COMPILER CONSTRUCTION (Recursive Descent Parser) **Fall 2019**

Lecture 14:15: Aug. 14,16 2019

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

Disclaimer: *These notes have not been subjected to the usual scrutiny reserved for formal publications. They may be distributed outside this class only with the permission of the Instructor.*

14.1 Problems in Top-Down Parsing

Several problems can complicate the top-down parsing. For example, the grammars with left recursion can cause termination problems. Choosing the wrong expansion rule necessitates backtracking. In fact the key issue is that in each step, the parser should choose the correct production to expand the growing fringe of the parse-tree.

Writing backtracking grammar requires care, the grammar must avoid left recursion. It must also ensures that single symbol lookahead suffices to determine the correct expansion at each step. In the following we discusses these issues.

Example 14.1 *Demonstrating Left-recursion.*

Let "$A \rightarrow Ab \mid b$" is left recursive grammar and it is required to derive the string $w = cdbbef$, using top-down parser, and b, c, d, e, f are terminal symbols. Let "$S \Rightarrow^* cd \uparrow Abef$" be the sentential form, where A is the next non-terminal to be expanded. If we replace A using production $A \rightarrow Ab$, the sentential form becomes $S \Rightarrow^* cd \uparrow Abbef$. Since, there is still non-terminal A at the same position, we again put the same substitution and $S \Rightarrow^* cd \uparrow Abbbef$, the process continues indefinitely, creating an infinite loop.

However, if we choose the production $A \rightarrow b$, in the original, the sentential form becomes $S \Rightarrow^* cdb \uparrow bef$, it correctly generates the final string $w = cdbbef$. But, if $w = cdbbbef$, with currently $S \Rightarrow^* cd \uparrow Abef$, it would require use of $A \rightarrow Ab$ once, then $A \rightarrow b$. But, there is no way to find out how many times $A \rightarrow Ab$ is to be used and $A \rightarrow b$ is to be used. This is the situation, where left-recursion is problem! □

14.1.1 Eliminating Left Recursion

A left-recursive grammar can cause a deterministic top-down parser to loop indefinitely by expanding the fringe of the parse-tree without generating a leading terminal symbol. Since, backtracking can only be triggered by a mismatch between the terminal symbol and the current input symbol, the loop will continue forever, since it does not generate terminal symbol that matches with current input symbol.

A grammar is left-recursive if it has a non-terminal A such that there is a derivation $A \Rightarrow^+ A\alpha$ for some string α. The top-down parsing method cannot handle left-recursive grammar, hence a transformation is needed to eliminate the left recursion. A left recursive production $A \to A\alpha \mid \beta$ could be replaced by a *non-left recursive* productions:

$$\begin{aligned} A &\to \beta A' \\ A' &\to \alpha A' \mid \varepsilon \end{aligned} \tag{14.1}$$

The transformation introduces a new non-terminal A', and transfers the recursion onto A'. It also adds a rule $A' \to \varepsilon$, where ε is empty string. The ε-production requires careful interpretation in the parsing algorithm. If the parser expands by rule $A' \to \varepsilon$, the effect is to advance the current *node* along the tree's fringe by one position. In the expression-grammar, the left-recursion appears in the E and T productions (see table **??**). We can change these productions to non-left recursion productions, as shown in Table 14.1.

Table 14.1: Right-recursion Grammar

Rule No.	Rule
1	$E \to TE'$
2	$E' \to +TE'$
3	$\mid -TE'$
4	$\mid \varepsilon$
5	$T \to FT'$
6	$T' \to \times FT'$
7	$\mid \div FT'$
8	$\mid \varepsilon$
9	$F \to (E)$
10	$\mid num$
11	$\mid id$

When compared these rules with the grammar of left-recursion, shown in Table **??** (Page no. **??**), we note that there is no left-recursion in the grammar, such grammar is called *right recursive grammar.*

Example 14.2 *Construct parse-tree for the expression $id * id + id$, using grammar shown in table (14.1).*

The derivation shown in Fig. 14.1 is top-down parsing, for the expression $id * id + id$.

