Lexical and Syntax Analysis

- Lexical Analysis
- The Parsing Problem
- Recursive-Descent Parsing
- Bottom-Up Parsing

The Parsing Problem (cont'd)

- 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

Recursive-Descent Parsing

- 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:

\[
\begin{align*}
<expr> & \rightarrow <term> \{ (+ \mid -) <term> \} \\
<term> & \rightarrow <factor> \{ (* \mid /) <factor> \} \\
<factor> & \rightarrow id \mid int\_constant \mid ( <expr> )
\end{align*}
\]
Recursive-Descent Parsing (cont’d)

• 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

Recursive-Descent Parsing (cont’d)

/* 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 == ADD_OP || nextToken == SUB_OP) {
      `lex()`;
      term();
   }
}

Recursive-Descent Parsing (cont’d)

/* Function `term`
   Parses strings in the language generated by the rule:
   `<term> -> <factor> {(* | /) <factor>}` */
void term() {
   printf("Enter <term>\n");
   /* Parse the first factor */
   factor();
   /* As long as the next token is * or /, call `lex` to get the next token and parse the next factor */
   while (nextToken == MULT_OP || nextToken == DIV_OP) {
      `lex()`;
      factor();
   }
   printf("Exit <term>\n");
} /* End of function `term` */
Recursive-Descent Parsing (cont'd)

```c
/* Function factor
 Parses strings in the language generated by the rule:
 <factor> -> id | int_const | (expr) */

void factor() {
    /* Determine which RHS */
    if (nextToken == ID_CODE || nextToken == INT_CODE) {
        /* For the RHS id or int-const, just call lex */
        lex();
        /* If the RHS is (expr) – call lex to pass over the left
         parenthesis, call expr, and check for the right parenthesis */
        else if (nextToken == LP_CODE) {
            lex();
            expr();
            if (nextToken == RP_CODE)
                lex();
            else
                error();
        } /* End of else if (nextToken == LP_CODE */
    } else error(); /* Neither RHS matches */
}
```

Recursive-Descent Parsing (cont’d)

• 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: FIRST(α) = {a | α =>* aβ}
  (If α =>* ε, ε is in FIRST(α))

Recursive-Descent Parsing (cont’d)

• 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
      For each nonterminal, A,
      1. Group the A-rules as A → α₁ | … | αₘ | β₁ | β₂ | … | βₙ
         where none of the β’s begins with A
      2. Replace the original A-rules with
         A → β₁A | β₂A | … | βₙA
         A’ → α₁A’ | α₂A’ | … | αₘA’ | ε

Recursive-Descent Parsing (cont’d)

• Pairwise Disjointness Test:
  – For each nonterminal, Aᵢ in the grammar that has more than one RHS, for each pair of rules, A → αᵢ and A → αⱼ, it must be true that
    FIRST(αᵢ) ∩ FIRST(αⱼ) = φ
• Examples:
  A → a | bB | cAb
  A → a | aB
Recursive-Descent Parsing (cont’d)

• Left factoring can resolve the problem

Replace

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

with

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

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

or

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

(the outer brackets are metasymbols of EBNF)

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

Bottom-up Parsing (cont’d)

• 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 \( S \Rightarrow^* \gamma = \alpha_1 \alpha_2 \Rightarrow^+ \alpha_1 \beta \alpha_2 \)

  – **Def:** \( \beta \) is a **simple phrase** of the right sentential form

    \( \gamma \) if and only if \( S \Rightarrow^* \gamma = \alpha_1 A \alpha_2 \Rightarrow \alpha_1 \beta \alpha_2 \)

Bottom-up Parsing (cont’d)

• Intuition about handles (cont’d):

  – 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
• 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.

• 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

\[(S_0X_1S_1X_2S_2...X_mS_m, a_i, a_{i+1}...a_n, \$)\]
Bottom-up Parsing (cont.)

- 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.

Structure of an LR Parser

- **Input**: $a_1 a_2 ... a_m$
- **Parse Stack**: $S_0 X_1 S_1 ... X_m S_m$
- **Top**: $X_m$
- **Goto**: $S = \text{GOTO}[S_m, A]$, where $r$ = the length of $\beta$.
- **Action Table**: 

Bottom-up Parsing (cont.)

- **Initial configuration**: $(S_0, a_1 ... a_n)$
- **Parser actions**:
  - If $\text{ACTION}[S_m, a_i] = \text{Shift } S$, the next configuration is:
    $$(S_0 X_1 S_1 X_2 S_2 ... X_m S_m a_i S, a_{i+1} ... a_n)$$
  - If $\text{ACTION}[S_m, a_i] = \text{Reduce } A \rightarrow \beta$
    and $S = \text{GOTO}[S_m, r]$, where $r$ = the length of $\beta$, the next configuration is:
    $$(S_0 X_1 S_1 X_2 S_2 ... X_m S_m A S, a_{i+1} ... a_n)$$

Bottom-up Parsing (cont.)

- **Parser actions (continued)**:
  - 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.
LR Parsing Table

<table>
<thead>
<tr>
<th>State</th>
<th>Action</th>
<th>Goto</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>id</td>
<td>55</td>
</tr>
<tr>
<td>1</td>
<td>+</td>
<td>56</td>
</tr>
<tr>
<td>2</td>
<td>(</td>
<td>57</td>
</tr>
<tr>
<td>3</td>
<td>)</td>
<td>54</td>
</tr>
<tr>
<td>4</td>
<td>$</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>$</td>
<td>2</td>
</tr>
<tr>
<td>6</td>
<td>$</td>
<td>3</td>
</tr>
<tr>
<td>7</td>
<td>$</td>
<td>4</td>
</tr>
<tr>
<td>8</td>
<td>$</td>
<td>5</td>
</tr>
<tr>
<td>9</td>
<td>$</td>
<td>6</td>
</tr>
<tr>
<td>10</td>
<td>$</td>
<td>7</td>
</tr>
<tr>
<td>11</td>
<td>$</td>
<td>8</td>
</tr>
</tbody>
</table>

A parser table can be generated from a given grammar with a tool, e.g., *yacc*

Bottom-up Parsing (cont.)

- A parser table can be generated from a given grammar with a tool, e.g., *yacc*

Summary

- Syntax analysis is a common part of language implementation
- A lexical analyzer is a pattern matcher that isolates small-scale parts of a program
  - Detects syntax errors
  - Produces a parse tree
- A recursive-descent parser is an LL parser
  - EBNF
- Parsing problem for bottom-up parsers: find the substring of current sentential form
- The LR family of shift-reduce parsers is the most common bottom-up parsing approach