction` will have access to the structure of the rule via its particular enumerated value. The following abbreviated Java code illustrates the new form of the `doAction` method in the class `ReduceEntry`.

```
public void doAction(PTable pt, Stack<StackEntry> st, Token t)
    throws ParseException {
    SyntaxTree val = null;
    switch (rule) {
    case STMT_LIST_R:
        break;
    case LIST_REC_R:
        break;
    case LIST_END_R:
        break;
    case STMT_DEC_R:
        break;
    case STMT_ASSIGN_R:
        break;
    case STMT_SL_R:
        break;
    }
    // val references an appropriate syntax tree structure
    int next = st.peek().getState();
    GotoEntry goE = (GotoEntry)pt.getEntry(next, rule.getLeftSym());
    st.push(new StackEntry(goE.getState(), val));
}
```

You will recognize the case labels to be the enumerated values in `RuleName` for *PDef-lite*. The idea is to place the appropriate stack processing code at each case. This method will be elaborated in the next section.

## 16.6 The Supplied Java Code

One of the appealing things about the LR parsing strategy is that any additional functionality that is grammar oriented can be easily added to the parser. One must simply identify what action to take on each shift operation and what action to take on each reduce operation. So having a parser for *PDef-lite*, we can extend it to generate a syntax tree by identifying what should be done on shift and reduce actions. At the end of the last section we also identified two small but vital other issues in implementing the syntax tree generator.

```
class StackEntry
```

The stack used by the parser must be modified to allow entries containing not just the state value but also a reference to a syntax tree component. There are two areas to look at. First, the class `Parser` must be defined to initialize the stack appropriately. Second, we must look at the class that defines the structure of the stack entries. The following code implements the class `StackEntry`, whose UML class diagram is in Figure 16.6.

```
public class StackEntry {
    private int      state;
    private SyntaxTree node;
    public StackEntry(int st, SyntaxTree t) {
        state = st;
        node = t;
    }
    public int getState() { return state; }
    public SyntaxTree getNode() { return node; }
    public String toString() {
        return "state:" + state;
    }
}
```

The class is used in the class `Parser` to declare and instantiate the stack. Another important point is to appropriately initialize the stack. For the LR parsing algorithm to work we initialize the stack with the state 1. Now the entry requires a second parameter which generally refers to a component of the syntax tree. In the initial stack entry we use the `null` reference for this initial entry. The reason we choose `null` is because the entry will never be referenced. A look at the various parse tables we have constructed shows that when the end of data is seen and that token is needed on the stack so that the final reduction can be carried out, that the parsing terminates without shifting the end of data token to the stack. So the final syntax tree reference will be found in that last entry (of two) on the stack. Here is the code responsible for initializing the stack in `Parser`.

```
private Stack<StackEntry> stack = new Stack<StackEntry>();

public StatementList parseProgram() throws ParseException {
    stack.push(new StackEntry(1, null));
    while (pTable.notDone()) {
        int st = stack.peek().getState();
        PTableEntry pte = pTable.getEntry(st, SymbolName.token2Symbol(currentToken));
        pte.doIt(pTable, stack, currentToken);
        if (pte.getClass().getName().equals("parser.ShiftEntry")) {
            currentToken = tokenStream.getNextToken();
        }
    }
    return (StatementList)stack.peek().getNode();
}
```

### The method `ShiftEntry.doAction`

In *PDef-lite* there are just two data-containing tokens, those for types and identifiers, with the rest of the tokens being punctuation. The syntax tree hierarchy has already accommodated these token classes in the form of the classes `TypeST` and `IdentST`. The new version of `ShiftEntry.doAction`, then, has the following form.

```
public void doAction(PTable pt, Stack<StackEntry> st, Token t)
    throws ParseException {
    SyntaxTree val = null;
    switch (t.getType()) {
    case TYPE_T:
        val = new TypeST(t);
        break;
    case IDENT_T:
        val = new IdentST(t);
        break;
    default: // for all other tokens -- i.e., punctuation
        // val == null from above
    }
    // val references an appropriate syntax tree structure or null
    st.push(new StackEntry(nextState, val));
}
```

### The method `ReduceEntry.doAction`

The most interesting code alterations for syntax tree generation are in the method `ReduceEntry.doAction`. This method must be written so that for each rule the correct syntax tree structure is generated. We will illustrate the strategy by analyzing the *PDef-lite* grammar rules in the context of the reduction process. Before we focus on the code for each rule, we should understand what considerations determine the code sequence for each rule. Rule 3, called `LIST_END_R`, will always be the first `L` rule to be reduced. So for every *PDef-lite* list of statements, rule 3 will reduce to generate the list with one element. So when rule 3 is reduced we can instantiate a new `LinkedList` and put the node value from the one popped stack entry into the list. Rule 2 will be reduced for all subsequent entries in the list. When rule 2 is reduced there will be a reference to a `LinkedList` object and a reference to another `Statement` object on the stack. The reduction process should place the reference to the statement onto the list. With this introduction we turn to the rule-by-rule design of the reduction actions.

#### 1. `SL -> lbT L rbT`

When this rule is reduced we will pop three entries from the top of the stack, as in the simple parser. The two entries for `lbT` and `rbT` will contain no data for us (i.e., `getNode` returns `null`) but the middle entry should contain a reference to a `LinkedList` object. We will use this reference as the parameter to construct a new `StatementList` object, which will be the final value for this rule.

```
temp = stack.pop();
```

```

LinkedList<Statement> mlist = stack.pop().getNode();
temp = stack.pop();
val = new StatementList(mylist);

```

2. L -> L commaT S

This rule is discussed above.

```

val = stack.pop().getNode();
temp = stack.pop();
SyntaxTree mstatement = stack.pop().getNode();
val.add2StmtList(mstatement);

```

3. L -> S

This rule is discussed above – we create a new `LinkedList` and add `S` to it.

```

val = new StatementList();
SyntaxTree mstatement = stack.pop().getNode();
val.add2StmtList(mstatement);

```

4. S -> identT assignT identT

When this rule is reduced we must create an `Assignment` object. The strategy should be clear: pop three entries and pass the syntax tree component from the first and third entries to the `Assignment` constructor. That gives us the final value. It is important to remember that when the class `Assignment` was defined we included two constructors. The relevant one here is that which takes two parameters of type `IdentST`.

```

IdentST idR = stack.pop().getNode();
temp = stack.pop();
IdentST idL = stack.pop().getNode();
val = new Assignment(idL, idR);

```

5. S -> typeT identT

When this rule is reduced we must create a `Declaration` object. We pop two entries and pass the syntax tree component from each to the `Declaration` constructor. That gives the final value. As for the previous rule, remember the second constructor for `Declaration`.

```

TypeST id = stack.pop().getNode();
IdentST ty = stack.pop().getNode();
val = new Declaration(ty, id);

```

6. S -> SL

When this rule is reduced we simply take the syntax tree reference from the entry popped from the stack and then include the same reference in the new node pushed on the stack.

```

val = stack.pop().getNode();

```

The code segments given above will be substituted into the appropriate switch statement cases in `ReduceEntry.doAction` below. Notice how the declarations of `val` and `temp` set up their use in the case code and, in the case of `val`, at the end of the method.

```

public void doAction(PTable pt, Stack<StackEntry> st, Token t)
    throws ParseException {
    SyntaxTree val = null;
    SyntaxTree temp = null;
    switch (rule) {
    case STMT_LIST_R:
        break;
    case LIST_REC_R:
        break;
    case LIST_END_R:
        { val = new StatementList();
          SyntaxTree mstatement = stack.pop().getNode();
          val.add2StmtList(mstatement);
        }
        break;
    case STMT_DEC_R:
        break;
    case STMT_ASSIGN_R:
        break;
    case STMT_SL_R:
        break;
    }
    // val references an appropriate syntax tree structure
    int next = st.peek().getState();
    GotoEntry goE = (GotoEntry)pt.getEntry(next, rule.getLeftSym());
    st.push(new StackEntry(goE.getState(), val));
}

```

There is one important thing to point out in the substitution of code segments. In the segments where a variable is declared, the segment must be surrounded (within the case) by curly braces. This has been illustrated above in the case for LIST\_END\_R.

## 16.7 Guided Development

Extending the PDef-*lite* syntax tree generator to syntax tree generator for PDef is quite straightforward. For convenience we reproduce the PDef grammar here.

```

0 P  → SL
1 SL → lbT L rbT
2 L  → L commaT S
3 L  → S
4 S  → identT assignT E
5 S  → typeT identT
6 S  → SL
7 E  → E addOpT T
8 E  → T
9 T  → T mulOpT F
10 T → F
11 F → intT
12 F → floatT
13 F → identT
14 F → lpT E rpT

```

You may also want to reference the SLR parse table for PDef given in Figure 16.1.

### ☛ **Activity 63** –

1. For each of the grammar rules numbered 7-14 picture what will happen when that rule is reduced, being sure to identify what should be on the stack at the time of the reduction. Based on this analysis, for each rule write a segment of code which should be executed when that rule is reduced. Reference the PDef-*lite* implementation for `ReduceEntry.doAction` above to insure that your code block will work when substituted into the `doAction` switch statement.
  2. Revisit grammar rule 4 of the PDef grammar and make adjustments from the PDef-*lite* PDef-*lite* implementation.
  3. Starting with the PDef parser, completed in the Guided Development of the previous chapter, make the appropriate changes to implement the syntax tree generation for PDef.
  4. Substitute the new parser package into the PDef implementation generated at the end of Chapter 14. Compile and test your new parser package.
-





**The Back-end**  
**– Synthesis Principles –**



# Chapter 17

## Introduction

In the Prelude we gave an overview of the structure of a language translator, describing it as the composition of two components, the front-end analyzer, dealing with syntax and static semantic checking, and the back-end synthesizer, which deals with code generation and optimization. Having dedicated the previous part of the book to the analysis phase, we devote this part to the synthesis phase.

In the chapters to come we will illustrate one technique for the formal specification of dynamic semantics and then apply the description in defining a basic code generation implementation, discuss intermediate code generation and code optimization, in the form of common subexpression elimination and register allocation.

We present an overview of the chapters in this last part of the book, marking each description with those Translator Principles.

### **An Example Language – FP :**

In this chapter we introduce an example language called FP, which stands for *functional programming*. FP will be the target language for synthesis algorithms discussed in the chapters to come. We define FP in terms of formal descriptions of syntax, static semantics, and dynamic semantics. While this language looks a lot different (syntactically) from PDef, the two languages actually have a lot of structure in common, which will make the implementation somewhat straightforward.

### **FP Code Generation:** [Correctness]

An initial activity in this chapter is to write a recognizer for FP – this is to be done as an independent project. We discuss new techniques where needed, but otherwise the reader is expected to build a recognizer via the techniques presented in the analysis part of the book. The bulk of the chapter focuses on the design and implementation of an FP code generator. Following this discussion the reader should be able to complete the complete implementation for FP, including analysis, synthesis, and the required run-time environment.

### **Intermediate Code Generation:** [Correctness, Portability]

To facilitate the Portability Principle it is beneficial to generate an intermediate form of a program, which is hardware independent, before actually producing the final executable, hardware dependent, form. In this chapter we discuss three-address code and how to define FP dynamic semantics in terms of this intermediate code. We also examine issues of design and implemen-

tation for FP code generation in terms of three-address code. Once again, the formal nature of the translation from FP to three-address code carries through the support for the Correctness Principle. In addition, three-address code facilitates certain kinds of code optimization – i.e., the Efficiency Principle. In this chapter we discuss algorithms for eliminating the computation of common subexpressions. The expression  $(x + (a - b)) * (a - b)$  has two occurrences of  $(a - b)$  – the code generator of Chapter 19 causes this expression to be evaluated two times. The architecture of the three-address code makes it possible to generate code that will evaluate the expression once and then use the value in place of the second occurrence of the expression. The implementation of this optimization is the focus of the second half of the chapter.

**MIPS Code Generation and Optimization:** [Efficiency]

In this chapter we discuss the basic problems associated with generating object code for the MIPS architecture. Since our target is an actual processor, the Efficiency Principle requires we make appropriate efficient use of the processor's facilities. Thus, we dedicate the second half of the chapter to a discussion of register allocation and its implementation in the code generator.

**Flow Graph Analysis:** [Efficiency]

This last chapter extends the discussion common subexpression elimination in Chapter 20. In that chapter we discussed the topic in the context of the FP selection statement. While FP has no repetition statements, they are also an important obstacle to common subexpression elimination. In this chapter we extend our algorithm to include repetitions statements with a single entry and exit.

## Chapter 18

# An Example Language – FP

The name FP stands for “functional programming” and implies that one programs by defining a set of functions and then applying them to solve a particular problem. Thus, an FP program should include function definitions and an expression, which is a combination of data and function applications. As is expected, FP has a small collection of built-in functions, the arithmetic and boolean operations, which can be used in defining functions and the program’s expression.

The syntax for function definitions is simple but possibly a bit different from the function definitions you are used to. Here is a sequence of three FP function definitions.

```
add x y = x + y;
mult x y z = x * y * z;
combine a b = a * (add a b) * (mult a b (a + b))
```

Some quick observations. First, a function’s formal parameters are not surrounded by parentheses, but rather are space separated. This form may be unfamiliar unless you have programmed with a modern functional language such as Haskell. Second, the semi-colon is a definition separator, not a terminator. The expressions on the right of each definition and the program’s own expression have the usual form of function-based arithmetic expressions.

There is an additional feature of FP that allows the programmer to use abstraction and thus limits the number of unique function names required. The feature is the nested function definition. Here is an example.

```
combine a b = a * (add a b) * (mult a b (a + b))
  where
    add x y = x + y;
    mult x y z = x * y * z
  endw
```

The **where**-clause, also called a definition block, introduces a new, local list of function definitions. The definitions are considered to be part of a new block and are not known outside that block.

These function definition elements are combined with an expression, analogous to the main function in a C++ program, gives the full form of an FP program. Notice in the following example of an FP program that the structure makes use of the **where**-clause.

```

3 + (combine 4 5)
where
  combine a b = a * (add a b) * (mult a b (a + b))
  where
    add x y = x + y;
    mult x y z = x * y * z
  endw
endw

```

There is one final feature of function definitions that makes it possible to define interesting recursive functions, the *selection* expression. Here is an example of a function that uses selection.

```
fact x = if x == 0 then 1 else x * (fact (n-1)) endif
```

The selection expression works as you expect. If the selection condition is true then the expression following **then** is evaluated, otherwise the expression following **else** is evaluated.

Since the only data type recognized by FP is integer, there is no need for formal declarations in a program. It is, of course, necessary to name function parameters in order to specify how the function combines them; specifying function parameters also allows a translator to determine the number of parameters that will be required in a call to the function.

There are restrictions on what constitutes a legal FP program. There can be only one function definition with a given name in a definition block. But function and parameter names can be redefined in nested definition blocks. Function calls must have a number of arguments matching the number of parameters in the function's definition. These restrictions form the low-level semantics of the language. There is a high level semantics that defines which results will be computed when an expression is evaluated. The easiest thing to say is that the functions evaluate exactly as they appear to – the semantics is based on the semantics of arithmetic expressions with which we should be familiar.

There are two special restrictions that avoid problematic translation situations in FP programs. First, it is not allowed that parameters of a function be used globally in nested function definitions. Second, it is not allowed to redefine a function name as one of its own parameters. This structure would normally be allowed, but it seems that for style purposes it is not an unreasonable restriction.

## 18.1 FP Syntax

### Alphabet of FP

The FP alphabet is the following set of characters.

```

alphabet = { a ... z
            A ... Z
            0 ... 9
            =
            + * - / %
            < > !

```

```
(
)
;
}
```

## FP Tokens

Figure 18.1 displays a description of the FP token classes in the form of regular expressions.

Token Class	Regular Expression	Termination Characters
addT	+	any character
subT	-	"
multT	*	"
divT	/	"
modT	%	"
lpT	(	"
rpT	)	"
scolonT	;	"
assignT	=	any character other than '='
compOpT	<   >	"
	<=   >=   ==   !=	any character
intT	0   [1 - 9][0 - 9]*	non-digit
identT	[a - zA - Z] <sup>+</sup>	non-letter
whereT	where	"
endwT	endw	"
ifT	if	"
thenT	then	"
elseT	else	"
endifT	endif	"

Figure 18.1: Regular Expression for FP Tokens

## FP Grammar

The grammar level of FP is defined by the context free grammar given in Figure 18.2. Based on this grammar each of the following is an FP program – notice that the value computed is also given.

- 12 \* (121 - 18)  
evaluates to 1236
- 12 \* (f 121 18)  
where f a b = a - b endw  
evaluates to 1236

- $(f\ 5\ 2) + (g\ 5)$   
 where  
      $f\ a\ b = h\ (a + b)\ (a - b)$   
         where  $h\ x\ y = x * y$   
         endw;  
      $g\ a = 3*a + 2$   
 endw  
 evaluates to 38

```

Program    → Exp [ Where ]
FnDefList → FnDef { scolonT FnDef }
FnDef     → identT ParamList assignT Exp [ Where ]
Where     → whereT FnDefList endwT
ParamList → identT { identT }
Exp       → subT Factor
          → FnApp
          → Selection
          → MExp
MExp     → MExp ( addT | subT ) Term | Term
Term     → Term ( multT | divT | modT ) Factor | Factor
Factor   → intT
          → identT
          → lpT Exp rpT
Selection → ifT BExp thenT Exp elseT Exp endifT
BExp     → Factor compOpT Factor
FnApp    → identT ArgList
ArgList  → Factor { Factor }

```

Figure 18.2: Context Free Grammar for FP Tokens

## 18.2 FP Static Semantics

Remember that static semantics refers to the translation-time semantics of identifiers and literal values – that is semantic properties that can be checked at translation time. In PDef, identifiers were used for a single purpose, as parameter names, and as such can have a single attribute, the type. In FP, identifiers are used in two different ways, as formal parameter names and as function names. In this section we will first provide definitions in FP for the notions of block and scope and, based on these, describe the static semantics of FP.

FP is block-structured like PDef, Java, and most other programming languages. The block structure of PDef is very simple because there is only one kind of declaration that can occur – the variable declaration. What complicates the notion of block for FP is the presence of function



definitions and the fact that a function definitions can be nested. Thus, the definitions of block and scope must be defined with precision.

### 18.2.1 Blocks

There are two uses of identifiers in FP: parameter names and function names. Each of these has its own declaration syntax. A parameter name is declared simply by including it in a parameter list in a function definition. A function name is declared via a function definition which has two and possibly three parts: the parameter list, the function's expression, and optionally a nested list of functions bounded by the **where/endw** keywords. So declarations are simple.

There are two kinds of blocks in FP. The outermost block coincides with the whole program (the expression plus its optional function definition list). In addition, every function definition determines a nested block starting with the function's first parameter and extending to the end of the function's definition, which is either the end of the function's expression or the **endw**, which marks the end of a function's nested **where** clause. Here is an example.

```
f (k 2) 3
where
  f a b = (g a b) + (h b a)
    where
      g x y = k (y + x);
      h a y = a - y
    endw;
  k x = 2 * x
endw
```

The block structure of this example is illustrated in Figure 18.3(a). The box surrounding the entire program marks of the outer-most block. That block holds four other blocks, each defined by a function definition. The block defined by **f**, as described above, extends from its first parameter to the **endw**, which marks the end of **f**'s **where** clause. **f** has two nested blocks, one defined by **g** and one by **h**. Finally, there's the block defined by **k**. Notice that while a function's block is nested, the function name resides in the outer block; so **f** and **k** are in the outer-most block, while **g** and **h** are in **f**'s block.

### 18.2.2 Declaration Scopes

FP has two kinds of declarations, parameters and function names, and each has its own scope rules. Parameters are easy since they can be declared only in the parameter list of a function. So, the define-before-use rule is natural. In fact the scope of a parameter extends from the parameter's declaration to the end of the function's expression – notice it does not extend through a **where** clause.

Function name declarations, on the other hand, follow the use-anywhere scope rule. You can see this in the example above where uses of **f** appear in the program's expression, even though the **f**'s definition hasn't appeared yet. The scope of a function name declaration also includes any nested **where** clauses for the function. This facilitates recursive function calls. We see evidence of the use-anywhere scope rule in Figure 18.3(b), where there are arrows pointing from a function

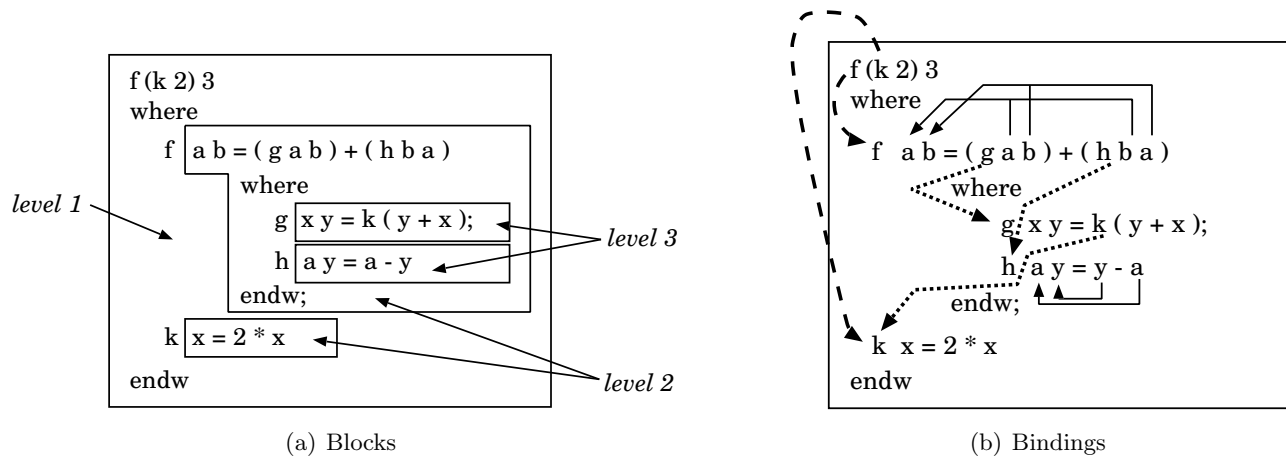


Figure 18.3: FP Block and Binding Structures

name use to the that function’s definition. While this rule is common in functional programming languages and makes programming very convenient, it does make the problem of code generation more challenging – namely, it is difficult to know the address of a function’s generated code before it has been generated! We will address this problem when we implement FP.

### 18.2.3 FP Static Semantic Rules

With these definitions in place we can state the following static semantic rules for uses of identifiers in an FP program. It is important that these rules are in addition to the general scoping rules presented in Figure 8.1 (see page 125).

1. A parameter name cannot be the same as the name of it’s function.
2. Each identifier must be in the scope of a declaration of the identifier.
3. In an expression, an identifier used as a variable must bind to a declaration of a parameter in the expression’s function, i.e., parameters can’t be used in nested function definitions.
4. In an expression, a call to a function with  $n$  arguments must be in the scope of a declaration for a function of the same name with  $n$  parameters.

On the attribute side, each kind of identifier has its own set of attributes. A parameter name has a single attribute – its location position in its parameter list (starting with position 1). A function identifier has two attributes – the number of declared parameters in its definition and its location, an attribute we will define specifically when we talk about the FP-machine architecture.

## 18.3 FP Dynamic Semantics

In Chapter 2 we discussed various methods of expressing dynamic semantics – we provided, in Section 2.4.3, an illustration by specifying an operational semantics for the language CL. In this chapter we will follow the same approach and define an operational semantics for the FP.

We will begin with a description of an abstract stack machine that will serve as the basis for the semantic description. We will call this the *FP stack machine* or simply the *FP machine*. This stack machine structure will then be employed in defining the semantic function `trans` for FP, thus providing the dynamic semantics for FP. The FP dynamic semantics will then be illustrated by evaluating `trans` on example FP programs.

### 18.3.1 The FP Stack Machine

The FP machine is a very simple pure stack machine and reflects much of the structure of the CL machine defined in Section 2.4.3. Unlike the CL machine, however, the FP machine must provide a convenient way to store the translated code for each function in an FP program as well as the code for the program's expression. The machine must also provide a mechanism for recursive function calls. While most of the structure of the FP machine will seem quite conventional, the memory for storing the executable code may not.

The architecture of the FP machine is illustrated in Figure 18.4 and comprises three components, the *control*, *code table*, and *run-time stack*. To understand the control and run-time stack you need to understand how code is stored for execution. The normal code structure, that used for imperative languages, is a linear address space, as we see in the physical processors in the devices we use. But due to the nature of FP and because we are going to write an interpreter we can be more creative. Rather than a linear address space (which is basically a 1-dimensional address space) we will use a 2-dimensional space. When code is generated for a function, its code will go into an assigned row of the code space – obviously each function will be allocated a different row. The code generated for a program's expression will always be translated into the first row (offset 0) in the code space. The order of the other functions is not important. You can now see that the “location” attribute of a function name is its assigned row number.

The control facilitates program execution and uses four registers in the process, as listed below.

- SP** – Stack Pointer (top of stack) register: holds the offset in the stack of the top element on the stack.
- FP** – Frame Pointer register: holds the offset in the stack of the base of the active stack frame.
- FN** – Function register, holds the address of the currently executing function, where the address is just the index of the code table row holding the code for the function.
- PC** – Program Counter register: holds the offset into the row for the currently executing function for the *next* instruction to be fetched.

The machine has a standard fetch/execute cycle. On the fetch side, the next instruction is retrieved from position PC on row FN in the code table. Immediately after the fetch, the value of PC is incremented. When a function call occurs, the called function's row number goes into FN and 0 goes into PC.

The control executes a machine language of 17 instructions and all, save the *push* instruction, take no argument. The important thing about the pure stack machine is that all data needed

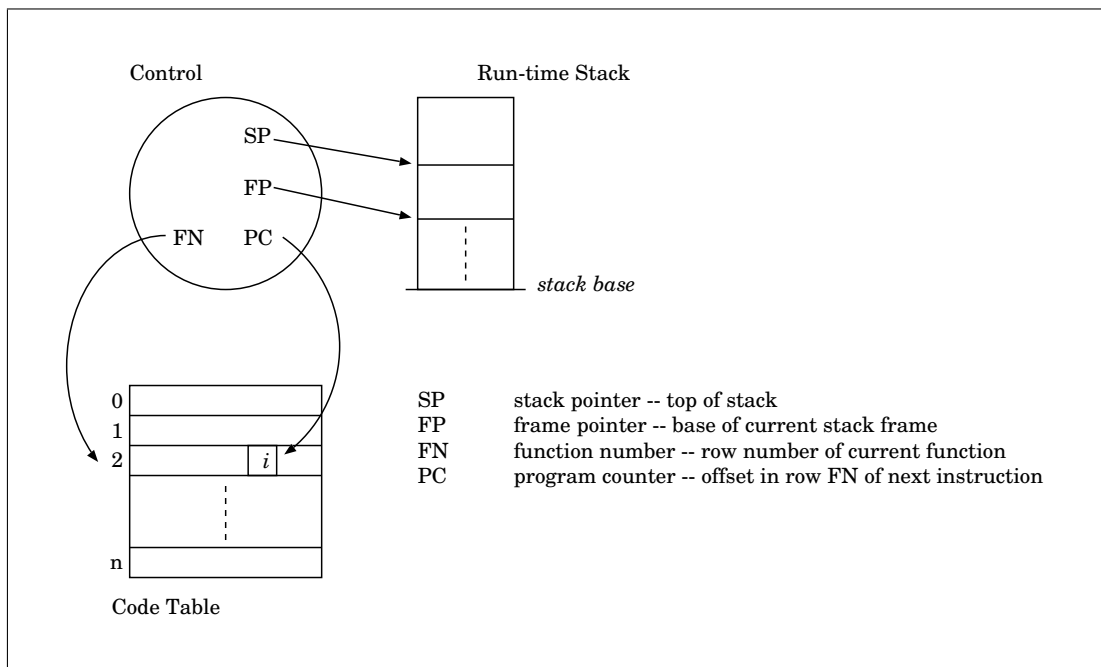


Figure 18.4: FP Machine Architecture

for instruction execution must be stored at the top of the stack. The execution of an instruction typically involves the control popping required arguments from the stack and then executing the instruction – if a result is generated it is pushed onto the stack.

Two instructions make this machine more than a simple calculator. The instructions `jmp` (a conditional jump) and `call` (function call) facilitate the change in the default sequential execution of instructions. The complete description of the 17 FP machine instructions is given in Figure 18.5.

### 18.3.2 Dynamic Semantics for FP

Since our approach to FP dynamic semantics is operational, we define the semantics in terms of an underlying FP machine. We base the semantic description on the FP syntactic structures, indicating for each structure the sequence of machine instructions that provide the required functionality, thus guaranteeing the Correctness Principle (see Figure 1.2.1 page 6). We will present the semantics and then give some examples. The semantics are defined in the form of a recursive function `trans`.

#### Identifier

`trans(V) = push offset, load`

where `offset` is the offset attribute for `V` found in its symbol table entry.

#### Number

`trans(N) = push N`

where `N` refers to its literal value.

#### Builtin Function Application

add, subt, mult, div, mod, lt, le, gt, ge, eq, neq	pop stack $\rightarrow$ p1 pop stack $\rightarrow$ p2 stack $\rightarrow$ (p2 op p1):stack
push x	stack $\rightarrow$ x:stack
load	pop stack $\rightarrow$ d stack[FP + d + 2] $\rightarrow$ a stack $\rightarrow$ a:stack
jmp	pop stack $\rightarrow$ d pop stack $\rightarrow$ a if a $\leq$ 0, d $\rightarrow$ PC
call	pop stack $\rightarrow$ fn pop stack $\rightarrow$ np FP $\rightarrow$ stack[SP - np] FN $\rightarrow$ stack[SP - np - 1] PC $\rightarrow$ stack[SP - np - 2] SP - np - 2 $\rightarrow$ FP fn $\rightarrow$ FN 0 $\rightarrow$ PC
return	FP $\rightarrow$ s pop stack $\rightarrow$ rv for (i = 0; i < SP - FP - 3; i++) pop stack pop stack $\rightarrow$ FP pop stack $\rightarrow$ FN pop stack $\rightarrow$ PC s $\rightarrow$ SP rv $\rightarrow$ stack[SP]
halt	stop fetch/execute cycle

”stack  $\rightarrow$  FP:stack” means that the value of the **FP** register is pushed onto the stack. Names p1, p2, x, a, d, etc., represent variables local to the machine implementation.

Figure 18.5: Semantics of FP Machine Instructions

$$\text{trans}(t1 \text{ op } t2) = \text{trans}(t1), \text{trans}(t2), \langle \text{bf} \rangle$$

where op is one of the builtin functions

$$+, *, -, /, \%, <, \leq, >, \geq, ==, /=$$

and  $\langle \text{bf} \rangle$  is the corresponding machine instruction for the builtin function (‘add’ for +, for example).

### FnApp

$$\begin{aligned} \text{trans}(f \ t1 \ \dots \ tn) &= \text{push } 0, \text{push } 0, \text{push } 0 \\ &\quad \text{trans}(t1), \dots, \text{trans}(tn), \text{push } n, \text{push } fn, \text{call} \end{aligned}$$

where fn is the row number in the code table for the code for f and n is the number of parameters for f.

The three ‘push 0’ instructions are to make space on the stack for storing the three register return values – i.e., FP, FN, PC.

### Selection

```
trans(if b then t else f endif) =
  trans(b), push x, jmp, trans(t), push 0, push y, jmp, trans(f)
```

where *b*, *t*, *f* are expressions, *x* is the address within the current code table row of the first instruction of ‘trans *f*’, and *y* is the address within the current code table row of the first instruction *after* ‘trans *f*’.

Note: The “push 0” in the code sequence is necessary to force the jump around the **else** code block.

### NegExp

```
trans(-e) = trans(e), push -1, mult
```

where *e* is an expression.

### Function Definition

```
trans(f p1 ... pn = e) = trans(e), return
```

where the function *f* has the associated expression *e*. Notice here that there is no reference to a possible nested function definition list! But definitions in a **where**-clause will be translated with exactly this same definition.

### Program

```
trans(e where fd1 ... fdn endw) =
  trans(e), halt, trans(fd1), ..., trans(fdn)
```

where *fd1*, ..., *fdn* are the function definitions and *e* the main expression. Of course, if *n* is zero then the where clause is missing – the program is simply an arithmetic expression to be evaluated.

## A Translation Example

To insure that the definition of **trans** is clear and to look ahead to the translation of FP programs to FP machine code, we will compute the function **trans** on the following simple FP programs.

```
f 2 3
where
  f a b = a + (g b);
  g a   = (m a) * (m a)
    where
      m x = x + x
    endw
endw
```

We begin by laying out the translation for the entire program. The following sequence shows the **trans** first applied to the program as a whole and then separating out the translation of the program expression.



```

load
push 1 // g takes 1 parameter
push 2 // g code at position 1 in
        // the code table
call   // call to g
add
return

```

The code for the translation of the entire program is shown in Figure 18.6. The number in parentheses next to the function name is the position in the code table of the function’s code – remember that the program expression goes into row 0. The rows are displayed in columns for obvious practical reasons!

	f 2 3 (0)	f (1)	g (2)	m (3)
0	push 0	push 1	push 1	push 1
1	push 0	load	load	load
2	push 0	push 0	push 0	push 1
3	push 2	push 0	push 0	load
4	push 3	push 0	push 0	add
5	push 2	push 2	push 1	return
6	push 1	load	push 3	
7	call	push 1	call	
8	halt	push 2	push 1	
9		call	load	
10		add	push 0	
11		return	push 0	
12			push 0	
12			push 1	
13			push 3	
14			call	
15			mult	
16			return	

Figure 18.6: Translation of the example program

We have defined the FP-machine and have also just seen an example of the code that should be generated for the expression `f 2 3`. To see the two at work at the same time, look at the diagram in Figure 18.7 (page 325), which displays the state of the FP-machine at each step in the execution of the translated program.





**Activity 64** –

For each of the following FP programs, apply the translation function `trans` as illustrated above.

1.  $3 * (4 + 6 / 2)$

2.  $2 + ( f\ 2\ (2*3)\ 4 )$

where

$$f\ x\ y\ z = x * y + z$$

endw

3.  $g\ ( f\ 2\ 3 )$

where

$$g\ a = f\ a\ a;$$

$$f\ x\ y = x * y$$

endw

---

## Chapter 19

# An FP Interpreter

Up to this point our interest in translation has been on the analysis end: check that the input is syntactically correct and that the input adheres to the static semantic rules. We are now ready to take on the back-end synthesis activity, that of converting the input into a form that can be executed on the target machine. We call this activity code-generation and in this chapter we take an initial, no frills approach to it. Our target language, as just presented, is the function language FP. There are three phases to this chapter. First, you will implement the front-end of a translator, applying those techniques learned in in the earlier chapter of the book. Second, we will discuss the design and implementation of a code-generator for the translator – this code generator will target the FP-machine presented in the last chapter. Third, we will discuss the design and implementation of a new code-generator whose target machine is a modified version of the FP-machine, one in which the program memory is linear rather than 2-dimensional. The modifications for this approach are not extensive, but the design must be examined carefully. To provide a general guide to the FP implementation, refer to the UML diagram in Figure 19.1.

When you have completed this chapter, you should have two working FP translators and a very basic model for any compiler you use.

### 19.1 Syntax Analysis

While the PDef experience of the earlier part of the book provides a good grounding for the front-end, the FP front-end has a couple of interesting new twists. To help out we will provide a path through the process, pointing out those twists and providing guidance for maneuvering around them. It is a good idea to review the appropriate tutorials (Chapters 12 – 15) before implementing each component of the front-end. The process should for the most part follow the patterns there.

After completing the work described in this section you will have a tokenizer and a parser for FP and the parser will generate a syntax tree. Because the components are the same functionally, from an external point of view, you should be able to use most of the driver classes as for the PDef implementations. For each component a synopsis of the relevant FP language structures will be given. The bulk of each section will discuss those aspects of the language that require new or revised techniques.

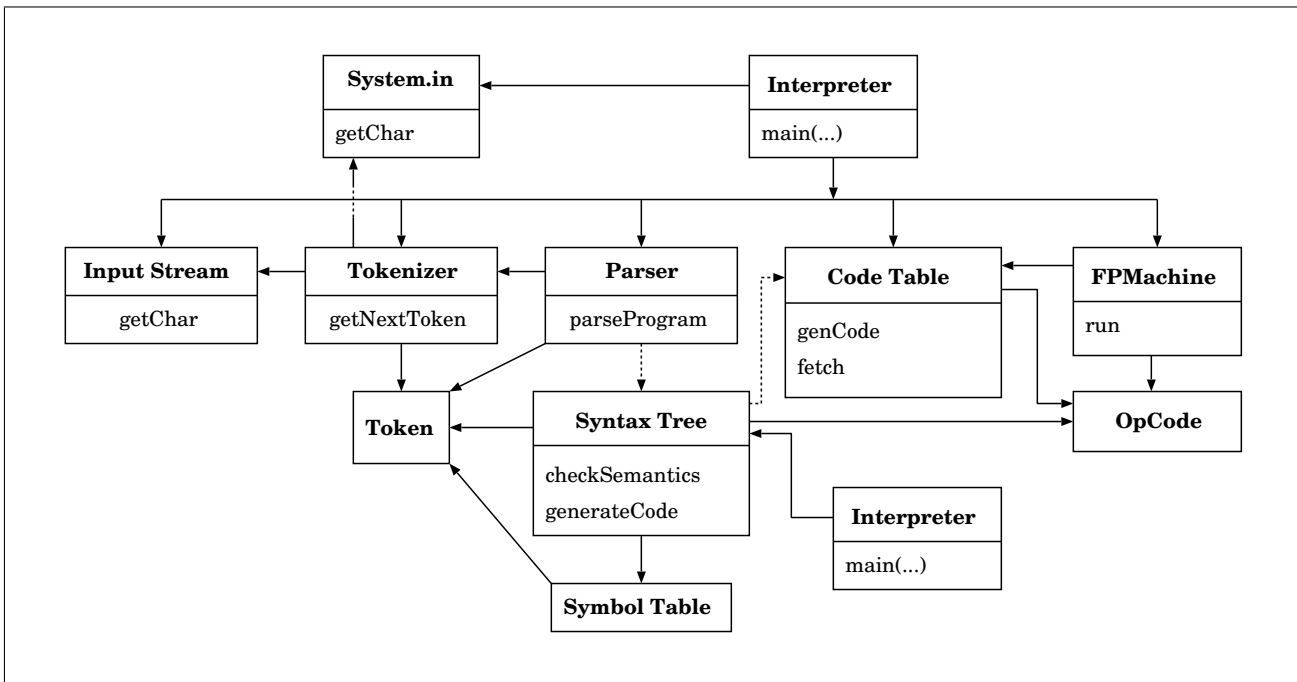


Figure 19.1: UML Class Diagram for FP Interpreter

### 19.1.1 Tokenizer for FP

The tokenizer for FP is quite similar to that for PDef. There are a few more key words and one-character operators, but those should be straightforward. On the other hand, there is a new problem with two-character operators – for example ‘<’ and ‘<=’. Here we have two tokens that start with the same character and then are different. You will need to draw an appropriate finite state machine and derive your solution. It is similar to the problem of knowing when an identifier token ends.

#### ☛ Activity 65 –

Implement an FP tokenizer based on the token description in Section 18.1.

---

### 19.1.2 Parser for FP

As was the case with the FP tokenizer, which you hopefully just completed, the FP parser is mostly very easy to implement. Just follow the strategies discussed and implemented in the paired Chapters 6 and 13. There is one twist in the FP grammar for the non-terminal `Exp` which needs special attention.

#### When a Form 4 parse rule is ambiguous

The grammar rule for `Exp` is the following.

```

Exp --> subT Factor
    --> FnApp
    --> Selection
    --> MExp

```

When we examine the various parse rule Forms from Chapter 6 (see Figure 6.9 **and** page 87) it is clear that **Exp**'s rule gives a choice – i.e., the rule is Form 4. While this rule is similar to that for **Exp** in the PDef grammar (see page 316), things get a bit more involved due to the presence of function calls. So it is a good idea to look back and read the discussion of Form 4.

Matching the approach for the non-terminal **Exp** we should first make a table with one row for each option and list in the first column the token alternatives and in the second the *first* set value for each alternative. For each option we determine the token or tokens that can start the option.

alternative	<i>first</i> set
subT Factor	subT
FnApp	identT
Selection	ifT
MExp	intT
	identT
	lpT

### ☛ Activity 66 –

Verify that the table is correct. Remember that when determining the *first* set for a non-terminal, if the rule does not start with a terminal symbol, then it will be necessary to look at the *first* set for the starting non-terminal – this process can, of course, ripple through multiple rules.

---

What we see from the table is that two of the options, **FnApp** and **MExp**, have the same token, **identT** in their *first* sets. Here is an example to consider.

```
f a b c = g ...
```

Here, the expression on the right starts with an identifier – the problem token just identified. The idea is that in the parser **currentToken** is **identT**, where the identifier is **g**. The discussion above indicates that we can't tell if the expression looks like an **FnApp** or an **MExp**. We could argue that since **g** isn't a parameter of **f**, **g** must be a function name. So in this case we would use the **FnApp** rule. The only problem with this argument is that it is context sensitive! Since our problem is a parsing situation, we can't count on such context sensitive information. So now what do we do?

Actually, we have seen this sort of thing before in the implementation of the finite state machine. In distinguishing two tokens that start with the same character, such as '**<**' and '**<=**', we have to be able to peek ahead to see if we have seen the entire token. But if we peek ahead we must make sure that the peeked character will still be the next character read. In our tokenizer we do this with the method **putBackChar**.

It would seem, then, that we need the same capability for tokens! When parsing the non-terminal `Exp` if we encounter an `identT` token we need to be able to look at the next token to see if that will tell us which case we have. There are two issues here. First, will looking ahead resolve the ambiguity and second, how can we provide a peeking mechanism for the tokenizer? We start with the first question.

### Does peeking help?

If we want to peek then we have to know what to look for. If `identT` is the first token in `FnApp` then the next possible token is easy to come by. You will find what follows `identT` in `FnApp` in the next Activity. So we turn to `MExp`. There are two cases, one where `MExp` involves multiple tokens and one where `MExp` has just one token. To solve the multiple token case we simply have to see what derivations (applying grammar rules in sequence) can be made starting from `MExp` where the first terminal to appear on the right is `identT`. Here are is a start with derivations.

$$\begin{aligned} \text{MExp} &\implies \text{Term} \\ &\implies \text{Factor} \\ &\implies \text{identT} \end{aligned}$$

Unfortunately, the only way `Factor` can be expanded to start with `identT` is to generate just a single token for `MExp` – we want multiple tokens. Here’s another try.

$$\begin{aligned} \text{MExp} &\implies \text{MExp addT Term} \\ &\implies \text{Term addT Term} \\ &\implies \text{Factor addT Term} \\ &\implies \text{identT addT Term} \end{aligned}$$

So this partial derivation verifies that `addT` can follow `identT` in `MExp`. This process is a bit tedious, but the grammar is small and there are only two rules for `MExp`.

### ☛ Activity 67 –

Determine all the tokens that can follow `identT` when it is the first token in `MExp`. Be complete. There are three tokens – you already know one of them. Show the derivations that verify your answers.

---

Now we want to look at the other case, when `MExp` has just one token - `identT`. This problem is a bit more involved. If we want to know what can come after `identT` and `MExp` has only one token, then we are looking for all the tokens that can follow `MExp` in a derivation – remember this is the definition of *follow*(`MExp`) (see Section 6.2.3 page 89). Here’s the definition to refresh your memory.

$$\begin{aligned} \text{follow}(X) = \{x \mid &x \text{ is a terminal symbol and } S \implies^* \alpha X x \beta \\ &\text{or} \\ &x \text{ is } \epsilon \text{ and } S \implies^* \alpha X \} \end{aligned}$$

We find this *follow* set by applying derivations in a way slightly different from that above. We don't want to start with `MExp` on the left, we want to start with other non-terminals and see if we can get `MExp` on the right – whatever follows `MExp` is in its follow set.

The best place to start is with the grammar. Take a look at the rules and see if you find any that can generate `MExp` followed by a token – all such tokens go into the *follow*(`MExp`). But there are other tokens that might belong to the *follow* set. One thing to notice is that `Exp` can generate `MExp` alone, which means that if a terminal can immediately follow `Exp`, then it can immediately follow `MExp` as well. This means that we must examine the rules that generate `Exp` as well! Are there any other non-terminals that can generate `MExp` alone? A scan of the grammar rules shows the answer is no. So we should focus on non-terminals whose rules have `Exp` on the right – this means the rules for the non-terminals `Program`, `Fndef`, `Factor`, and `Selection`. We also have to be aware of rules where these non-terminals appear on the right and factor them in as well: the rules for `FndefList` and `Where`.

So let's try one of these. `Factor` has a rule with `lpr Exp rpt` as right-hand side. So we can do the following derivation.

$$\begin{aligned} \text{Factor} &\Rightarrow \text{lpr Exp rpt} \\ &\Rightarrow \text{lpr MExp rpt} \end{aligned}$$

In this case `rpt` follows `MExp`: if `MExp` is the single token `identT` then `rpt` can follow. Here's a derivation starting with `Program`.

$$\begin{aligned} \text{Program} &\Rightarrow \text{Exp Where} \\ &\Rightarrow \text{MExp Where} \\ &\Rightarrow \text{MExp whereT FndefList endwT} \end{aligned}$$

Thus, the token `whereT` is also in *follow*(`MExp`).

### Activity 68 –

1. Determine *follow*(`MExp`) following the procedure just demonstrated. Show all derivations that determine an element of the follow set. There are 12 tokens to be found.
2. Argue that `thenT` is **not** in *follow*(`MExp`).

---

Though it gives you a massive hint on some of the Activities above, we display in the following table all the tokens that can follow `identT` in either `FnApp` or `MExp`. What we see, with some relief, is that the tokens in the `FnApp` row are disjoint from those in the `MExp` row.

alternative	starts with	second token
<code>subtT Factor</code>	<code>subtT</code>	
<code>FnApp</code>	<code>identT</code>	<code>identT, intT, lpT</code>
<code>Selection</code>	<code>ift</code>	
<code>MExp</code>	<code>intT</code>	
	<code>identT</code>	<code>multT, divT, modT, addT, subtT, whereT, endwT, periodT, colonT, rpT, elseT, endifT</code>
	<code>lpT</code>	

Now assuming a mechanism to put back a token onto the token stream, we should be able to sketch the parsing strategy for `Exp`. The crucial property, of course, is that the possible second tokens do not overlap. This means that we can use a switch structure on the peeked token value to determine whether to call `parseFnApp` or `parseMExp`. With this analysis in hand we can complete the `parseExp` method that started this long discussion.

```

switch (cT.getType()) {
case subT:  consume(subT); parseFactor(); break;
case ifT:   parseSelection(); break;
case intT:
case lpT:   parseMExp(); break;
case identT:
    {
        Token next = tins.getNextToken();
        // now put back the token since that was just a peek
        switch (next.getType()) {
        case intT: case identT: case lpT:
            parseFnApp(); break;
        case multT:   case divT:   case modT:   case addT:   case subT:
        case whereT:  case endwT:   case scolonT: case rpT:   case elseT:
        case endifT: case periodT:
            parseMExp(); break;
        default:
            { String out = "multT, addT, subT, whereT, endwT, periodT, ";
              out += "scolonT, rpT, thenT, elseT, endifT"
              throw new ParseException(out, cT);
            }
        }
    }
    break;
default: throw new ParseException
        ("Expected subT, ifT, intT, lpT, or identT", cT);
}

```

We draw your attention to the comment on the third line of the case for the `identT` token. Notice that as soon as we have read that second token we want to put it back on the token stream. We wisely saved the value so we can continue our analysis knowing that the token stream is in an appropriate state.