If we concatenate the leaves of parse-tree in Fig. 14.1 from left to right, we get $id * id + id$. Note that when ε is concatenated with any symbol, the result is that symbol. We note two things in this derivation,

1. There is no left-recursion, and
2. There is no ambiguity in the parse trees, because for each expression there is one and only one derivation tree possible.

Note that depending on any given expression, a parse-tree can be constructed in similar way. □

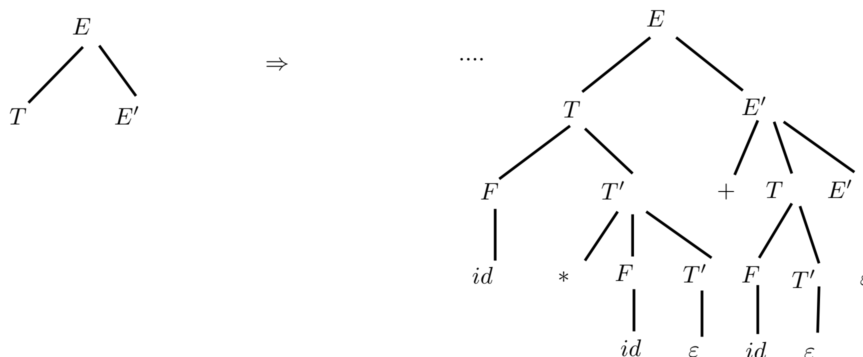

Figure 14.1: Parse-tree for the expression: $id * id + id$

At each step, for a top-down parse-tree, the main problem is of determining the production to be applied for a left-most non-terminal, say A. Once a production is chosen, the rest of the parsing process consists of matching the terminal symbols in the terminal body of a production, with the input string. The top-down parsing we are going to use is called *recursive descent parsing*, which may require backtracking to find the correct A-production to be applied.

A *predictive parsing* is a special case of recursive descent parsing, where no backtracking is required. This is because the predictive parsing chooses the correct A-production by looking ahead at the input a fixed number of symbols, typically we may look ahead at *one* (that is, the next input symbol). For example, in the Fig. 14.1, while applying the production at T' at the left-subtree, we need to match the first symbol in the production body, i.e., $\times$, in addition to the *head* of the rule. When this matched it is sufficient condition of look-ahead *one* symbol. The third case in the production of T' has ε as the first symbol in the production body, which does not match with any symbol in input sentence. So, we are in a position to correctly choose the production $T' \to \times\ F\ T'$, and not $T' \to \varepsilon$.

The class of grammars for which we can construct predictive parsers that look for k symbols ahead in the input is called $LL(k)$ grammar. The Grammar just now we have mentioned above to look ahead one symbol, is called $LL(1)$ Grammar.

14.2 Recursive-Descent Parsing

A recursive-descent parsing program consists of a set of procedures, one for each non-terminal of the production rules. The execution begins with the procedure for the start symbol, which halts and announces a success if its procedure body scans the entire input string. If start symbol and productions are $S \to X_1X_2...X_n$, then the procedure for S calls procedure for X_1, then procedure for X_2, upto X_n. The X_i may recursively call further procedures, and so on. Hence the name "recursive descent". The Algorithm 1 shows the algorithm for recursive descent parsing.

Note that the above procedure is non-deterministic, since it begin by choosing the A-production to apply in a manner that is not specified.

The general recursive-descent may require backtracking, i.e., it may require repeated scans over the input. However, the backtracking is rarely used in practice to parse programming

Algorithm 1 Recursive-descent parsing Algorithm-I for a non-terminal symbol A

```
1: void A(){
2: Choose a production, say A → X_0 X_1 ....X_{k-1}
3: for (i = 0 to k − 1) do
4:    if (X_i is a non-terminal) then
5:       call procedure X_i
6:    else
7:       if (X_i is equal to current input symbol 'a') then
8:          advance the input to the next symbol, as well to next X_i
9:       else
10:         /* an error has occurred */
11:      end if
12:   end if
13: end for
14: }
```

language constructs. Even for NLP backtracking is not efficient, and tabular methods (discussed in the sections to follow), like dynamic programming [5] are used.

