3.1 Introduction

- **Syntax and semantics provide a language’s definition**

  - **Syntax**: the form or structure of the expressions, statements, and program units
    Specify which strings of characters from the language’s alphabet are in the language
    E.g.,
    - `<sentence> -> <subject> <verb> <noun>`
    - `<while_stmt> -> while (<bool_expr>) <stmt>`

- **Semantics**: the meaning of the expressions, statements, and program units

  - Syntax and Semantics are closely related
  - Syntax and Semantics should be used together
3.2 Informal definition of languages

- A **language**: is a set of sentences
- A **sentence (word)**: is a string of characters over some alphabet
- A **lexeme**: is the lowest level syntactic unit to describe identifiers, literals, operators, and special words (e.g., `sum, *`, `18`)
- A **token** is a category of lexemes (e.g., `ID`, `PLUS_SIGN`)

Lexemes are instances (values) of tokens

---

Formal Definition of Languages

- **Alphabet**
  - Finite, nonempty set of symbols: $\Sigma$
  - Example: $\Sigma = \{0, 1\}$, $\Sigma = \{a, b, \ldots, z, A, B, \ldots, Z\}$

- **Strings**
  - Finite sequence of symbols (characters) chosen from some alphabet
  - Empty String ($\emptyset$): the string with 0 occurrences of symbols
  - Length of a String ($|w|$): number of symbols in the string, e.g., $|\emptyset| = 0$, $|01| = 2$

---

Formal Definition of Languages

- **Powers of an Alphabet** $\Sigma$
  - The set of all strings of a certain length from $\Sigma$.
  - E.g., $\Sigma = \{\emptyset\}$ if $\Sigma = \{0, 1\}$, then $\Sigma = \{0, 1\}$, $\Sigma = \{00, 01, 10, 11\}$
  - The set of all strings over an alphabet $\Sigma$ is denoted $\Sigma$, i.e.,
    $\Sigma = \Sigma \cup \Sigma \cup \Sigma \cup \Sigma \cup \ldots$

- **Language** ($L$)
  - A set of strings all of which are chosen from some $\Sigma$ ($L \subseteq \Sigma$)
  - E.g., English is a set of strings over the alphabet that consists of all the letters.
  - E.g., C: legal programs are a subset of the possible strings that can be formed from the ASCII characters (alphabet)
3.3 Formal Methods of Describing Syntax

- Backus–Naur Form and Context-Free Grammars
  - Most widely known method for describing programming language syntax
  - Metalanguages that are used to describe another language
  - Has four components
    1. a set of tokens, known as terminal symbols
    2. a set of nonterminals
    3. a set of productions (rules) of the form: LHS $\rightarrow$ RHS
    4. A designation of one of the nonterminals as the start symbol

- A rule can be recursive: if its LHS appears in its RHS

  E.g.,
  
  \[
  \begin{align*}
  \text{declaration_stmt} & \rightarrow \text{type_id} \text{id_list} \\
  \text{id_list} & \rightarrow \text{id} , \text{id_list} \\
  \text{type_id} & \rightarrow \text{int} | \text{char} | \text{float} | \text{void}
  \end{align*}
  \]

- A grammar is a generative device for defining languages
- Sentences of a language are generated through a sequence of applications of the rules, beginning with the start symbol.
- A sentence generation is called a derivation
- Replace the LHS with the RHS in each derivation step

  E.g., A Grammar for Simple Assignment Statements:

  \[
  \begin{align*}
  \text{assign} & \rightarrow \text{id} = \text{expr} \\
  \text{id} & \rightarrow \text{A} | \text{B} | \text{C} \\
  \text{expr} & \rightarrow \text{id} + \text{expr} | \text{id} * \text{expr} | (\text{expr}) | \text{id} \\
  \text{A} & \rightarrow \text{B} \times (\text{A} + \text{C})
  \end{align*}
  \]
3.3 Formal Methods of Describing Syntax

