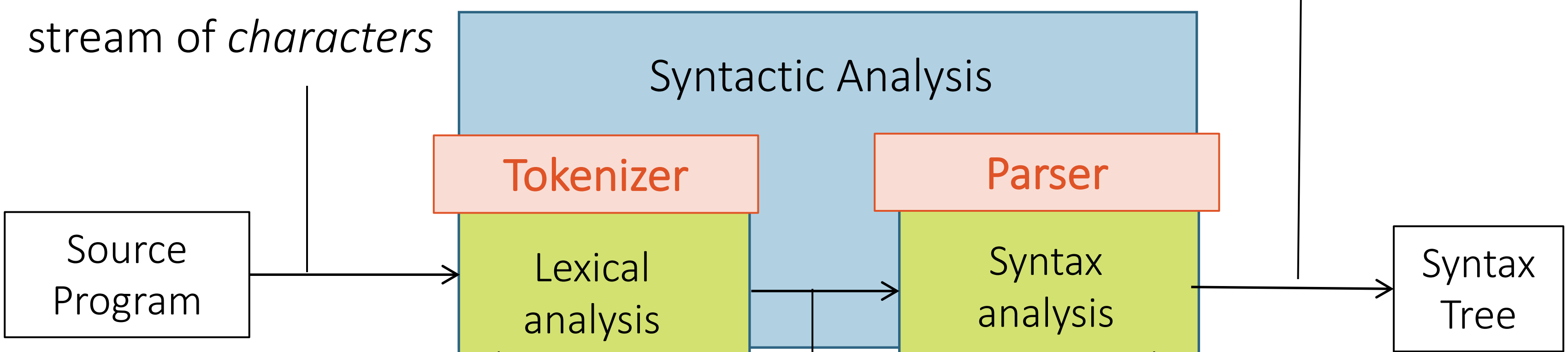
Describing Languages Syntactically

Regular expressions & regular grammars
Context free grammars

Subphases of Syntactic Analysis

Parser generates an *AST* representation of the source program

Source program is viewed as a stream of *characters*



Tokenizer assembles characters into a *lexeme* and classifies the lexeme into a *token type*

Parser assembles tokens into *phrases* according to the source language's grammar

Tokenizer provides a stream of *tokens* to the parser

Tokens

A lexeme is a sequence of characters in the source program that matches the pattern for a token

- “word” from a program
- Ex: `while`, `char`, `+`, `z`, `42`

A token is a pair consisting of the lexeme (its spelling) and token type

- Ex: `int answer = 42;` contains the following tokens

  ```
  int (keyword)
  answer (identifier)
  = (operator)
  42 (constant)
  ; (symbol)
  ```

- All tokens of the same kind can be interchanged without affecting the program’s phrase structure

Tokens, Lexical Analyzer, and Parser

The lexical analyzer (also known as “scanner” or “lexer”)

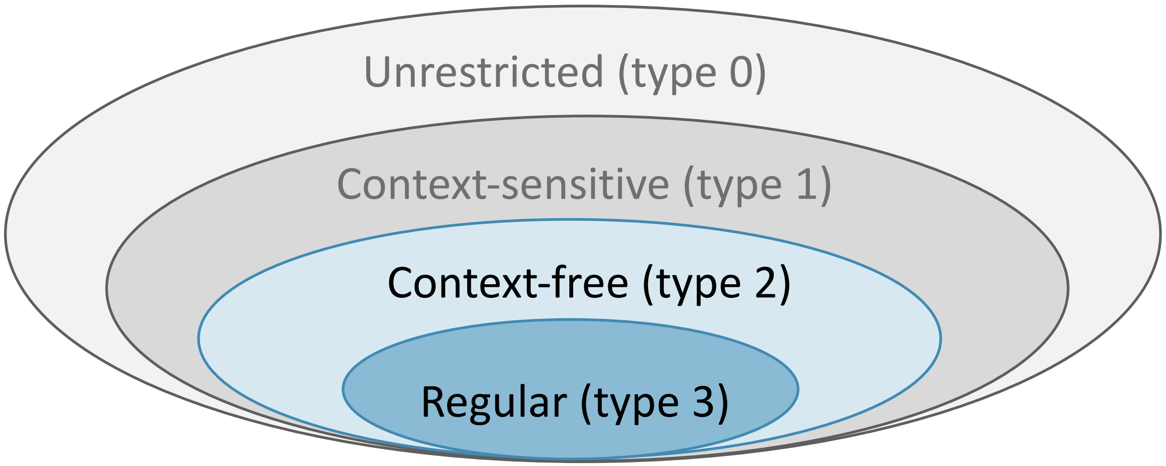
- Reads characters from the source file, assembling them into lexemes
- Needs clear rules about how to assemble lexemes and identify their token type
- Skips over comments and white space

