Reminders

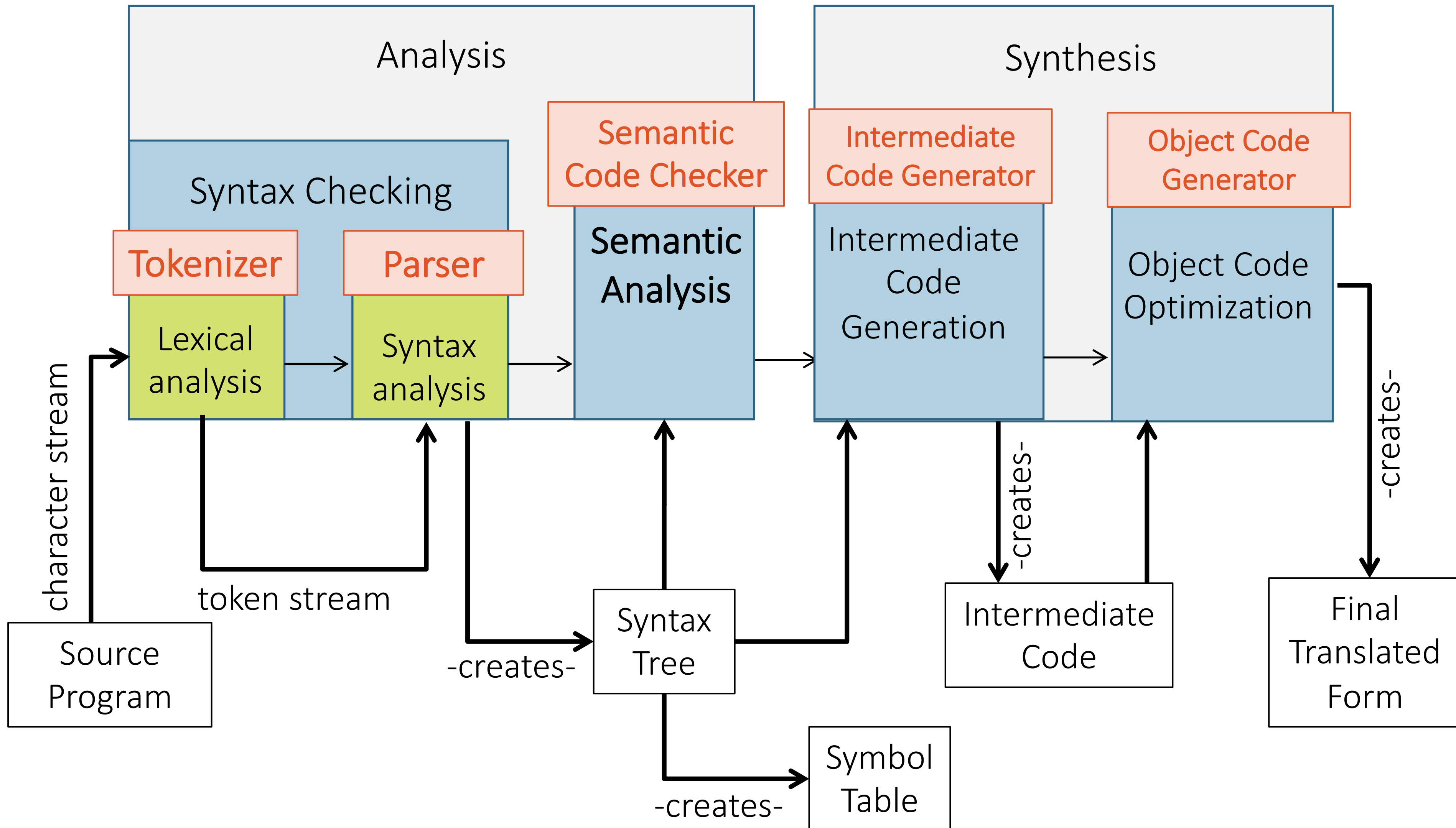
- Environment setup due Fri 9/4
- Lab 2 due Mon 9/7
- Formal topic, team, and, sources due Fri 9/11
  - Check Teams ‘Announcements’ first so there’s no overlap
  - Prior to formal assignment submission, let me know your topic in a few words to reserve it

