Chapter 4 (c) parsing

Top down vs. bottom up parsing

- The parsing problem is to connect the root node $S$ with the tree leaves, the input.
- **Top-down parsers:** starts constructing the parse tree at the top (root) of the parse tree and move down towards the leaves. Easy to implement by hand, but work with restricted grammars. Examples:
  - Predictive parsers (e.g., LL(k))
- **Bottom-up parsers:** build the nodes on the bottom of the parse tree first. Suitable for automatic parser generation, handle a larger class of grammars. Examples:
  - shift-reduce parser (or LR(k) parsers)
- Both are general techniques that can be made to work for all languages (but not all grammars!).

Parsing

- A grammar describes the strings of tokens that are syntactically legal in a PL.
- A recognizer simply accepts or rejects strings.
- A generator produces sentences in the language described by the grammar.
- A parser construct a derivation or parse tree for a sentence (if possible).
- Two common types of parsers:
  - bottom-up or data driven
  - top-down or hypothesis driven
- A recursive descent parser is a way to implement a top-down parser that is particularly simple.

Top down vs. bottom up parsing

- Both are general techniques that can be made to work for all languages (but not all grammars!).
- Recall that a given language can be described by several grammars.
- Both of these grammars describe the same language
  
  \[
  \begin{align*}
  E & \rightarrow E + \text{Num} \\
  E & \rightarrow \text{Num} \\
  E & \rightarrow \text{Num} + E
  \end{align*}
  \]

  - The first one, with it’s left recursion, causes problems for top down parsers.
  - For a given parsing technique, we may have to transform the grammar to work with it.
Parsing complexity

• How hard is the parsing task?
• Parsing an arbitrary Context Free Grammar is \(O(n^3)\), e.g., it can take time proportional the cube of the number of symbols in the input. This is bad!
• If we constrain the grammar somewhat, we can always parse in linear time. This is good!
• Linear-time parsing
  – LL parsers
    • Recognize LL grammar
    • Use a top-down strategy
  – LR parsers
    • Recognize LR grammar
    • Use a bottom-up strategy

Top Down Parsing Methods

• Simplest method is a full-backup, recursive descent parser
• Often used for parsing simple languages
• Write recursive recognizers (subroutines) for each grammar rule
  – If rules succeeds perform some action (i.e., build a tree node, emit code, etc.)
  – If rule fails, return failure. Caller may try another choice or fail
  – On failure it “backs up”

Recursive Decent Parsing Example

Example: For the grammar:

\[
\langle \text{term} \rangle \to \langle \text{factor} \rangle \{(*|/)\langle \text{factor} \rangle\}^* 
\]

We could use the following recursive descent parsing subprogram (this one is written in C)

```c
void term() {
    factor();     /* parse first factor*/
    while (next_token == ast_code ||
        next_token == slash_code) {
        lexical();  /* get next token */
        factor();   /* parse next factor */
    }
}
```
Problems

• Some grammars cause problems for top down parsers.
• Top down parsers do not work with left-recursive grammars.
  – E.g., one with a rule like: E -> E + T
  – We can transform a left-recursive grammar into one which is not.
• A top down grammar can limit backtracking if it only has one rule per non-terminal
  – The technique of rule factoring can be used to eliminate multiple rules for a non-terminal.

Left-recursive grammars

• A grammar is left recursive if it has rules like
  \[ X \rightarrow X \beta \]
  Or if it has indirect left recursion, as in
  \[ X \rightarrow A \beta \]
  \[ A \rightarrow X \]
• Why is this a problem?
• Consider
  \[ E \rightarrow E + \text{Num} \]
  \[ E \rightarrow \text{Num} \]
• We can manually or automatically rewrite a grammar to remove left-recursion, making it suitable for a top-down parser.

Elimination of Left Recursion

• Consider the left-recursive grammar
  \[ S \rightarrow S \alpha | \beta \]
  \[ S \rightarrow \beta S' \]
  \[ S' \rightarrow \alpha S' | \epsilon \]
  S generates all strings starting with a \( \beta \) and followed by a number of \( \alpha \)
  Can rewrite using right-recursion

More Elimination of Left-Recursion

• In general
  \[ S \rightarrow S \alpha_1 | \ldots | S \alpha_n | \beta_1 | \ldots | \beta_m \]
  \[ S \rightarrow \beta_1 S' | \ldots | \beta_m S' \]
  \[ S' \rightarrow \alpha_1 S' | \ldots | \alpha_n S' | \epsilon \]
  All strings derived from \( S \) start with one of \( \beta_1, \ldots, \beta_m \) and continue with several instances of \( \alpha_1, \ldots, \alpha_n \)
  Rewrite as
General Left Recursion

- The grammar
  
  \[ S \rightarrow A \alpha | \delta \]

  \[ A \rightarrow S \beta \]

  is also left-recursive because

  \[ S \rightarrow^+ S \beta \alpha \]

  where \( \rightarrow^+ \) means “can be rewritten in one or more steps”

- This indirect left-recursion can also be automatically eliminated

Summary of Recursive Descent

- Simple and general parsing strategy
  - Left-recursion must be eliminated first
  - … but that can be done automatically

- Unpopular because of backtracking
  - Thought to be too inefficient

- In practice, backtracking is eliminated by restricting the grammar, allowing us to successfully predict which rule to use.