### Being more sensible

The code above looks good and we could use it, but there is another consideration that will encourage us to adopt a different structure. Notice that after seeing an identifier we are really only interested in calling `parseFnApp` or `parseMExp`, so if we determine that one of these is not appropriate then we should simply call the other. This would mean leaving it up to this “other” parse method to determine if there is an error with the peeked token. In addition, there is the possibility that we will not correctly identify the follow set, or having correctly identified it, we may accidentally omit



one of the tokens. Since there are only three possibilities for the second token of `FnApp`, we will use that as our checked case and call `parseMExp` in the default case. This approach yields the following code that you should find be a bit easier to understand.

```

switch (cT.getType()) {
case subT:  consume(Tokens.Token.subT); parseFactor(); break;
case ifT:   parseSelection(); break;
case intT:
case lpT:   parseMExp(); break;
case identT:
    {
        Token next = tins.getNextToken();
        // now put back the token since that was just a peek
        switch (next.getType()) {
        case intT:
        case identT:
        case lpT:
            parseFnApp(); break;
        default:
            parseMExp(); break;
        }
    }
    break;
default: throw new ParseException
        ("Expected subT, ifT, intT, lpT, or identT", cT);
}

```

Remember, if we get to the default case and there really is an error, it is `parseMExp` that will identify it.

### Implementing token putback

That looks a bit complex, but it is the kind of analysis and implementation necessary for certain grammar rules. There is, however, another problem that has to be solved – namely, how to implement the commented line, i.e., how to put back a token.

If you look back at the tokenizer code you will see that in the method `putBackChar`, the character is really put back on the input stream object via a method called `reset`. The important thing in the case of the input stream is that, since we are putting a character back, we put it back where it originated, i.e., the input stream object. We will follow the same principle in putting back a token: implement a `Tokenizer` method, `putBackToken`, that will take care of the putback process. The idea is to have the tokenizer retain the last value returned by `getNextToken` and then return that same value just when a putback has been performed. We can manage this by defining a boolean data member which is true exactly when a token putback has just been carried out.

Here is a sequence of activities that will lead to a correct implementation of the method `putBackToken`.

1. Return to the FP tokenizer. At the end of the method `getNextToken` you have a local variable `token` of type `Token`. You should move the declaration for that variable so that it is a data member of `Tokenizer`.

There is another data member needed in the `Tokenizer` class, namely a `boolean` data member that will have value `true` if a token was just put back and `false` otherwise. Add this data member to the class and name it `didTokenPutBack`; initialize it to `false`.

2. Now we implement the method `putBackToken`. The implementation is very simple. The method when called should simply set the flag `didTokenPutBack` to `true`.
3. Now for the fun part. The real trick is to make sure that when `getNextToken` is called that the correct token is returned. We will do this by inserting a new method between the parse methods and what is `getNextToken`. The new method simply check's the flag and acts accordingly, resetting the flag if necessary. So here it is in two steps. First, change the name of our old friend `getNextToken` to `inputNextToken`; change nothing but the name.

Then implement a new method named `getNextToken` as follows: if `didTokenPutBack` is `true` then simply return the current value of the data member `token` and reset the value of `didTokenPutBack` to `false`. If, on the other hand, the putback flag is `false`, call `inputNextToken` to input the next token from the character stream and then return that token. Notice that in the process the value of `token` changes to the new value. Also notice that the method always returns the value of the data member `token`.

The completion of these steps should lead to a correct recursive descent FP parser.

## 19.2 Static Semantic Analysis

The design strategy for the FP syntax tree is the same as that applied in the PDef syntax tree tutorial (Chapter 12) – implement one object class for each non-terminal in the grammar. But as in the PDef implementation, we will profit from a flattening of the FP grammar (see page 254) before implementing the syntax tree.

### 19.2.1 A Syntax Tree for FP

The FP grammar is context free because that is a necessary structure for implementing a parser. But context free grammars, because they are non-ambiguous, are more complex than an equivalent ambiguous grammar. Remember how the parser constructs the syntax tree: a parse method checks the grammatical correctness of a component and only then instantiates a syntax tree node to represent the component. In PDef the syntax tree structure for expressions was described in a single grammar rule:

$$\text{Exp} \implies \text{Exp opT Exp}$$

Whereas the corresponding grammar rules for parsing expressions were defined by the following rules.

```

Exp      ⇒ Exp ( addT | subT ) Term | Term
Term     ⇒ Term ( multT | divT | modT ) Factor | Factor
Factor   ⇒ intT
          ⇒ identT
          ⇒ lpT Exp rpT

```

The first, flattened, grammar rule is ambiguous but requires far fewer syntax tree nodes. We want to flatten the FP grammar in preparation for implementing the syntax tree hierarchy. Here, for convenience, is the original FP grammar.

```

Program  ⇒ Exp [ whereT ]
FnDefList ⇒ FnDef { colonT FnDef }
FnDef    ⇒ identT ParamList assignT Exp [ Where ]
Where    ⇒ whereT FnDefList endwT
ParamList ⇒ identT { identT }
Exp      ⇒ subT Factor
          ⇒ FnApp
          ⇒ Selection
          ⇒ MExp
MExp     ⇒ MExp ( addT | subT ) Term | Term
Term     ⇒ Term ( multT | divT | modT ) Factor | Factor
Factor   ⇒ intT
          ⇒ identT
          ⇒ lpT Exp rpT
Selection ⇒ ifT BExp thenT Exp elseT Exp endifT
BExp     ⇒ Factor compOpT Factor
FnApp    ⇒ identT ArgList
ArgList  ⇒ Factor { Factor }

```

The first thing to notice is that the top part of the grammar (the five rules above those for `Exp`) are not a problem – they can remain as they are.

As was the case with `PDef`, flattening the FP grammar will focus on the rules for expressions. The first thing you should notice is that the five rules starting with that for `MExp` are just the same as in `PDef`, so the same flattening should work here. But then, as well, we notice that `Exp` can generate `MExp` alone and, thus, the rule for `MExp` can become a rule for `Exp` and we can get rid of `MExp`.

Now there are secondary effects. Since `MExp` generated `Factor` in the original grammar, `Factor` in the rules for `BExp` and `ArgList` can be replaced by `Exp`. Finally, we notice that the definition of `BExp` looks a lot like the `Exp` structure – the only difference is the `compOpT` token. Since this has the same shape as `Exp`, we will drop the rule for `BExp` and understand that the `opT` in the `Exp` rule covers the comparison operations as well. Of course with the demise of `BExp`, we must replace it by `Exp` on the right side of the rule for selection.

We have arrived at the final flattened version of the FP grammar that will be the source of the FP syntax tree structure. These structures, of course, can be determined by applying the design strategies described in Section 7.3. To facilitate discussion we will call this final grammar `FP*`.

```

Program    => Exp [ Where ]
Where      => whereT FnDefList endwT
FnDefList => FnDef { scolont FnDef }
FnDef     => identT ParamList assignT Exp [ Where ]
ParamList => identT { identT }
Exp       => subtT Exp
          => FnApp
          => Selection
          => Exp opT Exp
          => intT
          => identT
Selection => ifT Exp thenT Exp elseT Exp endifT
FnApp    => identT ArgList
ArgList  => Exp { Exp }

```

There is an important point that needs to be made here. Before starting the flattening process we said we wanted to form a new grammar more suited for an FP syntax tree. In fact the grammar above, while perfectly suited for an FP syntax tree, FP\* is not equivalent to the original FP grammar – there are strings described by FP\* grammar that are not in the FP language. The use of `Exp` for selection condition means that, syntactically, the selection condition could be a simple function call or another selection, whereas the original grammar described it simply as the comparison of two factors. So, the language FP is a subset of the language of FP\*. The reason that FP\* works is because those strings described by FP\* will be flagged as errors by the parsing process, and no syntax tree will result. Only those strings described by FP will result in a syntax tree.

### ☛ Activity 69 –

At this point, implement the syntax tree structure just discussed for FP. Follow the method illustrated in the syntax tree tutorial in Chapter 12.

---

## 19.2.2 Symbol Table Design

Once the syntax tree is structured we must determine how to structure a symbol table and how to integrate it into the syntax tree – this is a good time for you to review the earlier discussion of these issues in Section 8.2.2. The structure of the symbol table can be the same as for PDef. A more interesting issue for an FP implementation is the nature of the entries in the symbol table.

Because there are two kinds of identifier declarations, both parameter names and function names, it would seem that a class hierarchy is necessary to describe the identifier attributes. A parameter would normally have a type associated with it, but since FP has only one type, integer, a parameter needs no such attribute. The only other important information about a parameter, required for later implementation, is its position in the parameter list. So a parameter will have the single attribute indicating its position (an integer value).

The fact that FP has only one type has an impact on function name attributes as well. Normally we would expect a return type attribute as well as a list of its formal parameter types; the first is unnecessary and for the second we only need to know the number of parameters. We also know

that a function must ultimately be associated with a row in the code table. For these reasons a function name will have two attributes, its row number and its number of parameters.

The bindings hierarchy displayed in Figure 19.2 can be used for symbol table entries. There is a third entry in the class `FunctionAttribute`, a data member named `backPatch`, whose purpose is to facilitate getting correctly generating function calls even though calls can occur before the function's declaration. We will discuss everything dealing with the complexity of generating correct function calls later in this chapter.

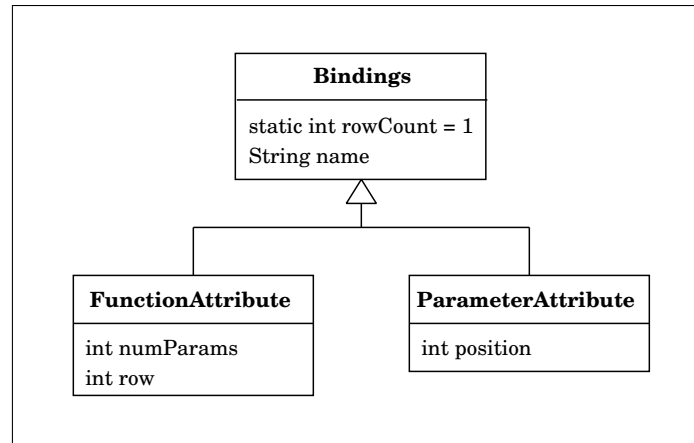


Figure 19.2: UML Class Diagram of Attribute Hierarchy

### 19.2.3 Integrating the Symbol Table into the Syntax Tree

Since we understand the block structures of FP, we should follow the strategy used in the PDef implementation and place references to our local symbol tables in those FP syntax tree classes that correspond to blocks – this means the classes `FnDefST` and `ProgramST`. Adding an appropriate data member (`localST`) to each of these two classes will integrate the symbol table to the syntax tree hierarchy.

#### ☛ Activity 70 –

Implement the symbol table and its integration into the FP syntax tree.

---

### 19.2.4 Implementing Semantic Checking

The FP language uses use-anywhere scoping on function names. From our discussions in Chapter 8, more particularly in Section 8.2.3, we know that use-anywhere scoping requires a two pass semantic checker. In the first pass the symbol table is created and filled and on the second pass the static semantic checking is carried out. In this section we will discuss these two passes.

## Filling the Symbol Table

In a survey of the FP\* grammar we note that the declaration of identifiers occurs only in function definitions and looking at the grammar we see that limits our symbol table building to elements that appear on the left side of the `assignT` token in `FnDef`. For this reason the traversal for filling the symbol table need not implicate the expression hierarchy. We can, instead focus our attention and traversal design strategy to those structures that guarantee we will traverse all function definitions. According to the grammar, that limits us to the first five rules in FP\* and our attention can be focused on traversal methods exclusively for those rules.

The one interesting situation that arises in symbol table construction is when an `FnDef` node is visited; we seem to be entering a nested block – that which begins with the function’s parameter list. But the name of the function, which is also in the `FnDef` node, belongs to the surrounding block. When the method `genSymbolTable` is called on the `FnDef` node, it will be necessary to pass the local symbol table of the surrounding block as a parameter so that (i) it can be set as the parent link of the local symbol table to be constructed and (ii) so that the function name and its attributes can be added to it (and not to the new local symbol table).

### ☛ Activity 71 –

Implement the syntax tree traversal that builds and fills the symbol table. Use the same approach as followed for building the symbol table for `PDef` – remember we split the symbol table generation and semantic checking, even though they occurred in one pass. Remember the trick of implementing two versions of `generateSymbolTable`, one with no arguments (for calling from the FP class) and one with the symbol table as argument for traversal.

---

## Checking Semantics

While all symbol table creation involves data contained in the `Program` and various `FnDef` syntax tree nodes, static semantic checking focuses exclusively on uses of identifiers, and these occur uniquely within the expression nodes.<sup>1</sup> Since the expressions are represented by data members of the `Program` and `FnDef` nodes, the `checkSemantics` traversal must be implemented on all syntax tree nodes – the expression nodes so checking of uses can be found and the `Program` and `FnDef` nodes so the checking can be forced down the syntax tree.

The semantic checking for FP does not have the same concerns as that for `PDef`. Since FP has a single data type expression types are already known, we don’t have to worry about determining a type to return from `checkSemantics`, as is necessary in `PDef`. On the other hand, where in `PDef` there is only one kind of identifier, in FP we have two. Thus, in FP semantic checking it is necessary to determine for each use of an identifier if the use (as a variable or function name) corresponds to a binding stored in the appropriate symbol table.

---

<sup>1</sup>Notice that, since FP has just a single type, the literal values in an FP program won’t have to be involved in semantic checking.

## Semantic exceptions

A final word related to exceptions for semantic checking. The hierarchy we used in the PDef implementation is relevant here but needs slight changes to be adapted to the FP context. The following semantic exception categories would seem to capture the static semantic errors that can be uncovered. For each category we include, in parentheses, the value(s) that distinguish a particular instance.

- not declared (identifier name)
- already declared (identifier name)
- function name used as parameter (identifier name)
- parameter name used as function (identifier name)
- parameter/argument mismatch (identifier name, # arguments present, # arguments expected)

While the first two are problems common to FP and PDef (and any statically typed block structured language), the last three are specific to FP.

### ☛ Activity 72 –

Implement the semantic checking traversal of the syntax tree. Be sure to generate the appropriate semantic exceptions.

---

## 19.3 Synthesis

Having defined the dynamic semantics of FP (see Section 18.3.2 above), the next step is to tackle something completely new, the synthesis phase – i.e., implement a code generator. The code generator will traverse the syntax tree (with integrated symbol table) and generate code for the FP-machine – when the FP-machine executes the generated code it will produce the result described by the FP semantics. The purpose of this section is to discuss the design and implementation of the code generator, focusing first on the design of the code table and then on the code generator itself.

### 19.3.1 Code Table Design

There are two components to define when implementing the code table: the code table itself, which is obvious, but also a structure for holding the operations that make up the translated code. The class `OpCode` will contain an enumerated type defining names for each of the machine operations. An `OpCode` object will have an instance variable to hold the particular opcode but also will have a second integer instance variable to hold a possible argument. The structure of the class `OpCode` is specified in Figure 19.3.

The design of the code table must satisfy the needs of code generation and the execution cycle of the FP-machine. Here are some important considerations for the design.

1. The purpose of the code table is to hold the sequences of FP-machine instructions. These instructions will be implemented as objects of the class `OpCode`. The class must hold a code for the machine instruction and must also accommodate an optional operand. This class is a simple abstraction class and will have accessor methods for each data member. So that we can conveniently refer to machine instructions by name, the class `OpCode` will also implement an enumerated type for these names.
2. The primary data member of the code table class will be the data structure that stores the `OpCode` references; it will be implemented as a two-dimensional array of `OpCode`. In addition, to make code generation easier, the code table class will have two additional data members for specifying the row and position where the next `OpCode` reference should be placed; we will refer to these as `currentRow` and `currentPosition` respectively. There should also be a debugging framework present in this class.
3. There will be several methods in the code table class oriented toward the filling of the code table.
  - (a) Before code generation can begin for a particular FP function definition, the code table must be told the row number into which code should be generated. For a particular function the row number is found in its symbol table entry. There will be a code table method, `setCurrentRow`, whose action is to set the value of `currentRow` to the value of the method's integer parameter; the method will also initialize `currentPosition` to zero.
  - (b) It will be convenient to implement three flavors of the method `genCode`, each of which will instantiate an `OpCode` object and place it at the current code table position and then increment the value of `currentPosition`. One flavor of `genCode` will take a single `OpCode` reference as parameter and will be called when a machine instruction with no argument is to be generated. A second flavor of `genCode` will take two parameters, an `OpCode` reference and an integer, and will be called only when the push instruction is to be generated. The third flavor of `genCode` will take a `String` parameter that contains the character representation for of an operation, either arithmetic or comparison. The character representation is obtained from a `Token` object representing an arithmetic or comparison operation.
  - (c) There are two other special methods defined in `CodeTable` that are used only when generating code for the selection expression. We repeat here the definition of `trans` for the selection expression as given on page 322.

```
trans(if b then t else f endif) =
    trans(b), push x, jmp, trans(t), push 0, push y, jmp, trans(f)
```

In this definition the values for `x` and `y` are not known until after the push instructions are generated. This means there must be a means for returning to a push instruction and changing its parameter. The method `getCodeCol` is used to get the offset of a push instruction whose parameter will be changed later. The other method is `setPushParam`; it takes two integer parameters, the first being the offset of a push instruction and the second being the new value for that instruction's parameter.

4. There will be a single method in the code table class oriented towards the FP-machine implementation. In particular, this method will be called `fetch` and return the `OpCode` reference,



which is located at the position in the code table specified by its two arguments. This method obviously implements the “fetch” step of the “fetch/execute” cycle for the FP-machine.

5. The final method we need to mention is the ever-present `toString`. This should be redefined to produce a listing of the code table. This method is essential for successfully testing and debugging the code generation implementation. There is also some utility in providing a mechanism for displaying just one row in the symbol table.

The relationships among the code table, opcode, and FP-machine classes are depicted in the UML class diagram in Figure 19.3.

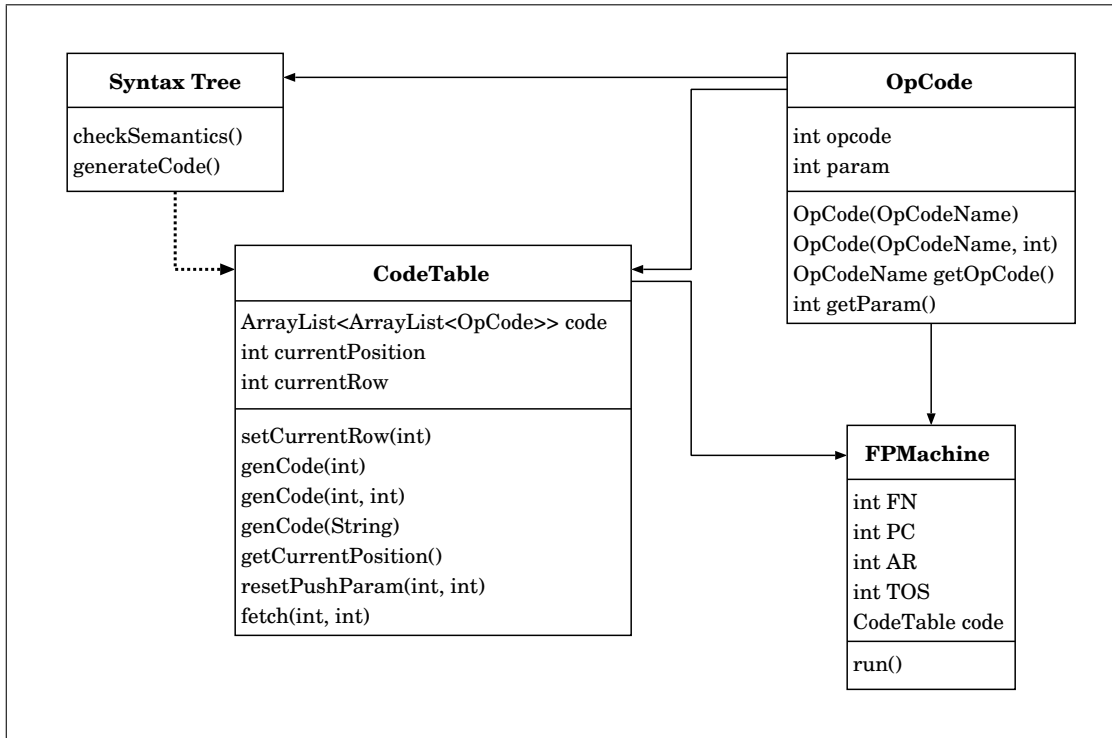


Figure 19.3: UML Class Diagram for the CodeTable, OpCode and Machine Classes

### 19.3.2 Code Generation

Code generation is accomplished via a traversal of the syntax tree, which means that, just as in the case of printing the syntax tree and performing semantic checking, we have to add a new group of methods to the syntax tree structure that will traverse a syntax tree and at the same time generate FP-machine code to implement the FP dynamic semantics. The new methods will be called `generateCode`. Having gone through this procedure several times earlier, notably for printing the syntax tree and for checking semantics, the bulk of this work is left as an exercise. However, since the code generation is new, we will go through three of the cases here to provide some guidance and models to use for the other cases.

First, so that we have all the necessary information at hand, here is the structure of the `OpCode` class. By the class structure we mean all instance variables and methods of the class while omitting

the code bodies for the methods.

```
public class OpCode {
    public enum OpCodeName {PUSH_I, LOAD_I, JMP_I, CALL_I, RETURN_I, HALT_I,
                            LT_I, LE_I, GT_I, GE_I, EQ_I, NE_I,
                            ADD_I, SUB_I, MUL_I, DIV_I, MOD_I};

    OpCodeName opcode;
    int        param;

    public OpCode(OpCodeName opcode) {...}
    public OpCode(OpCodeName opcode, int param) {...}
    public OpCodeName getOpCode() {...}
    public int getParam() {...}
    public String toString() {...}
}
```

Before starting the three example cases there is an important point to mention. As the symbol table was central to the functioning of the semantic checker, the symbol table is also central to code generation. In particular, the symbol table facilitates checking the attributes of identifiers, both variable and function, during code generation. As was also the case with the semantic checker in PDef, it will be necessary in some cases to have the symbol table as a parameter to the method `generateCode`. We will see this strategy reflected in the examples below.

Remember that the process requires that we look at each class in the syntax tree structure and determine what action is required to generate appropriate code and to continue the traversal. We will focus on three syntax tree classes, `FnDefST`, `ExpEXP` and `FnAppEXP`. For reference, here are the relevant parts of the definition of the function `trans`, which you will remember, defines the dynamic semantics of FP (see pages 320-322).

### Builtin Function Application

$$\text{trans}(t1 \text{ op } t2) = \text{trans}(t1), \text{trans}(t2), \text{OP}$$

where `op` is one of the builtin functions

$$+, *, -, /, \%, <, <=, >, >=, ==, /=$$

and `OP` is the corresponding `OpCode` object for the FP-machine operation `op`.

For example, if `op` were `+`, then `OP` would be the object

$$\text{new OpCode(OpCode.OpCodeName.ADD_I)}.$$

### FnAppEXP

$$\begin{aligned} \text{trans}(f \ t1 \ \dots \ tn) = \\ \text{PUSH\_I } 0, \text{ PUSH\_I } 0, \text{ PUSH\_I } 0, \text{ trans}(t1), \dots, \\ \text{trans}(tn), \text{ PUSH\_I } n, \text{ PUSH\_I } fn, \text{ CALL\_I} \end{aligned}$$

where `n` is the number of parameters for the function `f` and `fn` is the code table row number assigned to the function `f`.

### Function Definition

$$\text{trans}(f \ p1 \ \dots \ pn = e) = \text{trans}(e), \text{RETURN\_I}$$

where the function `f` has the associated expression `e`. Notice here that there is no reference to a possible nested function definition list!

We now list the three classes and describe the appropriate actions. In each of the three cases we will review the state of the particular syntax tree class along with the signature of the class's `generateCode` method.

`class ExpEXP`

The class `ExpEXP` represents an expression that applies a built-in arithmetic operation to two arguments. The promised state and signature for `ExpEXP` are as follows.

```
ExpressionST expL;
ExpressionST expR;
Token        op;
void generateCode(CodeTable ct, SymbolTable st);
```

The definition of `trans` that corresponds to `ExpEXP` is that for the built-in operations. Accordingly the left then the right expression references must be translated and then the specific instruction is generated corresponding to `op`. This simply implements the computational sequence for a stack machine – put the operands on the stack and then apply the operation. So the sequence of code generation operations should be as follows.

```
expL.generateCode(ct, st)
expR.generateCode(ct, st)
ct.genCode(op.getName())
```

Here the first two lines continue the depth-first traversal of the syntax tree – no specific code is generated here. In the third line, however, an opcode corresponding to `op` is generated. The call to `genCode` in this case takes a `String` parameter which will contain the name of the operation obtained from `op`.

`class FnAppEXP`

This case is actually quite similar in spirit to the previous one: translate each of the arguments and then generate the call to the function. The definition of `trans` for function application, however, is actually a bit more involved than that. The state of `FnAppEXP` and its `generateCode` signature are as follows.

```
Token name;
LinkedList<ExpressionST> argList;
void generateCode(CodeTable ct, SymbolTable st);
```

Code is generated in three phases. First, code is generated to create space on the stack to store return information for the call, then code is generated for each of the arguments, and finally code is generated for the call. There are two pieces of data that must be determined in order to complete the code generation: the number of arguments and the row number of the function. The first is just the length of the list `argList`; the second is accessible from the symbol table entry for the function name via a method named `getRowNumber`. The code generation is as follows.

```

FunctionBinding fnAttr = (FunctionBinding)(localST.findBinding(fname));
row = fnAttr.getRowNumber();
numParams = argList.length();

// push three 0's for space to save registers
for (int i = 0; i < 3; i++) ct.genCode(OpCode.OpCodeName.PUSH_I, 0);

call generateCode(ct, st)
    on each element of argList

ct.genCode(OpCode.OpCodeName.PUSH_I, numParams);
ct.genCode(OpCode.OpCodeName.PUSH_I, row);
ct.genCode(OpCode.OpCodeName.CALL_I);

```

Similarly to the previous case, it is the calls to `generateCode` on the elements of `argList` that guarantee that the syntax tree traversal continues.

#### class FnDefST

It is good to remember exactly what we are generating code for in this case: it is the code associated with the function itself, i.e., the code to be executed during a call to the function. When this code is accessed, the first part will be the code for the specified expression and that will be followed by the code that does the return from the function call. The definition of `trans` in this case clearly lays this out: translate the expression and then do the return! Of course we must also be concerned with continuing the traversal, and this comes in calling `generateCode` on each element in the (optional) function definition list. The state and `generateCode` method for `FnDef` are as follows.

```

Token                name;
LinkedList<identT>   paramList;
ExpressionST         exp;
LinkedList<FnDef>    defList;
SymbolTable          localST;
void                 generateCode(CodeTable ct);

```

Unlike the earlier cases, the `generateCode` method for `FnDef` requires no `SymbolTable` parameter; that is because the table it needs is part of its state – `localST`. One subtle point must be taken into account. When the code is generated for `exp` we must make sure that it is generated into the correct row of the symbol table. So we must first determine the row number corresponding to the name `name` and then tell the code table to set that particular row number. It is critical that this be done immediately before calling `generateCode` on `exp`. The code generation proceeds as follows.

```

FunctionBinding fnAttr = (FunctionBinding)(localST.findBinding(fname));
row = fnAttr.getRowNumber();

ct.setRowNumber(row);
exp.generateCode(ct, localST);
ct.genCode(OpCode.OpCodeName.RETURN_I);

```

```

    call generateCode(ct)
      on each element of defList;

```

### SelectionST hint

But a comment is really appropriate for the `generateCode` method for `SelectionST`. The reason is that while generating code you have to generate certain `PUSH_I` instructions at a point where you don't yet know the parameter for the instruction. It is always the case that these unknown values are offsets in the code for `JMP_I` instructions. At certain points in the code generation, then, you must be able to obtain the current offset value and then go back to a previously generated `PUSH_I` instruction and patch in the correct offset. The semantics of the `SelectionST` function is given by the `trans` function, so the offsets of interest in the code generation process are all indicated. A good thing to do is to write out the appropriate FP-machine code for some simple selection function and then see how it can be generated as a sequential activity.

### ☛ Activity 73 –

Design and implement the `generateCode` methods in the other classes in the syntax tree. Combine all the implementations to produce the complete code generator.

---

## 19.4 Generating Linear Code

There are two points to be made about the FP code generation we have just discussed. First, the strategy of using a two-dimensional array for the code table was very convenient because it facilitated the generation of the function code and, more importantly, the function call code. Second, the strategy was not just a convenient trick. Many modern pure functional programming languages, such as Haskell, generate code for functions in this way. The nature of the execution environment and the nature of the generated code makes this structure advantageous.

But FP is a very simple functional language, and as much like Haskell as it is like the language C. If we wanted to generate code for FP as we would for C, that is for a machine with a standard linear memory structure, then the two-dimensional array for code generation isn't appropriate. In this section we will look at the interesting changes that must be made to the FP code generator if we want to generate linear code.

The machine we will use is very similar to the FP-machine, but instead of a program memory that is two-dimensional it will be defined with a linear memory space. We will refer to this new machine as the  $FP^+$  machine. The modifications necessary to the FP-machine derive from our desire to generate linear code. Generating code for a linear address space means that we will no longer have need for the FN (function row number) register. This one change ripples through a small set of the FP-machine's instruction set.

We will assume the following for program execution: A loaded program will have its first instruction at position zero in the code table – this instruction is the first instruction generated for



```

g (f 2 3)
where
  f a b = a + (g b);
  g a   = (m a) * (m a)
        where
          m x = x + x
        endw
endw.

```

Figure 19.5: A Simple FP Program

**FnApp**

```

trans(f t1 ... tn) =
  push 0, push 0, trans(t1), ..., trans(tn), push n, push fn, call

```

where *fn* is the offset in the linear code table of the first instruction of *f*.  
The two ‘push 0’ instructions are to make space on the stack for storing the two register return values.

**A Translation Example**

Consider the simple program displayed in Figure 19.5. According to the semantic definition above we should see, at a high level, the following structure for the generated code.

```

trans( g (f 2 3) ), halt, trans ( f a b = a + (g b) ),
trans( g a = (m a) * (m a) ), trans( m x = x + x )

```

Before continuing there is a particular situation that occurs in this translation that wouldn’t have occurred in the translation of FP. Notice that in this code there are two calls to the function *g*. The problem is that the starting offset for *g* will not be known when the code is generated for the first call. So in generating the code for function calls there must be a technique for insuring that the correct offset is associated with each call – the technique is referred to as *back-patching* and will be introduced shortly.

The code generated for the program above is displayed in the table in Figure 19.6. Each column of the table holds the code generated for either the main expression (left-most column) or for one of the three function definitions, as implied in the translation layout above. Also, each statement is matched with its code table offset. One thing to take note of here is that the addresses of the functions’ code are *f* - 13, *g* - 24, and *m* - 40.

**19.4.2 FP Code Generation**

Based on the FP semantics defined for the FP<sup>+</sup>-machine, we can enhance the old FP interpreter, whose construction is described in Section 19.3.2, so that it will generate appropriate FP<sup>+</sup> machine code. We will discuss this enhancement in several phases.

- Discuss the basic problem of back-patching, i.e., generating correct addresses for function calls.

main	f	g	m
0 - push 0	13 - push 1	24 - push 1	40 - push 1
1 - push 0	14 - load	25 - load	41 - load
2 - push 0	15 - push 0	26 - push 0	42 - push 1
3 - push 0	16 - push 0	27 - push 0	43 - load
4 - push 2	17 - push 2	28 - push 1	44 - add
5 - push 3	18 - load	29 - push 40	45 - return
6 - push 2	19 - push 1	30 - call	
7 - push 13	20 - push 24	31 - push 1	
8 - call	21 - call	32 - load	
9 - push 1	22 - add	33 - push 0	
10 - push 24	23 - return	34 - push 0	
11 - call		35 - push 1	
12 - halt		36 - push 40	
		37 - call	
		38 - mult	
		39 - return	

Figure 19.6: Generated Code for the Program in Figure 19.5

- Describe modifications to the class `SymbolTable` (see Section 19.2.2 for the original definition).
- Describe modifications to the class `CodeTable` (see Section 19.3.1 for the original definition).
- Describe the implementation of the method `generateCode` in the FP `SyntaxTree` hierarchy.

Before we start, notice that the FP<sup>+</sup> machine has the same operations as the original FP-machine. Thus, we don't have to make any modifications to the class `OpCode`.

## Generating Linear Code

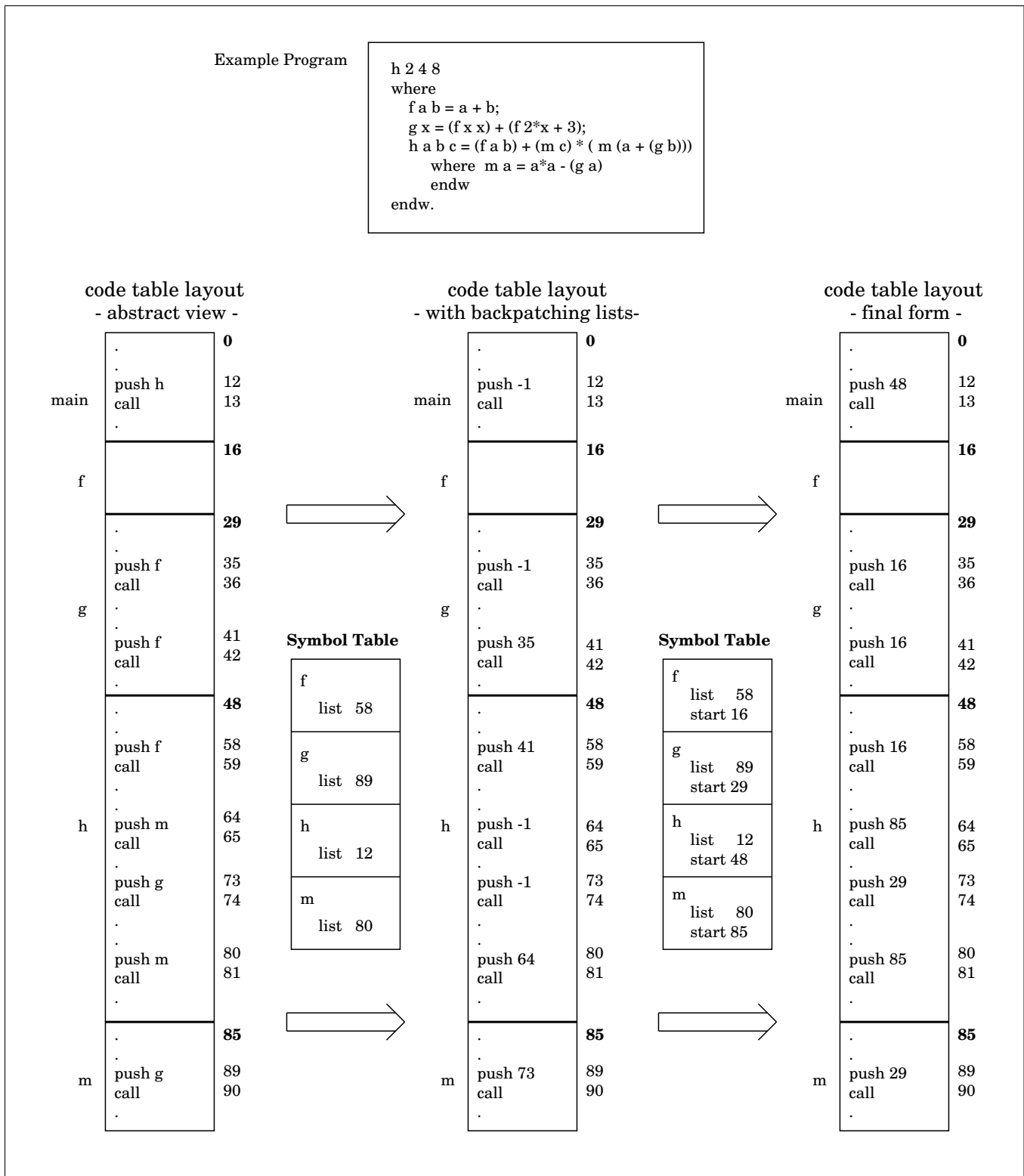
The code generation process for the FP<sup>+</sup> machine has many strategies in common with that for FP. This shouldn't be surprising since the two languages are almost identical and since the target machine language (i.e., the stack machine's) are very similar. The biggest problem for the FP<sup>+</sup> code generation is the fact that the code table is linear.

Of course in the FP-machine each row is linear, so we might guess that those classes, for which `generateCode` for FP simply generates linear code, will remain unchanged in the FP<sup>+</sup> implementation. This guess is true and the implementations for `generateCode` in `Exp`, `NegExp`, `Number`, `Identifier`, and `Selection` will remain unchanged. The classes `FnDef`, `FnDefExp`, and `FnApp`, will have to be considered in light of the linear code table. In addition, of course, we must consider the implementation of `generateCode` in the new version of `Program`, which now contains a reference to the main expression in addition to the list of function definitions.

## Function Calls and Function Definitions – Back-patching

With a linear code table the processing of function calls and definitions becomes more complex. This new complexity is due to the fact that we can't determine the address for a function before





its definition has been processed. There are two aspects to this problem. First, in generating code, when we come across a function call we cannot be guaranteed to have the address for the call. This means that for a function  $f$  we must keep track of the code table locations for all of its calls.

The strategy, called *back-patching*, is illustrated in the code table representations in Figure 19.7. In the left-most code table we see an abbreviated listing of the code generated for the FP example program, in the box at the top. In the listing, the parameter of a function call's push instruction is abstracted simply as the name of the function, knowing that eventually it must be replaced by the function's starting address. The middle code table, which is accompanied by abbreviated symbol table entries for the functions, shows how we use the push parameters to keep track of the locations of the calls to each function. Thus, for each function we have a *back-patch list* whose first position is in the symbol table, and each successive location is the parameter to a push statement. The code table on the right, also accompanied by symbol table entries, shows the final form of the code table after back-patching, with the address for each function finally placed in the push statement preceding its call.

### Symbol Table Changes

The changes to the symbol table entry structures is easiest shown via a diagram, a slight modification to that in Figure 19.2 – the new structure is displayed in Figure 19.8. In the diagram we see that in the `FunctionAttribute` class the location data member has been renamed `address`. There is also a new data member in `Bindings` named `backPatch`. It is there to facilitate the back-patching algorithm.

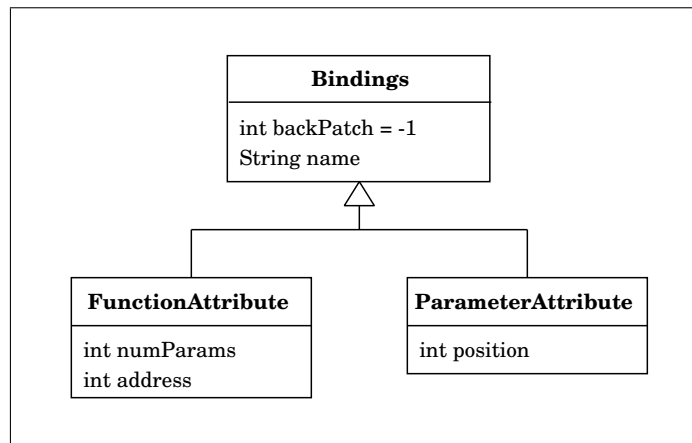


Figure 19.8: UML Class Diagram of Attribute Hierarchy

There must also be a mechanism for retrieving the value of the head of a function's back-patch list for use in the back-patch algorithm. This access will be provided by new accessor and setter methods for this instance variable: we will assume the method names `getBPList` and `setBPList`.

### Code Table Changes

The fact that the code table is two-dimensional in the FP implementation has a big impact on the structure of the class `CodeTable`. There are five entries from that class that need to be eliminated for

the current project: namely the declarations for the data members and methods `maxFunctions`, `fnPsn`, `setRowNumber`, `getCodeRow`, `getCodeCol`. After those deletions there is one addition. We need a method, analogous to the deleted `getCodeCol`, to return the offset (in memory) where the next opcode will be generated – the method is called `getCodePosition`. Here is the new code table definition.

```
public class CodeTable {
    private int maxInstructions;
    private Debug debug;

    private OpCode[] codeTable;
    private int nextPsn; // position in codeTable of next available position

    public CodeTable(boolean db)

    public void genCode(OpCode.OpName op)
    public void genCode(OpCode.OpName op, int param)
    public void genCode(String name)
    public void setPushParam(int psn, int value)
    public int getCodePosition()
    public OpCode fetch(int pc)
    public String toString()
}
```

#### ☛ Activity 74 –

Implement the class `CodeTable` for FP<sup>+</sup>, as described above. The method `toString` will be important in testing the code table later.

---

In following the earlier description of the back-patching strategy we can see a need for the following operations in `CodeTable`.

- set the value of an existing push instruction's parameter
- retrieve the value of an existing push instruction's parameter
- run the back-patch algorithm

The first of these operations is already provided by the code table method `setPushParam`. The second of these can be provided by an accessor method taking a code table index as parameter. The following provides the functionality.

```
public int getPushParam(int psn) {
    return codeTable[psn].getParam();
}
```

The third operation also requires a new method, which will be called separately for each function. The idea will be to traverse the back-patch list and substitute the function address for each push parameter on the list. Of course care must be taken to recover the existing push parameter, before changing it, since it is the index of the next entry on the back-patch list. This method will take two parameters, the address of the function in the code table and the index where the back-patch list begins. Here is the structure.

```
public int backPatch(int address, int bpList) {
    if (bpList == -1) return;

    // bpList != -1
    while (bpList != -1) {
        // set the parameter for codeTable[bpList] to address
        // recover the push parameter at codeTable[bpList]
    }
}
```

### Activity 75 –

Complete the modifications to `FunctionEntry` and to `CodeTable` as described above.

---

### FnApp

The method `FnApp.generateCode` will be responsible for generating the call to a function in a function application. The generation of the code for the arguments to the function will be unchanged. But the other details of the code generation will change. First, we need only allocate two positions on the stack for return information, as opposed to the three positions for the interpreter implementation.

The tricky part of this method is generating the push instruction containing the address for the call. As we've seen above, this address must be the index of another call to this function. What we want to do is to "push" this new push instruction onto the back-patch list. That means that this current one will become the head of the list. So the current head, whose index is stored in the symbol table entry for the function, must be used as the push parameter for the current instruction. Then, the index of the current push instruction must be stored in the symbol table entry as the new index of the head of the list. Here is the structure of this method.

```
public void generateCode(SymbolTable st, CodeTable ct) {
    debug.show("--->>> FnApp::genCodeTable()");

    FunctionEntry fe = (FunctionEntry)(st.findEntry(name));
    int head = fe.getBPList();

    // generate space to save return info
    ct.genCode(OpCodes.OpName.Mpush, 0);
}
```

```

    ct.genCode(OpCodes.OpName.Mpush, 0);

    // generate code for arguments
    for (Expression exp : argList)
        exp.next().generateCode(st, ct);

    // generate call
    // ...

    debug.show("<<<--- FnApp::genCodeTable()");
}

```

### Activity 76 –

Complete the implementation of the method `FnApp.generateCode`.

---

#### FnDef

The only real change from the FP implementation of `FnDef.generateCode` (see page 344) is that now we must determine the starting address for the function (in the linear code table) and record that address in the function name's entry in the symbol table.

When we look at the corresponding FP method, we see that the code is generated first for each of the nested function definitions and then for the current function definition's expression. We described this process above as generating the code for the expression first and then for the nested function definitions. In fact, the order doesn't really matter. But since we generate the main expression's code before generating code for the first level function definitions, it makes sense to maintain that same pattern here. So the code generation for this object should have the following structure.

```

public void generateCode(SymbolTable st, CodeTable ct) {
    debug.show("--->>> FnDef::generateCode()");

    FunctionEntry fnAttr = (FunctionEntry)(st.findEntry(name));
    int row = ct.getCodePosition();
        // this is the address of the next code table position
        // to be used by 'genCode' -- i.e., it is the starting
        // address for this function
    fn.setAddress(row);
        // save this address in the function's symbol table entry

    exp.generateCode(localST, ct);
    ct.genCode(OpCodes.OpName.Mreturn);

    for (FnDef entry : defList)

```

```

        entry.generateCode(localST, ct);

        debug.show("<<<--- FnDef::generateCode()");
    }

```

But we aren't quite finished with this method. There is one more operation to be carried out. When we return from `FnDef.generateCode` we will not enter another context where the nested functions defined in the where-clause can be referenced. This means that this would be a good time to back-patch the calls to these functions. This simply requires that we traverse the local symbol table and back-patch each function entry we encounter.

### ☛ Activity 77 –

Explain why it is unwise to wait until translation is finished to do the back-patching.

---

To accomplish this we will implement a new method in `CodeTable` which will take the local symbol table as parameter and then run through it back-patching the function entries encountered. That method will have the following structure.

```

public void backPatchFunctionEntries(SymbolTable symTable) {
    for (SymbolTableEntry entry : symTable) {
        if (entry.getClass().getName().equals("symbolTable.FunctionEntry")) {
            int address = (FunctionEntry) entry.getAddress();
            int bpList = (FunctionEntry) entry.getBPList();
            backPatch(address, bpList);
        }
    }
}

```

We make use of this method by calling it at the end of `FnDef.generateCode`. The call is inserted between the for-loop and the final debug statement.

It is important to notice that in the implementation of `backPatchFunctionEntries` above the for loop uses the object `symTable` as though its class, `SymbolTable`, implements the `Iterable` interface. Because the table of symbols within `SymbolTable` is implemented as a `LinkedList`, we can easily modify the definition of `SymbolTable` so that it implements the `Iterable` interface. To do so we must modify the class definition line and also implement the method specified for the interface. Here are the necessary alterations.

```

public class SymbolTable implements Iterable<SymbolTableEntry> {
    ....
    public Iterator<SymbolTableEntry> iterator() {
        return symbols.listIterator();
    }
}

```

### Program

This class represents a complete program and its `generateCode` must specify the order in which code is generate for its components. Fortunately the code is exactly the same as for the FP-machine implementation.

### ☛ Activity 78 –

Complete the implementation of the FP code generation for the FP<sup>+</sup>-machine. Test it the same way you tested the FP-machine implementation.

---





# Chapter 20

## Intermediate Code Generation

The experience with code generation for the FP language in Chapter 19 might lead us to believe that code generation in general is not a difficult problem. The advantage of having generated code for the FP and FP<sup>+</sup> machines is that the machines are very simple – they lack features of a physical processor that make code generation difficult. But because they are not like physical processors we didn't have to worry about the effective use of processor resources – i.e., data storage and execution time (see the Efficiency and Portability Principles in Section 1.2.1).

