4. Lexical and Syntax Analysis
4.1 Introduction

- Language implementation systems must analyze source code, regardless of the specific implementation approach.

- Nearly all syntax analysis is based on a formal description of the syntax of the source language (BNF).

- 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).

- 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.
4.1 Introduction (continued)

- Reasons 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 for character strings
- A lexical analyzer is a “front-end” for the parser
- Identifies substrings of the source program that belong together - lexemes
- Lexemes match a character pattern, which is associated with a lexical category called a token
- `sum` is a lexeme; its token may be `IDENT`
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:

1. Write a formal description of the tokens and use a software tool that constructs table-driven lexical analyzers given such a description.
2. Design a state diagram that describes the tokens and write a program that implements the state diagram.
3. Design a state diagram that describes the tokens and hand-construct a table-driven implementation of the state diagram.

We only discuss approach 2.
4.2 Lexical Analysis (continued)

- **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
  - Reserved words and identifiers can be recognized together (rather than having a part of the diagram for each reserved word)
    - Use a table lookup to determine whether a possible identifier is in fact a reserved word
4.2 Lexical Analysis (continued)

- Convenient utility subprograms:
  - `getChar` - gets the next character of input, puts it in `nextChar`, determines its class and puts the class in `charClass`
  - `addChar` - puts the character from `nextChar` into the place the lexeme is being accumulated, `lexeme`
  - `lookup` - determines whether the string in `lexeme` is a reserved word (returns a code)
4.2 Lexical Analysis (continued)

Implementation (assume initialization):

```c
int lex() {
    switch (charClass) {
    case LETTER:
        addChar();
        getChar();
        while (charClass == LETTER ||
               charClass == DIGIT) {
            addChar();
            getChar();
        }
        return lookup(lexeme);
        break;
    case DIGIT:
        addChar();
        getChar();
        while (charClass == DIGIT) {
            addChar();
            getChar();
        }
        return INT_LIT;
        break;
    } /* End of switch */
} /* End of function lex */
```
4.3 The Parsing Problem

- Goals of the parser, given an input program:
  - Find all syntax errors; For each, produce an appropriate diagnostic message, and recover quickly
  - Produce the parse tree, or at least a trace of the parse tree, for the program

- Two categories of parsers
  - Top down - produce the parse tree, beginning at the root
    - Order is that of a leftmost derivation
  - Bottom up - produce the parse tree, beginning at the leaves
    - Order is that of the reverse of a rightmost derivation

- Parsers look only one token ahead in the input

- Top-down Parsers
  - 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
4.3 The Parsing Problem (continued)

- Bottom-up parsers
  - Given a right sentential form, $\alpha$, what substring of $\alpha$ is the right-hand side of the rule in the grammar that must be reduced to produce the previous sentential form in the right derivation

- The Complexity of Parsing
  - Parsers that work for any unambiguous grammar are complex and inefficient ($O(n^3)$, where $n$ is the length of the input)
  - Compilers use parsers that only work for a subset of all unambiguous grammars, but do it in linear time ($O(n)$, where $n$ is the length of the input)
4.4 Recursive-Descent Parsing

- **Recursive Descent Process**

- There is a subprogram for each nonterminal in the grammar, which can parse sentences that can be generated by that nonterminal.

- EBNF is ideally suited for being the basis for a recursive-descent parser, because EBNF minimizes the number of nonterminals.

- A grammar for simple expressions:
  
  \[
  \begin{align*}
  <expr> & \rightarrow <term> \ { ( + \ | - ) <term> } \\
  <term> & \rightarrow <factor> \ { ( * \ | / ) <factor> } \\
  <factor> & \rightarrow \ id \ | \ ( <expr> )
  \end{align*}
  \]

- Assume we have a lexical analyzer named `lex`, which puts the next token code in `nextToken`.

- The coding process when there is only one RHS:
  
  - For each terminal symbol in the RHS, compare it with the next input token; if they match, continue, else there is an error.
  