- Complete Forms on Teams about joining in-person v. remote next week
  - I need an accurate headcount to assign groups

Describing Languages

Semantic Analysis

Recall: phases of a translator



Key ideas

- Need specifications of language syntax & semantics that produces implementations accepting the same sentences & producing the same meaning for them
- Syntax **checking** during translation is complete
  - we can be certain that a sentence is or is not in the language
- Semantic **checking** is incomplete
  - we cannot “check” that a program is “meaningful”
- Need to discuss semantic **checking** versus **specification**
- Formally specifying semantics is trickier than syntax
- A language’s type system is the bridge between syntax and semantics

Type systems

A type system is (1) a mechanism to define types and associate language constructs with them, and (2) a set of rules for

- **Type equivalence**: when are the types of two values the same
- **Type compatibility**: when can a value of a given type be used in a given context
- **Type inference**: rules that determine the type of a language construct based on how it's used

Static and dynamic typing

- Recall that types serve an expected set of operations...so to determine if a requested operation/operand pairing is legal, we need to know the operand's type
- Statically typed languages have the constraints that
  - a single type is associated with a variable through the variable's lifetime
  - the types of all variables and expressions can be determined at compile time
  - Example: C, C++, Java Haskell…
- Dynamically typed languages allow the type of a variable to change as the program runs
  - Example: Perl, Python, JavaScript…

Static typing

- Compile time checking minimizes amount of checking at run time
- Requires certain information be available at compile time:
  - For each operation we need to know arguments info (number, type, order) and result type
  - Type associated with a variable at declaration, which may not change during the variable's lifetime
  - Type inference rules can be used to determine the type of a literal (if the language does not require an explicit association)
    - Done in conjunction with **lexical rules** defining how to specify literals of the language-supported types: 3 is an integer, 3.0 is a real, '3' is a char, and "3.0" is a string

Static typing – some questions

- Rule 1: **declaration before use**
  - Ex: is a variable declared before it used?
- Rule 2: **type compatibility**
  - Ex: is an expression type-consistent?

- What do you remember is the limitation of RGs and why we need CFGs?
- Does a CFG have the expressive power to determine if a program written in a statically typed language is compliant with these two rules?

No! The semantic analysis phase takes AST as input and annotates it with type information that is used to determine if these rules have been followed.

Dynamic typing

- Type checking done **at run time**; requires
  - Type information stored with each data object
  - Before each operation, check the types of the operator's arguments
  - The result must also be tagged with its type
- Advantages:
  - Promotes flexibility (a variable can change types as necessary during the execution of the program)
  - Frees the programmer from most concerns about typing (including type declarations)
- Disadvantages: flexibility and freedom come at a cost:
  - Programs become more difficult to debug
  - Type information takes up extra space
  - Type checking requires execution of extra code, slowing down the execution of the program

Specifying dynamic semantics

A dynamic semantics description formally specifies behavioral characteristics of the language.

Methods:

- Axiomatic semantics
- Denotational semantics
- Operational semantics

Axiomatic semantics

- **Assertion** – a predicate that describes the **state** of a program at any point in its execution
  - **Precondition** – what is true ***before*** the statement executes
  - **Postcondition** – what is true ***after*** the statement executes

- Axiomatic semantics allows us to logically derive a series of assertions by reasoning about the behavior of each individual statement in the program
  - begin with the program postcondition
  - work backwards to the program's first statement and its precondition

- The result is a proof of program correctness

Denotational semantics

- More rigorous than the other methods
- Meaning of the language entities is represented by mathematical objects (denotations) which can be manipulated in ways that are more rigorous (exacting) than we can manipulate language entities

- For each language entity, define a mathematical object and a mapping function that maps instances of the entity onto instances of the mathematical object (which is said to *denote* the meaning of its corresponding syntactic entity)

Operational semantics

- A non-mathematical approach to the specification of the semantics of a language
- Provide a definition of program meaning by **simulating the program’s behavior** on a machine model that has a very simple instruction set and memory organization (e.g., a stack machine)
- Map each language statement to statements in the machine model
  - the meaning of **those** simple statements determines the meaning of the language’s statements

Calculator Language (CL)

- An extension of the Integer Expression Language
- Supports variable declarations and assignments
- Static typing, so recognizes the declaration before use and the type compatibility rules

Abstract syntax

Once parsing is complete, some elements of the input can be discarded. Compare the following….

