CSCE 314
Programming Languages
Syntactic Analysis

Dr. Hyunyoung Lee
What Is a Programming Language?

• Language = syntax + semantics

• The syntax of a language is concerned with the form of a program: how expressions, commands, declarations etc. are put together to result in the final program.

• The semantics of a language is concerned with the meaning of a program: how the programs behave when executed on computers.

• Syntax defines the set of valid programs, semantics how valid programs behave.
Programming Language Definition

• **Syntax**: grammatical structure
  - **lexical** – how words are formed
  - **phrasal** – how sentences are formed from words

• **Semantics**: meaning of programs
  - **Informal**: English documents such as reference manuals
  - **Formal**:
    1. **Operational semantics**: execution on an abstract machine, e.g.,
       \[<x:=c,s>\rightarrow [\ s[x \mapsto s(c)] \ ]\]
    2. **Denotational semantics**: meaning defined as a mathematical function from input to output, definition compositional, e.g.,
       \[\[(x:=c))(s)\rightarrow s[x \mapsto [[c]]_s]\]
    3. **Axiomatic semantics**: each construct is defined by pre- and post- conditions, e.g., \{y \leq x\} z:=x; z:=z+1 \{y < z\}
Language Syntax

• Defines *legal* programs:
  programs that can be executed by machine
• Defined by *grammar rules*
  Define how to make “sentences” out of “words”
• For programming languages
  • Sentences are called statements, expressions, terms, commands, and so on
  • Words are called tokens
  • Grammar rules describe both tokens and statements
• Often, grammars alone cannot capture exactly the set of valid programs. Grammars combined with additional rules are a common approach.
Language Syntax (Cont.)

- Statement is a sequence of tokens
- Token is a sequence of characters
- Lexical analyzer
  produces a sequence of tokens from a character sequence
- Parser
  produces a statement representation from the token sequence
- Statements are represented as parse trees (abstract syntax tree)
Backus–Naur Form (BNF)

• BNF is a common notation to define programming language grammars

• A BNF grammar $G = (N, T, P, S)$
  • A set of non-terminal symbols $N$
  • A set of terminal symbols $T$ (tokens)
  • A set of grammar rules $P$
  • A start symbol $S$

• Grammar rule form (describe context-free grammars):
  <non-terminal>
  ::= <sequence of terminals and non-terminals>
Examples of BNF

• BNF rules for robot commands
  A robot arm accepts any command from the set 
  \{up, down, left, right\}

• Rules:
  <move> ::= <command> | <command> <move>
  <command> ::= up
  <command> ::= down
  <command> ::= left
  <command> ::= right

• Examples of accepted sequences
  • up
  • down left up up right
How to Read Grammar Rules

- From left to right
- Generates the following sequence
  - Each terminal symbol is added to the sequence
  - Each non-terminal is replaced by its definition
  - For each |, pick any of the alternatives
- Note that a grammar can be used to both generate a statement, and verify that a statement is legal
- The latter is the task of parsing – find out if a sentence (program) is in a language, and how the grammar generates the sentence
Extended BNF

- **Constructs and notation:**
  - $<x>$: nonterminal $x$
  - $<x> ::= \text{Body}$: $<x>$ is defined by Body
  - $<x> <y>$: the sequence $<x>$ followed by $<y>$
  - $\{<x>\}$: the sequence of **zero or more** occurrences of $<x>$
  - $\{<x>\}+$: the sequence of **one or more** occurrences of $<x>$
  - $[<x>]$: **zero or one** occurrence of $<x>$

- **Example**
  - $<\text{expression}> ::= <\text{variable}> | <\text{integer}>$
  - $<\text{expression}> ::= <\text{expression}> + <\text{expression}> | ...$
  - $<\text{statement}> ::= \text{if } <\text{expression}> \text{ then } <\text{statement}>$
    
    $\{ \text{ elseif } <\text{expression}> \text{ then } <\text{statement}> \}+$
    $[ \text{ else } <\text{statement}> ] \text{ end } | ...$
  - $<\text{statement}> ::= <\text{expression}> | \text{return } <\text{expression}> | ...$
Example Grammar Rules (Part of C++ Grammar)

