COMPILER CONSTRUCTION (Non-recursive Predictive parser) **Fall 2019**

Lecture 17: Aug. 22,23 2019

Instructor: K.R. Chowdhary *: Professor of CS*

17.1 Non-recursive predictive parsing

A non-recursive predictive parser can be built by maintaining a stack a stack explicitly, rather than implicitly via recursive calls. The parser mimics a leftmost derivation. If w is the input that has been matched so far, then the stack holds a sequence of grammar symbols α such that,

$$S \Rightarrow^* w\alpha \tag{17.1}$$

The parser we are going to use is called *table driven* parser (see Fig. 17.1), with following arrangements:

- It has an input buffer that contains the string to be parsed with \$ as the end marker symbol,
- a stack containing a sequence of grammar symbols, and
- a parsing table.

The output is parse-tree. The bottom of the stack also holds the end marker symbol \$. Initially the symbol on top of \$ symbol in stack is start symbol of the grammar.

The non-recursive predictive parser constructs a top-down parse-tree. The parser is controlled by a program that read X, the symbol on top of the stack, and a – the current input symbol. If X is non-terminal, the parser chooses an X-production by consulting entry $M[X, a]$ in the parsing table M. In addition, a code be executed here, say, the code for constructing a node for a parse-tree. If X is a terminal symbol, then it checks for a match between the terminal symbol X and current symbol input a. if matched, the terminal is popped from stack, input pointer is advanced to next symbol, and the process repeats for next symbol on the stack. In case X is terminal but not matching with input symbol, then it is case of error.

The behaviour of the parser can be described in terms of *configurations*, which give the stack contents and the remaining input.

The Algorithm 1 describes how the configurations can be manipulated. The working of the algorithm is as follows: Initially, the parser is in a configuration with w \$ in the input buffer

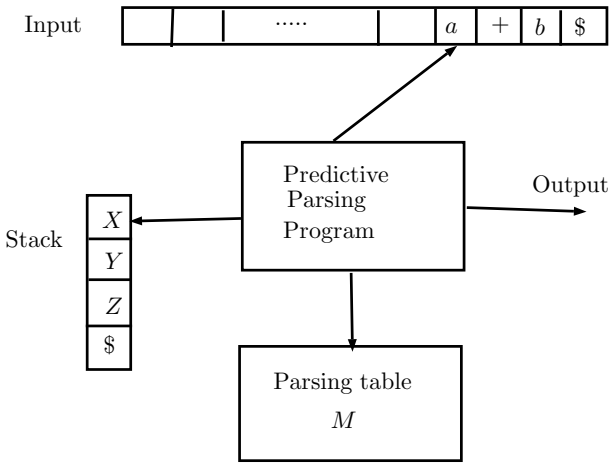

Figure 17.1: Table driven predictive parser

and the start symbol S of grammar G is on the top of the stack, i.e., just above \$. The algorithm uses the predictive parsing table M to produce a predictive parsing for the input.

Every output action in the algorithm corresponds to construction of parse-tree, and each output action is a step in construction of parse-tree, which adds a subtree under one of the child node of the already partly constructed parse-tree. The stack stores sentential form its with left-most symbol on top of the stack.

In the following we consider an example to demonstrate steps of parsing.

Example 17.1 *Parse the expression* $id + id \times id$.

Consider the grammar given below:

$$\begin{aligned} E &\to T\ E' \\ E' &\to +T\ E' \mid \varepsilon \\ T &\to F\ T' \\ T' &\to \times F\ T' \mid \varepsilon \\ F &\to (E) \mid id \end{aligned} \tag{17.2}$$

and the parsing table **??** (Page no. **??**), the moves of the non-recursive predictive parser (Algorithm 1) steps are shown in table 17.1. These moves corresponds to leftmost derivation:

$$E \Rightarrow TE' \Rightarrow FT'E' \Rightarrow ....$$

Note that the sentential form in this derivation corresponds to the input that has already been matched, in the column "matched", followed by stack contents. If we concatenate these columns in the order they exist in table, we get the usual sentential form. The top of the stack is to the left in the table. When we consider bottom up parsing, it will be more