The parser only cares about token types, which it uses to construct phrase structures

- but must retain lexemes for operators, literals & identifiers to do contextual analysis later, and, eventually, code generation

Grammars

- Noam Chomsky – linguist who defined a hierarchy of grammar classes, two of which are relevant to us



- **Regular expressions** and **regular grammars** describe the construction of tokens or terminals in the language
  - A lexical analyzer can be built from a regular grammar (Lex, Flex)
- **Context-free grammars** (**CFGs**) describe the syntax of a language
  - A parser can be built from a CFG (Yacc, Bison)

Regular Expressions (REs)

Describe the structure of terminals or strings in a language

- Think of a RE as a pattern for string generation

REs are closed under the following operations:

- Concatenation: if A and B are REs, then **A·B** (*read as A prepended to B*) is a RE.
- Union: if A and B are REs, then **A|B** (*read as A or B*) is a RE.
- Kleene Star: if A is a RE, then **A*** (*read as 0 or more occurrences of A*) is a RE.
- Kleene Plus: if A is a RE, then **A+** (*read as 1 or more occurrences of A*) is a RE.
- Optional: if A is a RE, then **A?** (*read as 0 or 1 occurrence of A*) is a RE.

If *R* is a RE, then *L(R)* is the language described by *R*.

A language is a **regular** language if it is described by some RE.

Examples of Regular Expressions (REs)

- R = ϕ (*empty set*)
- R = ε (*empty string*)
- R = a (*single character* a)
- R = a|b (a *or* b)
- R = ab (*the string* ab)
- R = a* (ε *or* a *or* aa *or* aaa *and so on*)
- R = a+ (a *or* aa *or* aaa *and so on*)
- R = a*(b+)a* (b *or* ab *or* ba *or* abb *or* bba *or* bb *and so on*)
- Digit = 0|1|2|3|4|5|6|7|8|9   can be written [0-9]
- IntegerLiteral = Digit+

Equivalence of REs and Finite Automata

Finite state machine (FSM), also known as finite automata (FA), is a state machine that takes a string of symbols as input and changes its state accordingly.

When a string is fed into the FA, it changes its state for each literal.

- If the input string is successfully processed and the FA reach its final state, it is ***accepted*** (i.e., the input string is a valid token of the language)

Languages recognized by FA are precisely the languages described by REs!

Any RE can be expressed as a Regular Grammar

- A (*right*) regular grammar (RG) consists of terminal symbols (alphabet), non-terminal symbols, a start symbol, and grammar rules consisting of one of the following forms:

  $A \rightarrow aB$ Right side contains exactly one non-terminal & it's the **rightmost**

  $A \rightarrow a$ Right side contains no non-terminals

  where *A* and *B* represent any single non-terminal, and *a* represents any single terminal or the empty string.

- We read $A \rightarrow aB$ as "*A* generates (produces) aB"

- Examples of rules that are not valid in a regular grammar:

  $A \rightarrow aBc$

  $B \rightarrow CD$

Regular Expression & Regular Grammar

Regular Expression:	Regular Grammar:
a*	$S \to \epsilon \| aS$
(a|b)*	$S \to \epsilon \| aS \| bS$
a*|b*	$S \to \epsilon \| A \| B$ $A \to a \| aA$ $B \to b \| bB$
a*b	$S \to b \| aS$
ba*	$S \to bA$ $A \to \epsilon \| aA$
(ab)*	$S \to \epsilon \| abS$

Example: RE and RG for C variable names

Regular Expression:

[a-zA-Z_][a-zA-Z_0-9]*

Regular Grammar:

Alphabet → Alphabet AlphaNumeric

Alphabet → a|b|…|A|B|C|…|Z| _

AlphaNumeric → Alphabet AlphaNumeric |
Numeric AlphaNumeric | ϵ

Numeric → 0|1|….|9

Example:
Integer Expression Language (IEL)

- Legal strings are any algebraic based on integer/float operations and integer/float literals

  Examples:

  2+3

  (10+3)*5

  ((6/2) - (8%3) * 5)

IEL Regular Expressions

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

IEL Regular Grammar

Operation → ‘+’ | ‘-’ | ‘*’ | ‘/’ | ‘%’

Punctuation → ‘(‘ | ‘)’

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

Limitations of Regular Grammars

- In a regular grammar, every production rule has one of the following forms:

  $$A \to aB$$
  $$A \to a$$

  where *A* and *B* represent any single non-terminal, and *a* represents any single terminal or the empty string.

