Lexical and Syntax Analysis (2)

In Text: Chapter 4

**Motivating Example**

- Consider the grammar
  
  \[ S \rightarrow cAd \]
  
  \[ A \rightarrow ab \mid a \]
  
- Input string: \( w = cad \)
- How to build a parse tree top-down?

**Recursive-Descent Parsing**

- Initially create a tree containing a single node \( S \) (the start symbol)
- Apply the \( S \)-rule to see whether the first token matches
  
  - If matches, expand the tree
    
    - Apply the \( A \)-rule to the leftmost nonterminal \( A \)
      
      - Since the first token matches both alternatives \( A_1 \) and \( A_2 \), randomly pick one (e.g., \( A_1 \)) to apply

  - Since the third token \( d \) does not match \( b \), report failure and go back to \( A \) to try another alternative
  
  - Rollback to the state before applying \( A_1 \) rule, and then apply the alternative rule
  
  - The third token matches, so parsing is successfully done

**Recursive-Descent Parsing Algorithm**

Suppose we have a scanner which generates the next token as needed. Given a string, the parsing process starts with the start symbol rule:

1. if there is only one RHS then
2. for each terminal in the RHS
3. compare it with the next input token
4. if they match, then continue
5. else report an error
6. for each nonterminal in the RHS
7. call its corresponding subprogram and try match
8. else // there is more than one RHS
9. choose the RHS based on the next input token (the lookahead)
10. for each chosen RHS
11. try match with 2-7 mentioned above
12. if no match is found, then report an error

**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:

\[
\begin{align*}
\text{<expr>} & \rightarrow \text{<term>} \hspace{1em} (\text{+ | -} \hspace{1em} \text{<term>} \\
\text{<term>} & \rightarrow \text{<factor>} \hspace{1em} (\text{* | /} \hspace{1em} \text{<factor>} \\
\text{<factor>} & \rightarrow \text{id} \hspace{1em} | \hspace{1em} \text{int_constant} \hspace{1em} | \hspace{1em} (\text{<expr>})
\end{align*}
\]

An Example

/* Function expr parses strings in the language generated by the rule: <expr> \rightarrow <term> ((+ | -) <term>) */
void expr() {
    printf("Enter <expr>\n");
    /* 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();
    }
    printf("Exit <expr>\n");
}

An Example (cont’d)

/* term */
/* Function term parses strings in the language generated by the rule: <term> \rightarrow <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 */
}

This particular routine does not detect errors

Convention: Every parsing routine leaves the next token in nextToken

/* Function factor parses strings in the language generated by the rule: <factor> \rightarrow id | int_constant | ( <expr> ) */
void factor() {
    printf("Enter <factor>\n");
    /* Determine which RHS */
    if (nextToken == ID_CODE ||
        nextToken == INT_CODE)
        /* For the RHS id, just call lex */
        lex();
    else if (nextToken == LP_CODE) {
        lex();
        expr();
        if (nextToken == RP_CODE)
            lex();
        else
            error();
    } /* End of else if (nextToken == ... */
    else error(); /* Neither RHS matches */
    printf("Exit <factor>\n");
}

Token codes

#define INT_LIT 10
#define IDENT 11
#define ASSIGN_OP 20
#define ADD_OP 21
#define MUL_OP 22
#define DIV_OP 24
#define LEFT_PAREN 25
#define RIGHT_PAREN 26
Recursive-Descent Parsing
(continued)

Trace of the lexical and syntax analyzers on (sum+47)/total

Next token is: 25 Next lexeme is { Enter <expr>
Enter <term>
Enter <factor>
Next token is: 11 Next lexeme is sum Enter <expr>
Enter <term>
Enter <factor>
Next token is: 21 Next lexeme is + Exit <factor>
Next token is: -1 Next lexeme is EOF

Key points about recursive-descent parsing

- Recursive-descent parsing may require backtracking
- LL(1) does not allow backtracking
  - By only looking at the next input token, we can always precisely decide which rule to apply
- By carefully designing a grammar, i.e., LL(1) grammar, we can avoid backtracking

Two Obstacles to LL(1)-ness

- Left recursion
  - E.g., id_list -> id_list_prefix ;
    id_list_prefix -> id_list_prefix, id | id
  - When the next token is id, which rule should we apply?
- Common prefixes
  - E.g., A -> ab | a
  - When the next token is a, which rule should we apply?

Common prefixes

- Pairwise Disjointness
  - Unable to decide which RHS should use by simply checking one token of lookahead
- Pairwise Disjointness Test
  - For each nonterminal A with more than one RHS, for each pair of rules, the possible first characters of the strings (FIRST set) should be disjoint
    - If A -> α₁ | α₂, then FIRST(α₁) \∩ \ FIRST(α₂) = \φ

LL(1) Grammar

- Grammar which can be processed with LL(1) parser
- Non-LL grammar can be converted to LL(1) grammar via:
  - Left-recursion elimination
  - Left factoring by extracting common prefixes

Left-Recursion Elimination

- Replace left-recursion with right-recursion
  id_list -> id_list_prefix ;
  id_list_prefix -> id_list_prefix, id | id
  id_list -> id id_list_tail
  id_list_tail -> ; | , id id_list_tail
Left Factoring
• Extract the common prefixes, and introduce new nonterminals as needed
  \[ A \rightarrow ab \mid a \]
  \[ A \rightarrow aB \]
  \[ B \rightarrow b \mid \epsilon \]

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

Non-LL Languages
• Simply eliminating left recursion and common prefixes is not guaranteed to make LL(1)
  \[ \text{stmt} \rightarrow \text{if condition then\_clause else\_clause} \mid \text{other\_stmt} \]
  \[ \text{then\_clause} \rightarrow \text{then stmt} \]
  \[ \text{else\_clause} \rightarrow \text{else stmt} \mid \epsilon \]
• How to parse "if C1 then if C2 then S1 else S2"?