In this chapter we look at both of these principles. For the portability principle we introduce a new target machine architecture – the *three-address code machine*. This new translator architecture will allow us to generate code in two phases. This structure was discussed in Section 1.5.3 and illustrated there in Figure 1.9 – the diagram is reproduced here in Figure 20.1 for convenience.

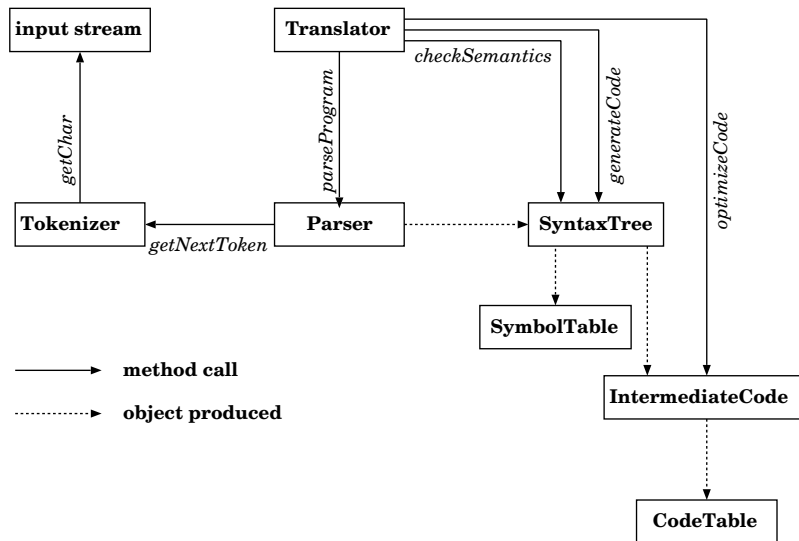


Figure 20.1: How the Translator Works

One way to read this diagram is, the translation is accomplished in a sequence of conversions: the input program code is first parsed and converted to a syntax tree form (equivalent to the input program code), the syntax tree is converted to an intermediate code program (equivalent to the syntax tree), which is finally converted to an equivalent final code table, containing the object

code to be executed on the target processor. The three-address architecture is the target of the intermediate code generation and acts as a boundary between the extended front-end (front-end plus intermediate code generation) and the object code generation. The extended front-end can be the same for a range of different target processors, while the object code generation must be re-implemented for each different processor. This is the essence of portability.

On the efficiency side, since the three-address code architecture has features common to the range of target architectures, optimizations to the generated three-address code can be made focusing on the characteristics of the hardware features common to all the processors. For example, the three-address code architecture has a set of general purpose registers and data can be manipulated only if it already resides in those registers. The registers behave as a cache between the processor and main memory, and it is this register cache that is the critical resource of the processor. The efficient management of the register cache is a central focus of optimizing code generation. Besides managing the cache, we will also find a useful cache connection to another optimization strategy – the identification of common sub-expressions whose repeated execution represents a waste of processor execution time. Think about computing even the simple expression  $(a + b) * (a + b)$ . Sub-expression removal will allow us to compute this by saving and reusing the first evaluation of  $(a + b)$  in place of the second.

In the rest of this chapter we will focus on basic issues of efficiency, having established above the simple strategy that splits code generation into two parts – the first hardware independent and the second hardware dependent. Our discussion of efficiency will be done presented in two steps. First we will focus on the design of a simple three-address code generator for the FP translator we have already implemented. In writing this code generator we will ignore register cache management in favor of a very simple but perhaps unexpected register-use strategy. This simple approach will facilitate the design of our new FP translator. With the three-address code generation designed, we will focus the second step on a strategy for generating three-address code that will eliminate common sub-expressions. This elimination takes place as code is being generated. We will first focus on the strategy and then discuss how to incorporate it into the three-address code generator.

## 20.1 Three-address Code

As an intermediate code architecture, three-address code must abstract processor structures while retaining the basic computational environment. In this spirit, the fundamental form of three-address code derives a simple assignment statement.

$$x = y + z$$

We have two variable values operated on and stored in a third variable – at the processor level this could be operate on two registers and store the result in a third register. To see how this form can be used in translation, consider the following FP function definition.

$$f\ a\ b\ c = (a - b) * ( b + c * (a - b) )$$

The expression on the right is easy to understand as is – i.e., from a programmer’s point of view. But if we want to describe this computation assuming we can never write down more than a three-address form then we have to rethink the expressions representation. For example, the first operation in computing the expression value is to compute  $(a - b)$  and then to store it somewhere.

The strategy will be to create temporary variables as we go along to store intermediate results. Using this strategy we can write down a new description of our expression as follows.

```
t1 = a - b
t2 = a - b
t3 = c * t2
t4 = b + t3
t5 = t1 * t4
```

One interesting thing to notice here is that this form matches the structure of the syntax tree for the expression, with each assignment representing an **Exp** node – in fact, the temporary variable on the left is a reference to the **Exp** object for the right side. In this way the three-address form is a linearized version of the syntax tree. So this three-address form should be convenient for code generation.

On the processor side we can wonder what we should make of the temporary variables. The variables **a**, **b**, and **c** are parameters and really just formal names or place holders for argument values computed before the call – so we would think of them as values in the stack frame for the call to **f**. The temporary variables could be thought of as registers – but values are often moved from registers back to a stack frame, so thinking of temporaries as registers is a bit restrictive. It is better to be non-committal and allow the possibility to think of them as registers or values stored on the stack frame for a call to **f**.

The three-address form, which is so natural for describing expression evaluation, can also be extended to facilitate describing general language structures, such as function call and branch operations. The table in Figure 20.2 shows the different forms for three-address codes. With each code there are four components: a name, two arguments, and a result. As is evident in the table, there are some codes for which all addresses are not used – an unused address is marked by a dash. Three of the entries (**Jump**, **Param**, **Label**) are interesting because they each have one argument but no result – even though these codes have no result, we will always put their single argument in the result position.

A program in the three-address code architecture is a sequence of three-address codes. There is an unbounded set of temporary variables that can be allocated as needed when generating three-address code. “Unbounded” may seem overdone, but again the assumption means we can avoid (until necessary) the constraints of real physical limitations.

One fundamental assumption of the three-address code architecture is that if you want to manipulate a value it must be in a temporary variable – it is this characteristic that separates the temporary variables from, for examples, the parameters in FP function definitions. Here is the sequence of three-address codes for our target expression.

$$(a - b) * (b + c * (a - b))$$

The codes should look familiar; the integer at the beginning of each line, not surprisingly, is the offset of the code in the sequence of generated code.

```
0 Copy(t1,a,-)
1 Copy(t2,b,-)
3 Sub(t3,t1,t2)
```

Name	Result	Argument 1	Argument 2	Meaning
Op (binary operation)	x	y	z	$x = y \text{ Op } z$
Neg (unary operation)	x	y	-	$x = -y$
Copy	x	y	-	$x = y$
Jump	x	-	-	goto x
JumpF	x	y	-	if !x goto y
Param	x	-	-	x is a parameter
FCall	x	y	z	$x = \text{Call } y, \text{ with } z \text{ parameters}$
FReturn	x	-	-	return x
Label	x	-	-	mark place in code
Halt	x	-	-	halt program returning x

Figure 20.2: Three-address Codes

```

4 Sub(t4,t1,t2)
5 Copy(t5,c,-)
6 Mul(t6,5,t2)
7 Add(t7,t2,t6)
8 Mul(t8,t1,t7)

```

What would be the three-address code for the function definition discussed above?

$$f \ a \ b \ c = (a - b) * (b + c * (a - b) )$$

Making use the FReturn and Label codes, we get the following sequence.

```

0 Label(f,-,-)
1 Copy(t1,a,-)
2 Copy(t2,b,-)
3 Sub(t3,t1,t2)
4 Sub(t4,t1,t2)
5 Copy(t5,c,-)
6 Mul(t6,5,t2)
7 Add(t7,t2,t6)
8 Mul(t8,t1,t7)
9 FReturn(t8,-,-)

```

Here we see how the return value is designated by the return from a call to `f`. How about the call `f 2 3 4`; how should that be encoded? We would need to generate the parameters and a call (we'll assume that the name `f` is a label for the first address in the code sequence for `f`). It is important to realize here that the first argument of each `Param` code is a data value. Before we can do anything with it it must be put in a temporary variable via the `Copy` command. We then use the command `Param` to process the temporary variable – maybe it would push the value on the computation stack or copy it to a special register just for function parameters.

```
Copy(t1,2,-)
```

```

Param(t1,-,-)
Copy(t2,3,-)
Param(t2,-,-)
Copy(t3,4,-)
Param(t3,-,-)
FCall(t4,f,3) // \texttt{f} applied to 3 parameters, find return value at \texttt{t4}

```

### An FP Program Example

To more completely illustrate the application of the three-address strategy we will give the equivalent three-address code sequence for the following program. We will do this in a similar fashion as for the FP<sup>+</sup> translation given in Figure 19.6. We will work with the following FP program.

```

f 2 3 4
where
  h a b = if (a < b) then a else b endif;
  f a b c = (g a) + (h b c)
  where
    g a = a * (a + a)
  endw
endw.

```

The conversion of this program into three-address code is presented in Figure 20.3.

main	h	f	g
0 - Copy(t0,2,-)	8 - Label(h,-,-)	19 - Label(f,-,-)	30 - Label(g,-,-)
1 - Param(t0,-,-)	9 - Copy(t4,a,-)	20 - Copy(t8,a,-)	31 - Copy(t14,a,-)
2 - Copy(t1,3,-)	10 - Copy(t5,b,-)	21 - Param(t8,-,-)	32 - Add(t15,t14,t14)
3 - Param(t1,-,-)	11 - Ge(t6,t4,t5)	22 - FCall(t9,g,1)	33 - Mul(t16,t14,t15)
4 - Copy(t2,4,-)	12 - JumpF(t6,10,-)	23 - Copy(t10,b,-)	34 - FReturn(t16,-,-)
5 - Param(t2,-,-)	13 - Copy(t7,t4,-)	24 - Param(t10,-,-)	
6 - FCall(t3,f,3)	14 - Jump(11,-,-)	25 - Copy(t11,c,-)	
7 - Halt(t3,-,-)	15 - Label(10,-,-)	26 - Param(t11,-,-)	
	16 - Copy(t7,t5,-)	27 - FCall(t12,h,2)	
	17 - Label(11,-,-)	28 - Add(t13,t9,t12)	
	18 - FReturn(t7,-,-)	29 - FReturn(t13,-,-)	

Figure 20.3: Three-address Code for an FP Program

#### 20.1.1 FP Semantics in Three-address Code

We will use the same operational approach here that we have applied earlier in Section 18.3.2: define a function `trans` that takes FP code semantic structures as parameter and produces an equivalent sequence of three-address code. Our function `trans` will have an extra capability – namely, besides generating a sequence of code, it will also return the name of the temporary variable in the “result”

position of the last code element generated. So `trans(a + b)` would generate the three code sequence above and return the value `t3` as the result temporary variable in the `Add` code entry.

### Identifier

```
trans(V) = Copy(t1,V,-)
return t1 = generateTemp()
```

### Number

```
trans(N) = Copy(t1,N,-)
return t1 = generateTemp()
```

where `N` refers to its literal value.

### Builtin Function Application

```
trans(x op y) = trans(x), trans(y), op(t1,t2,t3)
return t1 = generateTemp()
```

where `op` is one of the arithmetic or Boolean builtin functions

```
+, *, -, /, %, ==, !=, <, <=, >, >=
```

and `trans(x)` returns `t2`, `trans(y)` returns `t3`

### FnApp

```
trans(f p1 ... pn) = trans(p1), ..., trans(pn),
                    Param(t1,-,-), ..., Param(tn,-,-), FCall(t0,f,n)
return t0 = generateTemp()
```

where `f` is the three-address code label of the first instruction of `f` and each `ti` is the result temporary variable from `trans(pi)`.

### Selection

```
trans(if a op b then t else f endif) =
    trans(a), trans(b), op(t0,ta,tb), JumpF(t0, l1),
    trans(t), Copy(t1,tt,-), Jump(l2,-,-), Label(l1,-,-),
    trans(f), Copy(t1,tf,-), Label(l2,-,-)
return t1 = generateTemp()
```

where `ta` and `tb` are the result temporary variables returned by `trans(a)` and `trans(b)`, `l1` and `l2` are labels, `tt` and `tf` are the result temporary variables returned by `trans(t)` and `trans(f)` respectively.

### NegExp

```
trans(-e) = trans(e), Neg(t1, t2,-)
return t1 = generateTemp()
```

where `t2` is returned by `trans(e)`.

### Function Definition

```
trans(f p1 ... pn = e) = Label(f,-,-), trans(e), FReturn(t1,-,-)
return t1
```

where `t1` is the result temporary variable returned by `trans(e)`.

Notice no reference is made to possibly nested function definitions. This is because the nesting is needed only to specify the scopes of identifiers. At code generation time the nesting is not required so definitions are simply translated one after another into the three-address code program.

### Program

```

    trans(e where fd1 ... fdn endw)
        = trans(e), Halt(t1,-,-),trans(fd1),...,trans(fdn)
    return t1

```

where `t1` is returned by `trans(e)`.

To insure there are no temporary variable conflicts, there is a private translation variable `nextTemp`, which is initialized to zero, and a function `getTemp()` which is defined as follows.

```

string getTemp() {
    return "t"+nextTemp++;
}

```

So `getTemp()` simply returns the string name for the temporary variable, and the name is guaranteed to be unique.

### ☛ Activity 79 –

Verify that the three-address code sequence in Figure 20.3 results from applying the semantic definition above to the FP program associated with that Figure.

---

## 20.1.2 Implementing Three-address Code Generation

In this section we are interested in exploring the necessary changes to make our FP translator generate three-address code rather than the FP<sup>+</sup> stack machine code. Of course, at a later point we expect to translate the generated three-address code into the machine code for some real processor. There are three activities to carry out. First we will define the structure of the class `OpCode` that defines the form of the codes for the three-address architecture. Second we will design and discuss the implementation of the code table into which three-address code will be generated. And finally we will address the three-address code generation, both design and implementation issues.

### Three-address Code Structure

While the three-address codes are simple enough, there is the problem that the argument can represent various values, from labels to literals to temporary variables – sometimes arguments entries are left blank. We will look at the various three-code instruction forms to see what kinds of values have to be represented in each of the address entries. The result entry, the first of the three, is a temporary variable in each instruction except `Jump` and `Label`, in which case the entry is a label. The second and third entries can be temporary variables, integers, identifiers (parameters), or labels.

These four kinds of entries (temporary variable, label, integer, parameter) must be interchangeable. This tells us that these entries must all exist under the same type – the type of a data member representing the second argument must admit all four kinds of data.

Since we will eventually use the intermediate code to generate final object code, we should be mindful of the way the argument entries might be used. For example, when a function call is generated we will need to know the number of parameters, the function’s label, and have a mechanism for back-patching. In addition, we will be doing some code optimization later in the chapter, and we may need information not otherwise represented in the three-address code form.

The solution to this problem will seem overly complex until we see the solution in action, then it will make sense. The idea derives from the fact that, at least for the function and parameter names, all the information we may need during optimization and object code generation is already in a symbol table entry. That’s good – if we need the number of arguments for a function, just look in the symbol table! But then what do we do with the temporary variables, integer values, and labels – they don’t have symbol entries? The overly complex idea is to store them in the symbol table as well. That way every entry in a three-address code will be a symbol table entry – we solve the typing problem and the information access information at the same time.

Putting temporary variables and labels in the symbol table is a bit strange, but the literal values into the symbol table? Literal values just don’t belong in a symbol table. But the advantages of including literals out ways any strangeness. To make it sound more reasonable, we will make sure a string version of the literal is stored as the name of a literal in it’s symbol table entry. This, in fact, will help facilitate the code optimization process to be discussed later. The structure we have described here for the symbol table entries is summarized in the UML class diagram in Figure 20.4. There are details about the `TempEntry` and `LabelEntry` not represented in the diagram but will be discussed in the next section.

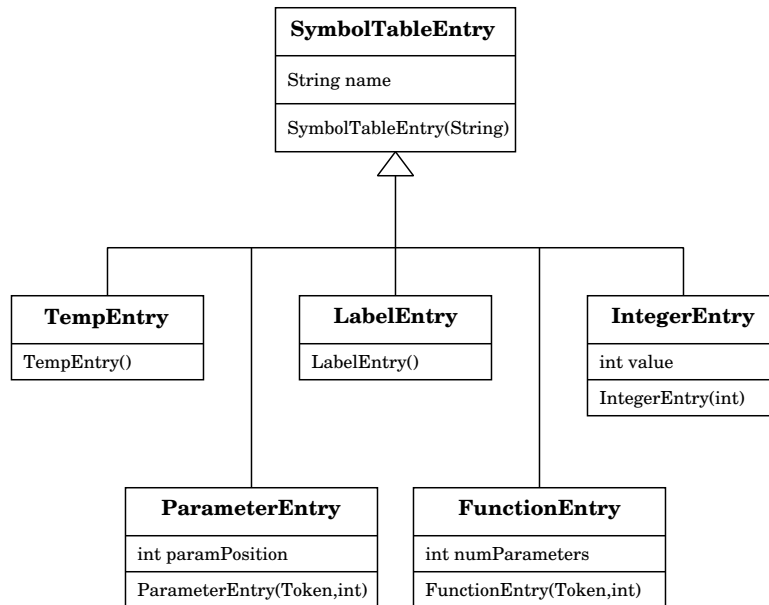


Figure 20.4: UML Class Diagram of `SymbolTableEntry` Hierarchy

This following definition for the class `OpCode` gives the flexibility to generate all the required



instruction types and to access and modify what are called the address entries.

```
public class OpCode {

    public enum OpName { Madd, Mmult, Msubt, Mdiv, Mmod, MNegate,
                        Meq, Mne, Mlt, Mle, Mgt, Mge, MCopy, MParam,
                        MFCall, MJumpF, MJump, MFReturn, MLabel, MHalt };

    OpName op;
    SymbolTableEntry adr1, adr2, adr3;

    public OpCode(OpName op, SymbolTableEntry a1,
                  SymbolTableEntry a2, SymbolTableEntry a3);
    public OpCode(OpName op, SymbolTableEntry a1, SymbolTableEntry a2);
    public OpCode(OpName op, SymbolTableEntry a1);

    public OpName getCode();
    public SymbolTableEntry getAdr1();
    public SymbolTableEntry getAdr2();
    public SymbolTableEntry getAdr3();

    public void setAdr1(SymbolTableEntry a);
    public void setAdr2(SymbolTableEntry a);
    public void setAdr3(SymbolTableEntry a);

    public String toString();
}

```

## The Code Table

Given the definition of `OpCode`, the definition of the code table class follows naturally. As was the case when we generated code for the FP translator on the FP<sup>+</sup> machine, we will do all three-address code generation via a traversal of the FP syntax tree. So our code table class will require `genCode` methods for each of the possible three code formats. It will also be convenient to be able to modify an argument in a particular generated code – say, based on its position in the code table. For data we must have an array of `OpCode` as the actual code table and to add the next code into the table there must be a variable to keep track of the next position available. This is all summarized in the following class definition.

```
public class CodeTable {
    private final int maxInstructions = 300;
    private Debug debug;
    private OpCode[] ct;
    private int nextPsn;

    public CodeTable(boolean db)

```

```

public void genCode(OpCode.OpName op);
public void genCode(OpCode.OpName op, SymbolTableEntry a1);
public void genCode(OpCode.OpName op, SymbolTableEntry a1,
                    SymbolTableEntry a2);
public void genCode(OpCode.OpName op, SymbolTableEntry a1,
                    SymbolTableEntry a2,
                    SymbolTableEntry a3);

public void genCode(OpCode op);

public void setAdr1(int psn, int value);
public void setAdr2(int psn, int value);
public void setAdr3(int psn, int value);

public int getCodePosition();

public String toString();
}

```

### Generating Temporary Variables and Labels

In our examples we have typically designated a temporary variable as a `t` followed by an integer value (for example `t0`, `t5`, `t12`) and we have done the same with label names – `l` concatenated to an integer value. Generating names in this style is easy: we make use of a static variables in the `TempEntry` and `LabelEntry` subclasses of `SymbolTableEntry`. The respective static variables will be incremented each time a new name is requested.

#### ☛ Activity 80 –

1. Write appropriate class definitions for the subclasses `TempEntry`, `LabelEntry`, and `IntegerEntry`. Remember that the value of the generated name is actually stored in `SymbolTableEntry`, so to set the generated name will require that you reference the superclass's constructor which, in the subclass, is called `super`. For integer remember that the small expression `""+val` (if `val` is an integer variable) is a string representation of the integer value in `val`.  
Be sure that label and temporary variable subclasses each have a static counter, as described above, and initialize it to 1.
2. Add two new methods to `SymbolTable` with the following signatures.

```

public TempEntry getNewTemp()
public LabelEntry getNewLabel()

```

These are analogous to the method `SymbolTable.addEntry`, but since we need the value created, these methods have return values. The methods are quite similar. The method `getNewTemp` acts as model. It is called whenever a new temporary variable is needed – this happens relatively frequently. The method should append the string `'t'` and the static counter (in `TempEntry`), thus generating a new temporary

variable name. A `TempEntry` object (with the generated name) is created, added to the symbol table, and then returned as the method return value. Following this description implement the two methods.

3. Add a method to `SymbolTable` that will add an integer value, if it hasn't already been added. The signature is as follows.

```
public void addInteger(int val)
```

This method will check to see if the integer's "name" is already in the table. If it is the method simply returns. If it is not it will add the entry value using the new `IntegerEntry` class. In this situation we only care that there is a single entry for each different numeric value.

## Implementing Three-Address Code Generation

Having set up the infrastructure above, the implementation of the code generation methods for the syntax tree classes is straight forward and parallels the code generation for the FP<sup>+</sup>-machine. But in our new context the semantics for the code generation methods is defined by the definition of `trans` given earlier on pages 362-363 in Section 20.1.1. Because the implementations of `generateCode` follow so closely the definition of `trans`, we will look at three representative implementations and leave the others as an exercise.

### `Identifier.generateCode`

For the leaves of an expression tree the code generation is a matter of instantiating a temporary variable and generating an `MCopy` instruction. For an identifier we must retrieve the symbol table entry for use in the generated instruction. One thing to realize is that the variable `name` in the following code is a variable defined in the class definition where this method is defined – so there's no magic going on here.

```
// Identifier.generateCode
public SymbolTableEntry generateCode(SymbolTable st, CodeTable ct) {

    ParameterEntry pe = (ParameterEntry)(st.findEntry(name));
    SymbolTableEntry result = st.getNewTemp();
    ct.genCode(OpCodes.OpName.MCopy, result, pe);
    return result;
}
```

All of the code generation methods for expression components have this basic structure, returning a reference to the temporary variable in `result` representing the last value to be computed. In the calls to `genCode` above, we have to first allocate a new temporary variable, via `st.getNewTemp()` and then use that generated temporary variable as the result component of the new instruction. That same temporary variable becomes the return value of the `generateCode` method. This is where our implementation of `getNewTemp` is important: since we increment the static count variable on

each request there's no danger of `getNewTemp` returning an already assigned temporary variable name.

#### `FnApp.generateCode`

Reviewing the definition for `trans` will reveal the design for `generateCode` for `FnApp`. There are a couple of important details. First, as indicated earlier, we will use the function name's entry from the symbol table to stand in place of the function label and thus make the third argument (the number of parameters) unnecessary (that value can be looked up in the symbol table entry if needed).

The second detail is the need to generate for each argument an `MParam` instruction containing a reference to the result temporary variable retrieved from the code generated for the parameter. Since these `MParam` instructions have to be generated after all arguments have been processed, it is necessary to save the result temporary variables for each argument as we go along. These temporary variables are kept in a `LinkedList` structure and made a part of the function name's entry in the symbol table. The list of argument result temporary variables comes in handy in a later section, so we will.

```
// FnApp.generateCode
public SymbolTableEntry generateCode(SymbolTable st, CodeTable ct) {
    // retrieve a reference to fe's symbol table entry
    // -- this also provides access to number of parameters
    FunctionEntry fe = (FunctionEntry)(st.findEntry(name));

    // generate code for each parameter to fe
    // -- at the same time make a list of the parameter result temporary variables
    LinkedList<TempEntry> params = new LinkedList<TempEntry>();
    for (Expression entry : argList) {
        SymbolTable t = entry.generateCode(st, ct);
        params.addFirst(t); // addFirst guarantees we'll see params
                           // in the appropriate order
    }
    // now use the list in params to generate required MParam entries
    for(TempEntry entry : params) {
        ct.genCode(OpCodes.OpName.MParam, entry);
    }
    TempEntry result = st.getNewTemp();
    ct.genCode(OpCodes.OpName.MFCall, result, fe );

    return result;
}
```

It should be noted, and at the risk of repetition, that in this translation we don't completely follow the definition of `trans(FnApp)` defined earlier. In that definition the original `FCall` code form was used which requires a label be generated to mark the beginning of the function's code segment. In that form we would expect the code line above to look more like

```
ct.genCode(OpCodes.OpName.MFCall, result, feLabel, numParams );
```

where `feLabel` would be a three-address code label and `numParams` would be the number of parameters required for the function. Our use of `fe` as a reference to the function name’s symbol table entry makes the earlier definition unnecessary – now the address of the function and the number of parameters are both found at the reference `fe`.

### ☛ Activity 81 –

Implement the `generateCode` methods for each of the subclasses of `SyntaxTree`, following the lead of the examples above and the definition of the `trans` function.

---

## 20.2 Code Optimization – Eliminating Common Sub-expressions

There are big advantages to adopting three-address code for intermediate code generation. The process of code generation produces a linear representation of the syntax tree, with most codes encapsulating the data at a syntax tree node. The two address arguments in a code (usually temporary variables) reference the node’s links to its two children while the result argument references the node’s parent – i.e., the address arguments point down the tree and the result argument points back up the tree. So if there are optimizations to be done they can take advantage of the fact that computation elements are encapsulated. So common sub-expression elimination (see this chapter’s introduction) is facilitated. Though FP is very simple, the three-address code for the selection expression represents accurately the non-linear execution pattern. So analyzing control flow should be facilitated in structured expressions and, in imperative languages such as Java or C++, structured statements such as selection and repetition. Finally, since the temporary variables are used like general purpose registers, the three-address code provides opportunities for register optimizations.

In the rest of this section we will focus on common sub-expression elimination first in simple arithmetic expressions, then in expressions with function calls, and finally in the presence of the selection expression. We begin with a few examples to illustrate the problem we are trying to solve. To begin, consider the following three-address code which would be generated for the expression on the right.

```
(MCopy,t1,a,-)          (a + 1) * b + a * (a + 1)
(MCopy,t2,1,-)
(Madd,t3,t1,t2)
(MCopy,t4,b,-)
(Mmult,t5,t3,t4)
(MCopy,t6,a,-)          <<
(MCopy,t7,a,-)          <<
(MCopy,t8,1,-)          <<
(Madd,t9,t7,t8)         <<
(Mmult,t10,t6,t8)
(Madd,t11,t5,t10)
```

If you trace carefully through the code there are some things that become immediately obvious. We seem to copy `a` into three different temporary variables. But since the value of `a` doesn’t change,

we could simply use the original value copied into `t1`. Notice that the same problems exist for the literal `1` and for the expression `(a + 1)`. The unnecessary codes above are marked. If we remove the marked codes and renumber the temporary variables we get the following simplified code sequence.

```
(MCopy,t1,a,-)          (a + 1) * b + a * (a + 1)
(MCopy,t2,1,-)
(Madd,t3,t1,t2)
(MCopy,t4,b,-)
(Mmult,t5,t3,t4)
(Mmult,t6,t1,t3)
(Madd,t7,t5,t6)
```

We save four instructions. If we think of the temporary variables as registers on a processor, then we have cut the number of registers needed by the same amount as well.

A similar optimization for expressions with function calls takes a bit more care. Consider the following non-optimized code for the expression on the right.

```
(MCopy,t1,a,-)          (f a b) + (f a b)
(MParam,t1,-,-)
(MCopy,t2,b,-)
(MParam,t2,-,-)
(MFCall,t3,f,-)        // the 'f' represents the address of f's code
(MCopy,t4,a,-)         <<
(MParam,t4,-,-)        <<
(MCopy,t5,b,-)         <<
(MParam,t5,-,-)        <<
(MFCall,t6,f,-)        <<
(Madd,t7,t3,t6)
```

We can see that the code which follows `(MFCall,t3,f,-)` is the same as the first five instructions — only the names of the temporary variables are different. We eliminate the five marked instructions, giving the following simpler sequence.

```
(MCopy,t1,a,-)          (f a b) + (f a b)
(MParam,t1,-,-)
(MCopy,t2,b,-)
(MParam,t2,-,-)
(MFCall,t3,f,-)        // the 'f' represents the address of f's code
(Madd,t4,t3,t3)
```

It is important that simplification of these code sequences depends on the fact that the values in the variables involved cannot change in the context of the expression. Of course, in imperative languages there are various statements that can cause the value of a variable to change — the most common being the assignment statement. So in such a context such simplifications of three-address code would require more analysis. But for the time being we will take advantage of the characteristics of FP, which simplify the optimizations to be discussed below.

## • Activity 82 –

Generate three-address code for each of the following expressions. Then carry out the optimization procedure demonstrated above.

1.  $(a + b) * (a + b)$
  2.  $(a + b) - (b + c) * b$
  3.  $a * (f a a) + a$
- 

### 20.2.1 Sub-expression Elimination – The Process

Before going too far it is good to make sure we understand the scope of our optimization activities. In doing common sub-expression elimination we are obviously interested only in those code elements that are involved in actual expressions, and these appear in FP programs as the main expression and then the defining expression for each function definition. In a program, then, we would do several independent sub-expression eliminations: one for the main expression and one for each of the function definition expressions. So we do the main sub-expression elimination (starting at offset zero) and when that reaches the `Halt` code we restart the process at the offset for the first instruction in the function expression. It is important also to recognize that since the processing of function expressions (and the main expression) are all independent, with each new expression, the temporary variable counter should be reset to 1. This means that the same temporary variable can appear in multiple expressions, but their independence means the variables won't interact.

One important thing to realize is that not every three-address code participates in the elimination process. The code types `Label`, `JumpF`, and `Jump` do not have result temporary variables, even though they do have result arguments. This is because they don't manipulate data. So when one of these instructions is encountered we will not search for copies.

Bringing the common sub-expression elimination to life requires a clear understanding of the problem and careful planning. In this section we will discuss the implementation strategy for basic sub-expression elimination in expressions without selection or function calls. Actually, our algorithm for sub-expression elimination is easy to follow but a bit messy to implement. We will describe the algorithm by first applying it to our earlier arithmetic expression. Assuming the start address and temporary variable counter are properly initialized, the process cycles through the following steps.

- Write down the instruction to be generated (without its result temporary variable) – but don't generate it.
- Search the instructions already generated for the instruction, ignoring the result entry.
- If the instruction isn't found then allocate a temporary variable, put it into the result slot in the instruction mockup, and then generate that instruction.
- If the instruction is found then generate nothing.

These steps are applied for each point of code generation – within the appropriate `generateCode` methods. For an expression we must imagine traversing its syntax tree and applying the steps at each node. Of course, the code generation is driven by FP's three-address code semantics. What follows is the sequence of code generation steps for the expression

$(a + 1) * b + a * (a + 1)$

but applying the four steps above at each point. At each step the element in the expression driving the code generation is underlined.

1.  $(\underline{a} + 1) * b + a * (a + 1)$   
 We want to generate (MCopy,-,a,-). It hasn't yet been generated so we generate temporary variable `t1` and generate the command.  
`0 (MCopy,t1,a,-)`
2.  $(a + \underline{1}) * b + a * (a + 1)$   
 We want to generate (MCopy,-,1,-). This isn't the same as the previous instruction, so we generate temporary variable `t2` and generate the command.  
`1 (MCopy,t2,1,-)`
3.  $(a + \underline{+} 1) * b + a * (a + 1)$   
 We want to generate (Madd,-,t1,t2). This instruction has not yet been generated, so we generate temporary variable `t3` and generate the command.  
`2 (Madd,t3,t1,t2)`
4.  $(a + 1) * \underline{b} + a * (a + 1)$   
 We want to generate (MCopy,-,b,-). We don't find this in the preceding instructions, so we generate temporary variable `t4` and generate the command.  
`3 (MCopy,t4,b,-)`
5.  $(a + 1) * \underline{*} b + a * (a + 1)$   
 We want to generate (Mmult,-,t3,t4). We don't find this in the preceding instructions, so we generate temporary variable `t5` and generate the command.  
`4 (Mmult,t5,t3,t4)`
6.  $(a + 1) * b + \underline{a} * (a + 1)$   
 We want to generate (MCopy,-,a,-). This matches the instruction at offset 0. Don't generate an instruction.
7.  $(a + 1) * b + a * (\underline{a} + 1)$   
 We want to generate (MCopy,-,a,-). Same as the previous point.
8.  $(a + 1) * b + a * (a + \underline{1})$   
 We want to generate (MCopy,-,1,-). This instruction matches that at offset 1. We don't generate an instruction.
9.  $(a + 1) * b + a * (a + \underline{+} 1)$   
 We want to generate (Madd,-,t1,t2). This instruction matches that at offset 2. We don't generate an instruction.
10.  $(a + 1) * b + a * \underline{*} (a + 1)$   
 We want to generate (Mmult,-,t1,t3). This doesn't match any previous instruction, so we generate temporary variable `t6` and generate the command.  
`5 (Mmult,t6,t1,t3)`



11.  $(a + 1) * b + a * (a + 1)$

We want to generate  $(Madd, -, t5, t6)$ . This doesn't match any previous instruction, so we generate temporary variable  $t7$  and generate the command.

6  $(Madd, t7, t5, t6)$

We summarize the results of this process in the table in Figure 20.2.1. The left column has the generated instructions, the middle column has the remembered temporary variable, and the right column indicates the sequence of code generation steps for the expression. Note that the token being examined is underlined.

Each time we identify an earlier instruction, we replace the result temporary variable for the instruction we would have generated by the result variable of the matched instruction. From the processor point of view, the value we want has already been computed and is stored in the result temporary variable. You might wonder why we check only for the one instruction – it seems that for the sub-expression  $(a + 1)$  we would have to check the whole structure, which involves more than one generated code. But, of course, when we examine the table above we can see that the data in the code for  $(a + 1)$  – that is,  $(Madd, t3, t1, t2)$  – actually does represent the operation and the two parameters since the temporary variables  $t1$  and  $t2$  are result variables for the argument expressions of the add operator.

### Activity 83 –

Produce non-optimized and optimized three-address code for the following expressions. In each case produce a table like the one in Figure 20.2.1

- $a + b * (a + b)$
- $2*(a + a * b)$
- $2*(a + a * b) + (a * b)$

---

Code Generated	Result Variable	Code Generation Steps
0 $(MCopy, t1, a, -)$	$t1$	$(\underline{a} + 1) * b + a * (a + 1)$
1 $(MCopy, t2, 1, -)$	$t2$	$(a + \underline{1}) * b + a * (a + 1)$
2 $(Madd, t3, t1, t2)$	$t3$	$(a + \underline{1}) * b + a * (a + 1)$
3 $(MCopy, t4, b, -)$	$t4$	$(a + 1) * \underline{b} + a * (a + 1)$
4 $(Mmult, t5, t3, t4)$	$t4$	$(a + 1) * \underline{b} + a * (a + 1)$
	$t1$	$(a + 1) * b + \underline{a} * (a + 1)$
	$t1$	$(a + 1) * b + a * \underline{(a + 1)}$
	$t2$	$(a + 1) * b + a * (a + \underline{1})$
	$t3$	$(a + 1) * b + a * (a + \underline{1})$
5 $(Mmult, t6, t1, t3)$	$t6$	$(a + 1) * b + a * \underline{(a + 1)}$
6 $(Madd, t7, t5, t6)$	$t7$	$(a + 1) * b + \underline{a} * (a + 1)$

Figure 20.5: Demonstration of Sub-expression Elimination

### 20.2.2 Eliminating Arithmetic Sub-expressions

The basic idea for implementing the sub-expression elimination is to alter the methods `generateCode` in the classes of the `Expression` hierarchy first to always return the resulting temporary variable and also to only generate new code if there isn't a match for the instruction to be matched. So, in the case of `Exp.generateCode` we might sketch the new implementation as follows.

```
public SymbolTableEntry generateCode(SymbolTable st, CodeTable ct) {
    // remember temp from expl
    SymbolTableEntry tLeft = expl.generateCode(st,ct);
    // remember temp from expr
    SymbolTableEntry tRight = expr.generateCode(st,ct);

    // Want to generate this.
    SymbolTableEntry try = new OpCode(opName, null, tLeft, tRight);
    // Does it exist earlier?
    SymbolTableEntry temp = ct.search4OpCode(try);
    if (temp == null) {
        // search failed: create new instruction with new result temporary
        SymbolTableEntry result = st.getNewTemp();
        // generate new instruction
        ct.generate(opName, result, tLeft, tRight);
        temp = result;
        // temp is now not null!
    }
    // temp != null
    return temp;
}
```

Remember that when we look back for matching instructions (via `search4OpCode`) we ignore the result entry; that's why we generate the temporary instruction `try` with the null parameter. Note that code generation methods need alteration only when they generate a three-instruction code that can be eliminated. So when the codes `Jump`, `JumpF`, and `Label` are to be generated the elimination process does not have to be called – these codes are just ignored as targets.

It is important to see in this code sketch that when we get the values `tLeft` and `tRight`, we don't know if they are result entries from just generated instructions or from previously generated instructions. Also, if we find that `tLeft` is a previous result, when we go off to generate code for the `expr`, we don't know how much code, if any, will be generated. The key point is that the value of `tLeft` is retained during `expr`'s code generation. This may be obvious from a programming point of view, but in tracing the sub-expression elimination examples above, that fact may not have been clear.

In order to finish the implementation of `Exp.generateCode`, we need to implement the method `CodeTable.search4OpCode`, which will require a mechanism for comparing the symbol table entries that can occupy the entries in a generated instruction.

## Comparing symbol table entries

It is obvious but is worth saying: to see if two three-address codes are the same we need to compare all four `OpCode` values in pairs. The op code names are enumerated type values so can be compared with the `==` operation. For the others this doesn't work. In the other three data members we need to compare the string names in the entries. But this is facilitated by the `SymbolTableEntry` method `getName`. Comparing the names is facilitated by adding to the class `SymbolTableEntry` a method `equals`, defined as follows.

```
public boolean equals (SymbolTableEntry val) {
    return name.equals(val.getName());
}
```

This method will be inherited by each of the subclasses.

## `CodeTable.search4OpCode`

There are two points in implementing the search for an equivalent instruction. First, we must have a place from which to start the search. The sub-expression elimination we have described is a local operation as described earlier. The starting point must be set for each expression, the main expression and each function expression. To manage this starting point, we will introduce into the class `CodeTable` a static instance variable `start` and a method for setting this variable.

```
public class CodeTable {
    ...
    private static int start = 0;
    ...
    public void setStart(int value) { start = value; }
    ...
}
```

So for each function definition encountered the value of `start` should be set to the first instruction generated for the function's expression.

We now address the search for previous instructions. The search is straight forward and can be accomplished with a sequential search. We begin the search at `start` and continue up to the value of `nextPsn`, which is where the next code is to be generated.

```
public TempEntry search4OpCode(OpCode op) {
    TempEntry t = null;
    boolean found = false;
    int i = CodeTable.start;
    while (!found && i < nextPsn)
        if (op.sameAs(ct[i]))
            found = true;
        else i++;

    if (found) t = (TempEntry)(ct[i].getAdr1());
}
```

```

    return t;
}

```

### ☛ Activity 84 –

There is another method mentioned here, `OpCode.sameAs`, which actually compares the data members of two `OpCode` objects. You should implement this method assuming the following signature.

```
public boolean sameAs(OpCode target)
```

---

This addition should complete the implementation of sub-expression elimination. It is important to recognize that the implementation is designed to work only on arithmetic expressions. If an expression contains function calls or selection expressions then the algorithm may not work.

### ☛ Activity 85 –

Complete the implementation of sub-expression elimination and test it on the following FP programs.

- $3 + 4$   
 where  $f\ a\ b = (a + b) * (a + b)$   
 endw.
  - $(2 + 4) * (4 - 2) * (4 - 2)$   
 where  $f\ a\ b = a + b * (a + b) / a;$   
 $g\ a\ b = a * a * a * a;$   
 $h\ x = (x + 2) * (x + 2);$   
 endw.
- 

### 20.2.3 Eliminating Common Function Calls

In this section we extend the sub-expression elimination to include function calls. Consider the following example.

```
f(a + 1) * f(a + 1)
```

In this expression there are four examples of common sub-expressions: `a` occurs twice as does `1`; in addition, both `(a+1)` and `f(a+1)` occur twice. While the first three of these common sub-expression can be found by our current implementation, the fourth cannot. But we could at least make a good guess at what the optimized code would be – here's a good guess.

```

0 MCopy(t1,a,-)
1 MCopy(t2,1,-)
2 Madd(t3,t1,t2)
3 MParam(t3,-,-)
4 MFCall(t4,f,-)  -- now we should be able to simply use t4 twice!
5 Mmult(t5,t4,t4)

```

So we have a good guess, but how do we automate a process? The main problem here is that once we have generated the function call for `f` our algorithm has forgotten the expressions processed as parameters. If we are going to identify a function call that has appeared before, we must have available the parameters for the call. To get a better perspective on this problem we will work through the translation of our example expression, applying subexpression removal as we currently know it. We will pay special attention to the second function call and leave details of the other code generation and sub-expression elimination to a minimum. To help track the process here is the syntax tree for our expression, which is central to the translation.

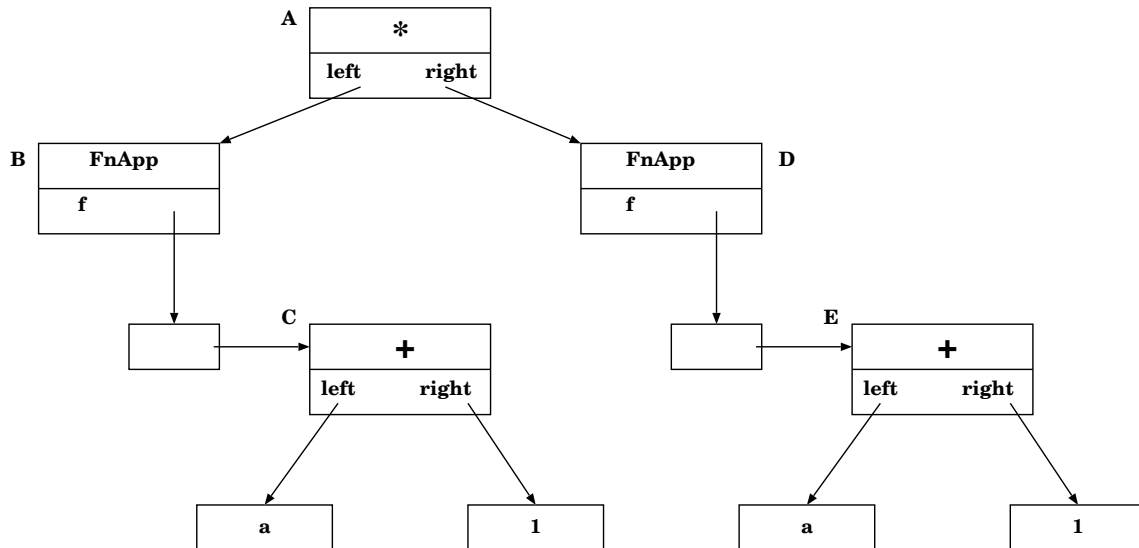


Figure 20.6: Syntax Tree for  $f(a + 1) * f(a + 1)$

- A** The first node is a multiplication node, so we translate each argument of the product.
- B** The left node is an `FnApp` node. We translate each argument and save a list of the temporary result variables from each argument translation.
- C** The only argument is an addition node. So we translate each argument of the sum.

The expression `a + 1` will translate to the following code.

```

0 MCopy(t1,a,-)
1 MCopy(t2,1,-)
2 Madd(t3,t1,t2)

```

The result variable is `t3` which we save in a list: `[t3]`.

**B** (resume)

We have the result variable list and so can generate the `MParam` instructions and the call instruction.

```
3 MParam(t3,-,-)
4 MFCall(t4,f,-) – we will save [t3].
```

**D** The right node is an `FnApp` node. Carry on as before.**E** The only argument is an addition node. We translate each argument.

The expression `a + 1` translates as follows.

1. Try instruction `MCopy(-,a,-)`: exists.  
The result variable `t1` is returned.
2. Try instruction `MCopy(-,1,-)`: exists.  
The result variable `t2` is returned.
3. Try instruction `Madd(-,t1,t2)`: exists.  
The result variable `t3` is returned.

The result variable is `t3` which we save in a list: `[t3]`.

We stop the process early because we have arrived at the crux point. Notice what has happened so far. We have translated the left argument to the top-level multiplication. We didn't talk about sub-expression removal because there was none to be done – also we ignored the problem of how to handle the first `FnApp` node because we wanted to get to the second one where all the action was bound to be. The one thing we did do was to save the list of argument return variables.

So we began the translation of the second `FnApp` node and processed the argument for the call to `f`. And what did we find? We found that the argument evaluation generated no code because each attempt found an earlier version of the proposed instruction. Notice the importance of the fact that finding a matching earlier instruction still returned the result variable from the earlier instruction – this allowed us to establish the list of return temporary variables for the arguments. We are now at a point where we must confront the search for an earlier call. We can resume the search.

**A** ...**D** ...**E** ...

The result variable is `t3` which we save `[t3]`.

**D** (resume)

We have the result variable list and will now look to see if this expression is a duplicate. We build an instruction `MFCall(-,f,-)`, save `[t3]`.

When we look back we see the instruction at address 4 is the same:

1. We ignore the first (result) argument as required.
2. We check the symbol table entries for each second argument and find the same names and same number of parameters.
3. When it comes to checking the parameter lists, we can see that they are the same, but we will have to automate this. This will be discussed below.

Passing all these tests means the calls are the same and we don't have to generate either the `MParam` instruction or the `MFCall` instruction. But we do know the correct return variable is `t4`, as it came from the first call.

#### A (resume)

Now we can generate the `Mmult` instruction as follows.

```
5 Mmult(t5,t4,t4)
```

So the big question is, “What should we do to save the parameter lists?” Fortunately, there is a simple answer. When we create our `MFCall(-,f,-)` instruction (so we can look for an earlier duplicate), we have the result parameter that is left intentionally blank. We can generate a new temporary variable and include it in the fabricated instruction. The trick is to also modify `TempEntry` so we can store in the temporary variable the param list just generated. But that's just half. Every time we generate an `MFCall` instruction we will store the generated parameter list in the result temporary variable. Now when a search is done for a matching `MFCall` instruction, all the algorithm has to do, in addition to comparing names and number of parameters, is to compare the parameter lists saved inside the first argument.

In order to put this to work, we must make changes to `FnApp.generateCode`. The following sequence describes the necessary processing for the altered code generation. You can look at the `FnApp.generateCode` code to see where things are done.