- Ex: Describe the language $\{a^n b a^m | n, m > 0\}$ with a regular grammar.

  $$S \to aS | aX$$
  $$X \to bY$$
  $$Y \to aY | a$$

Limitations of Regular Grammars

- In a regular grammar, every production rule has one of the following forms:

  $$A \to aB$$
  $$A \to a$$

  where *A* and *B* represent any single non-terminal, and *a* represents any single terminal or the empty string.

- We cannot build a regular grammar to describe the language $\{a^n b^n | n > 0\}$

- This implies we cannot design a regular grammar to check for balanced parenthesis/braces.

Context Free Grammars (CFGs)

- A grammar is a quadruple (Σ, V, S, R) four components:
  - a finite set Σ of **terminal** symbols – the alphabet of the grammar,
  - a finite set V of **non-terminals** symbols,
  - a unique **start symbol** symbol $S \in V$
  - a finite set of **grammar rules** (or **productions**) R, with each rule having the form $\alpha \rightarrow \beta$, where α and β are strings of non-terminals and terminals - i.e., $\alpha, \beta \in (\Sigma \cup V)^*$
- A context free grammar (CFG) has the restriction that α is a single non-terminal; a context sensitive grammar does not.
- A language that can be generated by a CFG is said to be a context free language.
- All regular grammars are context-free, but not all context free grammars are regular.

IEL Context Free Grammar (& RE)

CFG

$$
\begin{array}{lll}
(1) & \text{Exp} & \rightarrow \text{Exp Op Exp} \\
(2) & & \rightarrow \text{intT} \\
(3) & & \rightarrow \text{lpT Exp rpT} \\
(4) & \text{Op} & \rightarrow \text{addT} \mid \text{subT} \mid \text{mulT} \mid \text{divT} \mid \text{modT}
\end{array}
$$

RE

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

Left and Right derivations

- Determined by the order in which we apply productions
- *Left*-derivation: expand the *leftmost* non-terminal first
- All tokens of the same kind can be interchanged without affecting the program's phrase structure
  - For example, literal 5 and 2 are both tokens of type intT

Example:
A left-derivation of $(5-2)*6$

Grammar:

$$\begin{aligned} &(1)\ \ \mathrm{Exp} && \rightarrow \mathrm{Exp\ Op\ Exp} \\ &(2) && \rightarrow \mathrm{intT} \\ &(3) && \rightarrow \mathrm{lpT\ Exp\ rpT} \\ &(4)\ \ \mathrm{Op} && \rightarrow \mathrm{addT} \mid \mathrm{subT} \mid \mathrm{mulT} \mid \mathrm{divT} \mid \mathrm{modT} \end{aligned}$$

Derivation:

Exp	→ <u>Exp</u> Op Exp	(1)
	→ lpT <u>Exp</u> rPt Op Exp	(3)
	→ lpT <u>Exp</u> Op Exp rPt Op Exp	(1)
	→ lpT intT <u>Op</u> Exp rPt Op Exp	(2)
	→ lpT intT subT <u>Exp</u> rPt Op Exp	(4)
	→ lpT intT subT intT rPt <u>Op</u> Exp	(2)
	→ lpT intT subT intT rPt mulT <u>Exp</u>	(4)
	→ lpT intT subT intT rPt mulT intT	(2)
	(5 - 2) * 6	

CFG: A simple example

Consider the following (imperfect!) CFG for arithmetic expressions

Grammar:

(1) $E \rightarrow E + E$

(2) $E \rightarrow E - E$

(3) $E \rightarrow E * E$

(4) $E \rightarrow E / E$

(5) $E \rightarrow (E)$

(6) $E \rightarrow id$

Find a left derivation of (a+b)*c

$E \rightarrow E * E$ (3)