Program → ‘(‘ StmtList ‘)’

StmtList → Stmt | Stmt ‘;’ StmtList

with

Program → StmtList

StmtList → Stmt | Stmt StmtList

Abstract syntax

Syntactic elements that were essential to parsing but have no **semantic** content can be discarded.

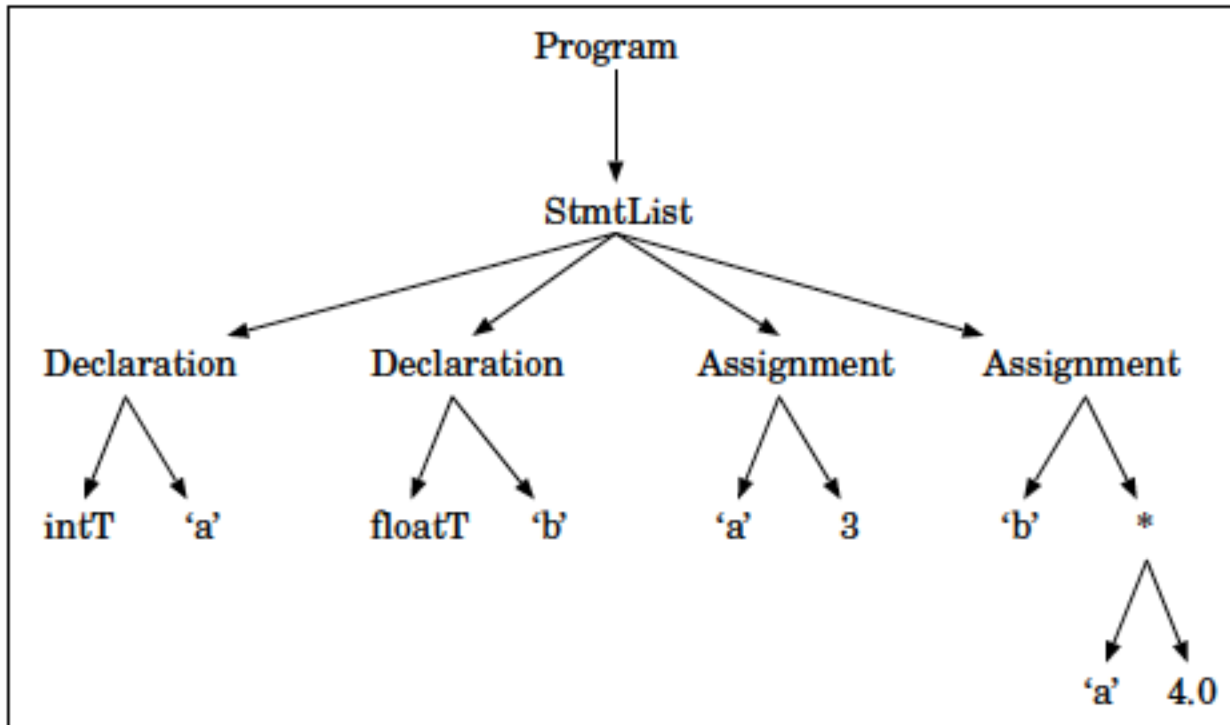


Figure 2.9: Syntax Tree for `int a, float b, a = 3, b = a*4.0`

Operational Semantics

- Define an abstract machine (CL Machine Instruction Set)
- Remove unneeded syntactic elements in the grammar
- For each rule in the grammar, define a mapping to CL machine instructions using the recursive trans function

```
1 Program → StmtList
     trans(Program) = push 0, ..., push 0, trans (StmtList) , halt
```
where the number of **push** *0 instructions = the number of declarations*

```
2 StmtList → Stmt
     trans(StmtList) = trans (Stmt)

3 StmtList → Stmt StmtList
     trans(StmtList) = trans (Stmt) , trans(StmtList)
```