1. Generate argument code as usual (some may disappear due to sub-expression elimination), saving each argument's result variable in the list `params`.
2. Generate a new result temporary variable and save `params` in this result variable.
3. Generate an `MFCall` instruction containing, as the result entry, the new result temporary variable holding the list `params`. Note: this generated instruction is only used for the search in the following step.
4. Search the already generated code for the constructed call instruction.
5. As in earlier code generation method, if the value returned by the search is `null` then a matching instruction wasn't found, so the list of `MParam` instructions must be generated followed by the `MFCall` instruction.

The modifications described above fit easily into the original `FnApp.generateCode`. Here is the implementation just described, with each step above annotated in the code by the step number and an arrow.

```
public SymbolTableEntry generateCode(SymbolTable st, CodeTable ct) {
    FunctionEntry fe = (FunctionEntry)(st.findEntry(name));
    LinkedList<SymbolTableEntry> params = new LinkedList<SymbolTableEntry>();
    // generate code for arguments; save variable t in the list params
    for (Expression entry : argList) {
        SymbolTableEntry t = entry.generateCode(st, ct);
        params.addFirst(t);
    }
}
```

```

    }
    // generate call
1 ==> TempEntry result = new TempEntry(); result.setParams(params);
    // result is the phantom result temporary variable but will
    // be the real one if no match is found
2 ==> OpCode op = new OpCode(OpCode.OpName.MFCall, result, fe );
3 ==> TempEntry temp = ct.search4FnCall(op);
4 ==> if (temp == null) {

4 ==>     for (TempEntry entry : params)
            ct.genCode(OpCode.OpName.MParam, entry);
        }
4 ==>     // Now generate the op that didn't find a match.
            ct.genCode(op);
            temp = result;
        }
    return temp;
}

```

This code raises a few more implementation problems. First, we must implement `search4FnCall`, which, fortunately, is very similar to `search4OpCode`. But in implementing `search4FnCall`, there is one more design problem to address. We will need a Boolean method to compare two `MFCall` instructions. A match will be confirmed if the two function names are the same and the number of parameters match and if the parameter lists are the same, i.e. the corresponding `TempEntry`s have the same name. You will resolve the remaining issues in the next activity.

### ☛ Activity 86 –

1. Make appropriate modifications to `TempEntry` so that the list `params` can be saved – setter and accessor methods are essential for this new data member.
2. Implement the method `FnApp.generateCode` as described above.
3. Implement the two methods for comparing `MFCall` instructions: `CodeTable.search4FnCall` and `OpCode.sameAsFnCall`. These should have the following signatures.

```

public TempEntry search4SameOp(OpCode op)
public boolean sameAsFnCall(OpCode target)

```

4. Implement the changes described and test the resulting translator on the following FP programs.
    - (f 2 3) \* (f 2 3) \* (f 2 1)  
where f a b = a + b  
endw
    - f 2 3  
where f a b = (g (a + b)) \* (g (a + b));  
g x = x \* (x + 1)  
endw
-



### 20.2.4 Sub-expression Elimination in Selection Expressions

Our common sub-expression elimination implementation allows more efficient code generation for expressions containing arithmetic operators and function calls. But FP<sup>+</sup> has one other type of expression – the selection expression. But this problem is considerably more difficult to manage. Consider the following selection example.

```
if (a < a + b) then (c + b) else (c + b) * b endif
```

In this example we see four occurrences of **b** and two of both **a** and **c**. We also have two occurrences of **(c+b)**. According to the sub-expression elimination strategy implemented above, we should be able to reuse the temporary variables associated with the first occurrence of each in subsequent occurrences. With the variable **a** we would expect its first occurrence to be associated with **t0**, and so each subsequent occurrence would be replaced by **t0**. The same can be said for **b**. We know it will be associated with **t1** and its subsequent occurrences will be replaced by the same temporary variable. But what about **c** and **(c+b)**? If the first occurrence of **c** is associated with **t3** can we replace the second occurrence with **t3**? The answer is no! Why? Because if the selection expression is false then the first **c** won't even be processed (at run time). The same is true for **(c+b)**.

Another problem arises when a selection is embedded in a larger expression. Consider, for example, the following expression.

```
(if (a < b) then c + b else c - b endif) * (c + b)
```

In this case we see two copies of the expression **(c+b)**. According to the strategy we discussed above, we would take the temporary variable associated with the first **(c+b)** and use it in the place of the second. But of course once again, the first **(c+b)** is in the true component of the selection and we don't know that it will be computed. If the false part is executed then the value of **(c+b)** will not have been executed, so there is not an already computed value to use.

Apparently, there are some rules we would have to follow in doing sub-expression elimination in the presence of selection expressions. First, any sub-expressions appearing before the **then** can be used to optimize the **then** and **else** parts. But the sub-expressions in the **then** part cannot impact the **else** part. At the other end, sub-expressions in the **then** or **else** parts cannot affect the sub-expressions following the **endif**.

What we need is a way to mark an instruction so that it cannot be matched in any instruction search. With such a mechanism, as we generate code for the **then** part, we would want to mark the generated instructions. In this way, when we generate code for the **else** part we would ignore the instructions of the **then** part. Similarly we would mark the instructions of the **else** part so that instructions from neither part can be used in optimizing the sub-expressions following the selection. More particularly, we can follow these steps when generating code in `Selection.generateCode`.

1. Generate code as usual for **bexp**.
2. Remember where the first instruction of **truePart** is generated.
3. Generate code as usual for **truePart**. Remember the result temporary variable for the **truePart**.
4. From the first instruction of **truePart** to **nextPsn-1**, mark each instruction as untouchable.

5. Remember where the first instruction of `falsePart` is generated.
6. Generate code as usual for `falsePart`. Add a copy instruction which copies the final result temporary variable to the saved final temporary variable of `truePart`
7. From the first instruction of `falsePart` to `nextPsn-1`, mark each instruction as untouchable.
8. Return the result variable of `truePart`, which we have also made the result variable for `falsePart`.

To facilitate marking, we can add a fifth instance variable called `marked` (type `boolean`) to `OpCode` as well as setter and getter methods. The `OpCode` constructor will initialize `marked` to `true`.

Our method basically factors out the selection branches for further sub-expression elimination. So for example, if the expression `(a + b)` appears for the first time in both the true and false parts of a selection, couldn't future versions of `(a + b)` be removed? The answer is yes, but our algorithm isn't sophisticated enough to deal with it. We could, for example, pull the expression `(a + b)` out in front of the selection somehow and then it would be evaluated before the selection and the occurrences within the two branches would then find the code during sub-expression elimination. Also, occurrences of the same expression after the selection would be able to be eliminated by the algorithm.

The problem we have discussed here for the selection expression is actually part of a larger problem that is present in imperative languages such as Java and C<sup>++</sup>. In these languages we have selection statements as well as repetition statements. Since these statements can be nested, the problem of knowing when a common sub-expression can be eliminated becomes considerably more difficult. Of course the fact that assignment statements can change the value of a variable only adds more complications. The technique used to solve these problems is called *Flow Graph Analysis* and is discussed briefly in the next part of the book.

### ☛ Activity 87 –

1. Modify `Selection.generateCode` to implement the sub-expression elimination strategy we have just described. Test your implementation on the following expressions.
  - (a) Just to make sure nothing broke!
 

```
f 2 3
  where f a b = (g (a + b)) * (g (a + b));
        g x   = x * (x + 1)
  endw
```
  - (b) `(a + b)*(c + b) + ( if (a < a + b) then (c + b) else (c + b) * b endif )*(a + c)*(c + b)`
2. This implementation for selection expressions isn't perfect. Describe situations in selection expressions in which sub-expressions could be eliminated but are not by the strategy implemented in the previous example. For each situation give a specific example and demonstrate that the implemented strategy eliminates certain sub-expressions but not all. For each example give the optimal generated code sequence.
3. For an expression such as `(a + b) * (b + a)` our current code generation strategy would find no common sub-expressions – but we know that addition is commutative so we should be able to use the value of `(a + b)` in place of `(b + a)`. Make

alterations to the sub-expression elimination implementation to take advantage of the commutativity of addition and multiplication.

---



## Chapter 21

# FP<sup>+</sup> to MIPS Code Generation

Despite the “FP<sup>+</sup>” in the chapter title, the MIPS code generation starts from the already generated three address code, which was generated from FP code. So FP and the FP and FP<sup>+</sup> stack machines are not of importance in the chapter. In this chapter we will address the basic problems associated with generating object code from three address code. For simplicity we will generate assembler rather than machine code. We have chosen the MIPS architecture as the object code target because the MIPS architecture and the MIPS SPIM simulator are so readily available.

### 21.1 Translating Three-address code to MIPS Assembler Code

Our first approach to code generation is simple and basic. Since we are translating from three-address code, no instruction will require more than three general purpose registers. We allocate the first three registers for this purpose, remembering that these registers cannot be used for long-term storage. So we must insure that when an instruction is complete all register values have been copied back to memory. We will also organize the stack as a traditional sequence of stack frames. Each stack frame will hold the function parameters, the function local variables, and two extra positions to save the function’s return information, i.e., the return address and frame pointer.

The implementation of the code generator will be in the form of a new class, `MIPSCodeTable`. The code generation methods in this class will take the already generated intermediate code table as a parameter, thus giving access to the three-address instructions needing translation.

#### 21.1.1 Addressing Parameters and Temporary Variables

In the translations that follow the names `n` and `address` will refer to `IntegerEntry` objects, `t1`, `t2`, `a1`, `a2` and `result` will refer to `TempEntry` or `ParameterEntry` objects, while `label` will refer to a `LabelEntry` object.

For temporary variables and parameters, there are methods `getPosition()` in both classes `ParameterEntry` and `TempEntry`, which return the appropriate byte-offset position value associated with the variable or parameter. Figure 21.1 shows how stack frames will be laid out – notice the values of the stack frame register (`fp`) and the stack pointer (`sp`).

Notice that the stack frame pointer `fp` points to the middle of the stack frame, with variables holding parameter values allocated in order above the stored return address and temporary variables

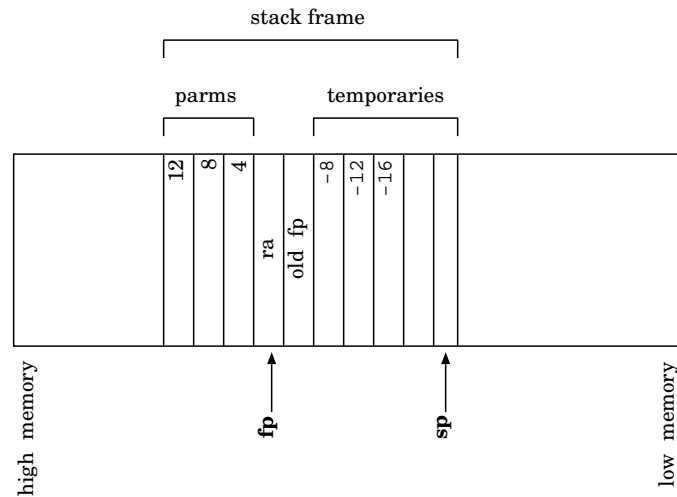


Figure 21.1: Stack Frame Layout

allocated in order below the return frame pointer. Since MIPS addresses are byte addresses, the offsets (from `fp`) for parameters are positive multiples of 4, so 4, 8, 12, etc. Temporary variables are located at offsets from `fp` that are negative multiples of 4 starting at -8. So parameter `n` is at offset  $4*n$  and temporary variable `n` is at offset  $-4*(n+1)$  – these computations are carried out by `getPosition`.

### 21.1.2 Three-address Code Semantics in MIPS Assembler

In this section we specify the semantics of the three-address code by giving translation equivalents in MIPS Assembler code. In the following discussions we use the standard MIPS assembler forms – see Appendix C for a review of MIPS architecture and assembly language. In addition we use the following notation to reference values computed during the code generation process.

- `t.offset`: `t` is either a `ParameterEntry` or `TempEntry` object. If `n` is the position value for a parameter entry `t`, then `t.offset` is  $4*n$ ; if `t` is a temporary variable then `t.offset` is  $(-4)*(n+1)$ .
- `n.getValue`: `n` is an `IntegerEntry` object. `n.getValue` denotes the numeric value stored in the object `n`.

In the following sequence of three-address forms, we will specify translation in the form of MIPS assembly code. A brief explanation is given in each case.

#### <Mallocate, `n`>

If we want to allocate space on the stack, then we need only increment the value of the stack pointer `sp`. We have to remember that the address is a byte address and that the stack grows downward in memory – thus the we add  $(-4*n.getValue)$  to `sp`.

```
addi $sp, $sp, (-4*n.getValue)
```

## &lt;Mnumber, t, n&gt;

We translate this to a two instruction sequence, the first instruction loads the value of **n** into a register and the second stores that registers value to the variable **t**.

```
ori $t0, $zero, (n.getValue)
sw $t0, t.offset($fp)
```

## &lt;Mparam, t&gt;

When a parameter is allocated on the stack there are two aspects – first the stack must be extended and then the parameter value needs to be loaded from the variable **t** and then stored into the newly alloated stack position (at address **sp**).

```
addi $sp, $sp, -4
lw $t0, t.offset($fp)
sw $t0, $sp
```

## &lt;Mcopy, t1, t2&gt;

This copy operation is simply loads the value into a register from variable **t1** and then stored from the register to the variable **t2**.

```
lw $t0, t2.offset($fp)
sw $t0, t1.offset($fp)
```

## &lt;Mnegate, t1, t2&gt;

This is similar to the previous translation, but with the value of **t1** negated before being stored. We use the machine instruction **sub** to facilitate the operation.

```
lw $t0, t2.offset($fp)
sub $t1, $zero, $t0
sw $t1, t1.offset($fp)
```

## &lt;Madd, t1, t2, t3&gt;

This works as well for **Msubt**. The translation is easy: load the two arguments, **t2** and **t3**, into registers, apply the **add** instruction to the two registers and store the value in the result register to the variable **t1**.

```
lw $t0, t2.offset($fp)
lw $t1, t3.offset($fp)
add $t2, $t0, $t1
sw $t2, t1.offset($fp)
```

## &lt;Mmult, t1, t2, t3&gt;

This works as well for **Mdiv** and almost for **Mmod**. The difference for **Mmod** is that the remainder on dividing two integers is found in the **Hi** register. So for both integer division and mod we do the division command `'div $t0 $t1` and then use **mflo**, for division, or **mfhi**, for mod.

```

lw $t0, t2.offset($fp)
lw $t1, t3.offset($fp)
mult $t0, $t1
mflo $t2
sw $t2, t1.offset($fp)

```

<Mlabel, label>

We extract the name in `label` and then (we'll assume it is `label`) we generate the following single line of code.

```
label:
```

<MbeqT, t1, t2, label>

This works as well for the other five comparison relations. The idea is to load the arguments, `t1` and `t2`, into registers, apply the `beq` instruction to these registers, and to take the specified branch if the comparison is true.

```
beq t1, t2, label
```

<Mjump, label>

Assuming the name extracted from `label` is `label`, we generate the following single line of code.

```
j label
```

<Mfcall, t, fn>

A function call is a complex but interesting translation. The strategy assumes that parameters have already been pushed onto the stack. We allocate space for the return data (i.e., the registers `ra` and `fp`) and store that data into the allocated spaces. The value of `fp` is readjusted (see the diagram in Figure 21.1) and then the call instruction is generated. Notice that the label generated with the `jal` instruction is simply the function's name; we use that name since we have used it as the label for the function's code segment. We know that after returning from the function call, that the stack must still be cleaned up. The code generated returns the saved register values to `fp` and `sp`, deallocates the 8 bytes for the return data and the parameter space, and finally stores the returned computed value in the variable `t`.

```

addi $sp, $sp, -8
sw $fp, 4($sp)
sw $ra, 0($sp)
addi $fp, $sp, 4
jal fn.name
lw $fp, 4($sp)
lw $ra, 0($sp)
addi $sp, $sp, 4*(2+fn.getNumParams)
sw $v0, t.offset($fp)

```



**<Mfreturn, t, n>**

The function return value, stored in variable `t`, must be loaded and then stored in a known location; in the MIPS environment return values are placed in the register `v0`. In addition we must deallocate the temporary variable space, whose size is retrieved from `n`. The actual return is accomplished by the `jr` instruction with the register `ra` as argument.

```
lw $v0, t.offset($fp)
addi $sp, $sp, (4*n.getValue)
jr $ra
```

**<Mhalt, t>**

This instruction is meant to transfer control back to the SPIM simulator; the final line below facilitates the transfer. Before returning, however, the value associated with the halt instruction will be stored in `v1`. Normally we might want to save the final value in `v0`, but that register is used by SPIM before halting. We use `v1` as an alternative.

```
lw $t0, t.offset($fp)
addu $v1, $zero, $t0
jr $ra
```

## 21.2 Generating the MIPS Assembler Code

The structure of the object code table has a similar structure to that of the intermediate (three address code) code table. Just as intermediate code generation was implemented in the `SyntaxTree` hierarchy, the object code generation will be implemented in the class `CodeTable`, which of course holds the intermediate code. In this section we will lay out the structure of the object code table class and provide specific strategies for implementing the code generation methods in `CodeTable`.

### 21.2.1 The Object Code Table

It is important to remind ourselves that the “object” code we generate we will be in MIPS assembly language. For our purposes in the basic code generator we will simply generate a string for each instruction, a string which contains the appropriate assembly code. So the data members of the object code table will be a `String` array and a variable `nextPsn` which indexes the next array position the next instruction can be generated. This suggests the following definition.

```
public class ObjectCodeTable {

    private final int MAX = 1000;
    private String[] oct = new String[MAX];
    private int nextPsn = 0;

    public ObjectCodeTable() {}
    public void generateCode(String instruction) {
        oct[nextPsn] = instruction;
    }
}
```

```

    nextPsn++;
}
public String toString() {
    String str = "";
    for (int i = 0; i < nextPsn; i++)
        str += oct[i] + "\n";
    return str;
}
public void outputObjectCode(.....) {
    for (int i = 0; i < nextPsn; i++)
        out.println( oct[i] );
}
}

```

### 21.2.2 Generating Object Code

In order to generate object code we must traverse the intermediate code table and, for each three-address instruction, generate an appropriate sequence of MIPS assembler statements. We've seen this sort of algorithm before: a switch statement which has a case for each entry in `OpCode.OpName`. The actual code generated in each case will be dictated by the specifications in Section 21.1 – in fact it is a good idea to mark the pages 386-389 so you can reference them easily.

We add a method `generateCode` to the class `CodeTable` which will have the following structure. Notice the initial calls to `oct.generateCode(...)`. These supply the preamble MIPS assembly code segments for all generated programs.

```

public void generateCode(ObjectCodeTable oct) {
    // we need int, OpCode, and OpCode.OpName variables
    // in some of the cases in the switch statement below
    OpCode.OpName code = OpCode.OpName.Mnoop;
    OpCode op = null;
    int val = 0;

    oct.generateCode(".text");
    oct.generateCode("main:");
    oct.generateCode("addu $fp, $zero, $sp"); // initialize the fp register

    for (int i = 0; i < nextPsn; i++) {
        op = ct[i];
        code = op.getCode();
        switch (code) {
            // This is the code to be discussed in this section
        }
    }
}

```

The code to be generated in most cases is simply the string containing the translated form we saw in the Section 21.1. But there are cases where some specific techniques are called for. We will

look at the specific code generation for certain of the switch cases and leave the remaining cases as easy exercises — easy because the process will be clear from the example cases. In the final code version of `generateCode` it may be possible to organize the switch cases to take advantage of common code segments. We will not worry about that in these explications.

Before embarking on the specifics of the cases, it is important to remind ourselves about the structure of a three-address instruction. These instructions are instances of the class `OpCode` which is described on page 365. The four critical methods from the class are the following.

```
public OpName getCode() { return op; }
public SymbolTableEntry getAdr1() { return adr1; }
public SymbolTableEntry getAdr2() { return adr2; }
public SymbolTableEntry getAdr3() { return adr3; }
```

Though the values returned by the `getAdr` methods are of type `SymbolTableEntry`, we will usually know the exact type of a returned value by the particular instruction being processed.

Remember that in the descriptions which follow you can reference three address code to MIPS assembly definitions on pages 386-389.

### Madd

Remember in each of the examples that the variable `op` has as value a particular three-address instruction. This means that `op` will respond to the methods listed above. This first example will illustrate how to extract data from `op`. Remember this general translation for `Madd` instruction.

```
lw $t0, t2.offset($fp)
lw $t1, t3.offset($fp)
add $t2, $t0, $t1
sw $t2, t1.offset($fp)
```

We must take this description and adapt it to the reality of `op`. The first line refers to the value `t2.offset`. The first task is to recover from `op` the value of its instance variable `adr2`. You will hopefully have noticed that instructions, other than `Mcopy`, always have variables that are temporary. The first line above, then, can be translated to the following.

```
val = ((TempEntry)(op.getAdr2())).getPosition();
mc.generateCode("lw $t0, " + val + "($fp)");
```

This form is standard and appears in most of the translation cases. Since we declared only the one integer variable, we have to do this formation in turn. If the value of `val` is 12, then the string generated by the second line will be as follows.

```
lw $t0, 12($fp)
```

This takes advantage of the Java feature which automatically calls the `toString` method on a variable when the variable is used in a position expecting a value of type `String`. Extending this example, then, the generated code for this example will be the following.

```

val = ((TempEntry)(op.getAdr2())).getPosition();
mc.generateCode("lw $t0, " + val + "($fp)");
val = ((TempEntry)(op.getAdr3())).getPosition();
mc.generateCode("lw $t1, " + val + "($fp)");
mc.generateCode("add $t2, $t0, $t1");
val = ((TempEntry)(op.getAdr1())).getPosition();
mc.generateCode("sw $t2, " + val + "($fp)");

```

Though not exactly the same, the other arithmetic operations can be translated in a similar manner. It is only necessary to adapt the translation descriptions using the techniques above.

### Mcopy

Remember that the translation for the copy instruction is as follows.

```

lw $t0, t2.offset($fp)
sw $t0, t1.offset($fp)

```

This would seem to be easy to convert using the techniques above. But remember, the copy instruction is the one instruction where a parameter name can appear. So the translation above ignores the ambiguity of the name `t2`. Since the methods `getPosition` exist only in the classes `ParameterEntry` and `TempEntry`, we must correctly cast the value of `t2` so that the correct method is called. The following code segment will correctly determine a value for the variable `val`, and with that value the obvious code generation follows.

```

if ( (op.getAdr2()).getClass().getName().equals("symbolTable.ParameterEntry") )
    val = ((ParameterEntry)(op.getAdr2())).getPosition();
else // assert op is of type TempEntry
    val = ((TempEntry)(op.getAdr2())).getPosition();
mc.generateCode("lw $t0, " + val + "($fp)");
val = ((TempEntry)(op.getAdr1())).getPosition();
mc.generateCode("sw $t0, " + val + "($fp)");

```

### MjumpF

The interesting thing in the conditional branch instruction is the use of the label. A label, remember, is of type `LabelEntry`. In order to get the name associated with the label we simply call the method `getName` and use that in the `beq` instruction. The code is generated as follows.

```

val = ((TempEntry)(op.getAdr1())).getPosition();
mc.generateCode("lw $t0, " + val + "($fp)");
mc.generateCode("beq $zero, $t0, " + ( (LabelEntry)(op.getAdr2()) ).getName());

```

### Mfcall

This is a complex translation, but the hard work was done earlier when the translation was described. Now converting that description into code is not difficult. But there are a couple of interesting

points. Remember that the third parameter to the `Mfcall` is the symbol table entry of the function name being called. We need access to pieces of information from the function name entry: the name of the function and the number of parameters for the function. The number of parameters is extracted in much the same way as positions were extracted earlier. In this case we make use of the method `getNumParams` from the class `FunctionEntry`. We retrieve the name of the function in much the same way and use the name as the label in the `jal` instruction. This analysis suggests generating the following code.

```

val = ((FunctionEntry)(op.getAdr3())).getNumParams();
mc.generateCode("addi $sp, $sp, -8");
mc.generateCode("sw $fp, 4($sp)");
mc.generateCode("sw $ra, 0($sp)");
mc.generateCode("addi $fp, $sp, 4");
mc.generateCode("jal " + ( (FunctionEntry)(op.getAdr3()) ).getName() );
mc.generateCode("lw $fp, 4($sp)");
mc.generateCode("lw $ra, 0($sp)");
mc.generateCode("addi $sp, $sp, " + (8 + val*4));
val = ((TempEntry)(op.getAdr1())).getPosition();
mc.generateCode("sw $v0, " + val + "($fp)");

```

### Mallocate

This instruction is used only when allocating stack space for temporary variables. To know how much space to allocate we need the number of temporaries, carried in first argument (an `IntegerEntry`); this value is multiplied by 4 and the result subtracted from the `sp` register – of course, subtracting from `sp` increases the stack. The following code illustrates that in a `String` expression we can include a computation.

```

val = ((IntegerEntry)(op.getAdr1())).getValue();
mc.generateCode("addi $sp, $sp, " + (-4)*val);

```

### ☛ Activity 88 –

After completing and testing the object code generation, copy your `FP+` development tree and in the copy make the necessary modifications so that `FP+` becomes a floating point functional language. This means that `FP+` will become a language supporting just one type – floating point. The modifications are mostly to the code generator, since the MIPS instructions for manipulating floating point numbers are not the same as those for integers. Don't forget to change the tokenizer so that it recognizes floating point constants rather than integers. There are slight changes also to the set of arithmetic operations.

---



## Chapter 22

# MIPS Code Optimization

In the previous chapter we finally put the finishing touches to a complete compiler for FP<sup>+</sup>. The code generated can be loaded into the SPIM simulator and executed. But our compiler fails to meet an important expectation. In the Prelude to this text we discussed the Efficiency Principle (see page 7) which states the requirement that a translator make efficient use of its target computational environment. Since the MIPS processor requires that data be in general purpose registers before being processed, these registers are a precious resource. But our compiler, as designed, makes use of only three of the 32 general purpose MIPS registers: `t0-t2` (i.e., registers 8-10), leaving the others unused. The result is that there is a large amount of time consumed moving data back and forth between memory and registers. In this chapter we focus on this performance issue, i.e., the efficient use of processor time by making effective use of the available general purpose registers.

### 22.1 Strategies for Register Allocation

In intermediate code generation we translate the input program form represented by the syntax tree into a sequence of three-address codes. A code, remember, has three parameters, the first is the result address and the second and third are argument addresses – e.g., `Add(t3,t1,t2)` describes a command to add the values in `t1` and `t2` and store the result at `t3`. But when we generate object code, how exactly should we think of these temporary variables? When we introduced three-address code we were a bit uncommitted as to the nature of the memory represented by the temporary variables, do they represent registers or main memory. Now we will look an answer to this question and discuss the consequences of the answer.

In the FP to MIPS code generation (Chapter 21) our assumption was that the temporary variables are main memory and the first three of the MIPS general purpose registers are special purpose – the first two are associated with the argument parameters and the third is associated with the result parameter – when the execution of a code is completed the value in the result register is transferred back to the space for the associated temporary variable. So we see the three registers associated with the three components of an executing code – nothing else. To say this is inefficient is an understatement: there is a lot of copying data between registers and memory and there are quite a few MIPS general purpose registers that go unused. So, while this scheme is easy for code generation, it is not good the Efficiency Principle.

At the other extreme we could associate each temporary variable with a register. The idea of

associating temporary variables with memory is inefficient but doable; but associating all temporary variables with registers is just not a feasible approach. An FP programmer would be implicitly limited in the complexity of the expressions that could be used in a program. Even if there were 1000 general purpose registers, tying a programming language to a restriction based on such a hardware characteristic is unreasonable. After all, the whole idea of high-level languages is to provide a buffer between problem solutions and hardware characteristics.

So we are left, as is often the case in computer systems, with a compromise between two unattractive options and, as usual, the solution is to take the middle road. In this case we want to see the set of MIPS general purpose registers as a cache between the arithmetic unit and main memory. We store all temporary variables in memory (on a run-time stack, for example) and when a value in memory is needed it copied into the cache so it can be used more efficiently. The value will remain in the cache until either it is no longer needed or its position in the cache is necessary for a new value. Notice that if a function has simple code and the number of temporary variable is less than or equal to the number of general purpose registers, then we are in the “temporary variables are registers” solution. When the number of temporary variables is too big, on the other hand, then it is the nature of the computation that will dictate how temporary variables and registers are associated.

### 22.1.1 Register Allocation as Cache Management

The problem we have just been discussing, for temporary variable management, is commonly known as the *register allocation problem*. Solving the register allocation problem will require, during code generation, that we keep track of which registers are in use and for which variables they store values. The code generation will also have to be sensitive to the situation where register data is in jeopardy, so that it can be saved back to memory. As implied in the section introduction, the register allocation problem can be solved if we think of the set of registers as a cache for variables stored in main memory. The real question is what criteria we should use to judge a solution.

In a memory hierarchy we have levels for data storage, where at each level the data access time is significantly faster than at the level immediately below. In a modern processor there are four levels to the memory hierarchy: the fastest level is the registers, next would typically be the L2-cache, then the L1-cache, and finally main memory itself. For our purposes we will ignore the processor’s memory caches and assume the hierarchy involves only memory and registers. Taking this point of view is convenient because it allows us to make use of known cache management strategies for implementing register allocation.

Cache management is based on the principles enumerated in Figure 22.1. The first two principles guarantee that data can freely flow into the cache when it is needed. The third principle guarantees that data is not lost and the fourth that the system is efficient. You may have wondered why the second capability didn’t include copying the data in the victim register back to memory, but that kind of protection of data is the responsibility of the third capability.

A common characteristic of any cache is that it is quite small in comparison to the capacity of the hierarchy level being cached. While memory can be giga-bytes in size the on-chip memory cache is less than a mega-byte and the set of registers (another cache) is considerably smaller yet, perhaps in the 100’s! So in real caches the limited amount of space is the real constraint, i.e., there will be times when the cache is full and cache replacement must occur. We will see that the strategy for choosing a victim cache entry can have a big impact on performance. This is where principle 4 is



**Cache Principle 1: Allocation**

When a data value not in the cache is referenced a cache entry must be allocated for its use.

**Cache Principle 2: Replacement**

When allocation is carried out but the cache is already full, a cache entry (we call it the *victim*) must be selected and allocated for the new demand.

**Cache Principle 3: Consistency**

Consistency must be maintained between the cache and the data space – i.e., when there is the chance of data loss, modified cache entries should be copied back their original level.

**Cache Principle 4: Performance**

Transfers between the cache and the lower level in the hierarchy must be minimized.

Figure 22.1: Cache Principles

most critical.

In the next four sections we will evolve a register allocation strategy by introducing the four cache principles one at a time. Each section will build on the previous one – augmenting the existing solution with strategies to guarantee the current level’s focus. In the process you will understand why omitting one principle can cause problems and how the strategy guarantees the problem can’t happen.

### 22.1.2 Cache Management I – Allocation

This first attempt at cache management of a set of general purpose registers is meant to guarantee only Cache Principle 1 – that is, if there’s a request for a register, the request will be granted. This strategy will work perfectly well if we have a small FP function and all its generated temporary variables can fit into the existing registers. But what if we get more ambitious and try a function that has more temporary variables than registers? Our strategy will be to throw an exception indicating no registers are available. The program will terminate! Not a very nice solution for the ambitious FP programmer, but for the programmer who thinks small it might work well (most of the time). Despite this fairly large flaw, this will be our first register management solution.

Speaking technically now, during code generation our register management policy will be the following: When a temporary variable is encountered in a three-address instruction we will request a register. If that variable is already associated with a register then we will be given that register’s number. If the variable is not associated with a register, a free register will be found and that register number will be returned. If there are no free registers the translation will terminate.

With this register management policy we must keep track of which registers are associated with which temporary variables. More particularly, we adopt the following design.

1. Establish a vector for all available registers. The index in the vector will correspond to the register number. The entries in the array will allow an entry which indicates the register is or is not allocated – initially all registers should be available.

2. When a register is needed for a variable in memory, call `getRegister` to search the available list for an available register, save the symbol table entry for the temporary variable in the entry, allocate the register.
3. If there is no register available then throw an exception, signaling that the translation should terminate. Notice that this will happen in `getRegister`.

These design points imply that the register vector will hold objects having at least two data members – one to indicate allocation status and the other to the symbol table entry for the associated temporary variable. As we will see in later sections, having the symbol table entry will facilitate the solutions to certain cache problems.

We have a solution that satisfies the Cache Principle 1, but we recognize that the solution doesn't satisfy Cache Principles 2-4. We turn to the second principle in the next section.

### 22.1.3 Cache Management II – Adding Replacement

Our solution to register allocation at this point is great for any program in which the number of temporary variables is less than or equal to the number of general purpose registers. Of course we know that there are lots of programs that fail this criterion. The only way to improve on our solution is to provide a mechanism to allow more temporary variables access to the registers. This can only be done if we have a mechanism to choose a register and re-assign it to the new temporary variable. If this sounds too easy, you will see in the next two sections that you are right. But for now we'll be naive about the process.

So, imagine we are generating code for a program, all the registers are currently associated with some temporary variable, and we arrive at an instruction containing a new, not before seen temporary variable. In order for the instruction to be executed the content of this new temporary variable must be in a register! We need to choose one of the registers (the victim) and reassign it to the new variable. Notice that whatever variable is unbound from the register will just become an unattached temporary variable – if we encounter it later on in the code sequence we will simply have to find it a register again.

Our interest in this section is the selection of the victim register. We could use a random number generator to determine the victim's register number, but as we will see in a moment, random choices can cause trouble. We also might not want to choose the last register that was assigned to a variable because that variable may be in use over the next few instructions – best to not rock the boat too much. So we will choose the *first-in-first-out* strategy, that is, we will choose as victim the register that has been linked to a variable for the longest time. If we are choosing our first victim, then we would select the first register since that was the first assigned. If we have to choose again the victim will be the second register and we will continue in that pattern. Notice when we've been through all the registers a second time we will have to start again with the first register.

This strategy's best feature is that it is easy to implement. We simply have a counter starting at the first register number and each time we allocate a register we increment the counter – the counter can be a data member in `CodeTable`. So in our solution from the previous section, instead of throwing an exception in `getRegister` when no register is free, we will have `getRegister` allocate in the normal way the register whose number is in the counter – that is allocate it as though it had been free.

Ok, there are some obvious problems with this solution, but they will be dealt with in the next couple of sections. We are building a solution by tackling the Cache Principles one at a time. Before moving to Cache Principle 3 there is one more problem that we should address in this section.

### Victim selection in multi-address instructions

There is a special problem that can occur in register allocation depending on the victim selection strategy we use. Actually, the strategy we have chosen is free of this problem, but most will suffer from this. In fact, when we talk about efficiency we will discover that our first-in-first-out strategy is not good enough and the new strategy does suffer from the new problem. So imagine generating code for a three-address instruction which has two temporary variables not currently in registers and assume there are no free registers. The first temporary variable would cause a victim to be found and then the second would do the same. But what if the second victim turned out to be the register allocated to the first request? Since all arguments in an instruction must be associated with registers for the instruction to be executed, we have to come up with a way of preventing already assigned registers from being victims again.

Fortunately a solution is easy. The trick is to keep track momentarily of which registers have been allocated for this instruction – call them “in use,” meaning not available as a victim. So if we were generating code for the three-address instruction

```
Madd t4 t5 t6
```

we would do something along these lines:

```
List using = new List()
reg1 = getRegister(t5, using)
using.add(reg1)
reg2 = getRegister(t6, using)
using.add(reg2)
reg3 = getRegister(t4, using)
// now generate the MIPS instruction
```

By sharing the list `using` with the method `getRegister`, `getRegister` can select an allocated register that is not in `using`. In this way when the return value is assigned to `reg3` we know it was chosen assuming that the other two previously allocated register could not be victims.

#### 22.1.4 Cache Management III – Adding Consistency

You may not have noticed, but in the last section we worried so much about selecting a victim register that we failed to see what could sometimes (not always) be a major problem – cache consistency. When we select a victim register and allocate it to another variable, the value of that variable will immediately be transferred to the register. But what if the variable that had been allocated to that register was not consistent with the corresponding register value? Maybe some computation had been progressing and the variable value (in the register) changed – now when the register is reallocated that computed value is lost!

The Cache Consistency Principle says we must maintain consistency between the cache and the memory being cached. So, when a victim is selected, the first thing to do is to copy the value in the

register back to the variable about to be cut off from the register – this is called *spilling*. Of course, in the future that variable might be referenced again and another register will have to be allocated, but we will know that the value in the variable is correct. This new twist is easy to implement. When a victim is chosen, before the register is reallocated it will be necessary to generate code that will properly return the register’s value to its associated variable. This can be accomplished by calling a special method `CodeTable.spill`, which would take the register number as argument. The method `spill` would get from `RegisterInfo` the offset of the associated temporary variable in the stack frame and then generate code to copy the value in the register to the stack frame location. Then the register can be officially reassigned.

There is another situation where cache consistency comes in – when there is a function call. In this case the function call is causing a complete switch in program context. The called function’s code will carry out a computation with no idea about the part of the program that made the call. So, it is essential at the time of a function call to spill all register values back to the stack frame. When the called function returns then spilled temporaries could be copied back to registers.

Of course, if we start again from ground zero there will be a lot of register allocation activity right after the return. Maybe it would be better to just copy all values back to the registers that were in use before the call. Then we would carry on as though the call hadn’t happened. Which approach is better?

#### ☛ Activity 89 –

Think about what to do after a return from a function call: start from ground zero or restore all registers right away. Decide which is a better choice and argue your case.

---

### 22.1.5 Cache Management IV – Adding Performance

We have now established a safe and dependable way to guarantee that, during object code generation, requests for registers will be satisfied – i.e., the first three Cache Principles have been addressed. Now we have to consider Cache Principle 4, which requires that the technique for granting register requests will minimize the transfers of data between the cache and memory. It is important to recognize that the allocation of a register to a temporary variable only causes one data transfer – and the data transfer only happens when a value is needed for computation. This is because the language FP has no assignment statements, so there is no way to specify a particular temporary variable to be assigned a second value. So our first allocation strategy causes no more data transfer than necessary; of course programs die if they need too many registers! We will address this problem in a moment, after we revisit consistency.

#### Consistency and efficiency

A second area where data transfer occurs is via the Consistency Principle. Here, we spill a register to its paired variable each time the register is reallocated. If a register has a value that has not changed since it was loaded, then saving it back to its variable is an unnecessary transfer. So we should find a way to cut out the unnecessary transfer. This can be done by keeping track of the state of the register – add a Boolean data member to `RegisterEntry` that is `true` if the register value is changed and not consistent with its paired temporary variable, and `false` otherwise. It

is usual to call this data member the **dirty bit** (a term that comes from cache management in operating systems). Thus, when a register is allocated the bit can be set to **false**, and if the value changes via computation (not loading from memory) set the dirty bit to **true**. Now the method `getRegister` can be modified to check the dirty bit when a register is reassigned – and spill the register value to its paired temporary if the bit is **true**. So when we reallocate registers data transfer is at a minimum.

You may well ask “How do we know when a register is dirty?” For FP the answer is easy. There are two kinds of three-address codes that generate a result. The instruction `MCopy(t1,x)` (where `x` is either a parameter or a literal) transfers the value of `x` to `t1`, but notice, in the case that `x` is a parameter, the value in `x` and `t1` are the same so are consistent – in fact, `t1` doesn’t really need a place in the stack frame. But for other instructions with a result parameter, the value is computed and stored in the result variable (let’s say it is `t1`). This means the value in `t1` would be stored in the stack frame. So when generating code for `MCopy` the temporary variable will be clean, while all other instruction translations will result in a dirty result temporary variable.

### Victim selection and efficiency

There is another efficiency problem lurking in our solution to selecting a victim register in reallocation. We chose the first-in-first-out strategy for convenience. It is easy to implement and it delays when a register is reallocated, seemingly, as long as possible. But in fact, we can do better. Consider an expression such as  $a*(a + b*(a - c)) + (c + a)*b$ . The variable `a` will be loaded into a register right away, and then will be accessed repeatedly through out the computation. This is a small example, but a very large one can have the same characteristic. It would seem that the register paired with `a` should never be reallocated since a new register would have to be reallocated right away to make the value of `a` available.

One thing to recognize is that our first-in-first-out strategy is independent of the ongoing computation. Choosing a victim register via a random number generator would also be computation independent. What is needed is a strategy based on the nature of the ongoing computation. To talk about the problem we need more vocabulary. We are interested in how often a new register request will occur – we use the term *register hit-rate*, or simply *hit-rate*, to refer to the frequency that a needed value is found already in a register. So when a register has to be allocated or reallocated the hit-rate goes down. We need a strategy that maximizes the hit-rate, or minimizes the *miss-rate* (when a value is NOT found in a register).

Imagine you are looking at the register allocation vector discussed earlier. If all registers are currently allocated but we need a register to allocate, which current register would be the best to choose from a hit-rate point of view? The answer has to come from a consideration of the three-address code sequence at hand. If we don’t want the hit-rate to drop we need to choose a register whose currently paired variable won’t be needed for a long time, in fact, for the longest time of all the register pairings. So we will choose the register that won’t be used for the longest time in the future. If we can always make this choice, we are guaranteed to have the highest hit-rate – i.e., it’s an optimal strategy. So when we have a register-miss, the replacement algorithm should determine the optimal victim register and allocate it.

Unfortunately, this approach requires that we look into the future, which isn’t so easy in general, but on computers it is hard because it depends on input data that is not yet known. In fact, this is exactly what makes page replacement in an operating system’s memory manager so difficult –

the need for future knowledge. But think about our situation for a minute. We are choosing this victim register during object code generation; we have the entire program already translated into three-address code – we can look into the future! We can look ahead in the code sequence and determine for all the registers (i.e., their paired temporary variables) when they will be accessed next. From this we can choose a register whose content won't be used for the longest time in the future.

It is often said that easy solutions have easy implementations, while difficult ones have hard implementations. This is a perfect example. If we want to implement the optimal strategy we will need to keep more information about the register/variable pairs in our allocated register array. Remember that when a temporary variable is associated with a register, the variable's symbol table entry is stored as part of the entry for the register in the register allocation vector. So we should be able to add data to the symbol table entry to help with this victim selection algorithm. We can add to `TempEntry` a new integer data member that holds the offset of the next access to the variable – if the variable isn't accessed again it could be simply the length of the code segment being translated.

This new information is used in two ways. When an already free register is allocated to a variable, the algorithm will scan down the three-address code looking for the next index where a reference to the variable occurs; that value is stored in the temporary variable's symbol table entry. When all registers are allocated and a register request comes in, the algorithm simply searches the array for the variable with the largest next use value. That register is allocated to the request.

## Examples

If we translate the expression  $2 * (2 + 3 * 2) / (3 - 2)$  several times, each time using a different number of registers, the benefit of our solution will be evident. Figure 22.2 displays the results of three translations, using 4, 5, and 6.

	Using 4 Registers	Using 5 Registers	Using 6 Registers
1	<code>addu \$fp, \$zero, \$sp</code>	<code>addu \$fp, \$zero, \$sp</code>	<code>addu \$fp, \$zero, \$sp</code>
2	<code>addi \$sp, \$sp, -28</code>	<code>addi \$sp, \$sp, -28</code>	<code>addi \$sp, \$sp, -28</code>
3	<code>li \$8, 2</code>	<code>li \$8, 2</code>	<code>li \$8, 2</code>
4	<code>li \$9, 3</code>	<code>li \$9, 3</code>	<code>li \$9, 3</code>
5	<code>div \$9, \$8</code>	<code>div \$9, \$8</code>	<code>div \$9, \$8</code>
6	<code>mfhi \$10</code>	<code>mfhi \$10</code>	<code>mfhi \$10</code>
7	<code>add \$11, \$8, \$10</code>	<code>add \$11, \$8, \$10</code>	<code>add \$11, \$8, \$10</code>
8	<code>sw \$10, -16(fp)</code>	<code>mult \$8, \$11</code>	<code>mult \$8, \$11</code>
9	<code>mult \$8, \$11</code>	<code>mflo \$12</code>	<code>mflo \$12</code>
10	<code>mflo \$10</code>	<code>sw \$10, -16(fp)</code>	<code>sub \$13, \$9, \$8</code>
11	<code>sw \$11, -20(fp)</code>	<code>sub \$10, \$9, \$8</code>	<code>sw \$8, -8(fp)</code>
12	<code>sub \$11, \$9, \$8</code>	<code>sw \$8, -8(fp)</code>	<code>div \$12, \$13</code>
13	<code>sw \$8, -8(fp)</code>	<code>div \$12, \$10</code>	<code>mflo \$8</code>
14	<code>div \$10, \$11</code>	<code>mflo \$8</code>	<code>addu \$v1, \$zero, \$8</code>
15	<code>mflo \$8</code>	<code>addu \$v1, \$zero, \$8</code>	
16	<code>addu \$v1, \$zero, \$8</code>		

Figure 22.2: MIPS Assembler Code for  $2 * (2 + 3 * 2) / (3 - 2)$  – Bounded Register Allocation

In this table it is interesting to scan from left to right to see the effect of adding an additional register each time. In the first column, when there are 4 registers available we can see in lines 8(10), 11(11), and 13(8) where the register in parentheses has been selected as victim in a register replacement. Since the code in the middle column has one more register available, we would expect to find that only two register replacements take place. This is true as seen on lines 10(10) and 12(8). Finally, in the right-most column there is a single replacement on line 11, replacing register 8. If we were to increase the number of registers to 7, then no replacements would take place.

### In the presence of selection

It's great that we seem to have an optimal solution to our problem. But you have been misled. Our discussions talked about scanning a code sequence; but the generated three-address code can hold segments of code that are not sequential – namely, the translated selection expressions. How will our strategy work when there is selection present? Will victim selection have to change? What we're really asking is whether identifying the next instruction where a register is accessed is impacted by a selection expression in the code sequence to come.

Imagine the sequence of three-address code generated for a large expression that contains selection sub-expressions. The generated code will begin, let's imagine, with code for ordinary arithmetic expression evaluation. At some point we will arrive at the code generated for the first selection expression. This will be followed by arithmetic expression evaluation and continue in this way, alternating between ordinary arithmetic code and selection code. We will refer to the segments of arithmetic expression as *blocks*. If we think about the true and false branches of a selection, the branches can also be blocks, unless they contain nested selection. But the idea is clear. Any segment of generated code bounded by the beginning of code, end of code, or branching code (or associated labels) will be blocks. So one approach to register allocation in the presence of selection would be to do register allocation for each block as it occurs – and by that we mean that register allocation would begin over on entering a block and be cleaned up on leaving a block. Of course, this is exactly what we do when there is no selection – there is only one block.