Non-LL Languages
• Define "disambiguating rule", use it together with ambiguous grammar to parse top-down
  – E.g., in the case of a conflict between two possible productions, the one to use is the one that occurs first, textually in the grammar
  – to pair the else with the nearest then
• "Disambiguating rule" can be also defined for bottom-up parsing

Table-Driven Parsing
• It is possible to build a non-recursive predictive parser by maintaining a stack explicitly, rather than implicitly via recursive calls
• The non-recursive parser looks up the production to be applied in a parsing table.
• The table can be constructed directly from LL(1) grammars

Table-Driven Parsing
• An input buffer
  – Contains the input string
  – The string can be followed by $, an end marker to indicate the end of the string
• A stack
  – Contains symbols with $ on the bottom, with the start symbol initially on the top
• A parsing table (2-dimensional array \( M(A, a) \))
• An output stream (production rules applied for derivation)
Input: a string w, a parsing table M for grammar G
Output: if w is in L(G), a leftmost derivation of w; otherwise, an error indication
Method:
\[ \text{set ip to point to the first symbol of w$} \]
\[ \text{repeat} \]
\[ \text{let X be the top stack symbol and a the symbol pointed to by ip;} \]
\[ \text{if X is a terminal or $, then} \]
\[ \text{if X = a then} \]
\[ \text{pop X from the stack and advance ip} \]
\[ \text{else error()} \]
\[ \text{else} \]
\[ \text{/* X is a non-terminal */} \]
\[ \text{if M[X, a] = X->Y_1 Y_2 … Y_k then} \]
\[ \text{pop X from the stack} \]
\[ \text{push Y_k, …, Y_2, Y_1 on to the stack} \]
\[ \text{output the production X->Y_1 Y_2 … Y_k} \]
\[ \text{end} \]
\[ \text{else error()} \]
\[ \text{until X = $} \]

An Example

- Input String: id + id * id
- Input parsing table for the following grammar

<table>
<thead>
<tr>
<th>E</th>
<th>T</th>
<th>E'</th>
<th>T'</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>E</td>
<td>E -&gt; TE</td>
<td>E</td>
<td>E</td>
<td>F</td>
</tr>
<tr>
<td>E'</td>
<td>E -&gt; TE'</td>
<td>E</td>
<td>E</td>
<td>F</td>
</tr>
<tr>
<td>T</td>
<td>T -&gt; FT</td>
<td>T</td>
<td>E</td>
<td>F</td>
</tr>
<tr>
<td>T'</td>
<td>T -&gt; FT'</td>
<td>T</td>
<td>E</td>
<td>F</td>
</tr>
<tr>
<td>F</td>
<td>F -&gt; id</td>
<td>F</td>
<td>E</td>
<td>F</td>
</tr>
</tbody>
</table>

LL Parsing

<table>
<thead>
<tr>
<th>Stack</th>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>$E$</td>
<td>id + id * id$</td>
<td></td>
</tr>
<tr>
<td>$ET$</td>
<td>id + id * id$</td>
<td>E -&gt; TE'</td>
</tr>
<tr>
<td>$ETF$</td>
<td>id + id * id$</td>
<td>T -&gt; FT'</td>
</tr>
<tr>
<td>$ETid$</td>
<td>id + id * id$</td>
<td>F -&gt; id</td>
</tr>
<tr>
<td>$ET'$</td>
<td>id * id$</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$E$</td>
<td></td>
<td>E' -&gt; ε</td>
</tr>
</tbody>
</table>

An Example

- Input String: id + id * id
- Input parsing table for the following grammar

<table>
<thead>
<tr>
<th>NON-</th>
<th>M</th>
<th>INPUT SYMBOL</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>E</td>
<td>+</td>
</tr>
<tr>
<td></td>
<td>E</td>
<td>*</td>
</tr>
<tr>
<td></td>
<td>F</td>
<td>(</td>
</tr>
<tr>
<td></td>
<td>F</td>
<td>id</td>
</tr>
<tr>
<td></td>
<td>T</td>
<td>id</td>
</tr>
<tr>
<td></td>
<td>T</td>
<td>id</td>
</tr>
<tr>
<td></td>
<td>T</td>
<td>id</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Stack</th>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>$E$</td>
<td>id + id * id$</td>
<td></td>
</tr>
<tr>
<td>$ET$</td>
<td>id + id * id$</td>
<td>E -&gt; TE'</td>
</tr>
<tr>
<td>$ETF$</td>
<td>id + id * id$</td>
<td>T -&gt; FT'</td>
</tr>
<tr>
<td>$ETid$</td>
<td>id + id * id$</td>
<td>F -&gt; id</td>
</tr>
<tr>
<td>$ET'$</td>
<td>id * id$</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$E$</td>
<td></td>
<td>E' -&gt; ε</td>
</tr>
</tbody>
</table>