Chapter 4
Lexical and Syntax Analysis

Topics
- Introduction
- Lexical Analysis
- Syntax Analysis
- Recursive-Descent Parsing
- Bottom-Up parsing

Language Implementation
- There are three possible approaches to translating human readable code to machine code:
  1. Compilation
  2. Interpretation
  3. Hybrid

Compilation

Introduction
- The syntax analysis portion of a language processor nearly always consists of two parts:
  - A low-level part called a lexical analyzer
    - Based on a regular grammar.
    - Output: set of tokens.
  - A high-level part called a syntax analyzer
    - Based on a context-free grammar or BNF
    - Output: parse tree.

Issues in Lexical and Syntax Analysis
Reasons for separating both analysis:
1) Simpler design.
   - Separation allows the simplification of one or the other.
   - Example: A parser with comments or white spaces is more complex
2) Compiler efficiency is improved.
   - Optimization of lexical analysis because a large amount of time is spent reading the source program and partitioning it into tokens.
3) Compiler portability is enhanced.
   - Input alphabet peculiarities and other device-specific anomalies can be restricted to the lexical analyzer.
Chapter 4: Lexical and Syntax Analysis

Lexical Analyzer
- First phase of a compiler.
- It is also called scanner.
- Main task: read the input characters and produce as output a sequence of tokens.
- Process:
  - Input: program as a single string of characters.
  - Collects characters into logical groupings and assigns internal codes to the groupings according to their structure.
  - Groupings: lexemes
  - Internal codes: tokens

Examples of Tokens
- Example of an assignment
  \[ \text{result} = \text{value} / 100; \]

<table>
<thead>
<tr>
<th>Token</th>
<th>Lexeme</th>
</tr>
</thead>
<tbody>
<tr>
<td>IDENT</td>
<td>result</td>
</tr>
<tr>
<td>ASSIGNMENT_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>

Interaction between lexical and syntax analyzers

Lexical Analysis
- Secondary tasks:
  - Stripping out from the source program comments and white spaces in the form of blank, tab, and new line characters.
  - Correlating error messages from the compiler with the source program.
  - Inserting lexemes for user-defined names into the symbol table.

Building a Lexical Analyzer
- Three different approaches:
  - Write a formal description of the tokens and use a software tool that constructs table-driven lexical analyzers given such a description (e.g., lex)
  - 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

State Transition Diagram
- Directed graph
- Nodes are labeled with state names.
- Arcs are labeled with the input characters that cause the transitions.
- An arc may also include actions the lexical analyzer must perform when the transition is taken.
- A state diagrams represent a finite automaton which recognizes regular languages (expressions).
Chapter 4: Lexical and Syntax Analysis

State Diagram Design

- A naive state diagram would have a transition from every state on every character in the source language - such a diagram would be very large.
- In many cases, transitions can be combined to simplify the state diagram
  - When recognizing an identifier, all uppercase and lowercase letters are equivalent
  - Use a character class that includes all letters
  - When recognizing an integer literal, all digits are equivalent - use a digit class

State Diagram: Example

Syntax Analyzer

- The syntax analyzer or parser must determine the structure of the sequence of tokens provided to it by the scanner.
- Check the input program to determine whether it is syntactically correct.
  - Produce either a complete parse tree or at least trace the structure of the complete parse tree.
  - Error: produce a diagnostic message and recover (gets back to a normal state and continues the analysis of the input program: find as many errors as possible in one pass).

Parser

- Two categories of parsers
  - **Top-down**: produce the parse tree, beginning at the root down to the leaves.
  - **Bottom-up**: produce the parse tree, beginning at the leaves upward to the root.

Conventions

- **Terminal symbols**: lowercase letters at the beginning of the alphabet (a, b, ...)
- **Nonterminal symbols**: uppercase letters at the beginning of the alphabet (A, B, ...)
- **Terminals or nonterminals**: uppercase letters at the end of the alphabet (W, X, Y, Z)
- **Strings of terminals**: lowercase letters at the end of the alphabet (w, x, y, z)
- **Mixed strings**: (terminals and/or nonterminals): lowercase Greek letters (α, β, δ, γ)

Top-Down Parser

- Build the parse tree in preorder.
  - Begins with the root.
  - Each node is visited before its branches are followed.
  - Branches from a particular node are followed in left-to-right order.
- Uses leftmost derivation.
Next sentential form?