To allow the backtracking, the code in the algorithm 1 needs to be modified as follows. First, we cannot use unique A-production as in line-number 2. So we must try each of the several productions in some order. Then failure at line, 10 is not ultimate failure, but suggest only that we need to return to line 2, and try another A-production. Only when there are no A-productions to try, we declare that an input error has occurred. For trying another A-production, we need to reset the input pointer to where it was when we first reached line 2. Thus, a local variable is needed to store this input pointer for future use.

Example 14.3 *Recognition of a string using the top-down algorithm 1.*

Consider the grammar $S \rightarrow c\ A\ d$, $A \rightarrow a\ b \mid a$. Assume that it required to generate the string $w = c\ a\ d$, using this grammar. The Fig. 14.2 shows the use of top-down parsing with backtracking facility to generate the string w.

In the top-down parsing there are two pointers in the expression of sentential form, where one points to the current symbol which might be required to be expanded (cptr), and another pointer (pptr) points to the previous position, to which it may be required to backtrack in case the remaining string does not match due to the current substitution. In the step 1, we have expand the symbol S and current pointer points to A, and previous pointer to original (0) position. In the step 2, when A has been expanded by "$a\ b$", the pointer advances to b, while previous pointer still points to 1. Knowing that there is no match between the string generated by parse-tree and input sentence, we backtrack to previous position of step 1, in the step 3. We check another substitution alternative, and find that it is $A \rightarrow a$, which is substituted, in step 4, and we note that word sequence $w = c\ a\ d$ is generated. □

14.3 Eliminating the need of Backtrack

Using the right-recursive grammar, the top-down parser can handle the string $x - 3 \times y$ without backtracking. To know this, we try to analyse how the parser makes decision that

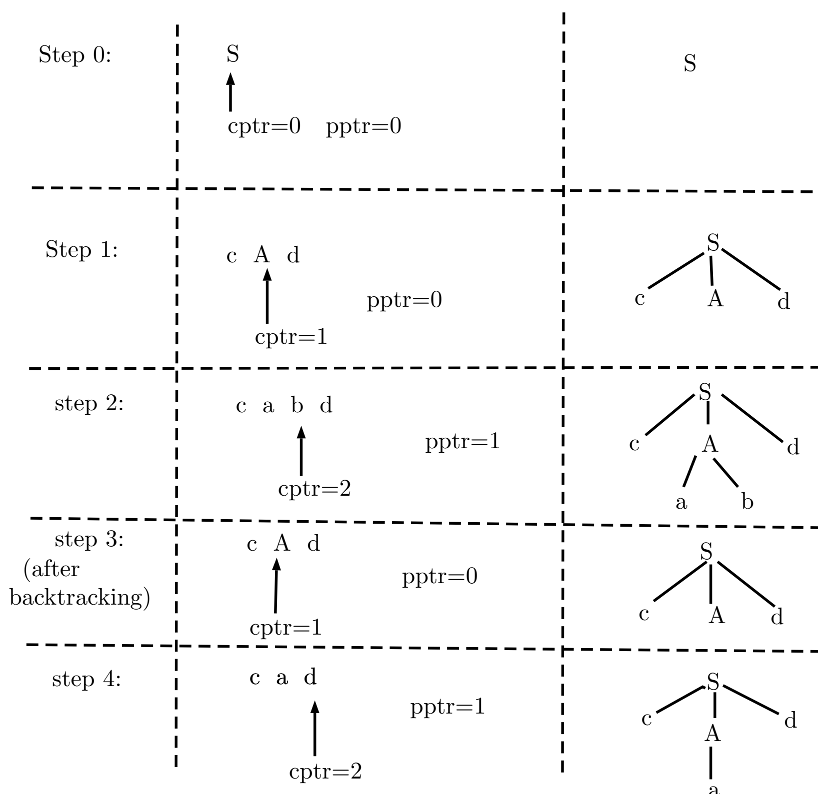

Figure 14.2: Steps in top-down parsing

it must retract through backtracking. The critical choice occurs when the parser selects a production using which it expands the lower fringe of the partly constructed parse-tree. When it tries to expand A, it must pick a rule $A \rightarrow \beta$. The algorithm we discussed (Algorithm **??**) picks the rule arbitrarily. It the parser can pick the appropriate rule, it could avoid the backtracking.