$\rightarrow (E) * E$ (5)

$\rightarrow (E + E) * E$ (1)

$\rightarrow (id + E) * E$ (6)

$\rightarrow (id + id) * E$ (6)

$\rightarrow (id + id) * id$ (6)

CFG: A simple example

Consider the following (imperfect!) CFG for arithmetic expressions

Grammar:

(1) E → E + E

(2) E → E - E

(3) E → E * E

(4) E → E / E

(5) E → (E)

(6) E → id

Find a left derivation of a * b + c

E → E + E	(1)
→ E * E + E	(3)
→ id * E + E	(6)
→ id * id + E	(6)
→ id * id + id	(6)

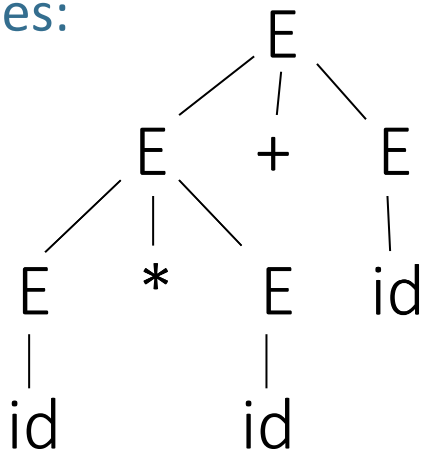
OR

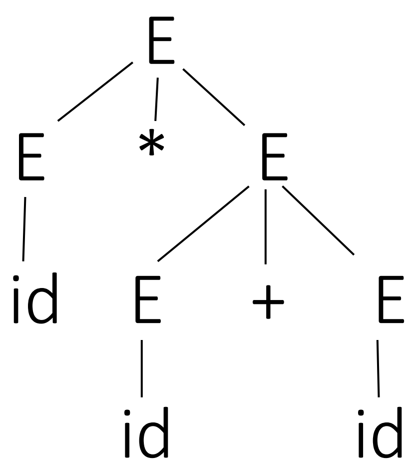
E → E * E	(3)
→ id * E	(6)
→ id * E + E	(1)
→ id * id + E	(6)
→ id * id + id	(6)

CFG: A simple example

Consider the following (imperfect!) CFG for arithmetic expressions

Parse Trees:





Find a left derivation of a * b + c

E → E + E (1)

→ E * E + E (3)

→ id * E + E (6)

→ id * id + E (6)

→ id * id + id (6)

OR

E → E * E (3)

→ id * E (6)

→ id * E + E (1)

→ id * id + E (6)

→ id * id + id (6)

Operator Precedence & Associativity

- A grammar that produces more than one parse tree for a sentence is *ambiguous.*
- **Precedence** determines the order in which operators of *different* levels of precedence are executed
  - Ex: a * b + c $\Rightarrow$ (a * b) + c
- **Associativity** determines the order in which operators of the *same* precedence are executed
  - Left associative – operations are grouped from the left
    - Ex: a + b + c $\Rightarrow$ (a + b) + c
  - Right associative – operations are grouped from the right
    - Ex: a = b = c; $\Rightarrow$ a = (b = c);   // valid in C
- Solution: Revise the grammar to remove the ambiguity – must describe the *same* language though!
  - Let the new grammar reflect operator precedence and associativity.

CFG for Arithmetic Expressions

Grammar:

(1) Expr $\rightarrow$ Expr + Term

(2) Expr $\rightarrow$ Term

(3) Term $\rightarrow$ Term * Factor

(4) Term $\rightarrow$ Factor

(5) Factor $\rightarrow$ (Expr)

(6) Factor $\rightarrow$ id

Find a left derivation of a * b + c

Expr $\rightarrow$ Expr + Term

$\rightarrow$ Term * Factor + Term

$\rightarrow$ Factor * Factor + Term

$\rightarrow$ id * Factor + Term

$\rightarrow$ id * id + Term

$\rightarrow$ id * id + Factor

$\rightarrow$ id * id + id

Left / Right Recursive Grammars

- Left Recursive: $S \rightarrow S\ a\ Q$
- Right Recursive $S \rightarrow Q\ a\ S$
- Later, we'll see that some parsing strategies cannot be applied to grammars with left recursion
  - There are techniques to remove left recursion