- Given a sentential form \( xA\alpha \), \( A \) is the leftmost nonterminal that could be expanded to get the next sentential form in a leftmost derivation.
- Current sentential form: \( xA\alpha \)
- \( A \)-rules: \( A? bB \), \( A? cBb \), and \( A? a \)
- Next sentential form?:
  - \( xbB\alpha \) or \( xcB\alpha \) or \( xa\alpha \)
- This is known as the parsing decision problem for top-down parsers.

Example:

<table>
<thead>
<tr>
<th>Equation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \langle S\rangle ::= \langle NP\rangle \langle VP\rangle )</td>
<td>&lt;S&gt; ::= &lt;NP&gt; &lt;VP&gt;</td>
</tr>
<tr>
<td>( \langle S\rangle ::= \langle aux\rangle \langle NP\rangle \langle VP\rangle )</td>
<td>&lt;S&gt; ::= &lt;aux&gt; &lt;NP&gt; &lt;VP&gt;</td>
</tr>
<tr>
<td>( \langle NP\rangle ::= \langle Proper_Noun\rangle )</td>
<td>&lt;NP&gt; ::= &lt;Proper_Noun&gt;</td>
</tr>
<tr>
<td>( \langle Det\rangle ::= \langle Noun\rangle )</td>
<td>&lt;Det&gt; ::= &lt;Noun&gt;</td>
</tr>
<tr>
<td>( \langle Verbose\rangle ::= \langle Noun\rangle \langle Noun\rangle )</td>
<td>&lt;NP&gt; ::= &lt;Noun&gt; &lt;Noun&gt;</td>
</tr>
<tr>
<td>( \langle Verb\rangle ::= \langle Verbose\rangle \langle NP\rangle )</td>
<td>&lt;VP&gt; ::= &lt;Verb&gt; &lt;NP&gt;</td>
</tr>
</tbody>
</table>

A miniature English grammar

Example: “Book that flight”

- Suppose the parser is able to build all possible partial trees at the same time.
- The algorithm begins assuming that the input can be derived by the designated start symbol \( S \).
- The next step is to find the tops of all trees which can start with \( S \), by looking for all the grammars rules with \( S \) on the LHS.
  - There are 3 rules that expand \( S \), so the second ply, or level, has 3 partial trees.

Top-Down Parser

- Parsers look only one token ahead in the input.
  - Given a sentential form, \( xA\alpha \), the parser must choose the correct \( A \)-rule to get the next sentential form in the leftmost derivation, using only the first token produced by \( A \).
- The most common top-down parsing algorithms:
  - Recursive descent - a coded implementation
  - LL parsers - table driven implementation
Bottom-Up Parser

- Bottom-up parsing is the earliest known parsing algorithm.
- Build the parse tree beginning at the leaves and progressing towards the root.
- Order corresponds to the reverse of a rightmost derivation.
  - The parse is successful if the parser succeeds in building a tree rooted in the start symbol that covers all of the input.

Example: “Book that flight”

- The parse beginning by looking to each word and building 3 partial trees with the category of each terminal.
  - The word book is ambiguous (it can be a noun or a verb). Thus the parser must consider two possible sets of trees.
  - Each of the trees in the second ply is then expanded, and so on.

Example: “Book that flight”

- In general, the parser extends one ply to the next by looking for places in the parse-in-progress where the right-hand-side of some rule might fit.
  - In the fifth ply, the interpretation of book as a noun has been pruned because this parse cannot be continued: there is no rule in the grammar with RHS $<\text{Nominal}> <\text{NP}>$.
  - The final ply is the correct parse tree.

Top-Down vs. Bottom-Up

- Advantage (top-down)
  - Never waste time exploring trees that cannot result in the root symbol ($S$), since it begins by generating just those trees.
  - Never explores subtrees that cannot find a place in some $S$-rooted tree
    - Bottom-up: left branch is completely wasted effort because it is based on interpreting book as Noun at the beginning of the sentence despite the fact no such tree can lead to an $S$ given this grammar.

Top-Down vs. Bottom-Up

- Disadvantage (top-down)
  - Spend considerable effort on $S$ trees that are not consistent with the input.
  - Firsts four of six trees in the third ply have left branches that cannot match the word book.
  - This weakness arises from the fact that they can generate trees before ever examining the input.