  - For each nonterminal symbol in the RHS, call its associated parsing subprogram.
4.4 Recursive-Descent Parsing (continued)

```c
/* Function expr
   Parses strings in the language generated by the rule:
   <expr> → <term> { (+ | - ) <term> } */
void expr() {
  /* Parse the first term */
  term();
  /* As long as the next token is + or -, call
     lex to get the next token, and parse the next term */
  while (nextToken == PLUS_CODE ||
         nextToken == MINUS_CODE) {
    lex();
    term();
  }
}

This particular routine does not detect errors

Convention: Every parsing routine leaves the next token in nextToken
```
A nonterminal that has more than one RHS requires an initial process to determine which RHS it is to parse

- The correct RHS is chosen on the basis of the next token of input (the lookahead)
- The next token is compared with the first token that can be generated by each RHS until a match is found
- If no match is found, it is a syntax error
/ * Function factor 
   Parses strings in the language generated by 
   the rule: <factor> -> id | (<expr>) */
void factor() {
    /* Determine which RHS */
    if (nextToken == ID_CODE) {
        /* For the RHS id, just call lex */
        lex();
    } else if (nextToken == LEFT_PAREN_CODE) {
        lex();
        expr();
        if (nextToken == RIGHT_PAREN_CODE) {
            lex();
            expr();
            if (nextToken == RIGHT_PAREN_CODE)
                lex();
        } else
            error();
    } else
        error();
    /* Neither RHS matches */
} /* End of else if (nextToken == ... */
else error(); /* Neither RHS matches */
}
4.4 Recursive-Descent Parsing (continued)

- **The LL Grammar Class**
  - The Left Recursion Problem
  - If a grammar has left recursion, either direct or indirect, it cannot be the basis for a top-down parser
  - A grammar can be modified to remove left recursion

- **The other characteristic of grammars that disallows top-down parsing is the lack of pairwise disjointness**
  - The inability to determine the correct RHS on the basis of one token of lookahead

- **Def:** $\text{FIRST}(\alpha) = \{a \mid \alpha \Rightarrow^* a\beta\}$ (If $\alpha \Rightarrow^* \varepsilon$, $\varepsilon$ is in $\text{FIRST}(\alpha)$)

- **Pairwise Disjointness Test:**
  - For each nonterminal, $A$, in the grammar that has more than one RHS, for each pair of rules, $A \rightarrow \alpha_i$ and $A \rightarrow \alpha_j$, it must be true that $\text{FIRST}(\alpha_i) \cap \text{FIRST}(\alpha_j) = \emptyset$
  - Examples:
    - $A \rightarrow a \mid bB \mid cAb$
    - $A \rightarrow a \mid aB$
4.4 Recursive-Descent Parsing (continued)

- Left factoring can resolve the problem Replace

\[<\text{variable}> \rightarrow \text{identifier} \mid \text{identifier} [<\text{expression}>]\]

with

\[<\text{variable}> \rightarrow \text{identifier} <\text{new}>\]

\[<\text{new}> \rightarrow \varepsilon \mid [<\text{expression}>]\]

or

\[<\text{variable}> \rightarrow \text{identifier} [[<\text{expression}>]]\]

(the outer brackets are metasymbols of EBNF)
4.5 Bottom-up Parsing

- The parsing problem is finding the correct RHS in a right-sentential form to reduce to get the previous right-sentential form in the derivation

- Intuition about handles:
  - Def: $\beta$ is the handle of the right sentential form
  - $\gamma = \alpha \beta w$ if and only if $S \Rightarrow^* \alpha A w \Rightarrow \alpha \beta w$
  - Def: $\beta$ is a phrase of the right sentential form
  - $\gamma$ if and only if $S \Rightarrow^* \gamma = \alpha_1 A \alpha_2 \Rightarrow^+ \alpha_1 \beta \alpha_2$
  - Def: $\beta$ is a simple phrase of the right sentential form $\gamma$ if and only if $S \Rightarrow^* \gamma = \alpha_1 A \alpha_2 \Rightarrow \alpha_1 \beta \alpha_2$
  - The handle of a right sentential form is its leftmost simple phrase
  - Given a parse tree, it is now easy to find the handle
  - Parsing can be thought of as handle pruning
4.5 Bottom-up Parsing (continued)