The disadvantage of this strategy is that, because the code is broken up into smaller blocks, the advantages of our earlier replacement strategy will be less apparent. In order to better understand the dynamics of the problem we'll look at the following expression.

```
a*(b + a) + ( if a < c then (a + b*(a + b)) else (a - c*a) endif )*(a + c)
```

First let's consider the block structure of the expression. The first block starts at the beginning of the expression and continues through the selection condition 'a < c', then the first branch occurs. So the second block starts just after the **then** and continues until the branch around the false branch. The third block starts just after the **else** and continues until the **endif** (which would generate a label code). The fourth and last block starts after **endif** and continues to the end of the expression. The following shows the blocks underlines.

```
a*(b + a) + ( if a < c then (a + b*(a + b)) else (a - c*a) endif )*(a + c)
```

So what will the impact be of doing register allocation by blocks. First, by the end of the initial block we will we will have filled several registers with the temporary variables in the corresponding generated code – for example, there will be temporary variable **t1** that will be associated with

a register and hold the value of the parameter `a`. This register/variable pairing should be used throughout the program. But if we start fresh with each block, then at the beginning of each block we will have to start over and allocate again registers to the temporary variables in the the new block. Remember that in sub-expression elimination each occurrence of `a` will be associated with `t1` in the in the three-address code. So we could easily keep the same register allocation, but with selection this becomes difficult to manage. An important thing to notice here is that when selecting a victim register, the search for the best victim will extend only to the end of the block, no further.

Our final piece of the register allocation puzzle is that dealing with performance. What we want to accomplish in register allocation is to make sure that generated run-time activity does not impose too much burden on the computation time. In the three preceding sections we have talked a bit about performance. Our victim selection was really all about performance and maximizing the hit-rate. But there are other things we can do to improve performance.

### Other performance enhancements

The optimal selection of a victim register is a very important performance strategy, but there are others we can find. You may have noticed that once the registers are full they will remain full. That is, we have no way of indicating that a register/variable binding is no longer needed. If we can break register/variable bindings when they are no longer needed it will increase the chance of free registers available when a request comes in.

The way to tackle this problem is to determine for each variable where its last reference is in the three-address code. We can do this by reading backwards through the three-address code and keeping track for each variable the first index where it is encountered. We can create another integer data member in `TempEntry` that would hold the index of the last access to a variable. The value can be determined when a variable is associated with a register. With this value in place, when the victim selection algorithm is running it can check, for each register, if the register should be freed. By running through the whole register list all possible freeings can take place.

Another situation that can impact performance is the consistency work done in the last section. We talked about restoring a register value to its associated variable when the register has been selected as a victim. But this restoring of register values is indiscriminate – we should, in fact, only copy back register values that have changed. We can tell when a value is changed in a register if a three-address code has as result variable the variable associated with the register. So a scan of the intermediate code will reveal if a register value needs to be copied back and we can record that fact in the temporary variable’s symbol table entry – call the new Boolean data member `dirty` and check it when a register is chosen as a victim.

### ☛ Activity 90 –

1. Construct the MIPS assembly code, as in Figure 22.2, that would result if there are just 3 registers for allocation. Note that this does not mean to use the translation resulting from the FP compiler from Chapter 21. It used 3 registers, but in a different way.

Also note that since we are translating three-address code, we could not get by with less than 3 registers.

2. First, translate the following expression into three-address code.



```
a*(b + a) + ( if a < c then (a + b*(a + b))
               else (a - c*a) endif )*(a + c)
```

Then apply the register allocation strategy we discussed for selection. Don't actually generate MIPS code, just focus on register allocations demanded by the three-address code.

---

## 22.2 Implementing Register Allocation

Though we described two separate strategies for register allocation in the previous section, we are really only interested in the bounded cache version. This is the one with the most flexibility and we know that the register replacement strategy will have the smallest effect on performance. However, the two strategies highlighted the two basic implementation problems for implementing register allocation: the search for free registers and the selection and replacement of a register when none are free. A third problem was addressed at the end when we described a strategy for freeing registers when they are no longer needed. In this section we will look at the implementation requirements for the solutions to these three problems.

### 22.2.1 Register Allocation

We already have a basic object code generator in the form of the class `ObjectCodeTable` developed for the simple FP<sup>+</sup> compiler. What we need to do is add a mechanism to track register usage and then to integrate its use into the existing code generation method, `ObjectCodeTable.generateCode`.

#### Register tracking

To keep track of register allocations and to facilitate the selection of a register to allocate we will implement a new class named `RegisterInfo`. The class will have an array of integer values that will represent the register/variable bindings, where index represents the register number and the value at the index is the variable's address.

A couple of important details. First, all variables, whether parameters or temporaries, are stored on the stack and addressed relative to the `fp` register. These offsets will always be positive for parameters and negative for temporaries (in fact  $< -1$  for temporaries). So to denote a free register we will use the value zero. Second, the index in the array isn't the actual register number, but rather an offset. We will assume that the cache of registers always starts at register `t0`, which is register #8. So when index  $n$  is allocated it is really the register  $n + 8$  that will be used.

The layout for the class `RegisterInfo` is shown in Figure 22.3. There are two class constants, one to indicate the MIPS number of the first allocatable register (`REG1`) and the other to indicate the number of allocatable registers (`MAXREGS`). The class has two instance variables: `registers` is the integer array for tracking allocations and `oct` is a reference to the externally created object code table. The variable `oct` is present to facilitate the small bit of code generation that is most easily accomplished in this class.

```

public class RegisterInfo {

    private final int REG1    = 8;
    private final int MAXREGS = 16;
    private int[] registers = new int[MAXREGS];
    ObjectCodeTable oct;

    public RegisterInfo(ObjectCodeTable oc)

    public int getSourceReg(int psn, int offset) throws Exception
    public int getTargetReg(int psn) throws Exception
    private int search4(int val)
    public void saveRegisters()
    public void zeroRegisters()
}

```

Figure 22.3: Interface for RegisterInfo

The class methods are relatively straightforward, but their functionality must be considered in the context, not of a single expression or function definition, but in the context of a complete FP<sup>+</sup> program, which may have many function definitions. A short description of each method follows.

#### RegisterInfo

The constructor takes a single parameter, the reference to the object code table, which will be assigned to the instance variable `oct`.

#### getSourceReg

This method is used to request a register that will be used to load a value from memory. The two parameters represent the word offset of the variable on the stack and the offset is the actual byte offset. This method searches for a free register and, if found, stores the position in the `registers` array and generates a load instruction using the allocated register's number and the offset parameter. The allocated register number + 8 is returned. If there is no free register available then an exception is thrown.

#### getTargetReg

This method is the same as that above except that, since we will use this register as a target, there is no need for loading a value.

#### search4

This private method is used by the two previous methods to search the array `registers` for a free register. The index is returned if one is found, or -1 if none is found.

#### saveRegisters

This method is used immediately before generating a MIPS function call instruction. The method traverses the array `registers`, generating a store instruction for each allocated register and resets each array value to 0.

### zeroRegisters

This method simply resets each position in the array `registers` to 0. This is used to zero out the register allocations when the `Mfreturn` and `Mhalt` instructions are translated. This will guarantee that when each new function's code is generated that the array `registers` will start out in a clean state (all 0's).

One consequence of the last two methods is important to understand. When a function call is made in an  $FP^+$  expression it is important to save the current registers so that the called function can use all registers as its generated code sees fit. When control returns from that call we know that the method `zeroRegisters` will be called. That means at the return, the registers that had earlier been allocated will no longer be allocated. Instead, the function code will have to bring back in any variable values it needs to reference again. In this way we do the reloading only as needed and under the control of the existing register allocation scheme.

### Code Generation Modifications

Object code generation is managed in the class `ObjectCodeTable`, which was defined in its simple form in Section 21.2. In this section we will discuss the additions and modifications to that definition necessary to implement register allocation. Figure 22.4 shows the layout of the class `ObjectCodeTable` reflecting changes for register allocation.

```
public class ObjectCodeTable {

    private final int MAX = 1000;
    private String[] oct = new String[MAX];
    private int nextPsn = 0;
    private RegisterInfo regInfo;

    public ObjectCodeTable()
    public void genCode(String op)
    public String toString()
    public void generateObjectFile(PrintWriter out)
    public void generatePreamble()
    public void generateCode(OpCode op) throws Exception
}

```

Figure 22.4: Interface for `ObjectCodeClass`

The first change is the addition of the instance variable `regInfo`, which is an instance of the class we just discussed. The code for the constructor of the class is a bit tricky since, as you will recall, the `RegisterInfo` constructor expects as parameter a reference to an `ObjectCodeTable` object. Fortunately, the tricky situation has a simple solution using the `this` self-reference variable – here is the code.

```
public ObjectCodeTable() {
    regInfo = new RegisterInfo(this);
}

```

The only other changes to the class `ObjectCodeTable` are contained in the method `generateCode`, which implements the translation of each different three-address instruction type. We will look at a couple of examples from which you should be able to complete the method.

If you look back at the simple code for `ObjectCodeTable.generateCode`, which was described in part in Section 21.2.2, you can anticipate the changes to be discussed by locating all the places where registers are used. At each position where a register is used we must add code to get a register number from `regInfo`. At each point where we have to request a register we have to be aware if the variable involved is acting as a source of data or as a target for a resulting value. We will look at the new translations for a few sample three-address instructions.

### Operations with three operands

Here is an example. First we look at the original code for a three address operation, reproduced here (also see page 391).

```
val = ((TempEntry)(op.getAdr2())).getPosition();
genCode("lw $t0, " + val + "($fp)");
val = ((TempEntry)(op.getAdr3())).getPosition();
genCode("lw $t1, " + val + "($fp)");
genCode(mipsOp + " $t2, $t0, $t1");
val = ((TempEntry)(op.getAdr1())).getPosition();
genCode("sw $t2, " + val + "($fp)");
```

In looking at this code we see that since each of the first two registers, `t0` and `t1`, have a load operation performed on them, the two registers are allocated to source variables. On the other hand register `t2` is used to store the operation's result to a target variable. So the resulting code our new version of `generateCode` will have the following structure.

```
val = ((TempEntry)(op.getAdr2())).getPosition();
r2 = regInfo.getSourceReg(val, (-4)*(val+1));
val = ((TempEntry)(op.getAdr3())).getPosition();
r3 = regInfo.getSourceReg(val, (-4)*(val+1));
val = ((TempEntry)(op.getAdr1())).getPosition();
r1 = regInfo.getTargetReg(val);
genCode(mipsOp + " $" + r1 + ", $" + r2 + ", $" + r3);
```

Two things to notice here. First, after determining the position value of the variable we call either `getSource` or `getTargetReg` to obtain the register numbers that will be used as arguments to the operation. Second, the new code contains only one call to `genCode`. This is because the load instructions are automatically generated by the calls to `getSourceReg`. Third, notice that the final statement, the call to `genCode`, is constructed in a different way, since now we must convert each register number to a string to form the parameter to `genCode`. Finally, there is no store instruction here. The value computed will remain in the assigned register until it becomes necessary to copy it back to memory. If this value is the final value computed, then it will be copied from this register to the register `v0` as the return value of a function.

## Mcopy

This instruction takes two parameters, the first is always a temporary variable but the second can be a parameter or a temporary. We saw in the simple case how this code must be generated (see page 392). That code must be modified so that `getSourceReg` is called in each branch of the selection statement and then the first parameters position can be determined, a target register allocated, and the MIPS instruction for performing the copy. The code follows.

```

if ( (op.getAdr2()).getClass().getName().equals("symbolTable.ParameterEntry") ) {
    val = ((ParameterEntry)(op.getAdr2())).getPosition();
    r2 = regInfo.getSourceReg((-1)*val, 4*val);
}
else {
    val = ((TempEntry)(op.getAdr2())).getPosition();
    r2 = regInfo.getSourceReg(val, (-4)*(val+1));
}
val = ((TempEntry)(op.getAdr1())).getPosition();
r1 = regInfo.getTargetReg(val, (-4)*(val+1));
genCode("add $" + r1 + ", $zero, $" + r2);

```

## Mfcall

The code generated for the function call instruction is virtually the same as for the simple code generator – see the simple code on page 392. In this case we need to worry about just two additional things. First, at the end of the code we must save away the return value of the function call. To do that we must allocate a register and then save the value (returned in register `v0`) to the allocated register. The code below does exactly that at the end. Notice again, however, that having stored the value in an assigned register that the value is not then copied back to memory.

```

val = ((FunctionEntry)(op.getAdr3())).getNumParams();
genCode("addi $sp, $sp, -8");
genCode("sw $fp, 4($sp)");
genCode("sw $ra, 0($sp)");
genCode("addi $fp, $sp, 4");
regInfo.saveRegisters();
genCode("jal " + ( (FunctionEntry)(op.getAdr3()) ).getName() );
genCode("lw $fp, 4($sp)");
genCode("lw $ra, 0($sp)");
genCode("addi $sp, $sp, " + (8 + val*4));
val = ((TempEntry)(op.getAdr1())).getPosition();
r1 = regInfo.getTargetReg(val, (-4)*(val+1));
genCode("add $" + r1 + ", $zero, $v0");

```

The second thing we notice in this code sequence is the call to `saveRegisters`. This insures that when the code for the called function begins executing, registers allocated in that code will not overwrite unsaved values from before the call. A related thing that should be remembered is that when the called function returns, it is up to that code to make sure the registers are once again.

This means that when a target register is allocated on the second to last line, there will be a register available!

### Mfreturn

The code generated for a return instruction is very similar to the code generated in the simple case. The two problems to address in this code relate to zeroing out the registers and determining the location of the return value, which we know must be copied to register `v0` – we know the variable name for the return value but must call `getSourceReg` to allocate a register.

```
val = ((TempEntry)(op.getAdr1())).getPosition();
r1 = regInfo.getSourceReg(val, (-4)*(val+1));
genCode("add $v0, $zero, $" + r1);
val = ((IntegerEntry)(op.getAdr2())).getValue();
genCode("addi $sp, $sp, " + (4*val));
regInfo.zeroRegisters();
genCode("jr $ra");
```

In the second line we determine which register contains the return value and, in the second to last line, we call `zeroRegister` to insure that the registers are all clean (and unallocated) before returning.

This is a good point to remind ourselves that when we call `getSourceReg` or `getTargetReg` a register for this variable may already have been allocated earlier (much earlier perhaps), in which case that register number is simply returned; if a register hasn't already be allocated then the methods search the array `registers` to find a free register to allocate.

## 22.2.2 Register Replacement

In order to add register replacement capability to our code generator we will have to make some additions to both `RegisterInfo` and `ObjectCodeTable`. To `RegisterInfo` will be added variables and methods facilitating recovery in case no register is free when register allocation is attempted. The primary change to `ObjectCodeTable` will be the capability to tell the register allocation methods (`getTargetReg`, `getSourceReg`) which registers are off limits for replacement. We need this in translating the three-address operations (`Madd r s t`, for example) so that a register allocated for the variable `s` isn't later chosen for replacement when we need a register for the result variable `r`. The following sections will illustrate the implementation details.

### Modifications to the class `ObjectCodeTable`

The only change to `ObjectCodeTable` is to the method `generateCode`. What we have to do is to keep track of, for each three-address instruction being translated, which registers have been allocated before each call to `getTargetReg` or `getSourceReg` — that is, which registers have been allocated *in translating the particular instruction*. So at the beginning of each call to `generateCode` we must establish an empty list (of register numbers) and then for each instruction, add to the list each register allocated. Obviously we only have to record two registers since a third register request must be the last. We illustrate the strategy by displaying a fragment of the code for the method.

```

public void generateCode(OpCode op, int psn) {
    ...stuff clipped...
    int[] inuse = { 0, 0 };
    code = op.getCode();
    switch (code) {
    case Madd:
        val = ((TempEntry)(op.getAdr2())).getPosition();
        r2 = regInfo.getSourceReg(val, (-4)*(val+1), inuse, psn); inuse[0] = r2;
        val = ((TempEntry)(op.getAdr3())).getPosition();
        r3 = regInfo.getSourceReg(val, (-4)*(val+1), inuse, psn); inuse[1] = r3;
        val = ((TempEntry)(op.getAdr1())).getPosition();
        r1 = regInfo.getTargetReg(val, inuse, psn);
        genCode("add $" + r1 + ", $" + r2 + ", $" + r3);
        break;
    ...stuff clipped...
    case Mdiv:
        val = ((TempEntry)(op.getAdr2())).getPosition();
        r2 = regInfo.getSourceReg(val, (-4)*(val+1), inuse, psn); inuse[0] = r2;
        val = ((TempEntry)(op.getAdr3())).getPosition();
        r3 = regInfo.getSourceReg(val, (-4)*(val+1), inuse, psn); inuse[1] = r3;
        val = ((TempEntry)(op.getAdr1())).getPosition();
        r1 = regInfo.getTargetReg(val, inuse, psn);
        genCode(mipsOp + " $" + r2 + ", $" + r3);
        genCode("mflo $" + r1);
        break;
    ...stuff clipped...
    case Mparam:
        genCode("addi $sp, $sp, -4");
        val = ((TempEntry)(op.getAdr1())).getPosition();
        r1 = regInfo.getSourceReg(val, (-4)*(val+1), inuse, psn);
        genCode("sw $" + r1 + ", 0($sp)");
        break;
    ...stuff clipped...
    }
}

```

The structure of the three instruction translations above illustrate how the technique is applied to three-address, two-address, and one-address instructions. Notice that when a particular instruction is translated (i.e., its corresponding `case` is executed) the array `inuse` has initially the value 0 in each position. This means that when `getSourceReg` is called in `case Mparam` and `inuse` is passed, it still has the value 0 in each position. In fact this is true in the first register allocation request in each example above — i.e., no registers are out of bounds. If there is a second register allocation request, then before the request is made the number of the first register to be allocated must be assigned to one of the positions in `inuse`. In this way we insure that the register will not be replaced. So we see this idea in the code for `Mdiv` and `Madd` — i.e., how to handle a request for a second register. Finally in the `Madd` case, before the third register request is made we assign the second register number to the other position in `inuse`. This will insure that the third request will

not replace either of the two already allocated.

The method `generateCode` has one more parameter – `psn`. This parameter indicates the position in the intermediate code table of the current instruction being translated. This parameter is passed as a parameter to the methods `getSourceReg` and `getTargetReg`. This parameter is discussed in the following section.

### Modifications to the class `RegisterInfo`

These modifications are more extensive. The idea is to replace the throwing of the exception by a call to a method that will carry out the register replacement and return the chosen register. That part is easy. The more interesting aspect of the replacement implementation is determining the best register to replace. The layout of the modified version of `RegisterInfo` follows.

```
public class RegisterInfo {

    private final int REG1 = 8;
    private final int MAXREGS = 16;
    private final int NOTUSED = -1;

    private int[] registers = new int[MAXREGS];

    private ObjectCodeTable oct;
    private CodeTable ct;

    public RegisterInfo(ObjectCodeTable oct, CodeTable ct)

    public int getSourceReg(int loc, int offset, int[] inuse, int psn)
    public int getTargetReg(int loc, int[] inuse, int psn)
    private int search4(int val)
    public void saveRegisters()
    public void zeroRegisters()
    private int findReg(int[] inuse, int psn)
    private int getBestReg(int [] regs)
}
```

There are a few things to pick up initially from this interface for the class. First, the constructor now has two parameters, the object code table and also the original intermediate code table. The second is needed so that we can determine which is the best register to replace (the optimal replacement strategy). Second, the methods `getSourceReg` and `getTargetReg` now have the “inuse” parameter – we’ve discussed the rationale for that parameter in the previous section – and a final parameter `psn` which indicates the position in the intermediate code table of the instruction currently being translated. The final two methods are new additions to the class. They collaborate to find the best register to replace. Now we will take a closer look at some of these method implementations.



### getSourceReg and getTargetReg

These two methods are close enough that we will examine only `getSourceReg`. In this method we must provide an alternative action to throwing an exception, as was done in the previous version of this method. The method is the same except when no register is found from the call to `search4`. All we have to do is replace the throw statement with a call to the method `findReg`. Here is an implementation for the method.

```
public int getSourceReg(int loc, int offset, int[] inuse, int psn) {
    // allocate a register and assign to variable at stack position 'psn'
    int reg = search4(loc);
    if (reg != -1) return reg+8;
    reg = search4(0);
    if (reg == -1) { // no register is free
        reg = findReg(inuse, psn);
    }
    // reg has now been allocated
    oct.genCode("lw $" + (reg+8) + ", " + offset + "(fp)");
    registers[reg] = loc;
    return 8+reg;
}
```

To understand this implementation we need to look at the method `findReg`.

### findReg

The purpose of this method is to find the optimal register to replace. This is done in two steps. First, the intermediate code table is traversed to determine for each register when the associated variable would be next referenced – these locations are recorded in an array. The resulting array is then passed to the method `getBestReg`, which will return the appropriate register number, and `findReg` returns the same register number as its return value.

`findReg`'s primary job is to produce the array described above. The difficulty is that we need to be able to process each type of three-address instruction – that means we must use a switch statement to filter out the different instruction types. For each instruction type we look at the variables referenced in the instruction and search for each one in the `registers` array (which contains the register/variable bindings). The method, with some of the cases omitted, is shown in Figure 22.5. The code for the missing cases should be easily reconstructed following the example of the cases that are presented.

### getBestReg

Not surprisingly, this method is a bit more straightforward. It takes as parameter the array produced in `findReg`. It searches that array first for a register that won't be referenced again and uses the first one if one exists. If all registers will be referenced again then it searches the array for a register whose next reference point is the largest. Once the target register is identified a MIPS instruction is generated to store the register's content back to its memory location. The code in Figure 22.6 clearly implements this strategy.

```

private int findReg(int[] inuse, int psn) {
    int[] regs = new int[MAXREGS];
    for (int j = 0; j < MAXREGS; j++) regs[j] = NOTUSED;

    if (inuse[0] != 0) regs[inuse[0]-8] = psn; // means it won't be chosen
    if (inuse[1] != 0) regs[inuse[1]-8] = psn; // means it won't be chosen

    int last = ct.getCodePosition();
    int p, r;
    OpCode op = null;
    int i = psn + 1;
    while (i < last && ( ct.getOpAt(i).getCode() != OpCode.OpName.Mhalt &&
        ct.getOpAt(i).getCode() != OpCode.OpName.Mfreturn )) {
        op = ct.getOpAt(i);
        switch (op.getCode()) {
        case Madd: // three variable references
            r = ((TempEntry)(op.getAdr1())).getPosition();
            p = search4(r);
            if (p >= 0 && regs[p] == NOTUSED) regs[p] = i;
            r = ((TempEntry)(op.getAdr2())).getPosition();
            p = search4(r);
            if (p >= 0 && regs[p] == NOTUSED) regs[p] = i;
            r = ((TempEntry)(op.getAdr3())).getPosition();
            p = search4(r);
            if (p >= 0 && regs[p] == NOTUSED) regs[p] = i;
            break;
        case Mnegate: // two variable references
            r = ((TempEntry)(op.getAdr1())).getPosition();
            p = search4(r);
            if (p >= 0 && regs[p] == NOTUSED) regs[p] = i;
            r = ((TempEntry)(op.getAdr2())).getPosition();
            p = search4(r);
            if (p >= 0 && regs[p] == NOTUSED) regs[p] = i;
            break;
        case Mfcall: // only one variable reference
            r = ((TempEntry)(op.getAdr1())).getPosition();
            p = search4(r);
            if (p >= 0 && regs[p] == NOTUSED) regs[p] = i;
            break;
        case Mallocate: // no variable references
            break;
        }
        i++;
    }
    return getBestReg(regs);
}

```

Figure 22.5: Abbreviated Code for RegisterInfo.findReg

```
private int getBestReg(int [] regs) {

    // is there a register that won't be used again?
    boolean found = false;
    int reg = 0;
    while (!found && reg < MAXREGS)
        if (regs[reg] == NOTUSED) found = true;
        else reg++;

    if (reg == MAXREGS) { // all will be used in future
        // select the one that won't be used soon
        reg = 0;
        for (int i = 1; i < MAXREGS; i++)
            if (regs[i] > regs[reg]) reg = i;
    }

    // save away the register content back to stack location
    if (registers[reg] > 0)
        oct.genCode("sw $" + (reg+8) + ", " + ((registers[reg]+1)*(-4)) + "(fp)");
    else
        oct.genCode("sw $" + (reg+8) + ", " + (registers[reg]*(-4)) + "(fp)");

    return reg;
}
```

Figure 22.6: Code for RegisterInfo.getBestReg

```

private int setLastUsed() {

    int last = ct.getCodePosition();
        // this will be 1 greater than the index of the last instruction
        // in the entire intermediate code table --- just a stopper!
    OpCode op = null;
    int i = psn;
    while (i < start && ( ct.getOpAt(i).getCode() != OpCode.OpName.Mhalt &&
                        ct.getOpAt(i).getCode() != OpCode.OpName.Mfreturn )) {
        op = ct.getOpAt(i);
        switch (op.getCode()) {
        case Madd: // three variable references
            ((TempEntry)(op.getAdr1())).setLastUsed(i);
            ((TempEntry)(op.getAdr2())).setLastUsed(i);
            ((TempEntry)(op.getAdr3())).setLastUsed(i);
            break;
        case Mnegate: // two variable references
            ((TempEntry)(op.getAdr1())).setLastUsed(i);
            ((TempEntry)(op.getAdr2())).setLastUsed(i);
            break;
        case Mfcall: // only one variable reference
            ((TempEntry)(op.getAdr1())).setLastUsed(i);
            break;
        case Mallocate: // no variable references
            break;
        }
        i++;
    }
}

```

Figure 22.7: Abbreviated Code for `RegisterInfo.freeRegisters`

### 22.2.3 Freeing Registers

The final element in the register allocation strategy is the freeing of registers once their bound variables will no longer be referenced in the current expression. One way to handle this is to traverse the intermediate code and record, in the `TempEntry` and `ParameterEntry` objects encountered, the index of the positions where the variables are seen. This means for a variable that is used a lot, that its “last used” entry will change often – but in the end it will have the appropriate value.

To facilitate this implementation we must first add a new instance variable, call it `lastUsed`, to the two symbol-table classes for variables – `TempEntry` and `ParameterEntry`. The algorithm for the search is similar to that of `findReg` but we must keep track of different information. We will not care about the array `registers` since this search will be done before we start generating code for an expression; similarly we won’t need the array `inuse` or to record our results in an array like `regs`, because we record the positions in each variable’s symbol table entry, which is part of the instruction. In Figure 22.7 we display the code for the method `RegisterInfo.freeRegisters`.

**Compiler Project**  
**– Compiling the IP Language –**



## Chapter 23

# Project Overview

In Tutorial I and II we discussed the basic implementation strategies for language recognition as well as for code generation. In the just completed Interlude II, we added techniques for linear code generation and for intermediate and object code generation and optimization. These experiences, grounded in the PDef and FP implementation projects, provide the necessary background to take on the implementation of a compiler for the imperative language IP, which was described earlier in Section 23.1.

This section will give an overview of the project and lay out the development phases that will lead to the completion of the IP compiler. The short chapters which follow will carefully describe each development phase, itemizing those activities that can be carried out without further direction and discussing new ideas necessary to complete the particular phase.

### 23.1 The Language IP

IP is an imperative language and is the most sophisticated of the three languages language introduced. An implementation of IP, in the form of a compiler, will be the focus of Tutorial III – the last part of this text. The purpose of this section is to present both informal and formal descriptions of IP, though a formal description of the IP semantics will wait for Tutorial III.

#### 23.1.1 Informal Definition of IP

This is a relatively simple language which follows the layout of a C program. That is, it is a sequence of function definitions with one of them being named `main`. Function definitions can have parameters, which can be either value or reference. The language has two data types, `int` and `float`, and the standard imperative statements for assignment, selection, repetition, and function call.

A simple program can consist of a single function but it must be named `main`. Here is an example which also illustrates the imperative statements (except for function call).

```
int gcd (int a, int b)
    while (b%a != 0)
        int tmp = b%a;
```

```

        b = a;
        a = tmp;
    endwhile;
    return a;
endfn

void main()
    int x; x = 10;
    int y; y = 32;
    y = gcd(x,z);
    print y;
endfn

```

There are important features of the language that are simplifications of corresponding features of other imperative languages such as C. First, variable declarations are not allowed at the highest program level, as is possible in C. Second, all variable declarations must appear at the top of a function definition – i.e., declarations are not treated as executable statements, as is done in C. Finally, the condition expressions in the repetition and selection statements have the same structure as in FP, that is, they are restricted to a comparison operator with two arithmetic expressions as arguments.

### 23.1.2 Formal Definition of IP

#### Alphabet of IP

The IP alphabet is the following set of characters.

```

alphabet = { a ... z
            A ... Z
            0 ... 9
            =
            + * - / %
            < > !
            ^
            (
            )
            ;
            .
            }

```

#### Lexical Level of IP

```

scolonT    ;
commaT     ,
lpT        (
rpT        )

```



```

refT      ^
assignT   =
addOpT    +
subOpT    -
multOpT   * | / | %
compOpT   < | <= | >= | > | == | !=
ifT       if
thenT     then
elseT     else
endiT     endif
whileT    while
endwT     endwh
endfT     endfn
printT    print
returnT   return
typeT     void | int | float
identT    [a-zA-Z]+
intT      0 | [1-9][0-9]*
floatT    (0 | [1-9][0-9]*).[0-9]+

```

### Grammar Level of IP

```

<Program> ::= <FnDefList>

<FnDefList> ::= <FnDef> [ <FnDefList> ]

<FnDef> ::= typeT identT lpT [ <FParamList> ] rpT <Body> endfnT

<FParamList> ::= <ParamDecl> [ commaT <FParamList> ]
<ParamDecl> ::= typeT [ refT ] identT

<Body> ::= <Block> [ <Return> ]
<Block> ::= <DeclList> <StList>

<Decl> ::= typeT identT
<DeclList> ::= empty | <Decl> colonT <DeclList>

<Statement> ::= <Assignment> | <Selection> | <While> | <FnApp> | <Print>
<StList> ::= empty | <Statement> colonT <StList>

<Assignment> ::= identT assignT <Exp>
<Selection> ::= if lpT <BExp> rpT thenT <Block> [ elseT <Block> ] endifT
<While> ::= while lpT <BExp> rpT <Block> endwhT
<FnApp> ::= identT lpT [ <ExpList> ] rpT
<Print> ::= printT <Exp>

```

```

<Return>      ::= returnT <Exp>

<BExp>        ::= <Exp> compOpT <Exp>

<FnApp>       ::= identT lpT <ExpList> rpT
<ExpList>     ::= <Exp> [ commaT <ExpList> ]
<Exp>         ::= subOpT <Factor>
               ::= <FnApp>
               ::= <Term> { ( addT | subT ) <Term> }
<Term>        ::= <Factor> { multT <Factor> }
<Factor>      ::= int
               ::= floatT
               ::= identT
               ::= lpT <Exp> rpT

```

### Semantics of IP

The semantics of IP is complex but should be familiar to anyone who has used an imperative language such as Java or C++. In this brief section we will discuss the static semantics of IP and describe informally its dynamic semantics.

The static semantics of a language is tightly bound to the data types of the language. In the case of IP there are three: `void`, `int`, `float`. The type `void` can be used in only one situation in IP, as the return type of a function. In this context the `void` return type designates a function as not returning a function value – i.e., the function is treated as a procedure and is called as a statement rather than as part of an expression. So the `void` type is not a type at all but a token that designates a procedure.

The type structure of IP, then, is determined by the types `int` and `float`, which behave in IP just as they do in PDef-*lite*. Since there is no `char` type in IP we will take the following table as defining the assignment compatibility for IP.

	int	float
int	C	C
float	NC	C

Remember that such a table is read left column to top row, i.e., `int` is assignment compatible(**C**) with itself and `float` while `float` is compatible with itself but not (**NC**) with `int`.

There are two situations in which assignment compatibility come into play: assignment statements and function arguments. Assignment statements follow the same rule as for PDef-*lite*. For function arguments, the type of an argument must be assignment compatible with the type of the corresponding formal parameter.

There are also two important rules that determine what kind of function arguments are permitted. If a function parameter is passed by value then its argument can be any expression whose type satisfies the assignment compatibility rule. If a function parameter is passed by reference, on the other hand, the only argument permitted is a variable whose type satisfies the assignment compatibility rule.

argument 1	argument 2	result type
int	int	int
float	int	float
int	float	float
float	float	float

Figure 23.1: Result Types for Mixed-mode Arithmetic

IP allows mixed mode arithmetic expressions. In FP there is only one type and consequently every expression must yield a value of that same type. But when an expression can mix values of type `int` and `float` there comes the question, "what is the type of the value computed?" In fact at a simpler level, if we add two values of different types what is the value of the result? This problem is addressed by having a rule to dictate the type resulting from all possible operations and operand types. For IP that rule is specified by the following table.

The dynamic semantics of IP is complex, though it should be familiar to anyone who has programmed in an imperative programming language such as Java or C++. The various statement types of IP behave in the same way as corresponding statements of other imperative languages; so assignment, repetition, selection, and function call statements behave in the familiar way. One important point that can be made here deals with the impact of reference parameter passage. In the context of IP reference parameter passage means that the value of an actual (reference) parameter can change value during execution – remember that the actual parameter is in the calling environment. This means that if a function call is part of an expression then other elements in the expression can change value during evaluation of the expression.

Consider, for example, the following simple IP program.

```
int f(int & a)
    a = a * 2;
    return a*a;
endfn
void main()
    int x = 10;
    print f(x) + x;
endfn
```

At first glance we might believe that when executed this program will print the value 410. But remember that `x` is a reference argument. When the first line in the function is executed, the value of `a` changes to 20. But since `a` is a reference parameter, the value of its argument also changes to 20. This means that the value of `x` is 20 when it is added to the value returned by the call `f(x)`. In other words, the program prints the result 20. We will see in the implementation of an IP compiler in Tutorial III how this functionality is implemented. It is also good to understand why in this situation the operation '+' is not commutative. Do you see what value is returned if the print statement is changed to `print x + f(x)`?

A complete description of the dynamic semantics of IP is presented in Tutorial III.

## 23.2 Project Overview

The structure of this tutorial parallels the structures of Tutorials I and II and consists of four phases: the front end (everything up through static semantic checking), intermediate code generation, common subexpression elimination for IP, and object code generation and optimization. While the first phase requires no new techniques it does require care and ultimately comprises a majority of the compiler's code. The other three phases draw heavily on the techniques discussed in Interlude II. For intermediate code generation we will expand on the three-address code used in the FP<sup>+</sup> implementation (Chapter 20.1.2) and extend the common subexpression elimination strategy (Section 20.2) for expressions to cover statements in IP. For object code generation we will delve more deeply into the functionality of the MIPS processor, in particular its floating point capabilities (see Chapters 21 and 22 for background).

In each chapter the particular software development activities will be described in numbered Activities. It is important to remember the value of careful planning and testing. Writing a suite of IP programs to test the emerging compiler is a valuable way to gain a deeper insight into design problems that arise. A good test suite is also useful in regression testing, which can expose errors caused by the addition of new functionality.

The remainder of this short introductory chapter provides a preview of the activities of each of the four development phases.

**Syntax and Static Semantics of IP** IP presents no new syntactic problems – you have seen all the necessary techniques in the earlier tutorials. A careful study of the grammar will be necessary, of course, in order to design the syntax tree class hierarchy. For symbol table design and integration it will be necessary to evaluate the scope rules of IP and then determine the appropriate places in the syntax tree hierarchy to locate local symbol tables.

Static semantic checking for IP is interesting in part because it combines characteristics of PDef and FP. In PDef you have encountered assignment statements and mixed mode arithmetic expressions. FP has function definitions but only one type. In IP we integrate the mixed mode arithmetic of PDef with the function definitions of FP to get functions that can have mixed parameter types. Checking the IP static semantics requires adapting the semantic checking techniques from the two previous tutorials. It is critical that the static semantic requirements for IP (see Section 23.1) be studied carefully before embarking on the design and implementation of this phase.

**Intermediate Code Generation** The IP intermediate code generation is based on three-address code. But the complexity of IP, compared to that of FP<sup>+</sup>, requires additions and modifications to the three-address code we discussed in the FP<sup>+</sup> context. IP brings into the code generation process three new issues:

1. imperative statements in the form of selection, repetition, and procedure call,
2. reference parameters for procedures, and
3. mixed mode arithmetic involving integer and floating point operations.

Because attacking these three issues simultaneously might be overly complex, the intermediate code generation process is divided into three phases as follows:

**Phase 1:** Define and implement the intermediate code generation for a subset of IP that includes all the statement structures of IP, but limited to integer values and value parameters for functions and procedures.

**Phase 2:** Augment the implementation of Phase 1 with the capability for reference parameters for procedure calls.

**Phase 3:** Augment the implementation of Phase 2 with the capabilities associated with mixed-mode arithmetic.

The idea is that the completion of each phase will result in a working translator for the particular sub-language. The work done in one phase should not have to be undone in the next. Of course, careless planning may mean that changes are necessary – but the ideal would be that each phase would result in additions and not changes.

**Optimization: Common Subexpression Elimination** As we saw in Chapter 20, common subexpression elimination is an optimization technique that can substantially improve both the execution time and the memory requirements of a program. The project's third development phase will examine the rationale for and benefits of subexpression elimination and will discuss critical issues associated with a phased implementation of a subexpression elimination algorithm. The steps of development involve subexpression elimination from

**Step 1:** IP expressions,

**Step 2:** IP sequences of assignment, procedure call, and/or print statements, and

**Step 3:** IP statement sequences including selection and repetition statements.

The goal of step 3 is, unfortunately, beyond the technical scope of this text. Rather than giving a complete solution to the problem, we will discuss an implementation strategy for applying the algorithm of step 2 to all (maximal) sequence of basic statements in an IP program – that includes the code blocks within selection and repetition statements. This implementation strategy is based on the formation of a flow graph for the generated three-address code. We will discuss the design of such a flow graph and the way it can be used to partially resolve step 3.

**MIPS Object Code Generation** The final development phase of the IP project focuses on the design and implementation of an object code generator. This section will also follow the three-phase approach taken for intermediate code generation. In each phase there are important aspects of the MIPS processor that must be reviewed. Of particular importance are the MIPS instructions related to reference parameters (Phase 2) and the structure and functionality of the MIPS floating point processor.



## Chapter 24

# Syntax and Static Semantics of IP

This part of the tutorial discusses the three initial components of the IP compiler, the tokenizer, parser, and syntax tree. We also discuss the processing that must be implemented in the syntax tree hierarchy to facilitate static semantic checking.

### 24.1 Parsing IP

#### Tokenizer

The tokenizer for IP poses no new implementation challenges, since the IP tokens are, in a certain sense, simply a mix of the tokens of PDef and of FP along with a few new reserved words.

```
scolonT    ;
commaT    ,
lpT       (
rpT       )
refT      ^
assignT   =
addOPT    +
subOPT    -
multOPT   * | / | %
compOPT   < | <= | >= | > | == | !=
ift       if
thenT     then
elseT     else
endiT     endif
whileT    while
endwT     endwh
endfT     endfn
printT    print
returnT   return
typeT     void | int | float
identT    [a-zA-Z]+
```

```
intT      0 | [1-9][0-9]*
floatT    (0 | [1-9][0-9]*).[0-9]+
```

## Parser

Most of the IP grammar structures have been seen earlier in PDef and FP. We repeat the grammar here for convenience.

```
<Program>      ::= <FnDefList>
<FnDefList>    ::= <FnDef> [ <FnDefList> ]
<FnDef>        ::= typeT identT lpT [ <ParamList> ] rpT <Block> [ <Return> ] endfnT
<ParamList>    ::= <ParamDecl> [ commaT <ParamList> ]
<ParamDecl>    ::= typeT [ refT ] identT
<Block>        ::= <DeclList> <StmtList>
** <DeclList>  ::= empty | <Decl> colonT <DeclList>
<Decl>         ::= typeT identT
<Statement>    ::= <BasicStmt> | <StructStmt>
<BasicStmt>    ::= ( <Assignment> | <FnApp> | <Print> ) colonT
** <StmtList>  ::= empty | <Statement> <StmtList>
<Assignment>  ::= identT assignT <Exp>
<Print>       ::= printT <Exp>
<Return>      ::= returnT <Exp>
<StructStmt>  ::= <Selection> | <Repetition>
<Selection>   ::= if lpT <BExp> rpT thenT <StmtList> [ elseT <StmtList> ] endifT
<Repetition> ::= while lpT <BExp> rpT <StmtList> endwhT
<BExp>        ::= <Exp> compOpT <Exp>
<FnApp>       ::= identT lpT [ <ArgList> ] rpT
** <ArgList>   ::= <Exp> [ commaT <ArgList> ]
<Exp>         ::= subOpT <Factor>
               ::= <FnApp>
               ::= <Term> { ( addT | subT ) <Term> }
<Term>        ::= <Factor> { multT <Factor> }
<Factor>      ::= int
               ::= floatT
               ::= identT
               ::= lpT <Exp> rpT
```

If we compare these grammar rules for IP to those of PDef and of FP, we see some interesting additions and differences. The relevant rules are preceded by ‘\*\*’. In the rest of this section we will discuss the three starred rules in turn.

### Parsing <DeclList>

The rule defining <DeclList> begins with the word `empty`, which is not a token but rather denotes the empty string.



```
<DeclList> ::= empty | <Decl> scolonT <DeclList>
```

This rule indicates that `<DeclList>` can be an empty list or be a declaration (`<Decl>`) followed by a semicolon followed by a `<DeclList>`. In this way the grammar rule describes a list of zero or more declarations (with semicolon). Looking ahead to the rule defining `<StmtList>` we see the same structure – however, the `<StmtList>` rule is starred for another reason.

We will focus on how to parse the rule for `<DeclList>`. The first thing to do is to expand the rule so that we see more clearly the options.

```
<DeclList> ::= empty
           ::= <Decl> scolonT <DeclList>
```

The question here is, how do we know which rule to parse. If the next token we see can start a `<Decl>` then we should parse the second rule. But the ‘empty’ is a bit of mystery. If the next token can’t start a declaration, then the only other possibilities are for tokens that can follow a declaration list. So we need to compute the first set for `<Decl>` and the follow set for `<DeclList>`, and hope the sets are disjoint. (See Section 9.3.3 for a refresher on first and follow sets.)

### Activity 91 –

Compute the first set of `<Decl>` and the follow set of `<DeclList>`. The first set is straightforward. To compute the follow set of `<DeclList>` you must look at the grammar rules to determine what can follow the non-terminal `<DeclList>`. The following rules may help in this determination.

```
<DeclList> ::= empty | <Decl> scolonT <DeclList>
<Block>   ::= <DeclList> <StList>
<FnDef>   ::= typeT identT lpT [ <FParamList> ] rpT <Block> [ <Return> ] endfnT
<StmtList> ::= empty | <Statement> <StmtList>
<Statement> ::= <BasicStmt> | <StructStmt>
<BasicStmt> ::= ( <Assignment> | <FnApp> | <Print> ) scolonT
<StructStmt> ::= <Selection> | <Repetition>
<Assignment> ::= identT assignT <Exp>
<Print>      ::= printT <Exp>
<Return>     ::= returnT <Exp>
<Selection>  ::= if lpT <BExp> rpT thenT <StmtList> [ elseT <StmtList> ] endifT
<Repetition> ::= while lpT <BExp> rpT <StmtList> endwhT
```

---

The answers to the previous exercise are then put to use in designing the method `parseDeclList`. The method, shown below, takes the standard form we have seen previously in the PDef and FP parsers. One caution is to be aware of the possibility that the first and follow sets for the two non-terminals might not be disjoint, and then lookahead might be required to resolve the ambiguity. In this case you should find no such ambiguity.

```
private void parseDeclList() {
```

```

// Grammar Rule: <DeclList> ::= empty | <Decl> scolonT <DeclList>
switch (currentToken.getType()) {
case typeT: // the only token in first(<Decl>)
    parseDecl();
    consume(Token.TokenType.scolonT);
    parseDeclList();
    break;
case ...: // fill in 'case token:' for each token in follow(<DeclList>)
    // do nothing
    break;
default:
    throw ParseException("Expected typeT, identT, ifT, whileT, printT, returnT, or
}
}
}

```

Actually, there are two other rules in which empty lists are described, the formal parameter list for a function definition and the corresponding argument list for a function call. But, in each of these cases the defining non-terminal, `<ParamList>` and `<ArgList>`, describe non-empty lists. The trick is that each of these lists is indicated as optional, implying that the empty lists are admissible. In these cases our strategy for implementing optional structures is used.

### ☛ Activity 92 –

Why would we use optional non-empty lists in the rules for `<ParamList>` and `<ArgList>` but non-optional possibly empty lists for `<DeclList>` and `<StmtList>`? As a strategy for discovering the answer, try altering the IP grammar so that `<StmtList>` is non-empty but its uses in the rules are optional.

### Parsing `<StmtList>`

We have already discussed one aspect of the non-terminal `<StmtList>`. But there is another detail of the statement list that is not so obvious. In Java and C++ we are used to the idea that every statement must be terminated with a semicolon. But in fact this isn't really true for all statements. If the body of a while loop is surrounded by curly braces, then there is no terminating semicolon. Actually, the rule is that basic statements are terminated by semicolons but right curly braces are not. In IP we don't use curly braces to bound blocks of code, rather, each structured statement has its own end token to do that job: `endwh` for `while` and `endif` for `if`. The structure of the rules defining `<StmtList>` make explicit that basic statements must be terminated by semicolons but no semicolon appears after `endwh` or `endif`.

### ☛ Activity 93 –

Design and implement the parse method `parseStmtList`. Remember to use the strategy from the previous exercise since statement lists can be empty.

**Parsing <ArgList>**

<ArgList> describes a comma separated list of function arguments. We saw almost the same structure for argument lists in FP , but there is a subtle difference. The FP definition

```
<Factor> [ <ArgList> ]
```

indicates a whitespace separated list of factors (<Factor>) while the IP definition

```
<Exp> [ commaT <ArgList> ]
```

specifies a comma separated list of expressions (<Exp>). Why the difference? The required commas in the IP case are the key! Consider the following legal function call in IP .

```
test(x , g(x))
```

The function `test` is being called on two arguments, a variable and a function call. If we wanted to write this in FP we would remove the parentheses and the commas getting the following.

```
test x g x
```

The FP notation becomes ambiguous – does `test` take two or three arguments? If the answer is two, then are the two arguments `x` applied to `g` and `x`, or the other way around? In FP we need parentheses around non-trivial arguments to remove the ambiguity. So in FP the argument list is a list of factors. The same problem is avoided in IP by having commas separate the arguments, eliminating any possibility of ambiguity.

### ☛ **Activity 94** –

Implement and test the parser for IP . Create a testing directory and put in it a suite of IP program files. There should be one set of programs that test that the parser can parse correct programs. The other set should have a program testing each possible syntactic error – focus on the places where exceptions can be thrown.

---

## 24.2 IP Syntax Tree Hierarchy

In this part of the project you will apply the techniques used in the PDef and FP projects to define and implement the IP syntax tree hierarchy, complete with symbol table. Since there are no new techniques to be discussed, what we will do instead is to give a new technique for describing the hierarchy. The technique requires that we describe not only the data members of the classes in the hierarchy, but also the subclass relationships. We will use a two column strategy, with the left

column describing each class and the right column describing the class's data members (type and name). To define a class name and specify its hierarchical relationship to other classes, we will use the notation  $A > B$  to indicate  $B$  is a subclass of  $A$ . For data members we use the notation  $A:B$  to indicate that  $A$  is of type  $B$ . The following row

Class name	Data members
<code>SyntaxTree&gt;Stmt&gt;Selection</code>	<code>BExp:condition, LinkedList&lt;Stmt&gt;:thenList, LinkedList&lt;Stmt&gt;:elseList</code>

defines the class `Selection` as being a subclass of `Stmt` which is a subclass of `SyntaxTree`: the row defined the class `Selection`. In addition, the class `Selection` has three data members, `BExp`, `thenList`, and `elseList`.