Predictive Parser

- A predictive parser uses information from the first terminal symbol of each expression to decide which production to use.

- A predictive parser is also known as an LL(\( k \)) parser because it does a Left-to-right parse, a Leftmost-derivation, and \( k \)-symbol lookahead.

- A grammar in which it is possible to decide which production to use examining only the first token (as in the previous example) are called LL(1)

- LL(1) grammars are widely used in practice.
  - The syntax of a PL can be adjusted to enable it to be described with an LL(1) grammar.

Example: consider the grammar

\[
S \rightarrow \text{if } E \text{ then } S \text{ else } S \\
S \rightarrow \text{begin } S L \\
S \rightarrow \text{print } E \\
L \rightarrow \text{end} \\
L \rightarrow ; S L \\
E \rightarrow \text{num} = \text{num}
\]

An S expression starts either with an IF, BEGIN, or PRINT token, and an L expression start with an END or a SEMICOLON token, and an E expression has only one production.
LL(k) and LR(k) parsers

• Two important classes of parsers are called LL(k) parsers and LR(k) parsers.
• The name LL(k) means:
  – L - Left-to-right scanning of the input
  – L - Constructing leftmost derivation
  – k – max number of input symbols needed to select a parser action
• The name LR(k) means:
  – L - Left-to-right scanning of the input
  – R - Constructing rightmost derivation in reverse
  – k – max number of input symbols needed to select a parser action
• So, a LL(1) parser never needs to “look ahead” more than one input token to know what parser production to apply.

Predictive Parsing and Left Factoring

• Consider the grammar
  \[ E \rightarrow T + E \mid T \]
  \[ T \rightarrow \text{int} \mid \text{int} \ast T \mid (E) \]
• Hard to predict because
  – For T, two productions start with \text{int}
  – For E, it is not clear how to predict which rule to use
• A grammar must be left-factored before use for predictive parsing
• Left-factoring involves rewriting the rules so that, if a non-terminal has more than one rule, each begins with a terminal.

Left-Factoring Example

• Consider the grammar
  \[ E \rightarrow T + E \mid T \]
  \[ T \rightarrow \text{int} \mid \text{int} \ast T \mid (E) \]
• Factor out common prefixes of productions
  \[ E \rightarrow T X \]
  \[ X \rightarrow + E \mid \epsilon \]
  \[ T \rightarrow (E) \mid \text{int} Y \]
  \[ Y \rightarrow * T \mid \epsilon \]

Left Factoring

• Consider a rule of the form
  \[ A \rightarrow a B_1 \mid a B_2 \mid a B_3 \mid \ldots \mid a B_n \]
• A top down parser generated from this grammar is not efficient as it requires backtracking.
• To avoid this problem we left factor the grammar.
  – collect all productions with the same left hand side and begin with the same symbols on the right hand side
  – combine the common strings into a single production and then append a new non-terminal symbol to the end of this new production
  – create new productions using this new non-terminal for each of the suffixes to the common production
• After left factoring the above grammar is transformed into:
  \[ A \rightarrow a A_1 \]
  \[ A_1 \rightarrow B_1 \mid B_2 \mid B_3 \ldots \mid B_n \]
Using Parsing Tables

• LL(1) means that for each non-terminal and token there is only one production
• Can be specified via 2D tables
  – One dimension for current non-terminal to expand
  – One dimension for next token
  – A table entry contains one production
• Method similar to recursive descent, except
  – For each non-terminal \( S \)
  – We look at the next token \( a \)
  – And chose the production shown at \([S,a]\)
• We use a stack to keep track of pending non-terminals
• We reject when we encounter an error state
• We accept when we encounter end-of-input

LL(1) Parsing Table Example

• Left-factored grammar
  \[
  \begin{align*}
  E &\rightarrow T \ X \\
  X &\rightarrow + \ E \mid \varepsilon \\
  T &\rightarrow ( \ E ) \mid \text{int} \ Y \\
  Y &\rightarrow \ast \ T \mid \varepsilon
  \end{align*}
  \]
• The LL(1) parsing table:

<table>
<thead>
<tr>
<th></th>
<th>int</th>
<th>*</th>
<th>+</th>
<th>(   )</th>
<th>$</th>
</tr>
</thead>
<tbody>
<tr>
<td>E</td>
<td>T X</td>
<td></td>
<td></td>
<td></td>
<td>T X</td>
</tr>
<tr>
<td>X</td>
<td></td>
<td></td>
<td>+ E</td>
<td>\varepsilon</td>
<td>\varepsilon</td>
</tr>
<tr>
<td>T</td>
<td>\text{int} Y</td>
<td></td>
<td></td>
<td></td>
<td>( E )</td>
</tr>
<tr>
<td>Y</td>
<td>\ast T</td>
<td>\varepsilon</td>
<td>\varepsilon</td>
<td>\varepsilon</td>
<td></td>
</tr>
</tbody>
</table>

Bottom-up Parsing

• YACC uses bottom up parsing. There are two important operations that bottom-up parsers use. They are namely shift and reduce.
  – (In abstract terms, we do a simulation of a Push Down Automata as a finite state automata.)
• Input: given string to be parsed and the set of productions.
• Goal: Trace a rightmost derivation in reverse by starting with the input string and working backwards to the start symbol.