- Leftmost (rightmost) derivation: the leftmost (rightmost) nonterminal is replaced by its definition in each step of derivation

Parse Trees
Grammars naturally describe the hierarchical syntactic structure of the sentences of the languages they define. The hierarchical structures are called parse trees

E.g., $A = B \cdot (A + C)$

4.1 Syntax Analysis – Introduction

- The syntax analysis portion of a language processor nearly always consists of two parts:
  - A low-level part called a lexical analyzer (mathematically, a finite automaton based on a regular grammar)
  - A high-level part called a syntax analyzer, or parser (mathematically, a push-down automaton based on a context-free grammar, or BNF)

4.1 Introduction (cont.)

- Reasons to use BNF to describe syntax:
  - Provides a clear and concise syntax description
  - The parser can be based directly on the BNF
  - Parsers based on BNF are easy to maintain because of their clear modularity
4.1 Introduction (cont.)

- Benefits to separate lexical and syntax analysis:
  - Simplicity – less complex approaches can be used for lexical analysis; separating them simplifies the parser
  - Efficiency – separation allows optimization of the lexical analyzer
  - Portability – parts of the lexical analyzer may not be portable, but the parser always is portable

4.2 Lexical Analysis

- A lexical analyzer is a pattern matcher that matches substring of a given string of characters against a given pattern
- A lexical analyzer is a “front-end” for the parser
- Are subprograms that produce the next lexeme and its associated token from the input and return them to the caller (syntax analyzer)

  E.g., \texttt{sum = oldsum - value / 100;}

<table>
<thead>
<tr>
<th>Token</th>
<th>Lexeme</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ident</td>
<td>sum</td>
</tr>
<tr>
<td>ASSIGN_OP</td>
<td>=</td>
</tr>
<tr>
<td>IDENT</td>
<td>oldsum</td>
</tr>
<tr>
<td>SUBTRACT_OP</td>
<td>-</td>
</tr>
<tr>
<td>IDENT</td>
<td>value</td>
</tr>
<tr>
<td>DIVISION_OP</td>
<td>/</td>
</tr>
<tr>
<td>INT_LIT</td>
<td>100</td>
</tr>
<tr>
<td>SEMICOLON</td>
<td>;</td>
</tr>
</tbody>
</table>
4.2 Lexical Analysis (cont.)

- The lexical analyzer is usually a function that is called by the parser when it needs the next token.
- Three approaches to building a lexical analyzer:
  - Write a formal description of the tokens and use a software tool that constructs table-driven lexical analyzers given such a description.
  - Design a state diagram that describes the tokens and write a program that implements the state diagram.
  - Design a state diagram that describes the tokens and hand-construct a table-driven implementation of the state diagram.

Lexical Analysis – DFA

- State diagrams are representations of a class of mathematical machines called finite automata. Finite automata can be designed to recognize a class of languages called regular languages.
  E.g.,
  Identifier: \[a-zA-Z][a-zA-Z0-9_]*\]
  Float: \[0-9]\[0-9]+\]

4.2 Lexical Analysis (cont.)

DFA

- A deterministic finite automaton (DFA) is a five tuple \( A = (Q, \Sigma, \delta, q_0, F) \), where:
  1. \( Q \): A finite set of states.
  2. \( \Sigma \): A finite set of input symbols.
  3. \( \delta \): A total function \( Q \times \Sigma \rightarrow Q \) (e.g., \( \delta(q0, a) = q1)\)
  4. \( q_0 \): The start state.
  5. \( F \): A set of final or accepting states.
4.2 Lexical Analysis (cont.)

- A transition diagram for a DFA $A = (Q, \Sigma, \delta, q_0, F)$ is a directed graph where:
  1. nodes represent states
  2. For each state $q$ in $Q$ and each input symbol $a$ in $\Sigma$, if $\delta(q, a) = p$, then the transition diagram has an arc from node $q$ to node $p$, labeled $a$
  3. There is an arrow into the start state $q_0$, labeled start
  4. Nodes corresponding to accepting states are marked by a double circle