Using this tabular form we can write out various possible structures for the syntax tree and experiment with alternatives. Since this strategy is purely for design, it might be prudent to select short class names to use in the table even though longer more descriptive names may be used in the code. For documentation purposes one could provide a final table with a key indicating the correspondence between class names used in the table and those used in code.

The following table defines one possible structure for the syntax tree for IP .

Class name	Data members
<code>SyntaxTree</code>	<code>SyntaxTreeDebug:debug</code>
<code>SyntaxTree &gt; FnDefList</code>	<code>LinkedList&lt;FnDef&gt;:fnDefs</code>
<code>SyntaxTree &gt; FnDef</code>	<code>Token:type, Token:name, LinkedList&lt;Decl&gt;:paramList, Block:stmts, Expression:return</code>
<code>SyntaxTree &gt; Block</code>	<code>LinkedList&lt;Decl&gt;:declList, LinkedList&lt;Stmt&gt;:stmtList, Expression:retValue</code>
<code>SyntaxTree &gt; Decl</code>	<code>Token:type, Token:name, boolean:ref</code>
<code>SyntaxTree &gt; Stmt</code>	---
<code>SyntaxTree &gt; Stmt &gt; Assignment</code>	<code>Token:target, Expression:source</code>
<code>SyntaxTree &gt; Stmt &gt; ProcCall</code>	<code>Token:fnName, LinkedList&lt;Expression&gt;:argList</code>
<code>SyntaxTree &gt; Stmt &gt; Print</code>	<code>Expression:exp</code>
<code>SyntaxTree &gt; Stmt &gt; Repetition</code>	<code>BExp:condition, LinkedList&lt;Stmt&gt;:stmtList</code>
<code>SyntaxTree &gt; Stmt &gt; Selection</code>	<code>BExp:condition, LinkedList&lt;Stmt&gt;:thenList, LinkedList&lt;Stmt&gt;:elseList</code>
<code>SyntaxTree &gt; Expression</code>	---
<code>SyntaxTree &gt; Expression &gt; Exp</code>	<code>Expression:leftExp, Token:op, Expression:rightExp</code>
<code>SyntaxTree &gt; Expression &gt; NegExp</code>	<code>Expression:exp</code>
<code>SyntaxTree &gt; Expression &gt; FnApp</code>	<code>Token:fnName, LinkedList&lt;Expression&gt;:argList</code>
<code>SyntaxTree &gt; Expression &gt; IntLit</code>	<code>Token:value</code>
<code>SyntaxTree &gt; Expression &gt; FloatLit</code>	<code>Token:value</code>
<code>SyntaxTree &gt; Expression &gt; IdentExp</code>	<code>Token:name</code>

The syntax tree hierarchy has just one situation that might appear a bit odd. There are two classes, `ProcCall` and `FnApp`, which have the same data member description. It would seem we could make some structural simplification here. But in fact, when we implement semantic checking and code generation, it will turn out that the two situations call for different processing.

### Activity 95 –

Implement the IP syntax tree hierarchy and link it into the parser. Be sure to include the capability to print the syntax tree. Test your code on the test suite programs that are syntactically correct.

## SymbolTable

We also need to define the symbol table and then integrate it at appropriate spots in the syntax tree hierarchy. IP is a block structured and statically typed language and, unlike Java and C++, does not allow declarations in statement blocks — you can see this in reviewing the IP grammar. So for the IP syntax tree the two obvious places for local symbol tables are in the classes `FnDefList` and `FnDef`. The symbol table in `FnDefList` will contain all the function entries while the those in the `FnDef` objects will contain the parameter and variable entries. This suggests the following changes to the syntax tree description above.

Class name	Data members
<code>SyntaxTree&gt;FnDefList</code>	<code>LinkedList&lt;FnDef&gt;</code> , <code>SymbolTable localST</code>
<code>SyntaxTree&gt;FnDef</code>	<code>Token:type</code> , <code>Token:name</code> , <code>LinkedList&lt;Decl&gt;:paramList</code> , <code>Block:stmtList</code> , <code>SymbolTable localST</code>

The basic symbol table structure used in the previous two tutorials provides a starting place for the IP implementation as well. As for the symbol table contents, there are clearly three kinds of declarations – for functions, variables, and parameters. The structure suggested in the following table follows naturally.

Class name	Data members
<code>SymbolTable</code>	<code>LinkedList&lt;SymbolTableEntry&gt;:localST</code> , <code>SymbolTable:parent</code> , <code>SymbolTableDebug:debug</code>
<code>SymbolTableEntry</code>	<code>String:name</code> , <code>Token:type</code> , <code>SymbolTableDebug:debug</code>
<code>SymbolTableEntry&gt;VariableEntry</code>	<code>int:psnInList</code>
<code>SymbolTableEntry&gt;ParameterEntry</code>	<code>int:psnInList</code> , <code>ref:boolean</code>
<code>SymbolTableEntry&gt;FunctionEntry</code>	<code>LinkedList&lt;ParamEntry&gt;:paramList</code>

There are two special considerations in the structure of the `FunctionEntry` class. First, we have added a data member to reference the function's parameter list. In FP all parameters are value parameters and of type `int`. This means that in FP the semantic checking and code generation for a function call does not have to reference the parameter list. But in IP a parameter can have one of two possible types and be either a value or reference parameter. Consequently, when checking semantics for a function call it is necessary to compare the type of an argument to that of the corresponding parameter; when generating code for a function call the nature of the parameter (value or reference) determines the code to be generated. So at both semantic checking and code

generation time a function's parameter list must be available. The easiest way to do this is to store a reference to the list in the function name's symbol table entry. Second, there is no entry for the function's address. This is a consequence of our code generation strategy, already previewed in the FP<sup>+</sup> implementation. We will use the name of the function to construct a unique label in our generated assembly program. In this way, no physical address will be needed.

A key to any software development project is to take advantage of previous projects to anticipate needs in future phases of development. This is particularly important when implementing a compiler. We know that we will be using a similar code generation strategy already employed in our FP<sup>+</sup> implementation. Thus we know that ultimately we will want to add entries to our symbol table structure for generated temporary variables, labels, integer values, and, since IP allows them, floating point values. Anticipating code generation, it will be convenient if we can at times treat all the variable entries in the same way. For this reason there will be a new abstract class `AddressableEntry` whose subclasses will be those for variable, temporary, and parameter entries. We can then specify a complete symbol table structure in the following table.

Class name	Data members
<code>SymbolTable</code>	<code>LinkedList&lt;SymbolTableEntry&gt;:localST,</code> <code>SymbolTable:parent, SymbolTableDebug:debug</code>
<code>SymbolTableEntry</code>	<code>String:name, Token:type,</code> <code>SymbolTableDebug:debug</code>
<code>SymbolTableEntry &gt; FunctionEntry</code>	<code>LinkedList&lt;ParameterEntry&gt;:paramList</code>
<code>SymbolTableEntry &gt; LabelEntry</code>	
<code>SymbolTableEntry &gt; IntegerEntry</code>	<code>int:value</code>
<code>SymbolTableEntry &gt; FloatEntry</code>	<code>float:value</code>
<code>SymbolTableEntry &gt; Addressable</code>	<code>int:offset, boolean:ref</code>
<code>SymbolTableEntry &gt; Addressable &gt; VariableEntry</code>	
<code>SymbolTableEntry &gt; Addressable &gt; ParameterEntry</code>	
<code>SymbolTableEntry &gt; Addressable &gt; TempEntry</code>	

### Activity 96 –

Implement the symbol table structure just described and make the required modifications to the syntax tree implementation. Be sure to provide a way to display the symbol table either imbedded in a syntax tree display or separately. Run your implementation on the valid component of the test suit.

## 24.3 Static Semantic Checking

Remember that static semantics refers to the semantics of identifiers that can be determined at translation time. In the PDef implementation we had identifiers that could be associated with a

single attribute, the type. In FP things were a bit more complex, with both function names and parameter names, each type having its own set of attributes. In IP we add to function definitions the possibility of local variable declarations in addition to the parameter declarations.

## Blocks and Scope

There are two types of blocks in an IP program: the one spanning the entire input file, the function definition block, and the second beginning with a function's parameter list and ending with its `endfn`, called a function block. Because there are no variables declared at the outer level of the function definition block, and because all variable declarations in function blocks must occur before any statement, we will adopt the following scope rule.

The scope of a declaration spans the immediate block within which the declaration occurs.

This means, of course, that function names are in the function definition block while the parameter and local variable declarations for a particular function are in the function's block.

As usual in statically typed block structured languages, names defined in an outer block can be redefined within a nested block. For IP that simply means that a function name could be redeclared in a function. This is the only way in which a name can be declared twice in overlapping (i.e., nested) scopes.

### 24.3.1 Semantic Checking

The required semantic checking should be pretty clear at this point, though there are a couple of special conditions that need to be remembered from the definition of IP in Section 23.1.2. First, we must be sure to make sure that functions called as procedures have a return type of `void`. In addition, when a function has a non-`void` return type then none of the parameters can be reference parameters.

It is also important to remember that the arithmetic expressions must be checked in the same way as in PDef – this means that each subexpression must have a type that ultimately gets factored into the type of the enclosing expression.

We should remember at this point that we added the parameter list as a data member in `FunctionEntry` in the symbol table. This is meant to facilitate semantic checking. In particular, when a function call is checked we must be able to compare the types of the function call's arguments against those of the function definition's formal parameters. The same processing is necessary for procedure calls.

The table in Figure 24.1 associates semantic checking actions with the classes in the syntax tree hierarchy. Based on this table you should be able to implement the method `checkSemantics` in each class of the syntax tree hierarchy.

Class name	Semantic checking action
<code>FnDefList</code> :	sequentially check semantics of each function in <code>fnDefs</code>
<code>FnDef</code> :	if <code>type</code> is <code>void</code> then check that no element in <code>paramList</code> is specified as a reference; check semantics of <code>stmts</code>
<code>Block</code> :	check semantics of each statement in <code>stmtList</code> ; verify that <code>retValue</code> is <code>null</code> only if the function return type is <code>void</code> ; check semantics of the expression <code>retValue</code> and verify that the deduced type of <code>retValue</code> is assignment compatible with the function's return type
<code>Decl</code> :	nothing to check
<code>Assignment</code> :	ensure that <code>target</code> is declared; check semantics of <code>source</code> and insure the return type of <code>source</code> is assignment compatible with <code>target</code>
<code>ProcCall</code> : †	check that <code>fnName</code> is declared, that its return type is <code>void</code> , and the number of actual params is the same as the number of formal parameters for <code>fnName</code> ; check semantics for the arguments in <code>argList</code> and that their return types are assignment compatible with the types of their corresponding formal parameters; check that each formal reference parameter is paired with a variable argument
<code>Repetition</code> :	check the semantics of <code>condition</code> and of each statement in <code>stmtList</code>
<code>Selection</code> :	check the semantics of <code>condition</code> and of each statement in <code>thenList</code> and in <code>elseList</code>
<code>Expression</code> :	subclasses must return a type when semantic checking is carried out
<code>Exp</code> :	check semantics of <code>leftExp</code> and <code>rightExp</code> and compute the return type based on the table in Figure 23.1 – return the computed type
<code>FnApp</code> : †	check that <code>fnName</code> is declared, that its return type is not <code>void</code> , and the number of actual params is the same as the number of formal parameters for <code>fnName</code> ; check semantics for the arguments in <code>argList</code> and that their return types are assignment compatible with the types of their corresponding formal parameters; return the type bound to <code>fnName</code> .
<code>NegExp</code> :	check semantics of <code>exp</code> and return its type
<code>IntLit</code> :	return type <code>int</code>
<code>FloatLit</code> :	return type <code>float</code>
<code>IdentExp</code> :	check that <code>name</code> is declared and return its type

† *The use of the new parameter list data member in `FunctionEntry` is critical here.*

Figure 24.1: Static Semantic Actions for IP



## Chapter 25

# IP Intermediate Code Generation

At this point you should have a working IP recognizer, i.e., your implementation should recognize syntactically and semantically correct IP programs and report appropriate parse errors and static semantic errors. In this chapter we will define the IP dynamic semantics by giving a translation from the syntax tree hierarchy to three-address code. One of the things we will find necessary in this effort is to alter the three-address code defined in Interlude II (see Figure 20.2) to accommodate the structures in IP.

There are two basic problems we will confront for the first time in this chapter. First, we will have to deal with both value and reference parameters; we have seen value parameters in the FP tutorial, but not reference parameters. Second, IP has multiple types and allows mixed-mode arithmetic. These capabilities require special care in semantic checking and in code generation – for example, we will need to be able to generate code to convert a value from one form (integer) to another (float). Since each of these problem areas is somewhat complex, we will approach the intermediate code generation in three phases:

**Phase 1:** Define and implement intermediate code generation for the subset of IP that has only integer values and value parameters, i.e., IP with reference parameters and the float type factored out.

**Phase 2:** Augment the implementation of Phase 1 by adding the reference parameter capability. This is IP without the floating point capability.

**Phase 3:** Augment the implementation of Phase 2 by integrating the floating point capability. This is the full IP language.

In each of the three phases we will go through the same sequence of activities: make any necessary modifications to the three-address instruction list, describe (changes to) the IP semantics based on the three-address instructions, discuss particular code generation problems associated with the particular phase.

### 25.1 Phase 1

The purpose of this phase is to provide an implementation base for the other two phases. The advantage of this approach is that in subsequent phases you can concentrate completely on the

particular feature being added. Our approach here will be to investigate the three-address code required for the Phase 1 implementation, present the Phase 1 dynamic semantics in terms of the enhanced three-address code, and finally discuss any particular problems to be encountered in implementing the intermediate code generation.

### 25.1.1 IP Three-address Code

The three-address code defined for the  $FP^+$  intermediate code generation is not quite strong enough for Phase 1. In looking at the differences between  $FP^+$  and IP we see several important things:

- there is a print statement;
- there is an assignment statement;
- while selection exists in both languages, selection in IP is a statement and has an optional else clause;
- there is repetition in the form of the `while` statement;
- there is a procedure call statement;

We will first look at these additions one at a time, to see if changes are warranted, and then summarize the Phase 1 three-address code.

#### print statement

The print statement is very simple since it only involves an expression. To insure that the intermediate code accurately specifies this statement we must add to the three-address code a new instruction denoting the print statement. It will take a single parameter, the temporary variable associated with its expression.

#### Assignment Statement

The assignment statement is not present in  $FP^+$ . However, its implementation in three-address code should simply involve evaluation of the expression and then assignment of that result to the variable on the left of the assignment. Since this can be handled by the three-address copy instruction, no new instructions are needed.

#### if statement

The translation of the IP selection statement is a fairly straightforward variation on the translation in  $FP^+$ . The only problem is dealing with the optional else. If the else is present then the translation is the same as before. If the else is absent then we must make sure that there is no jump instruction at the end of the true block. In the semantics we will describe this in two cases. This requires no new three-address instructions.

#### while Statement

For  $FP^+$  code generation we found that “jump on false” and “jump” were the appropriate three-address instructions for implementing selection. When evaluating a `while` statement, if the loop condition is true we do not want to branch around the loop body – only if the condition is false. This sounds exactly like the situation for the then-body of the selection

statement. At the bottom of a loop body we must always return to the top of the loop to re-evaluate the loop condition – this is best done with the absolute jump instruction. So to accommodate the while loop the  $FP^+$  three-address code is adequate.

### Procedure call

The procedure call in  $IP$  is similar to the function call in  $FP^+$  except there is no value to return. This fact means that the `FCall` and `FReturn` instructions for  $FP^+$  are not appropriate. Instead we need to add two new instructions in which the return value is factored out. We will call these `PCall` and `PReturn`.

Our analysis of these differences between  $IP$  and  $FP^+$  leads to an extended three-address code for  $IP$  which is presented in Figure 25.1. Notice that some instruction names have been altered; the changes are to prepare for the later integration of the floating point type. Instructions with ‘`Int`’ added will have counterparts with ‘`Flt`’ replaced by ‘`Flt`’.

Name	Result	Source 1	Source 2	Meaning
<code>IntLit</code>	<code>x</code>	<code>y</code>	-	<code>x = y</code> , where <code>y</code> is an integer literal
<code>IntNeg</code>	<code>x</code>	<code>y</code>	-	<code>x = (-1) * y</code>
<code>IntOp</code> (binary operation)	<code>x</code>	<code>y</code>	<code>z</code>	<code>x = y IntOp z</code>
<code>IntCopy</code>	<code>x</code>	<code>y</code>	-	<code>x = y</code>
<code>Jump</code>	<code>x</code>	-	-	<code>goto x</code>
<code>JumpF</code>	<code>x</code>	<code>y</code>	-	<code>if !x goto y</code>
<code>PrintInt</code>	<code>x</code>	-	-	<code>print x</code>
<code>Param</code>	<code>x</code>	-	-	<code>x</code> is a parameter
<code>PCall</code>	<code>x</code>	-	-	<code>Call x</code>
<code>PReturn</code>	-	-	-	<code>return</code>
<code>FIntCall</code>	<code>x</code>	<code>y</code>	-	<code>x = Call y</code>
<code>FIntReturn</code>	<code>x</code>	-	-	<code>return x</code>
<code>Label</code>	<code>x</code>	-	-	<code>x</code> is a label in the three-address code
<code>Halt</code>	-	-	-	halt program (no return value)

Figure 25.1: Three-address Code for Phase 1  $IP$

### 25.1.2 Definition of $IP$ Dynamic Semantics

The dynamic semantics of  $IP$  is defined by a function `trans`, which is defined by the following sequence of recursive definitions. The list defines `trans(x)`, where `x` is a component of the  $IP$  syntax.

#### Literal

$$\text{trans}(N) = \text{IntLit}(t1, N, -)$$

where `N` is an integer literal.

#### Variable Name

$$\text{trans}(V) = \text{IntCopy}(t1, V, -)$$

where  $V$  is an identifier, either a variable or parameter.

### Arithmetic and Comparison Operations

$$\text{trans}(x \text{ op } y) = \text{trans}(x), \text{trans}(y), \text{op}(t1, t2, t3)$$

where  $\text{op}$  is one of the arithmetic

$$+, *, -, /, \%, <, <=, ==, !=, >=, >$$

and  $t2$  and  $t3$  are the result temporary variables from  $\text{trans}(x)$  and  $\text{trans}(y)$ , respectively. [Notice, that since we have only integer values, there is no other processing necessary when generating code for these operations.]

### NegExp

$$\text{trans}(-e) = \text{trans}(e), \text{IntNeg}(t1, v, -)$$

where  $e$  is an integer expression and  $v$  is the result temporary variable from the translation of  $e$ .

### FnApp

$$\begin{aligned} \text{trans}(f \text{ p1 } \dots \text{ pn}) = \\ \text{trans}(p1), \dots, \text{trans}(pn), \text{Param}(t1, -, -), \dots, \text{Param}(tn, -, -), \text{FCall}(t0, f, n) \end{aligned}$$

where  $f$  is the offset in the code table of the first instruction of  $f$ . Each  $t_i$  is the temporary result variable from  $\text{trans}(p_i)$ .

### Assignment

$$\text{trans}(x = \text{exp}) = \text{trans}(\text{exp}), \text{IntCopy}(x, v, -)$$

where  $v$  is the temporary result variable from  $\text{trans}(\text{exp})$ .

**Selection** There are two cases to consider.

$$\begin{aligned} \text{trans}(\text{if } b\text{exp} \text{ then } t \text{ endif}) = \\ \text{trans}(b\text{exp}), \text{JumpF}(v, l1, -), \text{trans}(t), \text{Label}(l1) \end{aligned}$$

$$\begin{aligned} \text{trans}(\text{if } b\text{exp} \text{ then } t \text{ else } f \text{ endif}) = \\ \text{trans}(b\text{exp}), \text{JumpF}(v, l1, -), \text{trans}(t), \text{Jump}(l2, -, -), \\ \text{Label}(l1), \text{trans}(f), \text{Label}(l2, -, -) \end{aligned}$$

In both cases  $v$  is the result temporary variable from  $\text{trans}(b\text{exp})$ .

### Repetition

$$\text{trans}(\text{while } b\text{exp } b \text{ endwh}) =$$

```
Label(l1,-,-), trans(bexp), JumpF(v,l2,-), trans(b), Jump(l1,-,-),
Label(l2,-,-)
```

where `v` is the result temporary variable from `trans(bexp)`.

### ProcCall

```
trans(f p1 ... pn) =
    trans(p1), ..., trans(pn), Param(t1,-,-), ..., Param(tn,-,-), PCall(t0,f,n)
```

where `fn` is the offset in the code table of the first instruction of `fn`. Each `ti` is the temporary variable holding result of evaluating `pi`.

### Print

```
trans(print e) = trans(e), PrintInt(v,-,-)
```

where `v` is the result temporary variable from `trans(e)`.

### Statement List

```
trans(s1; ...; sn) = trans(s1), ..., trans(sn)
```

### FnDef

The correct translation depends on whether the return type is `void`.

`void`

```
trans(void f (p1,...,pn) body endfn) = Label(f), trans(body), PReturn(-,-,-)
```

`non-void`

```
trans(t f (p1,...,pn) body; return e; endfn) = Label(f), trans(body), trans(e),
where v is the result temporary from trans(e). [ The return has been pulled out of the
function body to make it explicit. ]
```

### Program

```
trans(fd1 ... fdn) = trans(fd1), ..., trans(fdn), Halt(t1,-,-)
```

As described this assumes that `fd1` is the function `main` (which is required). If `main` is actually `fi`, where `i > 1`, then the translation of `main` must still appear first.

### 25.1.3 IP Intermediate Code Generation

In Section 20.1.2 we defined classes `OpCode` and `CodeTable` to facilitate three-address (intermediate) code generation for `FP+`. The structures we defined there should work almost as they are for the intermediate code generation for `IP`. When looking at the code for `OpCode` the first thing that catches your eye is the enumerated type `textttOpName`, which contains the names of the three-address instruction names. This enumerated type must be extended to include all the new instructions appearing in the table in Figure 20.1.2.

#### ☛ Activity 97 –

Make the appropriate changes to the enumerated type `OpName`. Remember that there are new names to be added and old names that need modification.

---

Implementing the code generation methods (`generateCode`) in the syntax tree hierarchy is the next step. Here again we can take a lead from the corresponding implementation for  $FP^+$ . For the most part, the semantics definitions for IP are the same as for  $FP^+$ , e.g., the definitions for `Identifier`, `Number`, `NegExp`, and the built in function applications. The corresponding code generation methods, then, should be the same or very similar to the corresponding methods in the  $FP$  implementation. For the other semantic definition components for IP, the method implementations should be clear from the recursive definition of `trans` in 25.1.2. But in our Phase 1 IP there are a couple of interesting cases that warrant a preliminary look.

Our general strategy for generating code will be that used first for  $FP^+$ , i.e., we will generate unique temporary variables as needed across each function definition. Look back to Section 20.1.1 in which we defined the  $FP^+$  semantics and the subsequent section in which code generation is described. The following sequence of exercises should lead you to a complete implementation of the Phase 1 intermediate code generator. In some of the exercises there are suggested implementations.

#### ☛ Activity 98 –

Begin the implementation of the intermediate code generation by coding `generateCode` in the syntax tree classes for which changes are minimal. The implementation for `Print`, for example, should be very similar to that in `IntNeg`; that for `Assignment` should be only slightly different.

---

#### ☛ Activity 99 –

##### The Function `main`

One easy thing to take care of before starting any code generation is to insure that the function `main` appears first in the function definition list. Since that list consists of entries of type `FnDef` and each `FnDef` object contains the defined function's name, it should be easy to write a method for the `Program` class that locates the `main` function definition and moves it to the front of the list. Implement this capability and convince yourself that it behaves as desired (by displaying the syntax tree, for example).

---

#### ☛ Activity 100 –

##### `ProcCall`

Looking at the IP semantic definition above we can see that the definition associated with `ProcCall` is very similar to that for `FnApp`. Since the semantics of `FnApp` are the

same in IP and FP<sup>+</sup>, the implementation of FnApp in both settings should be the same. Consequently, the implementation of code generation for ProcCall should be a modification of that of FnApp. Compare the following implementation with that for the FP<sup>+</sup> implementation of FnApp.

```
// ProcApp.generateCode
public void generateCode(SymbolTable st, CodeTable ct) {
    // retrieve function's symbol table entry
    SymbolTableEntry fn = st.findEntry(name);

    // process the arguments and store the resulting temporary
    // variables on the list 'params'
    LinkedList<TempEntry> params = new LinkedList<TempEntry>();
    for (Expression entry : argList) {
        TempEntry t = (TempEntry)(entry.generateCode(st, ct));
        params.addFirst(t);
    }
    // now generate the Param instructions
    for (TempEntry entry : params) {
        ct.genCode(OpCode.OpName.Mparam, entry);
    }
    // finally, generate the call instruction
    ct.genCode(OpCode.OpName.Mpcall, fn);
}
```

Integrate this implementation into your system and test it out.

## Activity 101 –

### Selection

There are two main differences between selection in IP and FP<sup>+</sup>. The first is that selection in IP does not return a value – it is a statement. The second is that the **else**-part of the statement is optional in IP, while it is required in FP<sup>+</sup>. The selection semantics was defined in two parts, depending on the presence of the **else**. The obvious strategy is to carry out the initial common action and then have a selection where one branch finishes the code assuming an else clause and the other finishes the code assuming no else clause. The following code results.

```
// Selection.generateCode
public void generateCode(SymbolTable st, CodeTable ct) {
    // allocate the label for the jumpF after the condition
    LabelEntry label1 = new LabelEntry();

    // generate code for condition evaluation and the true part
```

```

SymbolTableEntry btemp = bexp.generateCode(st, ct);
ct.genCode(OpCodes.OpName.MjumpF, btemp, label1);
truePart.generateCode(st, ct);

if (falsePart != null) {
    // generate code if else is present
    LabelEntry label2 = new LabelEntry();
    ct.genCode(OpCodes.OpName.Mjump, label2);
    ct.genCode(OpCodes.OpName.Mlabel, label1);
    falsePart.generateCode(st, ct);
    ct.genCode(OpCodes.OpName.Mlabel, label2);
}
else { // generate label1 if no else
    ct.genCode(OpCodes.OpName.Mlabel, label1);
}
}

```

Implement this method and then the corresponding one in `Repetition` – the selection implementation should give all the guidance necessary for repetition.

---

## Activity 102 –

### Function definition

A function definition is another syntactic component which has two flavors, one similar to that its counterpart in  $FP^+$  and the other being a variation on that pattern. The problem to deal with here is that the code generated differs slightly depending on whether there is a non-void return value. In both cases we must generate code for the entry into the function (allocate space for temporary variables)

It is important to notice that this `generateCode`

```

public void generateCode(CodeTable ct) {
    FunctionEntry fn = (FunctionEntry)(st.findEntry(name));

    // facilitates getting count of temporary variables to allocate on stack
    IntegerEntry val = new IntegerEntry(0);

    TempEntry.resetCount(); // reset count of temporary variables
    ct.genCode(OpCodes.OpName.Mlabel, new LabelEntry(fn.getName()));

    block.generateCode(st, ct);

    // handle return, with three choices of return type
    if (type.getName().equals("void")) {
        ct.genCode(OpCodes.OpName.Mpreturn, val);
    }
}

```



```

    }
    else { // return type == int or float
        SymbolTableEntry temp = exp.generateCode(st, ct);
        ct.genCode(OpCode.OpName.MfIntReturn, temp, val);
    }
    //
    val.setValue(TempEntry.getCount() + declList.size());
}

```

While it seems that complete code has been given for this method, there are certain elements that might indicate modifications to other components. For example, you will need to implement the methods `resetCount` and `getCount` in `TempEntry` in the symbol table.

---

### ☛ Activity 103 –

Complete the Phase 1 intermediate code generator. Carefully test the completed system – you will want to read carefully the ccode sequences generated.

---

## 25.2 Phase 2

Reference parameters are familiar if you have programmed with languages of the Algol family of languages, Pascal and Ada being well known examples. There are two standard methods of parameter passage, value and reference, and both are allowed in IP . In this section we will review the reference parameter concept and determine any necessary additions to the IP three-address code, show how reference parameters are factored into the semantics of IP , and finally discuss how our initial code generation must be adjusted for reference parameters.

### 25.2.1 Reviewing Reference Parameters

The idea with a reference parameter is to pass the address of the argument rather than its value. One consequence is that an argument passed to a reference parameter must have an addressable location – i.e., it must be a variable. To identify the impact of reference parameters we will examine the following simple IP code example.

```

int f(int ^ x, int y)
    x = x + y;
    return g(x, x + y);
endfn
int g(int ^ a, int b)

```

```

    a = 10;
    return a + b;
endfn
void main()
    int r;
    r = 5;
    print f(r, 10);
    print r;
endfn

```

To understand what happens when this program executes the first element we focus on is the call to the function `f` in the first print statement in `main`. In the call we pass `r` to the first reference parameter `x` in `f`; the argument value 10 is passed to the parameter `y`. But while the value stored on the stack for `y` is 10, the value stored on the stack for `x` is the address for `r`. Fine, we knew that all along. Apparently, when we generate code for creating the parameter value for `x` on the stack, we must have a three-address instruction that says “copy the address” not the value. That’s our first discovery – we need a new kind of copy instruction to copy the address of the variable – call it `copyAddr`.

Now what happens when we execute the first statement in `f`? The value `x + y` will be computed and stored at `x`. How is this computation impacted by the fact that `x` is a reference parameter? The impact is that any reference to `x` is actually passed through the address contained in `x`. So in evaluating the assignment statement in `f`, the value we compute on the right hand side will ultimately be stored, not at `x`, but at the address in `x`. This is not the same as a normal copy. So we need a second kind of copy instruction, say `copyToRef(x,y)`, which takes the value of `y` and stores it at the address in `x`.

Continuing with the evaluation of the assignment statement in `f`, what two values do we add together when we evaluate `x + y`? Since `y` is a value parameter we will use its passed value, which is 10. Since `x` is a reference parameter we must take, not its value, but the value at the address it contains – that means the value of the original argument `r` in `main`. That value is 5. Apparently, we need a third new copy instruction, which is the opposite of `copyToRef`, and we will call it `copyFromRef`.

So we compute the value 15 and store it at the address in `x`, which means that the value of `r` in `main` will change to 15. But what happens when we call `g` in the return statement for `f`? We will pass the value of `x + y` (which now is 25) to the second parameter `b` of `g`, but `x` will be passed to `a` which is a reference parameter in `g`! The value parameter is easy to understand, the value 25 will be associated with `b` in `g`. But what about passing `x`? When we pass a reference parameter as an argument to another reference parameter we pass, not the address of `x`, but the address stored in `x`. When a reference parameter is passed to a reference parameter, it is simply the value (an address) that is passed on. But the usual copy command will handle this situation.

### ☛ Activity 104 –

So what value does the function `g` produce? And what is the final output from executing the program? Show, through a careful trace, that the values printed by the program will be 35 followed by 10.

The result of this analysis is that we need three new three-address instructions to handle the presence of reference parameters. Here are the instructions and appropriate descriptions.

Name	Result	Source 1	Source 2	Meaning
IntCopyFromRef	x	y	-	$x = *y$ †
IntCopyToRef	x	y	-	$*x = y$ †
CopyAddr	x	y	-	$x = \&y$ †

† Here the symbols ‘\*’ and ‘&’ denote dereferencing and address operators, as in C

### 25.2.2 Reference Parameter Semantics

The discussion above that determined the new three-address instructions is summarized in the following semantic description. It turns out that the impact of reference parameters can be restricted to the four cases which follow.

#### Identifier Name

$$\text{trans}(V) = \text{CopyFromRef}(t_1, V, -)$$

where  $V$  is a reference parameter.

#### Assignment

$$\text{trans}(x = \text{exp}) = \text{trans}(\text{exp}), \text{CopyToRef}(x, v, -)$$

where  $x$  is a reference parameter and  $v$  is the result temporary variable from  $\text{trans}(\text{exp})$ .

#### ProcCall

$$\begin{aligned} \text{trans}(f(p_1, \dots, p_n)) = \\ \text{trans}(p_1), \dots, \text{trans}(p_n), \text{Param}(t_1, -, -), \dots, \text{Param}(t_n, -, -), \text{FCall}(t_0, f, n) \end{aligned}$$

where  $fn$  is the symbol table entry for  $f$ . Each  $t_i$  is the result temporary variable from  $\text{trans}(p_i)$ .

It is important to recall that void functions, i.e., those involved in procedure calls, can have value and reference parameters. So for the argument  $p_i$  there are two possibilities depending on the nature of the  $i$ th formal parameter:

value:  $\text{trans}(p_i) = \text{Copy}(t_i, p_i, -)$

reference:  $\text{trans}(V) = \text{CopyAddr}(t_i, p_i, -)$

### 25.2.3 Intermediate Code Generation for Reference Parameters

In order to correctly implement the semantics above we must identify the code generation methods that require modification. If we look at the three semantic rules above we can see that identifiers are the focus and that those variable names can appear in three contexts: the left side of an assignment, in an expression, as an argument to a reference parameter. So we must focus on the syntax tree classes `Assignment`, `IdentExp`, `ProcCall`, and `FnApp`.

#### Activity 105 –

### Assignment

The modifications for assignment code generation are relatively easy. From the semantic rule for assignment we see that our original version satisfies the case where the left side is not a reference. So we need only distinguish when the left side is or is not a reference. Remember that the only way for an identifier to be a reference is for it to be a formal parameter designated as a reference.

Implement the necessary changes to `Assignment.generateCode`.

---

### Activity 106 –

#### IdentExp

When an identifier is encountered in an expression we must assess, as in the assignment case, whether or not the identifier is a reference. The modifications should be straightforward if based on the result of the previous exercise.

Implement the necessary changes to `IdentExp.generateCode`.

---

### Activity 107 –

#### ProcCall

In generating code for `ProcCall` the impact of reference parameters occurs only at the point where the argument list is being processed. The argument list and formal parameter list must be traversed in parallel, identifying those parameters that are reference parameters. Semantic checking will already have verified that reference parameters are paired with identifier expressions. So in traversing the argument and parameter lists, arguments for value parameters are translated as in Phase 1 parameters for reference parameters must be translated as described in the semantics description. Implement the necessary changes to `ProcCall.generateCode`.

---

### Activity 108 –

At this point you should have a completed Phase 2 implementation. Make sure that it functions and then test it on a range of IP programs involving reference parameters. In addition, run the implementation on the suite of programs from Phase 1 to insure that nothing has broken.

---

## 25.3 Phase 3

Adding multiple types and mixed-mode arithmetic to our growing IP implementation is the topic of this section. While it is not surprising that handling reference parameters requires a bit of work, it might seem Phase 3 will be pretty straightforward. In fact it is not straightforward and there are several problems to deal with.

### 25.3.1 Three-address Code

One of the keys to our code generation has been the use of temporary variables to associate with computed values. If we are to have multiple types, it will be necessary to associate types with the temporary variables. For this reason we will add to the class `TempEntry`, in the symbol table hierarchy, a new data members `type`, a reference to a `Token` object, and `ref`, a boolean value which is true if the value represented is a reference.

#### ☛ Activity 109 –

Implement the necessary changes to the class `TempEntry`. Test the changes.

Thinking in terms of the IP three-address code, we have just provided a mechanism so that temporary variables in three-address instructions will hold information about their type. The other problem presented by code generation is the necessity, in mixed-mode arithmetic, to be able to convert at run-time a value of integer type into an equivalent value in floating point type. To facilitate this we add a new three-address instruction called `Convert`.

In addition to this new instruction it is important to recognize those existing instructions that will require floating point versions. All copy instructions need need flavors, all arithmetic and comparison operations as well. There are also floating point versions needed for the instructions `IntLit`, `IntNeg`, `PrintInt`, and `IntReturn`. The following table summarizes these new instructions.

Name	Result	Source 1	Source 2	Meaning
<code>Convert</code>	<code>x</code>	<code>y</code>	-	<code>x = float(y)</code> ( <code>x:float, y:int</code> ) †
<code>FltLit</code>	<code>x</code>	<code>y</code>	-	<code>x = y</code> , where <code>y</code> is a floating point literal
<code>FltNeg</code>	<code>x</code>	<code>y</code>	-	<code>x = (-1.0) * y</code>
<code>FltOp</code> (arithmetic operation)	<code>x</code>	<code>y</code>	<code>z</code>	<code>x = y FltOp z</code>
<code>FltOp</code> (comparison operation)	<code>x</code>	<code>y</code>	-	<code>x FltOp y</code>
<code>PrintFlt</code>	<code>x</code>	-	-	<code>print x</code>
<code>FFltCall</code>	<code>x</code>	<code>y</code>	-	<code>x = Call y</code>
<code>FFltReturn</code>	<code>x</code>	-	-	return <code>x</code> , or <code>float(x)</code> , if necessary
<code>FltCopy</code>	<code>x</code>	<code>y</code>	-	<code>x = *y</code> †
<code>FltCopyFromRef</code>	<code>x</code>	<code>y</code>	-	<code>x = *y</code> †
<code>FltCopyToRef</code>	<code>x</code>	<code>y</code>	-	<code>*x = y</code> †

† `float(y)` denotes a mechanism in object code for converting `y` to floating point form.

### 25.3.2 Semantics of Mixed-mode Arithmetic

There are five contexts in which multiple types and mixed mode arithmetic impact IP semantics: parameter binding during function and procedure call translation (that's two), the appropriate translation for `NegExp`, the coercion (when necessary) of arguments in comparison and arithmetic binary operations, and the assignment statement. The semantic details follow.

#### Literal

`trans(N) = FltLit(t1,N,-)`

where `N` is a floating point literal.

#### NegExp

There are two cases here depending on the type associated with `exp`.

type of <code>v</code>	generated code
<code>int</code>	<code>trans(-exp) = trans(exp), IntNeg(t,v,-)</code>
<code>float</code>	<code>trans(-exp) = trans(exp), FltNeg(t,v,-)</code>

where `v` is the temporary result variable from `trans(exp)`.

#### Arithmetic and Comparison Operations

Assume `Op` is one of the IP built in arithmetic or comparison operations. We want to define `trans(expL Op expR)`. Assume that `tL` and `tR` are the temporary result variables from `trans(expL)` and `trans(expR)`, respectively. We also assume that `tx` is the temporary result variable from an application of `Convert`.

type of (tL,tL)	generated code
(int,int)	<code>trans(expL Op expR) = trans(expL), trans(expR), IntOp(t,tL,tR)</code> t has type int
(int,float)	<code>trans(expL Op expR) = trans(expL), Convert(tx,tL), trans(expR), FltOp(t,tx,tR)</code> t has type float
(float,int)	<code>trans(expL Op expR) = trans(expL), trans(expR), Convert(tx,tR), FltOp(t,tL,tx)</code> t has type float
(float,float)	<code>trans(expL Op expR) = trans(expL), trans(expR), FltOp(t,tL,tR)</code> t has type float

#### ProcCall

The definition of `trans` is the same as in the Phase 2 semantics, except for the handling of the translation of the argument list. In this case we must worry about converting integer arguments that are paired with formal floating point parameters.

Assume `pi` is an argument and `Pi` is the paired formal parameter (and is not a reference parameter). Based on the types of the two, here are the possible translations for `pi`.

types of (pi,Pi)	translation
(int,int)	trans(pi) = trans(pi), Copy(ti,v,-)
(float,float)	trans(pi) = trans(pi), Copy(ti,v,-)
(int,float)	trans(pi) = trans(pi), Convert(t,v,-), Copy(ti,v,-)
(float,int)	This is a semantic error.

The type of `ti` will be `int` in the first case and `float` in the next two cases.

Reference parameters are handled in the same manner as in Phase 2.

**FFltCall** In this case dealing with floating point parameters is identical to the description for **ProcCall**. The other impact of the floating point type is seen in the function return type. There are two cases depending on the type of the expression to be returned.

### FnDef

The correct translation depends on the return type. There is no change for return types of `void` or `int`. In the case of a return type of `float` and a return expression whose type is `int` that a conversion is required before the return.

Case 1

```
trans(float f (p1,...,pn) body; return e; endfn) = Label(f), trans(body),
trans(e), FFltReturn(v,-,-)
assuming that the type of the expression e is float and that v is the result
temporary variable from trans(e).
```

Case 2

```
trans(float f (p1,...,pn) body; return e; endfn) = Label(f), trans(body),
trans(e), Convert(t,v,-), FFltReturn(t,-,-)
assuming that the type of the expression e is int and that v is the result
temporary variable from trans(e) and t is the temporary variable from
Convert.
```

### Print

```
trans(print e) = trans(e), PrintFlt(v,-,-)
```

where `e` is a floating point expression and `v` is the result temporary variable from `trans(e)`.

### Assignment

From the Phase 2 semantics we know that the general form for `trans` in this case is

```
trans(x = exp) = trans(exp), 'copycommand'(x,v,-)
```

where `v` is the result temporary variable of `trans(exp)` and `'copycommand'` is either `Copy` or `CopyToRef` (based on the Phase 2 semantics).

If we consider the possible types for `x` and `v`, we have the following new translations.

types of (x,y)	translation
(int,int)	trans(x = exp) = trans(exp), 'copycommand'(x,v,-)
(float,float)	trans(x = exp) = trans(exp), 'copycommand'(x,v,-)
(float,int)	trans(x = exp) = trans(exp), Convert(t,v), 'copycommand'(x,t,-)
(int,float)	This would be a semantic error.

where in each case `v` is the result temporary variable of `trans(exp)`.

### 25.3.3 Intermediate Code Generation for Mixed-mode Arithmetic

The modification of intermediate code generation methods to accommodate multiple types and mixed-mode arithmetic all involve the insertion of `Convert` commands. This solution pattern will be illustrated by a single code example – that for assignment.

```
// Assignment.generateCode
public void generateCode(SymbolTable st, CodeTable ct) {
    TempEntry t = exp.generateCode(st, ct);
    Token sourceType = t.getType();
    Token targetType = target.getType();
    if ((sourceType.getName().equals("int") && (targetType.getName().equals("float")))
        TempEntry t1 = st.addTemp(sourceType, false);
        ct.genCode(OpCodes.OpName.Mconvert, t1, t);
        t = t1;
    }
    if (target.isRef())
        ct.genCode(OpCodes.OpName.McopyToRef, x, t);
    else
        ct.genCode(OpCodes.OpName.Mcopy, x, t);
}
```

In addition, any occurrence of variables or expressions in the code generation methods will require checking for floating point or integer variables. There are several changes and all are similar to the following, for the class `IdentExp`.

```
public SymbolTableEntry generateIMCode(CodeTable ct, SymbolTable st) {
    // determine symbol table entry and type for 'name'
    AddressableEntry var = (AddressableEntry)(st.findEntry(name));
    Token type = var.getType();
    // generate new result temporary variable
    TempEntry t = st.addTemp(); t.setType(type);
    OpCodes.OpName instr = null;
    // determine which of four cases we have
    if (var.isRef()) { // is a ref
        if (var.getType().getName().equals("int")) // an int ref
            instr = OpCodes.OpName.McopyFromRefInt;
        else // it must be a float ref
            instr = OpCodes.OpName.McopyFromRefFlt;
    }
    else { // its not a reference
        if (var.getType().getName().equals("int")) // an int
            instr = OpCodes.OpName.McopyInt;
        else // it must be a float
            instr = OpCodes.OpName.McopyFlt;
    }
    // now generate the three-address instruction!
```



```
        ct.genCode(instr, t, var);
    return t;
}
```

### ☛ Activity 110 –

Complete the IP intermediate code generation phase by making appropriate modifications to the methods `generateCode` in the necessary classes of the syntax tree hierarchy. It will be important to reference carefully the IP semantics defined in the chapter. Upon completion test your implementation to insure that nothing from Phase 1 or 2 has been broken and that the new semantics of Phase 3 is properly implemented.

---



## Chapter 26

# IP Common Subexpression Elimination

We have seen in the  $FP^+$  implementation how common subexpression elimination can be carried out automatically on expressions. Adapting the  $FP^+$  elimination process to IP expressions is straightforward. But IP has other computational structures, basic and structured statements, far more complex than the expressions of  $FP^+$ . Extending common subexpression elimination to those structures provides the focus for this phase of the IP project. We will address this project phase in three steps:

**Step 1:** extend the elimination process to IP expressions;

**Step 2:** extend Step 1 to sequences of simple statements (assignment, print, procedure call);

**Step 3:** discuss the extension of Step 2 to include selection and repetition statements.

### 26.1 Step 1 – Extending $FP^+$ Elimination to IP Expressions

The general strategy we applied to  $FP^+$  for common subexpression elimination (see Section 20.2.1) can be directly applied to our IP implementation. All of the work takes place in the **Expression** subtree methods `generateCode`. Remember that these methods return a value which is the result temporary variable for the code generated by the particular `generateCode` method. The temporary variable is understood to be the name of a location where the result of the computation will be stored when computed.

Remember that the elimination process begins by setting a starting point for the searches for matching three-address instructions. Rather than simply generating the appropriate instruction we use a method `ct.search4SameOp` to see if a previously generated instruction carries out the same computation; the value returned by `ct.search4SameOp` is the result temporary variable for the instruction found (or null if not found). The steps of the process are enumerated here:

1. Instantiate a mock instruction object that matches the one to be generated in all but the first address argument – use `null` for that parameter.
2. Call `ct.search4SameOp` with the mock instruction to see if there is an earlier instruction that carries out the same computation – i.e., matches the mock instruction in all but its first address argument.

3. If `search4SameOp` returns `null` then a match was not found and `genCode` must be called to generate the appropriate instruction. Remember the result temporary variable for that instruction
4. If the search returns a non-null value, that value is a reference to the earlier matching instruction.
5. Have `generateCode` specify the temporary variable from step 3 or 4 as its return value.

### Activity 111 –

To reinforce your memory, translate each of the following IP expressions into two different sequences of three-address code: one using no subexpression elimination and the other employing the subexpression elimination process. Assume all variables are integers.

1.  $x * (x + y)$

**Answer:**

CopyInt(t1,x,-)	CopyInt(t1,x,-)
CopyInt(t2,x,-)	CopyInt(t2,y,-)
CopyInt(t3,y,-)	AddInt(t3,t1,t2)
AddInt(t4,t2,t3)	MultInt(t4,t1,t3)
MultInt(t5,t1,t4)	

2.  $(x + y) * (x + y)$
3.  $x + ( f(x,x+y) * f(x,x+y) )$

The first step in extending the subexpression elimination to IP is to identify all the statements in IP which have expressions as components. In fact, each statement including the return statement has an expression. We must edit the `generateCode` method in each statement type's syntax tree class and insert a call to `ct.setStart()` just before the call of `generateCode` on the classes expression data member. That may sound a bit complicated, so here is the code that should result for the class `Assignment`.

```
// Assignment.generateCode
public void generateCode(SymbolTable st, CodeTable ct) {
** ct.setStart();
    TempEntry t = exp.generateCode(st, ct);
    Token sourceType = t.getType();
    Token targetType = target.getType();
    if ((sourceType.getName()).equals("int") && (targetType.getName()).equals("float"))
        OpCode op = new OpCode(OpCode.OpName.Mconvert, null, t);
        SymbolTableEntry temp = ct.search4SameOp(op);
        if (temp == null) {
            temp = st.addTemp(sourceType, false);
            ct.genCode(OpCode.OpName.Mconvert, temp, t);
        }
    }
}
```

```

        t = temp;
    }
}
if (target.isRef())
    ct.genCode(OpCodes.OpName.McopyToRef, x, t);
else
    ct.genCode(OpCodes.OpName.Mcopy, x, t);
}

```

The line marked by ‘\*\*’ shows where the `setStart` call is made.