Algorithm 1 Predictive parsing (Input: \$ S on stack, w\$ in input buffer)

```
1: Set ip to point to first left-most symbol of w;
2: Set X equal to top stack symbol;
3: ; repeat while stack is not empty
4: while (X ≠ $) do
5:    if (X is a) then
6:       pop the stack and advance ip;
7:    else
8:       if (X is terminal) then
9:          error();
10:      else
11:         if (M[X, a] is an error entry) then
12:            error();
13:         else
14:            if (M[X, a] = X → Y1Y2...Yk) then
15:               output the production X → Y1Y2...Yk;
16:               pop the stack;
17:               push YkYk−1...Y1 onto stack with Y1 at top;
18:               set X to top of stack symbol;
19:            end if
20:         end if
21:      end if
22:   end if
23: end while
```

natural to show the top of the stack to the right. The input pointer points to the left most symbol of the string in the INPUT column. □

Table 17.1: Moves of predictive parser for input $id + id \times id$

MATCHED	STACK	INPUT	ACTION
	$E\ \$$	$id + id \times id\ \$$	
	$TE'\ \$$	$id + id \times id\ \$$	Output $E \to TE'$
	$FT'E'\ \$$	$id + id \times id\ \$$	Output $T \to FT'$
	$id\ T'E'\ \$$	$id + id \times id\ \$$	Output $F \to id$
id	$T'E'\ \$$	$+id \times id\ \$$	matched id
id	$E'\ \$$	$+id \times id\ \$$	Output $T' \to \varepsilon$
id	$+TE'\ \$$	$+id \times id\ \$$	Output $E' \to +TE'$
$id+$	$TE'\ \$$	$id \times id\ \$$	match $+$
$id+$	$FT'E'\ \$$	$id \times id\ \$$	Output $T \to FT'$
$id+$	$id\ T'E'\ \$$	$id \times id\ \$$	Output $F \to id$
$id + id$	$T'E'\ \$$	$\times id\ \$$	match id
$id + id$	$\times FT'E'\ \$$	$\times id\ \$$	Output $T' \to \times FT'$
$id + id\times$	$FT'E'\ \$$	$\times id\ \$$	match $\times$
$id + id\times$	$idT'E'\ \$$	$id\ \$$	Output $F \to id$
$id + id \times id$	$T'E'\ \$$	$id\ \$$	match id
$id + id \times id$	$E'\$$	$id\ \$$	Output $T' \to \varepsilon$
$id + id \times id$	$\$$	$\$$	Output $E' \to \varepsilon$

17.2 Review Questions

1. Can the values in $FIRST$ and $FOLLOW$ be non-terminals?
2. Find out the $FIRST(A)$ in the following cases:

   (a) $A \to aA$
   (b) $At\ aAB$
   (c) $A \to BbA$
   (d) $A \to \varepsilon$

17.3 Exercises

1. Show that for the grammar $S \to aSa \mid aa$, which generate all the strings of even length, a recursive descent parser recognizes strings $aa, aaaa, aaaaaaaa$, but not $aaaaaa$.
2. Do we need $LL(1)$ grammar to select keyword, i.e., is is necessary to lookahead for first character in the body of a production to check if that production is to be selected, e.g., *if*, *then*, etc.
3. Given the grammar in above question, parse the expression $w = aaaba$ using predictive parser. Construct a table. Construct a parsing table to show all moves with progressive contents of stack, input, output, parse-tree, and matched symbols.
4. Given the productions for some grammar $S \to aBS$, $B \to ABb \mid \varepsilon$, $A \to aA \mid \varepsilon$, construct the predictive parser table for the above productions.
5. Explain in your own words, the construction procedure of predictive-parser table.
6. Explain in descriptive form, in your own words, the predictive parser algorithm.
7. List all the moves for parsing each of the following expressions, for table-driven predictive parser.

   (a) $+XY$
   (b) $XY+$
   (c) (XY)
   (d) $X()$
   (e) XX
   (f) $(+XY)$
   (g) $()+$
   (h) $\times()$

   Also answer the followings:

   i. Are the rules presented in this chapter sufficient for error recover for above errors?
   ii. What special situation you come across in certain errors, for which the rules discussed so far are not sufficient?
   iii. Make your own suggestions for every other expression for error recovery.

References

[1] Compilers: Principles, Techniques, and Tools (2nd Edition) by Alfred V. Aho, Monica S. Lam, et al., Sep 10, 2006.

[2] Compiler design in C (Prentice-Hall software series) by Allen I Holub, Jan 1, 1990.

[3] Engineering a Compiler, by Keith D. Cooper and Linda Torczon, Morgan Kaufmann Publishers, 2004.

[4] Tools for Large-scale Parser Development, Proceedings of the COLING-2000 Workshop on Efficiency In Large-Scale Parsing Systems, 2000, pp. 54-54, `http://dl.acm.org/citation.cfm?id=2387596.2387604`.

[5] `https://www.antlr.org/`