- **Shift-Reduce Algorithms**
  - Reduce is the action of replacing the handle on the top of the parse stack with its corresponding LHS
  - Shift is the action of moving the next token to the top of the parse stack

- **Advantages of LR parsers:**
  - They will work for nearly all grammars that describe programming languages
  - They work on a larger class of grammars than other bottom-up algorithms, but are as efficient as any other bottom-up parser
  - They can detect syntax errors as soon as it is possible
  - The LR class of grammars is a superset of the class parsable by LL parsers
4.5 Bottom-up Parsing (continued)

- LR parsers must be constructed with a tool

- Knuth’s insight: A bottom-up parser could use the entire history of the parse, up to the current point, to make parsing decisions
  - There were only a finite and relatively small number of different parse situations that could have occurred, so the history could be stored in a parser state, on the parse stack

- An LR configuration stores the state of an LR parser
  - \((S_0X_1S_1X_2S_2\ldots X_mS_m, a_ia_{i+1}\ldots a_n\$)\)
4.5 Bottom-up Parsing (continued)

- LR parsers are table driven, where the table has two components, an ACTION table and a GOTO table
  - The ACTION table specifies the action of the parser, given the parser state and the next token
  - Rows are state names; columns are terminals
  - The GOTO table specifies which state to put on top of the parse stack after a reduction action is done
  - Rows are state names; columns are nonterminals
4.5 Bottom-up Parsing (continued)

- Initial configuration: \((S_0, a_1 \ldots a_n$)\)

- Parser actions:
  - If \(\text{ACTION}[S_m, a_i] = \text{Shift } S\), the next configuration is:
    \((S_0X_1S_1X_2S_2\ldots X_mS_m a_iS, a_{i+1} \ldots a_n$)\)
  - If \(\text{ACTION}[S_m, a_i] = \text{Reduce } A \rightarrow \beta\) and \(S = \text{GOTO}[S_{m-r}, A]\), where \(r =\) the length of \(\beta\), the next configuration is:
    \((S_0X_1S_1X_2S_2\ldots X_{m-r}S_{m-r}AS, a_ia_{i+1} \ldots a_n$)\)
  - If \(\text{ACTION}[S_m, a_i] = \text{Accept}\), the parse is complete and no errors were found
  - If \(\text{ACTION}[S_m, a_i] = \text{Error}\), the parser calls an error-handling routine

- A parser table can be generated from a given grammar with a tool, e.g., yacc
### 4.5 Bottom-up Parsing (continued)

<table>
<thead>
<tr>
<th>State</th>
<th>Action</th>
<th>(\text{id})</th>
<th>(\text{+})</th>
<th>(\text{*})</th>
<th>(\text{(})</th>
<th>(\text{)})</th>
<th>(\text{$})</th>
<th>Goto</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>S5</td>
<td>S4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>E</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>S6</td>
<td></td>
<td></td>
<td></td>
<td>accept</td>
<td></td>
<td>T</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>R2</td>
<td>S7</td>
<td>R2</td>
<td>R2</td>
<td></td>
<td></td>
<td>F</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>R4</td>
<td>R4</td>
<td>R4</td>
<td>R4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>S5</td>
<td></td>
<td>S4</td>
<td></td>
<td></td>
<td></td>
<td>8</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>R6</td>
<td>R6</td>
<td>R6</td>
<td>R6</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>S5</td>
<td></td>
<td>S4</td>
<td></td>
<td></td>
<td></td>
<td>9</td>
<td>3</td>
</tr>
<tr>
<td>7</td>
<td>S5</td>
<td></td>
<td>S4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>10</td>
</tr>
<tr>
<td>8</td>
<td>S6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>R1</td>
<td>S7</td>
<td>R1</td>
<td>R1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>R3</td>
<td>R3</td>
<td>R3</td>
<td>R3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>R5</td>
<td>R5</td>
<td>R5</td>
<td>R5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>