Having set a starting point for the subexpression searches to start, we must move on to modifying the `generateCode` methods in the `Expression` hierarchy. A problem we encounter in IP is that depending on the types involved and/or the nature of a parameter, there may be multiple possibilities for the mock instruction. In this case mock instructions of each kind must be generated. An example will illustrate the nature of the modifications needed.

```

// IdentExp.generateCode
public SymbolTableEntry generateCode(SymbolTable st, CodeTable ct) {
    SymbolTableEntry pe = st.findEntry(name);
    // 1 create one of two possible mock instructions
    OpCode op = null;
    if (pe.isRef())
        op = new OpCode(OpCodes.OpName.McopyFromRef, null, pe);
    else
        op = new OpCode(OpCodes.OpName.Mcopy, null, pe);
    // 2 search for matches
    // 4 this also sets the value if null isn't returned!!
    SymbolTableEntry temp = ct.search4SameOp(op);
    if (temp == null) {
        // 3 if null generate a real instruction (one of two forms!)
        temp = st.addTemp();
        if (pe.isRef())
            ct.genCode(OpCodes.OpName.McopyFromRef, temp, pe);
        else
            ct.genCode(OpCodes.OpName.Mcopy, temp, pe);
    }
    // 5 return the appropriate result temporary variable
    return temp;
}

```

### Procedure Call Argument Lists

There is one unusual, but not difficult, case to mention – so it is not forgotten in the rush. A procedure call contains a list of expressions, so we should be able to apply the elimination process to them. We could reset the search starting point before processing each argument, but there is an alternative. Because at run time all the parameters will be evaluated before the procedure is

called, there can be no changes in the state of the variables in the argument list. So we can do the subexpression elimination across all the arguments – set the starting point before the loop processing the arguments. Notice that this is done automatically for function calls since function calls are expressions.

### ☛ Activity 112 –

Following the previous two code examples, implement common subexpression elimination on IP expressions. The way to test your implementation is to carefully construct examples and check that the appropriate subexpressions have been eliminated. You will want to make your programs simple with appropriate expression examples. Read the generated intermediate code to see if the elimination is correct.

---

## 26.2 Step 2 – Extending Subexpression Elimination I

In this section we address the implementation of subexpression elimination across sequences of basic statements, meaning assignment, print, and procedure call statements. The problem that emerges in this situation is that the state of the computation can change. When a variable's value changes earlier instructions involving that variable are no longer valid for reuse. We will investigate the problem and then point the direction to an appropriate implementation.

### 26.2.1 Understanding the Problem

The subexpression elimination strategy discussed above is a simple one – but we can do better. Consider the following sequence IP statements.

```
x = (a + b) * (b - a);
print (a + b);
```

If we apply our Step 1 strategy to these two statements we will be able to eliminate the second loads of `a` and `b` in the assignment and that's all. But it is easy to see that since `a` and `b` don't change in the first assignment, we should be able to eliminate the expression `a + b` from the print statement, making use of the computation in the first assignment. Here are two translations for this sequence.

CopyInt(t1,a,-)	CopyInt(t1,a,-)
CopyInt(t2,b,-)	CopyInt(t2,b,-)
AddInt(t3,t1,t2)	AddInt(t3,t1,t2)
SubtInt(t4,t2,t1)	SubtInt(t4,t2,t1)
MultInt(t5,t3,t4)	MultInt(t5,t3,t4)
CopyInt(x,t5,-)	CopyInt(x,t5,-)
CopyInt(t6,a,-)	PrintInt(t3,-,-)
CopyInt(t7,b,-)	

```
AddInt(t8,t6,t7)
PrintInt(t8,-,-)
```

Here's another example that is a bit more interesting.

```
a = (x + b);
x = (x + b) * (b - a);
print (x + b);
```

In visually scanning this example for common subexpressions we see there are four uses of `b` and three of `x + b`. Once we have a copy instruction for `b` in the first assignment we will be able to reuse it three times. Is the same true for the `x + b`? Sadly, no! Because the value of `x` changes in the second statement we will not be able to reuse the `x + b` computation in the third assignment. Can we use it in the second assignment? Happily, yes! The expression in the second assignment is evaluated before the value of `x` is changed. Here is the optimal three address code for the three statements.

```
CopyInt(t1,x,-)
CopyInt(t2,y,-)
AddInt(t3,t1,t2)
CopyInt(a,t3,-)
CopyInt(t4,a,-)
SubtInt(t5,t2,t4)
MultInt(t6,t3,t5)
CopyInt(x,t6,-)
CopyInt(t7,x,-)
AddInt(t8,t2,-)
PrintInt(t8,-,-)
```

### Activity 113 –

Show how the code generation will change if we change the first assignment statement in the most recent example has `b` on the left side of the assignment.

```
b = (x + b);
x = (x + b) * (b - a);
print (x + b);
```

---

While we didn't look at an example with a procedure call, clearly if a procedure call has no reference variables then it will be just like a print statement – it will have expressions but will not change the state of the computation. If there are reference parameters, then each one must be treated as a variable on the left of an assignment – i.e., assume it has changed as a result of calling the procedure. We are left with the question, how can we extend our subexpression elimination strategy to sequences of assignment, print, and procedure call statements?

### 26.2.2 Implementing Subexpression Elimination I

What we learn from the examples above is that as we generate intermediate code we must recognize when a variable changes; based on the location of that change we must modify the range of our search space. So for each variable we should keep track of where it last changed and then use that position to determine the starting point of the next search.

That sounds pretty good. But what if we are confronted with an expression  $x + y$  where the most recent change to  $x$  was at instruction 10 and the most recent change to  $y$  was at instruction 22? Where do we start our search? The answer is obviously to start the search at the most recent of the two positions, i.e., start at instruction 22. What about a more complex expression?

$$x * (x + y*z)$$

As the code generation process hits each operation there are two arguments with two valid start points. It seems that when an operation is applied that the subexpression, for example  $y*z$ ) will get a starting point that is the maximum of the constituent parts. Similarly for  $x + y*z$ , etc.

Each `generateCode` method in an expression class already returns a result temporary variable. We can associate with each variable a start point and then each `generateCode` method will compute a new start point based on those of its constituent parts. If we make the start point a data member of each temporary variable then each `generateCode` method will in essence return both the result temporary variable and the appropriate starting point for that particular expression (as represented by the temporary variable). Actually, while this is a good idea for our current context, it will not extend well when we want to do subexpression elimination over structured statements. An alternative idea is to create a new object, one that binds the variable name with a starting location, and then use this object as the return value of `generateCode`.

Here are the implementation details we have identified. We need to maintain a list of current variable/location bindings – the best place to maintain the list is in the class `CodeTable`, the repository for the generated intermediate code. To store these bindings we will define a new class `Binding`. The definition of `Binding` and its relationship to `CodeTable` are shown in the UML class diagram in Figure 26.2.2. Notice that in `CodeTable` there are methods for not only searching the list `bindings` but also for adding, removing, or replacing bindings. We can now see how these new structures are used to implement subexpression elimination by looking at the new `generateCode` methods in a couple of the syntax tree classes.

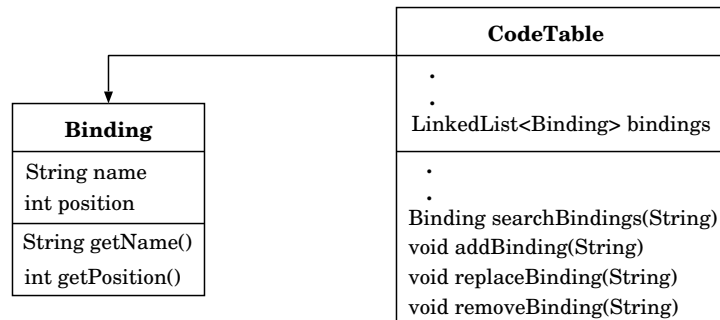


Figure 26.1: Integrating List of Bindings into `CodeTable`



To begin we look back to page 457 to see the code for `generateCode` in `Identifier`. This code implements the expression only version of elimination. There are a few things that must be updated in the code. First, before calling `search4SameOp` we must know where to search. We can find this out by recovering the binding for the identifier in question.

```
Binding bind = ct.searchBindings(name);
```

Assuming we actually found a binding, then we need to call `ct.setStart()` before calling `search4SameOp`. At the end, rather than simply returning the temporary variable for the variable, we need to return the current binding for the temporary variable, whether it was created or found earlier.

Before looking at the code, there is a curious result we can glean from the innocuous call to `searchBindings`. What does it mean if the search returns a `null` value? If the return value is `null` then there mustn't yet be a binding for the variable. But since a binding is created as soon as a value is assigned to the variable, a return of `null` can only mean that the variable doesn't yet have a value. This is an obvious semantic error – we are trying to use a variable in an expression before that variable has been assigned a value! So we should follow the call to `searchBindings` with a check for `null`, with an appropriate `throw` statement in case the check is true.

Here is the updated version of `generateCode`. [Added lines are marked by '\*' on the far left.]

```
public Binding generateCode(SymbolTable st, CodeTable ct) {
    SymbolTableEntry pe = st.findEntry(name);
    * Binding bind = ct.searchBindings(name);
    * if (bind == null)
    *     throw new SemanticException("variable may not have been assigned a value.");
    * ct.setStart(bind.getPosition());
    OpCode op = null;
    if (pe.isRef())
        op = new OpCode(OpCode.OpName.McopyFromRef, null, pe);
    else
        op = new OpCode(OpCode.OpName.Mcopy, null, pe);
    SymbolTableEntry temp = ct.search4SameOp(op);

    if (temp == null) {
        temp = st.addTemp();
        if (pe.isRef())
            ct.genCode(OpCode.OpName.McopyFromRef, temp, pe);
        else
            ct.genCode(OpCode.OpName.Mcopy, temp, pe);
    *   ct.addBinding(temp);
    }
    * return ct.searchBinding(temp.getName());
}
```

The return statement may look a bit odd. When we call `search4SameOp` we get a temporary variable in return; it is either `null` or there exists a binding for the variable. If there is no binding then we create a new one in the last line of the selection. The return statement simply searches for the binding we know to be there.

Another interesting syntax tree class to examine is `Exp`, which generates the code for binary operations. Here is the simple subexpression elimination version of `generateCode`.

```
public SymbolTableEntry generateCode(SymbolTable st, CodeTable ct) {
    SymbolTableEntry tLeft = expl.generateCode(st,ct);
    SymbolTableEntry tRight = expr.generateCode(st,ct);

    SymbolTableEntry try = new OpCode(opName, null, tLeft, tRight);
    SymbolTableEntry temp = ct.search4OpCode(try);
    if (temp == null) {
        // if not then create a new instruction with a new result temporary
        SymbolTableEntry result = st.addTemp();
        ct.genCode(opName, result, tLeft, tRight); // generate the new instruction
        temp = result;
    }
    return temp;
}
```

To modify this implementation we can first take into account that `generateCode` now returns a reference to a `Binding` object; this means replace each `SymbolTableEntry` by `Binding` in the first three lines. What follows the searching should look almost like the corresponding code for the `Identifier` version; the one difference being that the `Exp` version doesn't have to worry about reference variables. What is new in this example is the determination of the search starting point. But the earlier example clearly indicated that we should use the larger of the positions derived from the bindings `tLeft` and `tRight`. The code that results follows.

```
public Binding generateCode(SymbolTable st, CodeTable ct) {
    Binding tLeft = expl.generateCode(st,ct);
    Binding tRight = expr.generateCode(st,ct);

    ct.setStart(MATH.max(tLeft.getPosition(), tRight.getPosition()));
    SymbolTableEntry try = new OpCode(opName, null, tLeft, tRight);
    SymbolTableEntry temp = ct.search4OpCode(try);
    if (temp == null) {
        // if not then create a new instruction with a new result temporary
        temp = st.addTemp();
        ct.genCode(opName, temp, tLeft, tRight); // generate the new instruction
        ct.addBinding(temp.getName());
    }
    return ct.searchBinding(temp.getName());
}
```

Finally, we will see how the technique is applied to `generateCode` in `Assignment`. This is a different sort of problem. Since this is not part of the `Expression` hierarchy we won't have to return a binding. At the same time the method does generate a copy command for a program variable, so a binding has to be established. This new binding will be added to the bindings list and either be new or will replace the current binding for the variable. This functionality is guaranteed by the method `replaceBinding`.

Notice that when we add a temporary variable to the bindings list it cannot replace anything since it was created new for its purpose. So we always use `addBinding` for temporary variables. A copy command with a program variable as target can only occur in the assignment statement and it is here that we use `replaceBinding`.

### ☛ Activity 114 –

Implement the remaining `generateCode` methods for the IP syntax tree. Make other adjustments necessary to test your implementation – this means, determine how to initialize the elimination algorithm for each function definition.

Testing this implementation is tricky since it will break if in the presence of selection or repetition, so use only basic statements in your test programs.

---

## 26.3 Step 3 – Extending Subexpression Elimination II

The last section leaves us in the awkward position of understanding how to eliminate common subexpressions from sequences of basic statements, but not being able to apply that capability to a general IP program – i.e., one containing selection or repetition. In this section we will discuss a particular strategy for solving this problem, i.e., for extending common subexpression elimination across the structured statements of IP. A general solution is complex and beyond the scope of this text, but we will sketch the strategy so that the associated problems become clear. As a consequence you will be able to extend the subexpression elimination in IP to include all basic blocks of statements, even if they occur in selection and repetition statements.

### 26.3.1 Understanding the Problem

When we look at the structured statements in IP, selection and repetition, we find that subexpression elimination becomes considerably more difficult. To understand why we will examine two examples. First, consider the following block of code which contains a simple selection statement.

```

a = x + y;      <<===
if x > y then
    x = y+1;
    print x+y;  <<===
else
    a = x-1;
    print x+y;  <<===
endif

```

This example illustrates some of the difficulties of implementing subexpression elimination in selection statements. First, since `y` is referenced before the selection and not changed in either branch we should be able to eliminate the references to `y` that occur in the selection. How do we detect that? Which references to `x` should we be able to eliminate, through reusing the reference in the first statement? The problematic reference is in the print statement in the true branch, the others

we should be able to eliminate. Again, how do we detect that? Finally, should we be able to reuse the computation of  $x + y$  before the selection in the two branches?

We see similar problems with the repetition structure, even though there is just a single block involved.

```
x = 10;
y = 5
while x > y
    a = y + 3;
    sum = sum + a;
    x = x - 1;
endwh
```

Consider the following possible translation of the first three lines of this example.

```
Copy(t1,10,-)
Copy(x,t1,-)
Copy(t2,5,-)
Copy(y,t2,-)
Label(labelA,-,-)
Gt(t3,t1,t2)
JumpF(t3,label,-)
```

Notice that the `Label` instruction is the target of the `Jump` instruction we know must be at the end of the loop's block.

This translation looks pretty straightforward – we have applied subexpression elimination twice in the loop condition since the values of  $x$  and  $y$  are referenced before the loop. When we look deeper into the loop we see, however, that the value of  $x$  will change. Why is that a problem? The problem occurs when we branch back to `labelA` at the top of the loop. Now the value of  $x$  is no longer the one computed before the loop, but rather the value computed in the loop. So it seems that we cannot eliminate the reference to  $x$ . What about the references to  $y$  in the loop – can they be replaced by the reference before the loop? Here the answer is yes, but again, how can we detect this fact?

The solution to these problems is to adopt a new way of visualizing the generated three-address code, so that the dependencies across the boundaries of the structured statements can be more easily managed.

### 26.3.2 Implementing Subexpression Elimination II

The implementation of common subexpression elimination across code sequences including selection and repetition is complex. In fact these techniques occupy a large percentage of advanced text books on compiler construction.<sup>1</sup> Because the development of a general algorithm subexpression elimination is beyond the scope of this text, we will focus on a more modest goal: implement subexpression elimination across all the blocks of basic statements in a program.

---

<sup>1</sup>Put in refs to dragon book and another ...

While a linear representation is adequate for implementing subexpression elimination over sequences of basic statements, it is not adequate when selection or repetition are present. The structure needed for such a mix of statements is one that matches the ways in which data can flow through a program. The structure we will adopt is called a *flow graph*: the nodes of such a graph are linear sequences of three-address code and the links are determined by jump instructions.

In this section we will discuss one possible definition of the flow graph for three-address code, a process by which we can generate such a graph, and finally possible strategies for optimizing the code represented in the flow graph.

## Flow Graph

Assume that we have the translation of an IP program into three-address code. A flow graph will contain the same sequence of instructions, but they will be distributed into nodes, called *blocks*. Links between nodes are meant to describe transitions into the top of the block and exits from the bottom of a block. In this section we will describe how the blocks are defined and the links between the nodes are defined. The result is the flow graph representation of the original IP program.

The first question is how should blocks be identified. Certainly a block must contain the three-address code generated for sequences of basic statements. But what about the label instructions generated for the structured statements, and the jump instructions? How should they affect the contents of blocks? Since labels mark entry points into the code sequence and links into a node are into the top of the block, it make sense to require labels to always be at the top of a node. Similarly, since jump instructions are transfers out of a code sequence and links from a node leave the bottom of the block, we put all jump instructions at the end of blocks.

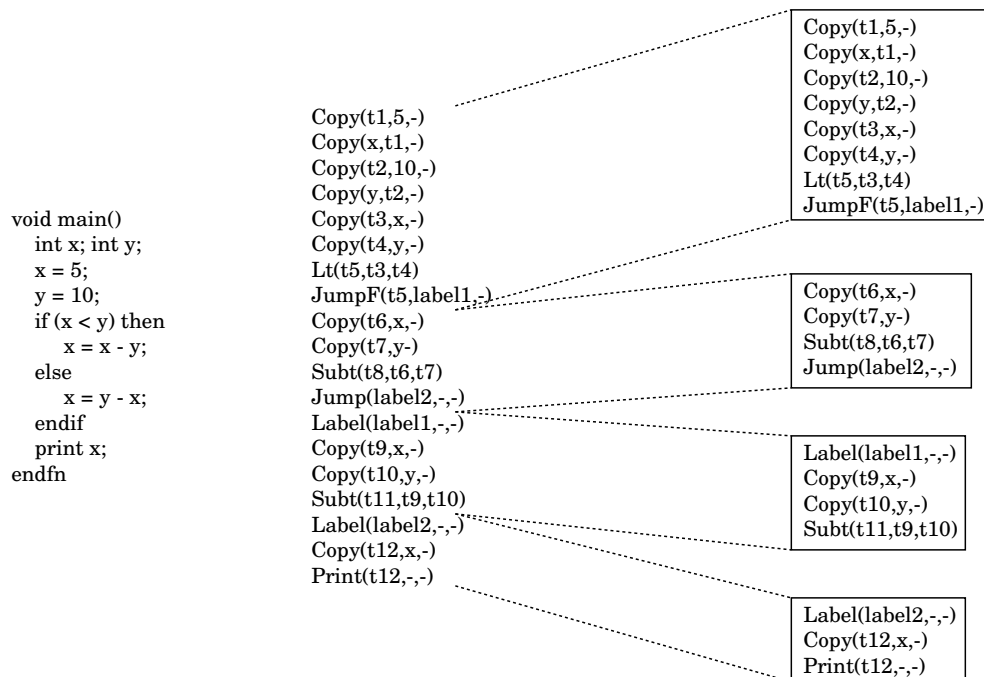


Figure 26.2: A program and its blocks

With these principles in mind, we define blocks via a construction process.

1. We first identify the instructions which can be at the head of a block, called the *leaders*:
  - the first instruction in the program is a leader,
  - any instruction immediately following a conditional jump is a leader, and
  - any statement that is the target of a conditional or unconditional jump is a leader.

Because the target of a jump instruction is always a label instruction, the third category of leaders will always be a label instruction.

2. A *block* is a sequence of instructions which begins with a leader and extends to but does not include the next leader.

By this definition the blocks are maximal sequences of instructions such that flow into the block is always at the top and flow out of the block occurs at the bottom. The block containing the program's first instruction is termed the *initial block*. Figure 26.2 displays three things: on the left is an IP program, in the middle is its translation into three-code instructions (no subexpression elimination applied) and finally on the right is the associated collection of blocks.

Now we can define a *link* in our flow graph. Let  $N_1$  and  $N_2$  be blocks. There is a link from  $N_1$  to  $N_2$  if one of three conditions holds.

- If  $N_2$ 's leader is a label instruction which is the target of a jump in  $N_1$ .
- If  $N_1$  ends with a conditional jump and  $N_2$  naturally follows  $N_1$  in the linear form of the program's translation.
- If  $N_1$  ends with an instruction other than a jump and  $N_2$  naturally follows  $N_1$  in the linear form of the program's translation.

If we apply this definition to three-address code in Figure BlockConversionExample then we produce the graph in Figure 26.3.

### ☛ Activity 115 –

Display the three-address code flow graph for the following IP program.

```

void main()
  int x; int y; int a;
  x = 10;
  y = 25;
  if x > y then
    a = 0;
    while (x < y)
      a = a + 1;
      x = x + 1;
    endwh;
  else
    a = x - y;

```

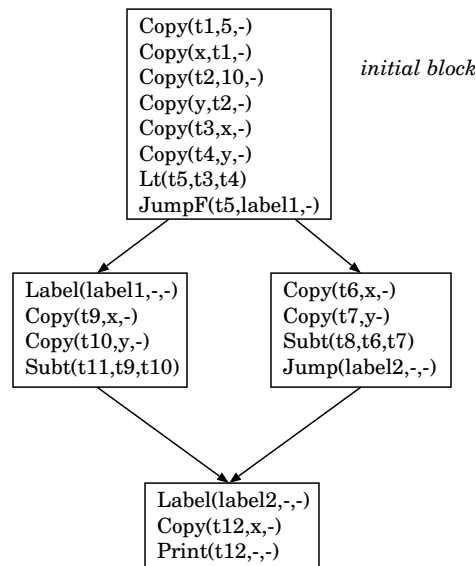


Figure 26.3: The flow graph corresponding to Figure 26.2

```

endif;
print a;
endfn

```

---

## Reconstructing the Three-address Code Sequence

There is another problem that needs to be addressed via the flow graph. Once a flow graph is created and expression elimination carried out, it is necessary to reconstruct a linear form of the three-address code in the graph. For this purpose there is an additional link required in each block. The new link will indicate for each block the *next block sequentially* in the original code sequence. We can see this idea in Figure 26.4, where the new sequential links are heavier than the flow links. With these in place a traversal of the graph via these links will allow us to produce a linear representation of the generated and optimized three-address code.

## Implementing a Flow Graph

It is convenient that the code in the blocks of a flow graph correspond to the maximal sequences of basic program statements (as translated). So within a block we can apply our common subexpression elimination algorithm. In fact, our strategy for generating the flow graph for a program will be to translate the program into blocks of three-address code where the subexpression elimination algorithm has been applied across the statements in the program's corresponding block of code.

The definition allows us to determine the start of each successive block. The generation of the blocks will take place via the `generateCode` methods in the syntax tree classes.

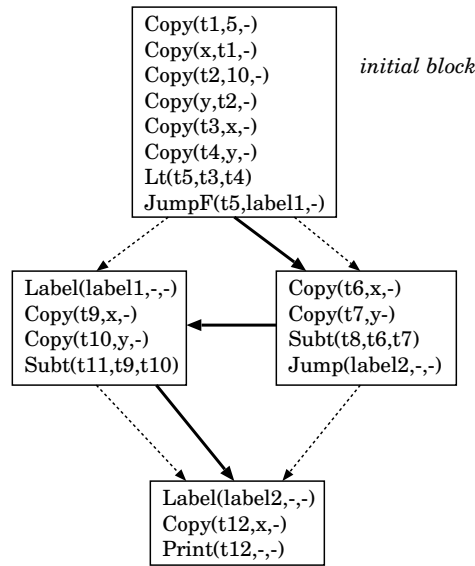


Figure 26.4: The flow graph from Figure 26.3 with sequential links

- A new block is started with the first instruction of each function statement list.
- In a **Selection** object the current block ends with the generation of the **MjumpF** instruction and a new block begins immediately after that. Continuing sequentially, the current block ends with the **Mjump** instruction and the new block begins with the label instruction that follows. That block ends just before the last label generated for the selection.
- In a **While** object a new block begins with the generation of the first label instruction. That block ends with the **MjumpF** instruction following the condition translation and a new block begins immediately after that. Subsequently, the current block ends with the **Mjump** instruction (at the bottom of the loop) and a new block begins with the label instruction that follows.
- The final block ends when the current function’s statement list has been processed.

Notice that in this process the sequential links can be generated directly from each block to the next. The flow links will be determined by locating the jump instructions and linking that block to the block containing the target label.

This construction description is adequate for purposes of implementing the the project’s subexpression elimination across basic blocks, which is the topic of the next section.

### 26.3.3 Eliminating Subexpressions

From the previous section you may have had the impression that you will have to implement a new structure for the flow graph. Since our goal is to apply subexpression elimination to the basic blocks, our only need is to identify the blocks – the flow links are not relevant. It turns out that we can do this implementation as we translate to three-address code. From the description in the previous section you should be able to identify in the **generateCode** methods where each block begins. At that point the subexpression elimination process can be initialized (or re-initialized).



As code generation proceeds the elimination of subexpressions will take place naturally but only within the current block. Once again using the description in the previous section, you will be able to identify where in the `generateCode` methods a block terminates. Interestingly, no action will be required at that point, because the more interesting point is where the next block begins!

So as intermediate code generation proceeds subexpression elimination will be constantly applied, but with each entry to a new block, the reference point for the elimination process is reset.

### ☛ Activity 116 –

You have already implemented the subexpression elimination algorithm for basic blocks. Now make the changes to your implementation to appropriately reset the search starting point at the entry to each flow graph block. Having done this you should have an intermediate code generator which generated three-address code which is optimized across all the basic blocks.

It is important to test this implementation thoroughly since this will be the input to the object code generation in the final development phase.

---

#### 26.3.4 Generalizing Subexpression Elimination

The real problem with extending subexpression elimination across block boundaries is one of tracking what bindings are valid on entering a new block. The problem is not too difficult if the only structure available is selection – repetition is the real culprit. Though we will not solve this problem here, we will discuss the strategy and the data structures necessary to set up a solution.

The first thing to consider is how the elimination applied to one block can affect the elimination in a block that follows. If we think about the initial block of a flow graph, we know that when we reach the end of the block subexpression elimination will have produced a list of bindings. As of the previous section, when we arrive at the next block we will restart with an empty list of bindings. Now if we want the bindings from the initial block to be available in the second block we have to know how to form a new binding list on entry to the second block. This seems obvious – just use the final binding list from the initial block as as the initial binding list of the second block. If the second block has only one input link and it is from the initial block then this strategy works fine. The problem is when there are two or more inputs!

#### in and out Lists

We will define two concepts the *in* binding list and the *out* binding list for a block. The idea is to be able to take a flow graph and determine the in and out lists for each block. Notice that for a block, once we know the in list for a block, our subexpression elimination algorithm will determine the out list for the block. So determining the in list for a block is the interesting question.

Consider the flow graph in Figure 26.5 where block *A* has two inputs coming from blocks *B* and *C*. The lists out(*B*) and out(*C*) are the reasonable candidates for in(*A*). Remember that if we encounter a reference to a variable *x* in block *A*, we would like to be able to use a previous

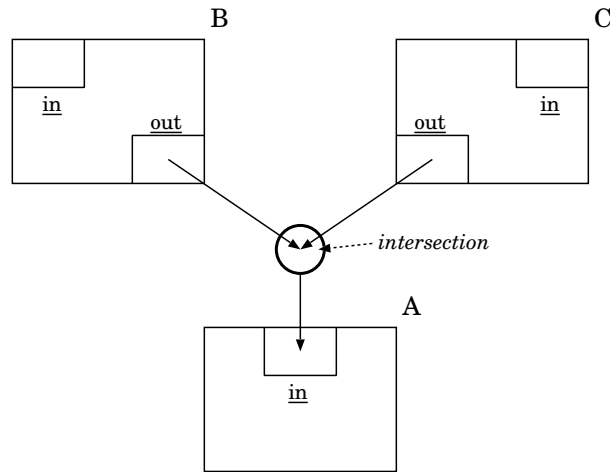


Figure 26.5: Determining the in list from two out lists

computation if possible. So if there is a binding for  $x$  in  $\text{out}(B)$  then we could use that. But what if we happen to follow the path from  $C$  in getting to the  $x$  in  $A$ ? There had better be a binding for  $x$  in  $\text{out}(C)$  as well. That's still not good enough, because if the binding from  $B$  assigns value 5 to  $x$  and that from  $C$  binds 10, then it won't work! Apparently what we must demand is that the bindings for  $x$  be identical along the two paths. So we've learned something interesting – the following identity which relates the in/out lists of linked blocks.

$$\text{in}(A) = \text{out}(B) \cap \text{out}(C)$$

This sounds too easy. The idea that  $\text{in}(A)$  is simply the intersection of two out lists must mask another problem. Consider the IP code segment and flow graph shown in Figure 26.6. We have exactly the situation just discussed. So is it easy to compute the in list for the block labeled  $A$  in this new diagram?

If we trace the links into the block  $A$  we see one comes from the initial block, no problem there, but the other one comes from the block  $B$  which is ultimately linked to from  $A$ . We have a cycle in the flow graph and the one of the inputs to  $A$  comes indirectly from  $A$ . So the determination of  $\text{in}(A)$  is ultimately dependent on  $\text{out}(A)$  which derives from  $\text{in}(A)$ ! This is why extending subexpression elimination across structured statements is difficult.

What we have learned is that in order to implement such an extension to subexpression elimination we must define a data structure to hold the flow graph and that each block in the flow graph must have in and out lists. Computing those lists is the hard part, and we leave discussions of the theory and implementation to more advanced books on compiler construction<sup>2</sup>.

<sup>2</sup>Give a couple of references.

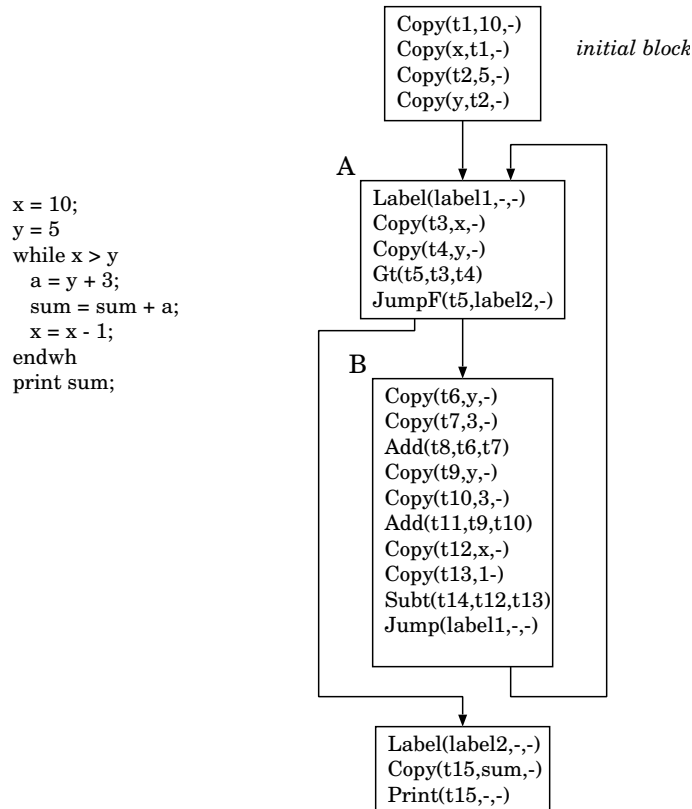


Figure 26.6: The flow graph of a repetition



## Chapter 27

# IP Object Code Generation

The final phase of the IP project is to implement object code generation. Because we do object code generation starting with three-address code and because our target machine is the MIPS, we will be able to use the translation strategies employed in the FP<sup>+</sup> object code implementation. The major change from the that implementation is the introduction of reference parameters for procedure calls, the printing capability, and the floating point arithmetic. In this chapter we will discuss the new code generation problems presented by these IP features and also some object code optimizations.

As we did for the intermediate code generation portion of the project, we will unfold the implementation of object code generation in terms of the same three incremental development phases:

**Phase 1:** Generate object code for the base IP language which has only integer variables and literals and allows no reference parameters.

**Phase 2:** Add to the Phase 1 implementation object code generation for the reference parameter capability.

**Phase 3:** Add to the Phase 2 implementation object code generation for floating point literals, variable types, and mixed-mode comparison and arithmetic expressions.

### 27.1 Phase 1 – The Base Implementation

A review of the Phase 1 IP intermediate code description will indicate that there are just three new three-address instructions not already used for the FP<sup>+</sup> implementation: the instructions `Preturn`, `Pcall` and `PrintInt`.

`Pcall`

The translation here is a small variation on the `MFIntCall` translation in FP<sup>+</sup>. The difference is that for a procedure call there is no return value to process.

```
FunctionEntry fe = (FunctionEntry)(op.getAdr1());
genCode("addi $sp, $sp, -8");
genCode("sw $fp, 4($sp)");
genCode("sw $ra, 0($sp)");
```

```

genCode("addi $fp, $sp, 4");
genCode("jal " + fe.getName() );
genCode("lw $fp, 4($sp)");
genCode("lw $ra, 0($sp)");
vInt = fe.getNumParams();
genCode("addi $sp, $sp, " + (8 + vInt*4));

```

### Prereturn

As in the translation of `Pcall`, this translation is the `MFIntReturn` without the processing of a return value.

```

vInt = ((IntegerEntry)(op.getAdr1())).getValue();
genCode("addi $sp, $sp, " + (4*vInt));
genCode("jr $ra");

```

### <PrintInt,t>

The `xspim` simulator provides several system calls to facilitate input and output of data. Each system call has an identifying integer code which must be loaded into the register `$v0` – for integer printing system call the code is 1. Here is the appropriate code to generate for this three-address instruction.

In the `PrintInt` instruction the `t` is assumed to be a register holding an integer value and the instruction specifies to display the value on the standard display. Notice that the system call requires the value of `t` be in register `$a0`.

```

li $v0, 1
add $a0, t, 0
syscall
jr $ra

```

### ☛ Activity 117 –

As indicated above, the `spim` system provides several system calls to facilitate input and output. Add a new statement type to `IP` which implements reading a single integer value. The syntax for the statement should be `'read x'`, where `x` is an integer variable or parameter. In integrating this statement into the implementation notice that its execution changes the value of its arguments, so this will have an impact on bindings used in subexpression elimination.

---

### ☛ Activity 118 –

Integrate register allocation into the object code generation for Phase 1. It is a good idea to do this in steps; first implementing the general purpose registers as an unbounded cache and, when that is completed and tested, extending the implementation to implement a bounded cache.

---

## 27.2 Phase 2 – Adding Reference Parameters

There are three new integer-based instructions, `CopyIntToRef`, `CopyIntFromRef`, `CopyAddr`, introduced into the IP three-address code to facilitate the implementation of reference parameters. Their use in the intermediate code generation for IP is discussed in Section 25.2 and their translation to MIPS assembler is summarized in the following table.

<CopyIntToRef, x, y>

In these first three instructions the assumption is that the value at `y` is an address. In this case the following translation is appropriate:

```
lw $t1, x
sw y, ($t1)
```

<CopyIntFromRef, x, y>

```
lw $t1, y
sw ($t1), x
```

<CopyAddr, x, y>

```
la $t1, y
sw $t1, x
```

### ☛ Activity 119 –

Add these translations to your Phase 1 implementation. Demonstrate your implementation works by translating appropriate Phase 2 type IP programs and executing the resulting MIPS assembler code on the SPIM simulator.

---

## 27.3 Phase 3 – Integrating Floating-point Capabilities

Before delving into the code generation problems posed by floating point arithmetic, it is important to understand what is offered by the MIPS processor. This section begins with a review of the MIPS floating point functionality and is followed by the discussion of three-address code translation for the IP floating point capabilities.

### 27.3.1 MIPS Floating Point Architecture Reviewed

Since IP only supports one floating point type, it should come as no surprise that that floating point type is the 32-bit flavor. This means that we can ignore all those double floating point capabilities of the MIPS architecture.

The first architectural structure to be aware of is the floating point co-processor – co-processor 1 (CP1). This processor provides all the floating point functionality and has its own set of general purpose registers – there are 32 of these (which means there can be 16 double registers). Co-processor 0 (CP0) is the stack-based general purpose processor, which provides all integer capabilities. There are a few categories of floating point instructions that will be of use in our object code generation. The technical details of the instructions are summarized in the table in Figure 27.1.

Load and Store	
<code>li.s fd number</code>	<code>number</code> is a floating point literal
<code>l.s fd address</code>	copy value at <code>address</code> to CP1 register <code>fd</code>
<code>s.s fd address</code>	copy value in CP1 register <code>fd</code> to <code>address</code>
Data Movement	
<code>mtc1.s rs fd</code>	copy value in CP0 register <code>rs</code> to CP1 register <code>fd</code>
<code>mfc1.s rd fs</code>	copy value in CP1 register <code>fs</code> to CP0 register <code>rd</code>
Arithmetic Operations	
<code>add.s fd fl fr</code>	<code>fd = fl + fr</code>
<code>sub.s fd fl fr</code>	<code>fd = fl - fr</code>
<code>mul.s fd fl fr</code>	<code>fd = fl * fr</code>
<code>div.s fd fl fr</code>	<code>fd = fl / fr</code>
<code>neg.s fd fs</code>	<code>fd = -fs</code>
Comparison Operations	
<code>c.eq.s fl fr</code>	set CP1 condition register to 1 if <code>fl == fr</code>
<code>c.lt.s fl fr</code>	set CP1 condition register to 1 if <code>fl &lt; fr</code>
<code>c.le.s fl fr</code>	set CP1 condition register to 1 if <code>fl &lt;= fr</code>
Branches	
<code>bclt label</code>	branch to <code>label</code> if condition code of CP1 is true
<code>bclf label</code>	branch to <code>label</code> if condition code of CP1 is false
Conversion	
<code>cvt.s.w fd fs</code>	<code>fs</code> contains integer and <code>fd = float(fs)</code>

Figure 27.1: MIPS 32-Bit Floating Point Instructions



### 27.3.2 Three-address Code Translation

The MIPS floating point capability requires careful attention in the design of object code generation. The approach of this section will be to list *all* three-address instructions and indicate for each the translation modifications required.

<MfltLit, t, n>

CP1 has a load immediate instruction corresponding to that of CP0.

```
li.s $f0, n.getValue
sw $f0, t.offset($fp)
```

<Mconvert,x,y>

In this instruction the value y is a variable containing an integer value – the instruction will convert the value to 32-bit floating point form and store it at x.

```
la $t1, y
cvt.s.w $t2, $t1
sw $t2, x
```

<McopyFlt, t1, t2>

The conversion to floating point here is easy.

```
l.s $f0, t2.offset($fp)
s.s $f0, t1.offset($fp)
```

<MfltNeg, t1, t2>

```
l.s $f0, t2.offset($fp)
neg.s $f1, $f0
s.s $f1, t1.offset($fp)
```

<MfltAdd, t1, t2, t3>

There are four arithmetic floating point instructions (MfltAdd, MfltSubt, MfltMult, MfltDiv) and their translations is demonstrated in the translation for MfltAdd.

```
l.s $f0, t2.offset($fp)
l.s $f1, t3.offset($fp)
add.s $f2, $f0, $f1
s.s $f2, t1.offset($fp)
```

<MfltEq, t1, t2, t3>

There are six three-address floating point comparison instructions (MfltEq, MfltNe, MfltLt, MfltLe, MfltGt, MfltGe) but only three MIPS floating point comparison operations (equal, less than, less than or equal). For this reason, for the three instructions MfltNe, MfltGt, MfltGe, we will use their negative operation and report the opposite result – i.e., for  $x > y$  we will compute the value  $x \leq y$  and report its negation. Since we want these instructions to behave as the integer instructions, we will generate a value of 0(false) or 1(true) and store that in the temporary result variable.

```

l.s $f0, t2.offset($fp)
l.s $f1, t3.offset($fp)
c.eq.s $f0, $f1
addi $t0, $zero, 1      // assume the result was true
bclt 4                  // branch around the next word
addi $t0, $zero, 0      // reset if result was false
sw $t0, t1

```

For `MfltGt` we take the opposite approach – we use the opposite operation and then store in `$t0` the negation of the result.

```

l.s $f0, t2.offset($fp)
l.s $f1, t3.offset($fp)
c.le.s $f0, $f1
addi $t0, $zero, 0      // assume the result was true (report false)
bclt 4                  // branch around the next word
addi $t0, $zero, 1      // reset (to true) if result was false
sw $t0, t1

```

With these translations, we can then use the standard `JumpF` without worrying about the types of the expressions compared.

<PrintFlt,t>

The differences for printing with floating point involve a different system call code and a floating point register for holding the value to print.

```

vInt = ((AddressableEntry)(op.getAdri())).getOffset();
genCode("li $v0, 2");
genCode("l.s $f12, " + vInt + "($fp)");
genCode("syscall");

```

<MfFltCall, t, fn>

The translation for `MfFltCall` is almost the same as that for `MfIntCall`. The one difference is that the return value is returned in `$f0`. This translation assumes that the return value will already have been converted to floating point form if necessary.

```

addi $sp, $sp, -8
sw $fp, 4($sp)
sw $ra, 0($sp)
addi $fp, $sp, 4
jal fn.name
lw $fp, 4($sp)
lw $ra, 0($sp)
addi $sp, $sp, 4*(2+fn.getNumParams)
s.s $f0, t.offset($fp)

```

<MfltReturn, t, n>

The return for a floating point function must retrieve its return value from the specified register `$f0`.

```
l.s $f0, t.offset($fp)
```

If the type of `t` is `int` then we must introduce the following instruction.

```
cvt.s $f0 $f0

addi $sp, $sp, (4*n.getValue)
jr $ra
```

### ☛ Activity 120 –

Implement the translations described above. This should complete the IP compiler. Be sure to adequately test the finished system.

---

### ☛ Activity 121 –

Add a statement `read` to the IP language. This means make the appropriate changes to the tokenizer, parser, syntax tree structure, etc., through final code generation. It is important to remember that in this instruction the argument to `read` will be treated as a reference parameter! Treat it accordingly in program statement translation.

---

### ☛ Activity 122 –

Add IP the `for` loop and/or a `do...until` loop – as in the previous exercise this involves changes to all components of the compiler. The `for` statement will not allow a declaration in its first argument. Also, the condition component should use the `<BExp>` syntax of IP and FP<sup>+</sup>. Think carefully about the semantics of each statement and the appropriate intermediate code to generate.

---

### ☛ Activity 123 –

Add the type `boolean` type to the language. This should include allowing full boolean expressions for loop and selection conditions. Begin with a straightforward implementation. When that works change the evaluation strategy for the expressions to do short-circuit evaluation.

---

## 27.4 Peephole Optimizations

In Chapter 22 we discussed the most critical object code optimization strategy – the allocation and management of general purpose registers. While register allocation is the most critical of optimizations, there are many other optimization strategies that increase the computation efficiency of generated object code.

The general strategy that we will discuss in this section is called **peephole optimization**. The idea is to scan the generated object code and watch for particular sequences of instructions that can be altered or eliminated. The size of the peephole is the number of instructions in the pattern being optimized. Peephole optimization is often driven by particular characteristics of the target processor.

**Madd a b c** – This 3-address instruction specifies that **a** will get the sum of **b** and **c**. This expression is translated into the following sequence of MIPS instructions:

```
lw b $t0
li c $t1
add $t2 $t0 $t1
sw $t2 a
```

If either of the values **b** or **c** happens to be a numeric literal then the code can be optimized by using the **addi** (add immediate) instruction. The restriction is that the numeric literal must fit into a 16-bit field in the MIPS instruction format. Assuming that **c** is the literal and has a value between 0 and  $2^{16} - 1 = 65535$ , then the translation can be the following:

```
lw $t0 b
addi $t1 $t0 c
sw $t1 a
```

Of course a similar translation can be done if the argument **b** is the literal.

The translation of these cases can be handled as special cases in the code generator when an Add 3-address instruction is encountered. The special case is checked for first, being sure to check that the numeric value fits within 16-bits.

**Mmult a b c (or Mdiv a b c)** – The **Mmult** operation specifies the multiplication of **b** and **c**, with the result stored in **a**.

```
lw $t0 b
lw $t1 c
mult $t0 $t1
mflo $t2
sw $t2 a
```

If it happens that one of the multiplication arguments is a numeric literal which is a power of 2, and that power is less than 32, then we can replace the multiplication instruction by a shift instruction, which is much faster. This also saves the loading of the literal since the shift instruction takes the shift amount as a 5-bit argument. Assuming that **c** is  $2^n$ , where **n** is less than 32, we can produce the following MIPS code.

```
lw $t0 b
sll $t1 $t0 n
sw $t2 a
```

A similar translation works for the case where `b` is the literal.

The case of division is similar except for two things. First, since division is not commutative, it is the second argument `c` which must be the power of 2. Second, for division we must shift the register right, that means using the MIPS instruction `srl`.

```
sw a b, lw a b or s.s a b, l.s a b
```

These sequences are often generated and may even occur after other optimizations have taken place. In the two cases the load instruction is not necessary.

These cases illustrate how one can use the features of the instruction set to generate better code in often fairly specific situations. In the examples above we gain two things: the new code uses fewer registers, thus freeing up a register for other uses; the code is shorter and uses faster instructions, so the code will execute faster.

It is important to notice that these optimizations cannot be carried out during intermediate code generation since 3-address instructions don't relate directly to processor instructions. Since these optimizations derive directly from the characteristics of the target processor they must be applied in the object code generation phase.

There is another set of optimizations that, while not really driven by processor characteristics, are conveniently handled during object code generation. There are several particular patterns we can watch for in the arithmetic instructions that will allow us to apply the following well known algebraic identities:

$$a + 0 = a, a - 0 = a, a * 1 = a, a / 1 = a, a * 0 = 0, 0 / a = 0$$

For the first four of these identities, when we find a 3-address instruction whose operands match the pattern on the left, then we need generate no code. For addition and multiplication, which are commutative, we will also watch for the reversed order of operands as well. For the last two identities, we will simply generate code to store the value zero at the designated place. The following table summarizes the code to be generated.

3-address code	object code
Madd x a 0, Madd x 0 a	no code
Mmult x a 1, Mmult x 1 a	no code
Msub x a 0	no code
Mdiv x a 1	no code
Mmult x a 0, Mmult x 0 a	sw \$zero x
Mdiv x 0 a	sw \$zero x
Mdiv x a 0	throw a translation exception!