A.5 Statements
statement:
  labeled-statement
  expression-statement
  compound-statement
  selection-statement
  iteration-statement
  jump-statement
  declaration-statement
  try-block
labeled-statement:
  identifier : statement
  case constant-expression : statement
default : statement
expression-statement:
  expression_opt :
compound-statement:
  { statement-seq_opt }
statement-seq:
  statement
  statement-seq statement
selection-statement:
  if ( condition ) statement
  if ( condition ) statement else statement
  switch ( condition ) statement
condition:
  expression
type-specifier-seq declarator = assignment-expression
iteration-statement:
  while ( condition ) statement
  do statement while ( expression ) ;
  for ( for-init-statement ; condition_opt ; expression_opt ) statement
for-init-statement:
  expression-statement
  simple-declaration
jump-statement:
  break ;
  continue ;
  return expression_opt ;
go to identifier ;
declaration-statement:
  block-declaration
Context Free Grammars

• A grammar $G = (N, T, S, P)$ with the set of alphabet $V$ is called context free if and only if all productions in $P$ are of the form
  
  $A \rightarrow B$

  where $A$ is a single nonterminal symbol and $B$ is in $V^*$.  

• The reason this is called “context free” is that the production $A \rightarrow B$ can be applied whenever the symbol $A$ occurs in the string, no matter what else is in the string.

• Example: The grammar $G = (\{S\}, \{a,b\}, S, P)$ where $P = \{ S \rightarrow ab \mid aSb \}$ is context free. The language generated by $G$ is $L(G) = \{ a^n b^n \mid n \geq 1 \}$. 

Concrete vs. Abstract Syntax

• Concrete syntax tree
  • Result of using a PL grammar to parse a program is a parse tree
  • Contains every symbol in the input program, and all non-terminals used in the program’s derivation

• Abstract syntax tree (AST)
  • Many symbols in input text are uninteresting (punctuation, such as commas in parameter lists, etc.)
  • AST only contains “meaningful” information
  • Other simplifications can also be made, e.g., getting rid of syntactic sugar, removing intermediate non-terminals, etc.
Ambiguity (1)

- A grammar is ambiguous if there exists a string which gives rise to more than one parse tree
- E.g., infix binary operators ‘-’
  \[
  \text{<expr>} ::= \text{<num>} \mid \text{<expr>} \text{ ‘-’ } \text{<expr>}
  \]
- Now parse 1 - 2 - 3

As (1-2)-3

As 1-(2-3)
Ambiguity (2)

- E.g., infix binary operators `+` and `*`
  
  
  
  \[
  \langle \text{expr} \rangle ::= \langle \text{num} \rangle \mid \langle \text{expr} \rangle + \langle \text{expr} \rangle \mid \langle \text{expr} \rangle * \langle \text{expr} \rangle \mid \langle \text{expr} \rangle == \langle \text{expr} \rangle
  \]

- Now parse 1 + 2 * 3

  As (1+2)*3
  
  As 1+(2*3)
Resolving Ambiguities

1. Between two calls to the same binary operator
   • Associativity rules
     • left-associative: a op b op c parsed as (a op b) op c
     • right-associative: a op b op c parsed as a op (b op c)
   • By disambiguating the grammar
     \[ \text{<expr> ::= <num> | <expr> '-' <expr>} \]
     vs.
     \[ \text{<expr> ::= <num> | <expr> '-' <num>} \]

2. Between two calls to different binary operator
   • Precedence rules
     • if op1 has higher-precedence than op2 then
       a op1 b op2 c => (a op1 b) op2 c
     • if op2 has higher-precedence than op1 then
       a op1 b op2 c => a op1 (b op2 c)
Resolving Ambiguities (Cont.)