# Appendix A

## Parse Table for FP

The language FP is right at the margin of allowing for hand-construction of a LR(0) transition graph. While a usable parse table is not presented here, a sketch of the LR(0) transition graph is displayed. "Sketch" means that the graph is not given in its complete form, but rather is given as a base diagram and then three other diagrams which give patterns that could be used in the construction of the final graph.

The complexity in the FP transition graph is due to the complexity of FP expressions. The base graph displayed in Figure A.1 shows the states which derive from that part of the grammar describing the function definition lists. The difficult part of the graph is clearly signaled by the pseudo transition to a circled state labeled "Patterns." This transition is meant to say, "there are transition patterns that occur frequently and they derive from states like this one." Because of the way in which transition graphs are defined, when a dotted form appears in a state and the dot appears before a non-terminal then a sequence of left-most dotted forms are added to the state based on the non-terminal. If the non-terminal to the right of the dot is **Exp** then the last 13 dotted forms seen in **St11** must appear.

As a consequence there must be transitions from **St11** for each of the following symbols:

**FnApp**, **Select**, **MExp**, **Term**, **Fact**, **if**, **id**, **int**, **'-'**, **'('**

There are two display problems. First, there are at least 10 transitions for any state containing a dotted form with the dot before **Exp** — that's a lot of transitions. Second, since such dotted forms appear frequently in the FP transition graph, there will be a lot of states with this same pattern of transitions. Thus, the transition to the circled "Patterns" means the transition patterns are exposed in the other three diagrams.

Figure A.2 shows the transition graph appropriate for a transition on the symbol **'-'**. Notice that there is another "Patterns" transition from **St24**. In addition, there is an implicit sequence of transitions, the four transitions from **St21**, which are present if a state has a dotted form with **Fact** following the dot. These four transitions can also be implied by the "Patters" transition.

Another required transition from **St11** is on the symbol **if**. The resulting transition graph is displayed in Figure A.3. Notice that there are four "Patterns" transitions, two from states with **Exp** following a dot and two from states with **Fact** following a dot. In each case you should understand the implied pattern of transitions.

Finally, in Figure A.4 we see the transition patterns from **St11** determined by the symbols **MExp**,

**Term** and **Fact**. Again we see occurrences of "Patterns" transitions for **Exp** and **Fact**. The four diagrams provide all the information needed to derive a complete LR(0) transition graph for FP .



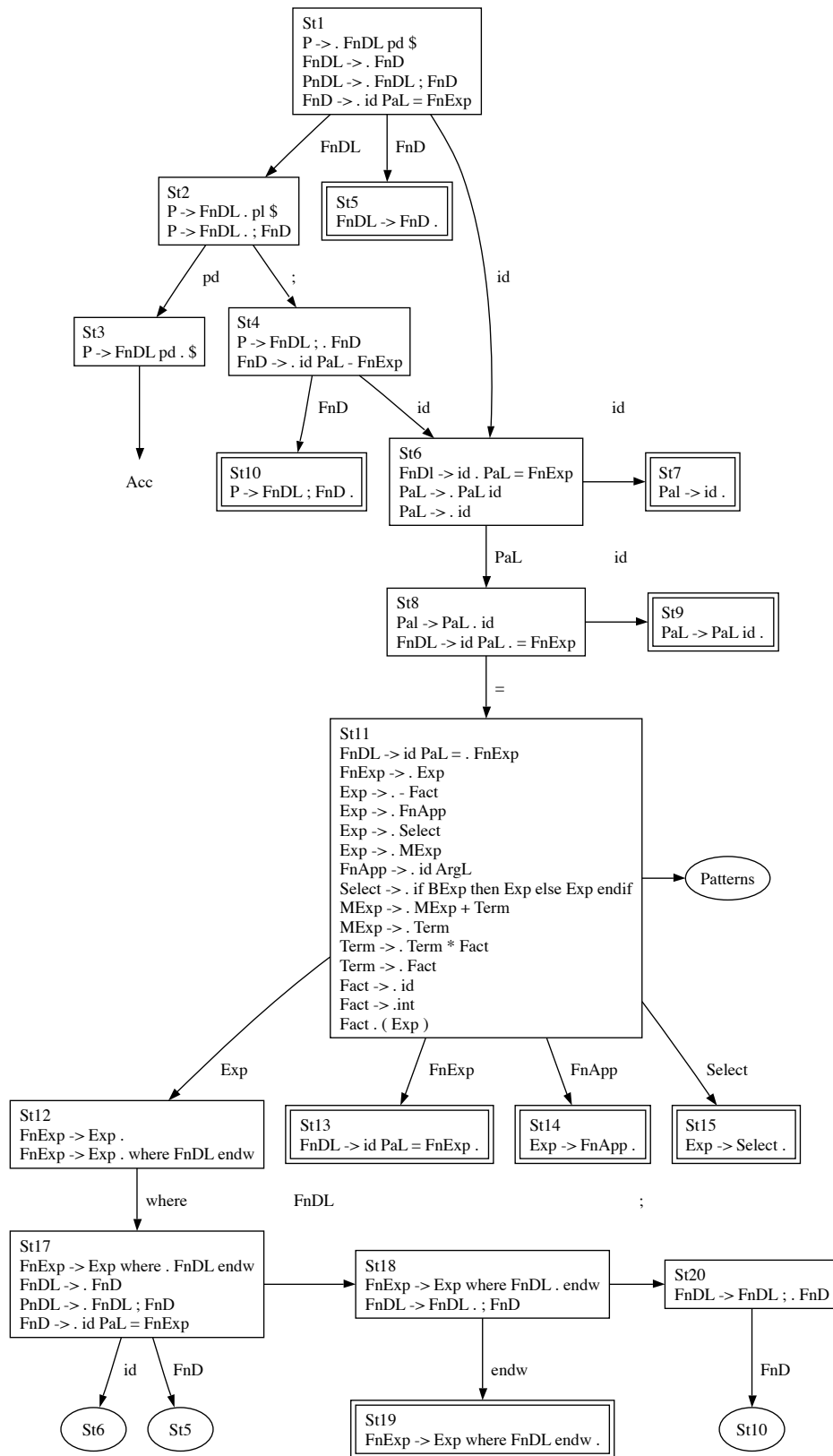


Figure A.1: Transition Graph I for FP

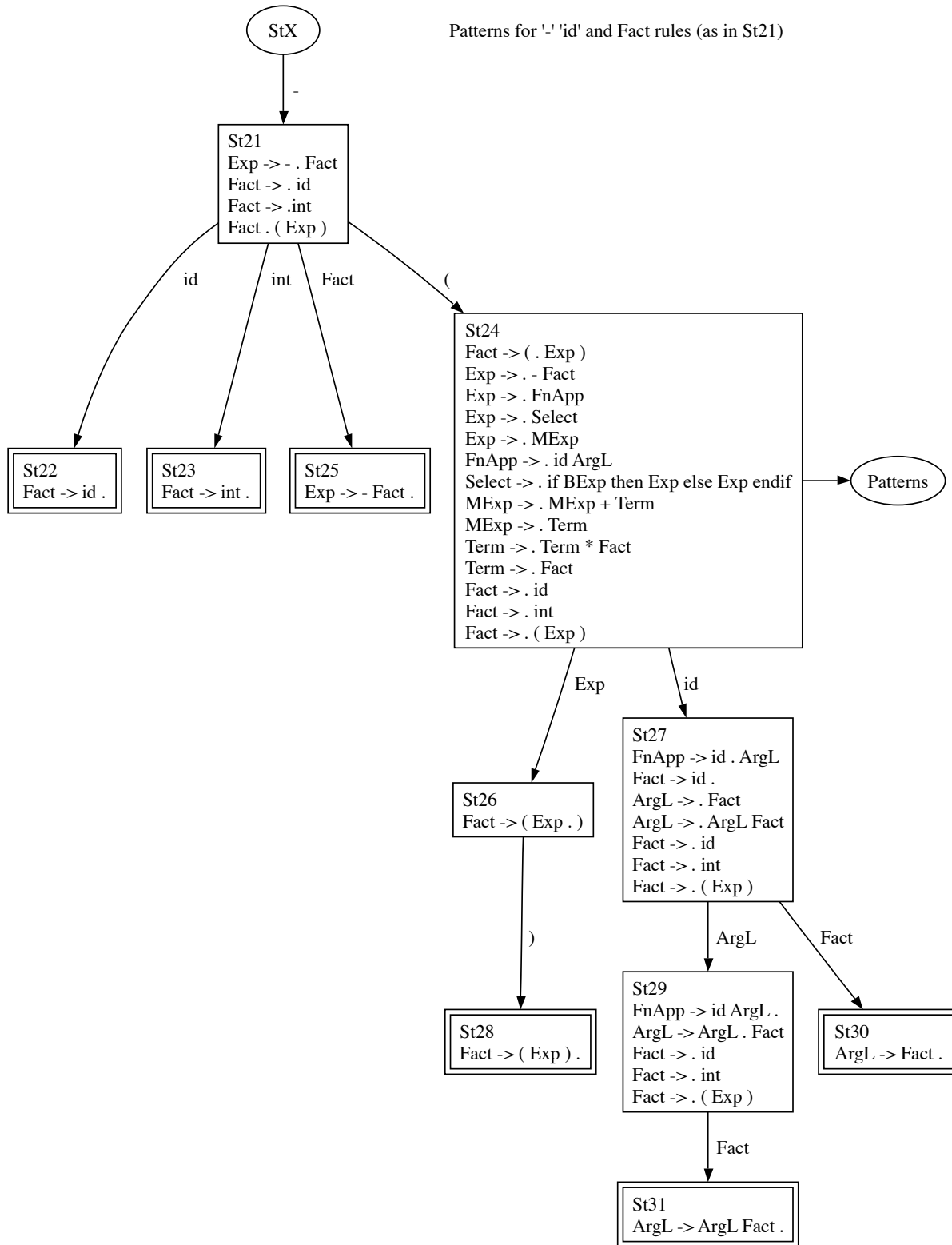


Figure A.2: Transition Graph II for FP

Patterns for 'if' in Select rule

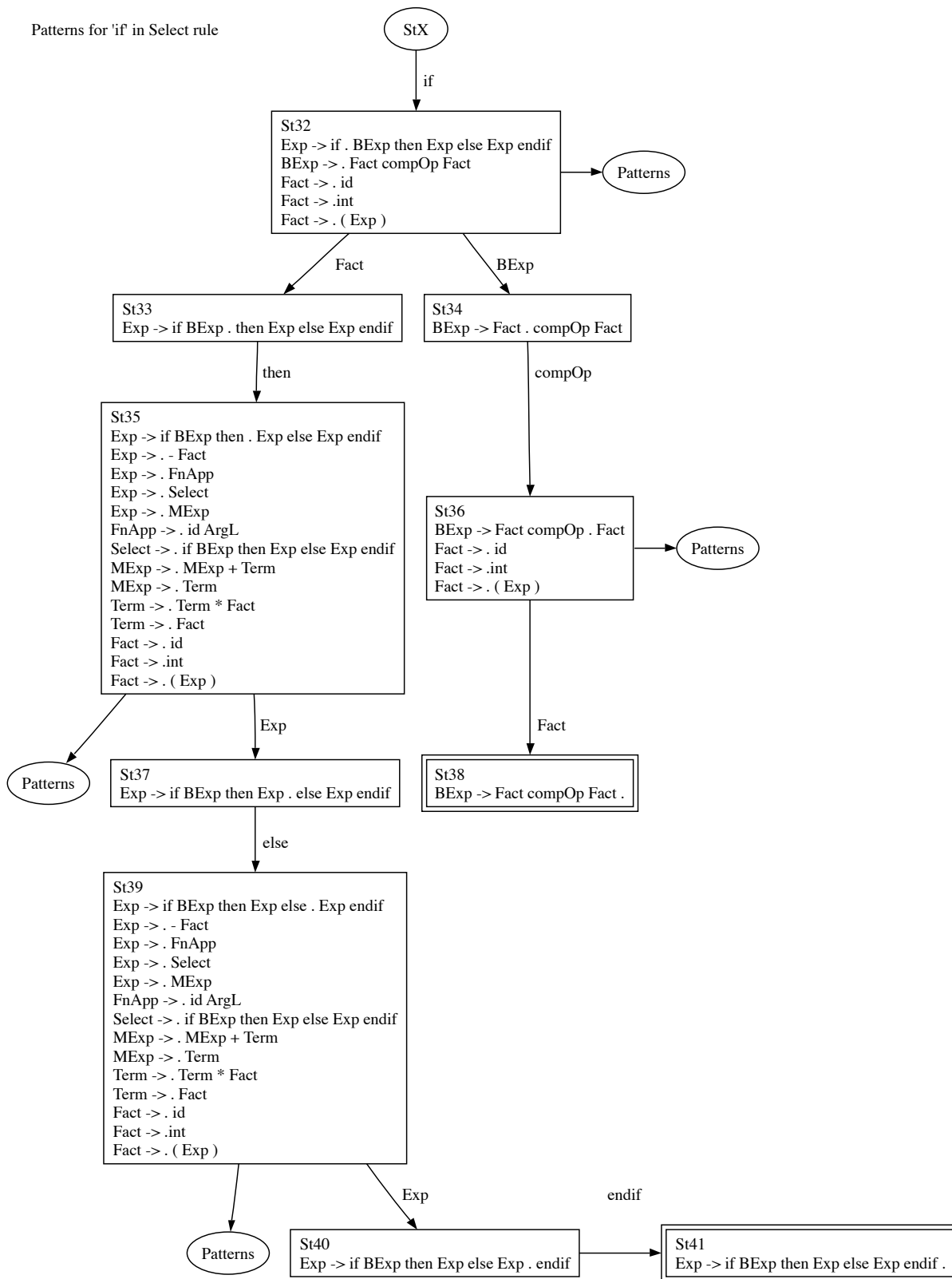


Figure A.3: Transition Graph III for FP

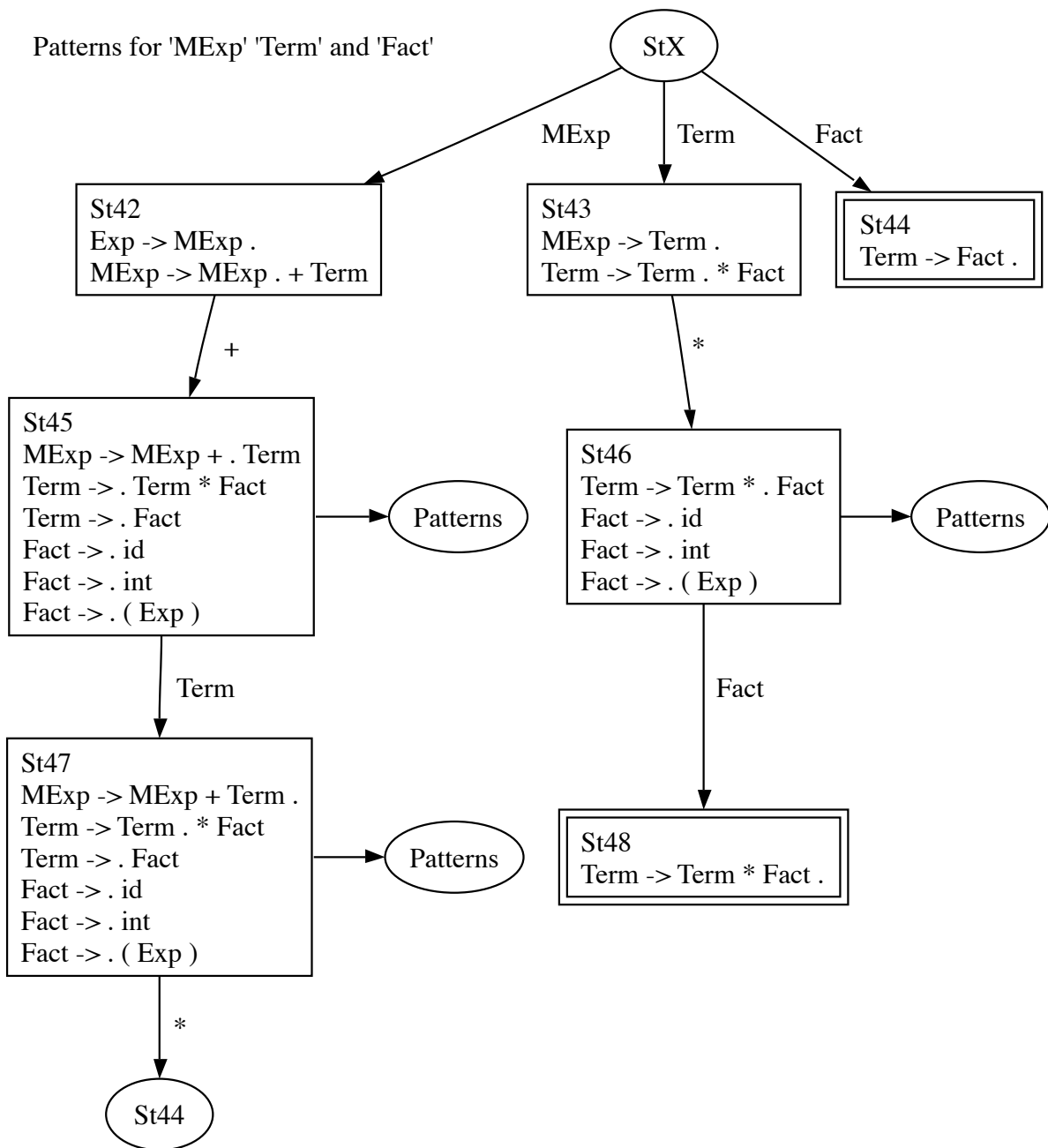


Figure A.4: Transition Graph IV for FP

## Appendix B

# FP Interpreter Driver Code

```
import java.io.FileReader;
import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.io.FileNotFoundException;
import java.io.IOException;

import debug.*;
import tokenizer.*;
import parser.*;
import syntaxTree.*;
import symbolTable.*;
import machine.*;
import exceptions.*;

public class FP {

public static void main(String[] args) {

    BufferedReader in = null; // the input character stream

    boolean echo      = false;
    boolean sntprint  = false; // signal printing syntax tree
    boolean symprint  = false; // signal printing symbol table
    boolean ctprint   = false; // signal printing code table rows > 0
    boolean ct0print  = false; // signal printing code table row 0

    int numArgs = args.length;

    if (numArgs < 1) {
        System.out.println("Not enough arguments!\n");
        System.exit(0);
    }
}
```

```

    }
    else { // args[0] is the data file name
        try { in = new BufferedReader(new FileReader(args[0])); }
        catch (FileNotFoundException e) {
System.out.print("Could not open file ");
System.out.print(args[0]);
System.out.println("");
System.exit(0);
        }
        if (numArgs > 1) { // args[1] holds debug flags
for (int i = 0; i < args[1].length(); i++) {
switch (args[1].charAt(i)) {
    case 'e': echo = true; break;
    case 't': Debug.initDebug(Debug.DebugFlags.TOKENIZER_FLAG); break;
case 'p': Debug.initDebug(Debug.DebugFlags.PARSER_FLAG); break;
case 's': Debug.initDebug(Debug.DebugFlags.SYNTAXTREE_FLAG); break;
case 'b': Debug.initDebug(Debug.DebugFlags.SYMBOLTABLE_FLAG); break;
case 'k': Debug.initDebug(Debug.DebugFlags.CODETABLE_FLAG); break;
case 'i': Debug.initDebug(Debug.DebugFlags.MACHINE_FLAG); break;
case 'n': sntprint = true; break;
case 'm': symprint = true; break;
case 'x': ctprint = true; break;
case 'y': ct0print = true; break;
}
}
        }
        try {
            Tokenizer tins = new Tokenizer(in, echo);
            Parser parse = new Parser(tins);
            CodeTable ct = new CodeTable();
            Machine cpu = new Machine(ct);

            Program st = parse.parseProgram();
            if (sntprint) st.print();
            st.genSymbolTable();
            if (symprint) st.printSymbolTable();
            st.checkSemantics();
            st.genCodeTable(ct);
            if (ctprint) ct.printCT();
            if (ct0print) ct.printFnCode(0);

            cpu.run();
            // display result

```

```
    System.out.println("Answer --> " + cpu.getResult());  
  
} // handle program-based errors  
    catch (FPException e) {  
System.out.println("\n" + e.toString());  
    }  
    catch (IOException e) {  
    System.out.println("\n" + e.toString());  
    }  
  
}  
  
}
```





# Appendix C

## Review of MIPS Assembly Language

This won't be a review of the entire MIPS architecture but rather just those parts necessary for our implementation. We will begin with the register structures and then discuss the SPIM assembly instructions and their semantics. Because FP<sup>+</sup> manipulates only integers we will focus on those aspects of the architecture that manipulate 32-bit integer values.

### C.0.1 Register Structures

MIPS has a RISC architecture, which means it employs a small, simple instruction set along with a large set of general purpose registers. The MIPS architecture has 32 registers, some of which, in the context of the SPIM simulator, have conventional uses. The table in Figure C.0.1 shows the register numbers, assembler names, and, for groups of registers, a description use and assembler conventions for their use. At the end of the table there are also three special purpose registers which are important to our code generation.

### C.0.2 MIPS Assembly Instructions

Because the code we will generate deals only with 32-bit integer data values, we will use just a portion of the MIPS assembly instructions. The nature of the FP<sup>+</sup> language requires that we make use of (integer) arithmetic and comparison instructions. Control transfer instructions are also needed to allow the translation of selection structures as well as function calls and returns from those calls.

#### **data transfer:**

The most basic need in programming is the ability to transfer data between main memory and general purpose processor registers. MIPS assembly language provides three instructions of relevance for our code generation. Each of the transfer instructions requires a register name and a memory address. For our purposes, we will always specify memory address either in a global segment or on a run-time stack. In each case memory will be referenced by giving a byte-offset to a particular register.

register number	register name	description (or conventional use)
0	<b>\$zero</b>	holds the constant value 0
1	<b>\$at</b>	reserved by the assembler
2-3	<b>\$v0, \$v1</b>	used for return values from function calls
4-7	<b>\$a0-\$a3</b>	(convention) assembler uses these for parameters to assembler functions
8-15	<b>\$t0-\$t7</b>	(convention) not saved across function call boundaries, so if needed by the caller, the caller must save them before a call and restore them after the return
16-23	<b>\$s0-\$s7</b>	(convention) a function using any of these must save them before using them and then restore values before the return
24-25	<b>\$t8-\$t9</b>	(convention) treated like <b>\$t0-\$t7</b>
26-27	<b>\$k0-\$k1</b>	used for interrupt/trap handler
28	<b>\$gp</b>	global pointer
29	<b>\$sp</b>	stack pointer
30	<b>\$fp/s8</b>	frame pointer (or treated like <b>\$s0-\$s7</b> )
31	<b>\$ra</b>	return address
	PC	address of next instruction to be executed
	HI	high order 32-bits from integer multiplication and division
	LO	low order 32-bits from integer multiplication and division

Figure C.1: MIPS Registers

<code>lw \$t, -24(\$s)</code>	If register <code>s</code> has value <code>m</code> , then register <code>t</code> will have transferred to it the 4 bytes (1 word) starting at memory location <code>m + (-24)</code> .
<code>sw \$t, -24(\$s)</code>	If register <code>s</code> has value <code>m</code> , then the word stored in register <code>t</code> will be transferred to it the memory location <code>m + (-24)</code> .

Another useful instruction is one which allows us to load a constant value directly to a register. When a constant data value is specified in an instruction that instruction is often referred to as an "immediate" instruction. The 'load immediate' instruction is an example. The load immediate instruction (`li`) is a pseudoinstruction, so we will use the following instruction to which it is usually translated.

<code>ori \$t, \$zero, 5</code>	The <code>ori</code> (or immediate) instruction when applied to zero and a value will always produce the value. So the <code>ori</code> instruction will produce the value (in this case 5) and store it in register <code>\$t</code> .
---------------------------------	---

Another pseudoinstruction for data transfer is the `move` instruction, which copies one register to another. This pseudoinstruction is typically implemented in terms of the `add` machine instruction, and it is that form we will use in place of `move`.

<code>add \$t, \$zero, \$s</code>	The register <code>t</code> will get the value in the register <code>s</code> .
-----------------------------------	---

### arithmetic operations:

We need to be able to translate five integer arithmetic operations `+`, `-`, `*`, `/`, `%`. Since addition is useful in various situations (for the `move` instruction above and for adjusting the addresses in the stack and frame pointer registers, for example) there are several versions provided by the processor instruction set. There is an immediate form of the addition operation as well as an unsigned version. The unsigned version behaves the same as the normal addition operation except it does not generate an integer overflow error. The details for these instructions shouldn't be surprising. There are two subtraction operations, normal and unsigned; there is no immediate form of this operation.

<code>add \$r, \$t, \$s</code>	<code>r = t + s</code> , generate overflow trap
<code>addu \$r, \$t, \$s</code>	<code>r = t + s</code> , generate no overflow trap
<code>addi \$r, \$t, v</code>	<code>r = t + v</code> , generate overflow trap – <code>v</code> is an integer constant
<code>sub \$r, \$t, \$s</code>	<code>r = t - s</code> , generate overflow trap
<code>subu \$r, \$t, \$s</code>	<code>r = t - s</code> , generate no overflow trap

The other three integer arithmetic operations are implemented as two operand instructions. The reason is that, in the case of 32-bit multiplication, the product is potentially 64-bits; in the case of division and `mod`, the integer division of two numbers produces two value, one the quotient and the other the remainder. In all three of these cases the result of the operation is stored in two special purpose registers named `HI` and `LO`. In order to access result of an operation, there are two instructions to transfer the result in each register to a general purpose register. Here are the details for these operations.

<code>mult \$t, \$s</code>	<code>[HI,LO] = t * s</code> , generate overflow trap
<code>multu \$t, \$s</code>	same as <code>mult</code> except generate no overflow trap
<code>div \$t, \$s</code>	<code>LO = t / s</code>
	<code>HI = t % s</code> , generate overflow trap
<code>divu \$t, \$s</code>	same as <code>div</code> except generate no overflow trap
<code>mflo \$r</code>	<code>r = LO</code>
<code>mfhi \$r</code>	<code>r = HI</code>

A final operation we need is the negation operation. The MIPS assembler supports a pseudoinstruction called `neg`, but it is usually translated using the machine instruction `sub`. We will use the `sub` version in code generation.

```
sub $t0, $zero,    to = (-1)*t1 [ same as neg $t0, $t1 ]
$t1
```

### branch operations:

There are two categories of branch operations, those which facilitate a return to the instruction after the branch, and those do not. In this section we discuss two instructions of the second category, one which branches unconditionally, and the other which branches conditionally – each is a machine instruction. Here is a summary of the two instructions.

<code>j label</code>	this will cause the PC register to take the address of instruction at the specified <code>label</code>
<code>beq \$t, \$s, label</code>	this will only change the value of PC if the values in the two specified registers are equal
<code>bne \$t, \$s, label</code>	similar to <code>beq</code> , with change occurring if the values are not equal

We can combine the conditional branch instructions with a comparison operation (“set less than”) to implement branches based on various comparisons as illustrated in the following two columns: the first has the MIPS pseudoinstruction and the second column has the translation in machine instructions.

<code>bge \$t1, \$t2, label</code>	<code>bne \$t2, \$t1, 12</code>
	<code>ori \$t3, \$0, 1</code>
	<code>beq \$0, \$0, 8</code>
	<code>slt \$t3, \$t2, \$t1</code>
	<code>bne \$zero, \$t3, label</code>

The “set less than” sets the result register with 0 or 1 depending on whether the value of the first source operand is less than that of the second source operand. The next section looks at comparison operations in more detail.

### comparison operations:

Arithmetic comparisons use the operations `<`, `<=`, `==`, `!=`, `>=`, `>` which are implemented in the MIPS assembler by the instructions `slt`, `sle`, `seq`, `sne`, `sge`, and `sgt`. But, except for `slt`,

these instructions are all pseudoinstructions. Since our goal is to use no such instructions, we must look for equivalent instruction sequences in the MIPS machine language. To understand the instruction sequences we will look at two cases.

The first case is for expressions such as  $a > b$ . In this case, since `slt` is a machine instruction, we can simply apply `slt` but to the operands in the opposite order (i.e., determining if  $a > b$  is the same as determining if  $b < a$ ). The second case is more interesting and involves the situation where there are two possible answer: the expression  $a \leq b$ . In this case we want to eliminate the equality case first, and then, if there aren't equal, determine if they are in the specified order. Here is an annotated sequence.

1. `bne $t2, $t1, 12` – skip next two instructions if the two values are the same
2. `ori $t3, $0, 1` – load 1 into `$t3`, signifying they were equal
3. `beq $0, $0, 8` – branch around the next instruction
4. `slt $t3, $t2, $t1` – check if  $a > b$  (i.e.,  $\$t2 > \$t1$ )
5. at this point `$t3` contains the result of the comparison

Assuming that we want to express ‘ $a \text{ op } b$ ’, where the values of  $a$  and  $b$  are in registers `t1` and `t2` and the result of the operation is to go into register `t3`, then the following table gives the pseudoinstruction paired with its equivalent in non-pseudoinstructions.

<code>sgt \$t3, \$t1, \$t2</code>	<code>slt \$t3, \$t2, \$t1</code>
<code>slt \$t3, \$t1, \$t2</code>	<code>slt \$t3, \$t1, \$t2</code>
<code>sge \$t3, \$t1, \$t2</code>	<code>bne \$t2, \$t1, 12</code> <code>ori \$t3, \$0, 1</code> <code>beq \$0, \$0, 8</code> <code>slt \$t3, \$t2, \$t1</code>
<code>sle \$t3, \$t1, \$t2</code>	<code>bne \$t2, \$t1, 12</code> <code>ori \$t3, \$0, 1</code> <code>beq \$0, \$0, 8</code> <code>slt \$t3, \$t1, \$t2</code>
<code>seq \$t3, \$t1, \$t2</code>	<code>beq \$t2, \$t1, 12</code> <code>ori \$t3, \$0, 0</code> <code>beq \$0, \$0, 8</code> <code>ori \$t3, \$0, 1</code>
<code>sne \$t3, \$t1, \$t2</code>	<code>beq \$t2, \$t1, 12</code> <code>ori \$t3, \$0, 1</code> <code>beq \$0, \$0, 8</code> <code>ori \$t3, \$0, 0</code>

**function call/return:**

The second category of branch instructions has the function call/return instructions. These instructions are always complex structures for a processor architecture. The call/return pair on some processors (CISC) carry out considerable register and stack manipulations. On a RISC processor, such as the MIPS, the call/return pair are much simpler, leaving more of the function call management to the programmer (i.e., compiler). The function call is a branch from which the program will return to the point immediately after the call. The call, then, must be able to somehow save the address to return to. At the other end, a call implies an eventual return from the call. The return statement (also a branch) must have access to the saved return address from the call. The following table summarizes these two branch statements – remember that they are used in pairs, a call paired with a return.

<code>jal label</code>	the value of PC is stored in the register <code>\$ra</code> and then is changed to the address of the instruction named by <code>label</code> . It is the responsibility of the callee to insure that at the return, the value of <code>\$ra</code> is correct.
<code>jr \$ra</code>	copies the value of <code>\$ra</code> to PC – this performs the return

It is important to emphasize here that it is the responsibility of the programmer (or compiler writer!) to manage other aspects of the function call. For example, if a stack frame needs construction for managing the resources of the function, it is the programmer who is responsible. If nested calls occur and stack frames become stacked, then it is the programmer who must save not only the return address but also the address of former stack frames.

The table in Figure C.0.2 summarizes the MIPS instructions described above along with their assembly language forms.

<code>lw \$t, n(\$s)</code>	<code>sw \$t, n(\$s)</code>	<code>ori \$t, \$zero, n (load immediate)</code>
<code>add \$r, \$t, \$s</code>	<code>addu \$r, \$t, \$s</code>	<code>addi \$r, \$t, v</code>
<code>sub \$r, \$t, \$s</code>	<code>mult \$t, \$s</code>	<code>div \$t, \$s</code>
<code>divu \$t, \$s</code>	<code>mflo \$r</code>	<code>mfhi \$r</code>
<code>sub \$t0, \$zero, \$t1 (negate)</code>	<code>slt \$t3, \$t2, \$t1</code>	<code>beq \$t, \$s, label</code>
<code>bne \$t, \$s, label</code>	<code>j label</code>	
<code>jal label</code>	<code>jr \$ra</code>	

Figure C.2: Summary of MIPS Assembly Instructions

# Index

- follow* set (*follow* function), 89
- first* function, 157
- follow* function, 158
- ambiguous grammar, 37
- analysis phase, 7, 8
  - semantic analysis, **8**
  - syntax analysis, **8**
- assembler, **4**
- assembly language, **4**
- associativity, *see* operator associativity
- attribute, 42, 165
  - inherited attribute, 169
  - synthesized attribute, 167
- attribute grammar, 163, 171
  - inherited attribute, 169
- back-end, **13**
- back-patching, 348–350, 354
  - back-patch list, 350–353
- block, 403
- block structured, 123
- BNF/EBNF, *see* grammar
- class invariant – example, 220
- CL - The Calculator Language, 40
- code generation, **9**
  - back-patching of function calls, *see* back-patching
  - generating linear code, 345
- code generator, 13
- code optimization, **9**
- computation, **3**
- context free grammar, 22
  - LL(1) grammar, 78
  - one token lookahead, 78
- context sensitive, 40, 163
- design strategy
  - clarity first, 182
  - debugging framework, 186
  - energize design, 182
  - theory is best, 182
- dotted form, 142
- dynamic semantics, 21
- exception handling, 183, 226–228
  - catching exceptions, 226
  - throwing exceptions, 226, 227
- finite-state machine, 63
  - deterministic, 63
  - non-deterministic, 69
- finite state machine, 63
- implementation strategies
  - $\epsilon$ -transition, 195
  - algorithm, 70, 196
  - end-of-file, 196
  - handling errors, 196
  - keywords, 194
  - peek ahead, 194
  - white-space, 194
- first set (first function), 87
- flow graph, 465, **467**
  - block, 465, 466
    - example, 465
  - example, 467
  - initial block, 466
  - leaders, 466
  - linearizing a flow graph, 467
  - link, 466
  - next sequential block, 467
    - example, 468
- FP
  - formal definition
    - alphabet, 314
    - grammar level, 315
    - token description, 315
  - FPDriverCode, 489–491
  - implementation

- code generation, 341–345
- code table, 339–341
- semantic checker, 337–339
- symbol table, 336–337
- syntax tree, 334
- informal definition, 313
- interpreter
  - system structure, 328
- operational semantics, 320–322
  - translation example, 322–345
- parse table, 483–488
- static semantics
  - block structure, 317
  - declaration scope, 317
- symbol table
  - integrated in syntax tree, 337
- FP+
  - code generation (stack code), 347–348, 350–355
    - `CodeTable.backPatchFunctionEntries`, 354
    - `CodeTable.backPatch`, 352
    - `CodeTable.getPushParam`, 351
    - `FnApp.generateCode`, 352
    - `FnDef.generateCode`, 353
    - class `CodeTable`, 350–352
    - class `FnApp`, 352
    - class `FnDef`, 353
    - class `Program`, 355
    - symbol table, 350
  - dynamic semantics
    - example (stack code), 347
    - in three-address code, 361–363
    - stack-based, 346–347
  - intermediate code generation
    - three-address code structure), 363–365
  - intermediate code generation (three-address code), 363–369
    - `FnApp.generateCode`, 368
    - `Identifier.generateCode`, 367
    - class `OpCode`, 365
    - code table, 365–366
    - example, 361
    - temporary variables & labels, 366
    - UML `SymbolTableEntry` hierarchy, 364
  - object code generation, 389–393
    - `CodeTable.generateCode`, 390
    - class `ObjectCodeTable`, 389
    - translation of `add`, 391
    - translation of `allocate`, 393
    - translation of `copy`, 392
    - translation of `fcall`, 392
    - translation of `jumpF`, 392
    - sub-expression elimination
      - `Exp.generateCode`, 374
      - `FnApp.generateCode`, 379
- FP machine, 319–321
  - registers
    - SP FP FN PC, 319
- front-end, **11**
- grammar
  - First* function, 87
  - first* function, *see first* function
  - follow* function, *see follow* function
  - ambiguous, 37
  - associativity, *see operator associativity*
  - BNF/EBNF, 32
  - context free, **27**
  - derivation, 29
    - left-most, 30
    - right-most, 30
  - flatten, **254**
  - grammar rules, 28
  - left regular, **35**
  - non-terminal symbol, 27
  - parse tree, 30
  - precedence, *see operator precedence*
  - regular, **35**
  - resolving ambiguity, 38
  - right regular, **35**
  - terminal symbol, 27
- high-level language, **4**
- IEL
  - token description
    - regular expression, 27
    - regular grammar, 35
- IEL (integer expression language), 23
- intermediate code generator, **10**
- IP
  - common subexpression elimination, 425
  - dynamic semantics



- Phase 1 in three address code, 439
- Phase 2 in three address code, 447
- Phase 3 in three address code, 450–451
- formal definition
  - alphabet, 420
  - block, 435
  - grammar level, 421, 428
  - scope of a declaration, 435
  - static semantics, 422–423
  - token level, 420, 427
- informal definition, 419
  - dynamic semantics, 423
- intermediate code generation (three address code)
  - Phase 1, 437, 441–445
  - Phase 1 table, 439
  - Phase 2, 437, 447–448
  - Phase 2 table, 447
  - Phase 3, 437, 452–453
  - Phase 3 table, 449
- intermediate code generator, 424
- object code generation
  - peep-hole optimization, 480–481
  - Phase 1, 473–475
  - Phase 2, 475
  - Phase 3, 475–479
- object code generator, 425
- parser, 424, 428–431
  - implementation, 428–431
- static semantics
  - semantic checker design, 436
- static semantic checking, 424
- subexpression elimination (basic)
  - implementation, 456–458
  - strategy, 455–456
- subexpression elimination (block spanning)
  - implementation, 464–469
- subexpression elimination (generalized), 469–470
  - in/out identity, 470
  - in/out lists, 469
- subexpression elimination (with branches)
  - strategy, 463–464
- subexpression elimination (with state change)
  - implementation, 460–463
  - strategy, 458–459
- symbol table, 433–434
- class hierarchy, 433, 434
- syntax tree, 424, 431–433
  - class hierarchy, 432, 433
- tokenizer, 424
  - implementation, 427–428
- Java classes/methods
  - BufferedReader**, 184
    - mark**, 184
    - read**, 184
    - reset**, 184
  - FileReader**, 184
  - LinkedList**, 185
  - ListIterator**, 185
    - hasNext**, 185
    - next**, 185
- Java features
  - exception handling, 183, 229
    - try-catch, 183
  - packages, 183
- Kleene star, 25
- language
  - dynamically typed, **6**
  - statically typed, **6**
- language, **17**, **22**
  - artificial/natural, 17
  - block structured, 123
  - formal view, 22
  - informal view, 17
- languages
  - IEL (integer expression language), 23
- left-associative operation, **34**
- left-most derivation, 30
- left-recursion, **34**
- lexical analysis, **8**
- lexical error, 8
- LR-parsing, 137
  - decorated grammar rule, 142
    - dotted form, 142
    - left-most dotted form, 142
    - terminal dotted form, 142
- examples, 137–141
- sets of dotted forms, 143
  - closure of, 143
- transition graph, *see* transition graph

## MIPS

- architecture, 476
  - floating point, 476
  - instruction summary, 498
  - registers, 493
- assembly language, 493–498
- instructions - arithmetic operations
  - add, addu, addi, sub, subu, 495
  - mult, multu, div, divu, mflo, mfhi, 496
- instructions - branch operations, 496
  - beq branch equal, 496
  - bne branch not equal, 496
  - j jump, 496
  - slt, sle, seq, sne, sge, sgt, 497
- instructions - comparison operations, 496
- instructions - data transfer, 493
  - add - add to register, 495
  - lw - load word, 493
  - ori - or immediate, 495
  - sw - store word, 493
- instructions - function call/return, 498
  - jal branch and link, 498
  - jr return, 498
- register structure, 494

natural language processing, 17

null string, 22

object code, 10

object code generator, **10**

one token lookahead, 78

operator associativity, 37

operator precedence, 36

parameter passing
 

- by reference, 445–447
- by value, 445–447

parser, **10**, 73
 

- as pluggable component, 295

parse tree, 30, **30**

parsing, 8, 73
 

- bottom-up, 73
  - handle, 79
  - shift operation, 79
  - table-driven, 137
- bottom-up
  - reduce, 79

- class invariant, 220
- error identification, 225–230
- error recovery, 231–234
- LL, 76
- LR, 80
- LR(0) parse table
  - accept action, 148
  - error action, 150
  - goto action, 147
  - reduce action, 147
  - shift/reduce conflict, 156
  - shift action, 147
- LR(0) parse table, 147
- LR parser
  - algorithm, 282
  - UML class diagram, 284
- non-LR(0) grammar structures, 154–156
  - arithmetic expressions, 154
  - possibly-empty lists, 155
- Parser Invariant, 220
- Parser Rule, 220
- possibly empty list, 429
- predictive parser, 78
- recursive descent, 73, 82
  - grammar-rule forms, 84–92
- shift/reduce parser, 80
- SLR, 156
- SLR parse table, 158–161
- table driven, 137, 147
- top-down, 73
  - recursive descent, 83
  - table driven, 83

PDef

- formal definition
  - alphabet, 58
  - grammar level, 60
  - static semantics, 60, 174
  - syntax, 58–60
  - token level, 59
- informal definition, 57
- language component
  - assignment, 57
  - declaration, 57
  - statement list, 57
- LR parser, 294–295
- parser implementation, 237–238

- semantic checking, 276–279
- SLR parse table, 161–162, 296
- syntax tree
  - UML diagram, 255
- syntax tree (LR parser), 306–307
- syntax tree (recursive descent), 253–257
- tokenizer
  - implementation, 209–212
- transition graph
  - expression component, 298
  - list component, 297
- PDef-light
  - finite state machine
    - transition actions, 206
  - parser (bottom-up)
    - Parser.parseProgram, 285
    - class Parser, 283
    - class AcceptEntry, 290
    - class ErrorEntry, 290
    - class GotoEntry, 289
    - class ParseTableEntry, 287
    - class ParseTable, 285
    - class ParseTable initialization, 293
    - class ReduceEntry, 288
    - class ShiftEntry, 288
    - enum Rule, 288
    - enum SymbolName, 286
    - error handling/recovery, 294
    - parse table, 152–154
  - parser (bottom-up) parse table, 291
  - parser (recursive descent), 84
    - Parser.parseProgram, 224
    - class Parser, 221
    - class ParseException, 229
    - class PdefException, 229
    - error recovery, 235–236
    - exception handling, 229–230
  - static semantics
    - class List, 263
    - class PDefSymbolTable, 267
    - class SymbolTable, 262, 267
  - symbol table
    - SymbolTable.addBinding, 267
    - SymbolTable.isAtThisLevel, 268
  - syntax tree
    - UML diagram, 245
- syntax tree (bottom-up)
  - ReduceEntry.doAction, 304–306
  - ShiftEntry.doAction, 304
  - StackEntry, 303
  - modifications to class Parser, 303
- syntax tree (recursive descent)
  - Parser.parseAssignment, 243
  - Parser.parseBlock , 248
  - Parser.parseStmtList, 249
  - class Parser, 243
  - class PDef, 241
  - display traversal algorithm, 250–257
  - object-oriented design, 243
  - parser modifications, 248
- token
  - implementation, 190
  - structure, 190
- tokenizer
  - class Debug, 203
  - class Tokenizer, 199
  - class Token, 199
  - implementation, 191
  - method Tokenizer.getNextToken, 202, 204
  - method Tokenizer.putBackChar, 202
  - structure, 190
  - transition graph, 292
- PDefL
  - formal definition
    - grammar level, 181
    - token level, 180
  - informal definition
    - static semantics, 181
- precedence, *see* operator precedence
- programming language
  - compiled, 5
  - declarative, 4
  - functional, 4
  - logic programming, 4
  - imperative, 4
  - interpreted, 5
- program state, 21
- register allocation, 396
  - as cache management, 396–403
  - cache management
    - cache allocation, 397
    - cache consistency, 397

- cache performance, 397
- cache replacement, 397
- implementation
  - `RegisterInfo.getSourceReg`, 406
  - `RegisterInfo.getTargetReg`, 406
  - `RegisterInfo.saveRegisters`, 406
  - `RegisterInfo.search4`, 406
  - `RegisterInfo.zeroRegisters`, 407
  - class `ObjectCodeTable`, 407
  - class `RegisterInfo`, 406
  - freeing registers, 416
  - register replacement, 410–413
  - register tracking, 405
- rationale, 405
- using an unbounded cache, 397
- using a bounded cache, 398
  - victim selection, 398
- regular expression, 23, **24**
- regular language, 24
- regular expression, 22
- right-associative operation, **34**
- right-most derivation, 30
- right-recursion, **34**
- run-time behavior, 19
- scope of a declaration, 123, 124
- semantics, 17
  - axiomatic, 43
  - denotational, 43
  - dynamic, 21, 43
    - black-box view, 21
    - state-change view, 21
  - dynamic semantics, 43
  - operational, 43
  - semantic checker, 21
  - static, 19
  - static semantics, 163
- Semantic Analysis, 97, 121
- semantic analysis, **8**
- semantic checker, **10**
- semantic checking, 9
- semantic error, 8
- semantic rules, 171
- spilling, 400
- state of a program, *see* (rogram state)21
- static semantics, 19
  - attribute, *see* attribute
  - bound, 42
  - scope of a declaration, **123**, **124**
  - scoping rules, 125
  - steps in semantic checking, 132
- string concatenation, **23**
- sub-expression elimination
  - for arithmetic expressions, 374–376
  - for function calls, 376–380
  - for selection expressions, 381–383
  - process, 371–373
- symbol table, **10**, 13, 122, **261**
  - design, 262
  - display algorithm, 271
  - exception handling, 264
  - filling the table, 269
  - syntax tree integration, 263
- syntax, 17
- syntax analysis, 8, **8**, 73
- syntax errors, 8
- syntax tree, **10**, 13, 45
  - class design
    - using grammar rule forms, 101–105
  - design, 100
  - example, 98–100, 103, 244–245
  - generation by LR parser
    - `ReduceEntry.doAction`, 302
    - `ShiftEntry.doAction`, 301
    - design strategy, 295–300
  - symbol table integration, 263
  - traversal
    - algorithm design process, 116
    - algorithm example, 117–119
- synthesis phase, 7, **9**
- three-address code, 358–361
  - dynamic semantics
    - in MIPS assembler, 385–389
  - instruction forms, 360
- token, 8, 18
- tokenizer, 10
  - handling errors, 196
  - token identification
    - longest substring, 68
- tokenizing, 8
- transition graph, 142–145
- translator, 3, **3**
  - back-end, **13**

- design principles, *see* translator design principles
  - correctness, **6**, 320
  - early warning, **6**
  - efficiency, **7**
  - portability, **7**
- front-end, **11**
- phases
  - analysis, 7
  - synthesis, 7
- properties, 5
- structure
  - data-driven view, 10
  - intuitive view, 7
  - object-oriented view, 11
- translator design principles, **6**
- tree traversal
  - algorithm intuition, 251
- type checking, **6**, 8
- type system, **21**, 42
  
- variable
  - L-value, **20**
  - R-value, **20**
  - use of
- variable
  - L-value, , 505