- Rewriting the ambiguous grammar:
  `<expr> ::= <num> | <expr> + <expr> | <expr> * <expr> | <expr> == <expr>`

- Let us give * the highest precedence, + the next highest, and == the lowest

  `<expr> ::= <sum> { == <sum> }`  
  `<sum> ::= <term> | <sum> + <term>`  
  `<term> ::= <num> | <term> * <num>`
Dangling-Else Ambiguity

- Ambiguity in grammar is not a problem occurring only with binary operators.

- For example,
  \[ \langle S \rangle ::= \text{if } \langle E \rangle \text{ then } \langle S \rangle | \]
  \[ \quad \text{if } \langle E \rangle \text{ then } \langle S \rangle \text{ else } \langle S \rangle \]

- Now consider the string:

  \[
  \text{if } A \text{ then if } B \text{ then } X \text{ else } Y \\
  \]

  1. \( \text{if } A \text{ then ( if } B \text{ then } X \text{ else } Y ) ? \)
  2. \( \text{if } A \text{ then ( if } B \text{ then } X ) \text{ else } Y ? \)
Chomsky Hierarchy

Four classes of grammars that define particular classes of languages

1. Regular grammars
2. Context free grammars
3. Context sensitive grammars
4. Phrase-structure (unrestricted) grammars

• Ordered from less expressive to more expressive (but faster to slower to parse)
• Regular grammars and CF grammars are of interest in theory of programming languages
Regular Grammar

• Productions are of the form
  \( A \rightarrow aB \) or
  \( A \rightarrow a \)
  where \( A, B \) are nonterminal symbols and \( a \) is a terminal symbol. Can contain \( S \rightarrow \lambda \).

• Example regular grammar \( G = (\{A, S\}, \{a, b, c\}, S, P) \),
  where \( P \) consists of the following productions:
  \( S \rightarrow aA \)
  \( A \rightarrow bA | cA | a \)

• \( G \) generates the following words
  \( aa, aba, aca, abba, abca, acca, abbba, abbca, abcba, \ldots \)

• The language \( L(G) \) in regular expression: \( a(b+c)^*a \)
Regular Languages

The following three formalisms all express the same set of (regular) languages:

1. Regular grammars
2. Regular expressions
3. Finite state automata

Not very expressive. For example, the language

$$L = \{ a^n b^n \mid n \geq 1 \}$$

is not regular.

Question: Can you relate this language L to parsing programming languages?
Answer: balancing parentheses
A finite state automaton $M=(S, I, f, s_0, F)$ consists of:
- a finite set $S$ of states
- a finite set of input alphabet $I$
- a transition function $f: S \times I \rightarrow S$ that assigns to a given current state and input the next state of the automaton
- an initial state $s_0$, and
- a subset $F$ of $S$ consisting of accepting (or final) states

Example:
1. Regular grammar
   
   $S \rightarrow aA$
   $A \rightarrow bA \mid cA \mid a$

2. Regular expression
   
   $a(b+c)^*a$

3. FSA

   ![Finite State Automata Diagram]
Why a Separate Lexer?

• Regular languages are not sufficient for expressing the syntax of practical programming languages, so why use them?

• Simpler (and faster) implementation of the tedious (and potentially slow) “character-by-character” processing: DFA gives a direct implementation strategy

• Separation of concerns – deal with low level issues (tabs, linebreaks, token positions) in isolation: grammars for parsers need not go below token level
Summary of the Productions

1. Phrase-structure (unrestricted) grammars
   \[ A \rightarrow B \text{ where } A \text{ is string in } V^* \text{ containing at least one nonterminal symbol, and } B \text{ is a string in } V^*. \]

2. Context sensitive grammars
   \[ lAr \rightarrow lwr \text{ where } A \text{ is a nonterminal symbol, and } w \text{ a nonempty string in } V^*. \text{ Can contain } S \rightarrow \lambda \text{ if } S \text{ does not occur on RHS of any production.} \]

3. Context free grammars
   \[ A \rightarrow B \text{ where } A \text{ is a nonterminal symbol.} \]

4. Regular grammars
   \[ A \rightarrow aB \text{ or } A \rightarrow a \text{ where } A, B \text{ are nonterminal symbols and } a \text{ is a terminal symbol. Can contain } S \rightarrow \lambda. \]