In the right-recursive grammar, the parser can always make a correct choice, by comparing the next word in the input stream , against alternative right-hand sides (body) of the rules, for the left most non-terminal on the fringe. In the following example we make use of right-recursive grammar to derive (parse) an expression correctly.

Example 14.4 *Parse the sentence $x - 3 \times y$ using right-recursive grammar (Table 14.1, page no. 14-2).*

The table 14.2 demonstrates the use of right-recursive grammar to derive the expression $x - 3 \times y$, which makes use of lookahead symbol for *one* position. Note that, grammar is called $LL(1)$ grammar.

We note that first two rules (1, 5) are obvious as the grammar has only one rule for each. Expanding F to id requires a choice. The choice becomes clear when we look for the current word in the input stream. Next, T' needs to be expanded; where rule number 6 and 7 do not match, hence rule 8 is selected. When this process continues, we accept the word. □

Table 14.2: Tabular representation of Right recursion

Rule	Sentential Form	Input Sentence
	E	$\uparrow x - 3 \times y$
1	$T\ E'$	$\uparrow x - 3 \times y$
5	$F\ T'\ E'$	$\uparrow x - 3 \times y$
11	$id\ T'\ E'$	$\uparrow x - 3 \times y$
$\rightarrow$	$id\ T'\ E'$	$x \uparrow -3 \times y$
8	$id\ E'$	$x \uparrow -3 \times y$
3	$id\ -T\ E'$	$x \uparrow -3 \times y$
$\rightarrow$	$id\ -T\ E'$	$x- \uparrow 3 \times y$
5	$id\ -F\ T'\ E'$	$x- \uparrow 3 \times y$
10	$id\ -num\ T'\ E'$	$x- \uparrow 3 \times y$
$\rightarrow$	$id\ -num\ T'\ E'$	$x - 3 \uparrow \times y$
6	$id\ -num \times F\ T'\ E'$	$x - 3 \uparrow \times y$
$\rightarrow$	$id\ -num \times F\ T'\ E'$	$x - 3 \times \uparrow y$
11	$id\ -num \times id\ T'\ E'$	$x - 3 \times \uparrow y$
$\rightarrow$	$id\ -num \times id\ T'\ E'$	$x - 3 \times y \uparrow$
8	$id\ -num \times id\ E'$	$x - 3 \times y \uparrow$
4	$id\ -num \times id$	$x - 3 \times y \uparrow$

Complexity Analysis Considering n number of symbols (words) in the input, there will be n number of comparisons with the to match the next *node* in the fringe of the tree. In the worst case, there can be n number of substitutions, if there are all non-terminals in the tree. Hence, the complexity is $n+n$, i.e., $O(n)$. Here we assumed that searching for proper rule in the rule-base, is done using hash-based search, and has complexity of $O(1)$.

We can now formalize the property that makes the right right-recursion expression grammar backtrack-free. At each point in the parse, the choice of an expansion is obvious because each alternative for the left-most non-terminal leads to a distinct terminal symbol. Comparing the next word in the input sentence against those choices provides the correct expansion.

14.4 Review Questions

1. Can the top-down parser handle left-recursive grammar? Justify with example.
2. What is left-recursive grammar? Why it is considered harmful?
3. Can there be right-recursive grammar? Is it good or bad?
4. Which of the following does not require backtracking?
   (A) Recursive descent-parsing (C) Both A and B
   (B) Predictive parsing (D) None of A or B
5. Predictive parsing is special case of recursive descent parsing. (True / False)
6. Recursive descent parsing is special case of Predictive parsing. (True / False)
7. Recursive descent parsing uses backtracking. (True/False)
8. A recursive descent parser is (select one or more correct choices):
   (A) Top Down (B) Bottom up
   (C) Backtracking (D) Not backtracking

14.5 Exercises

1. In the productions,

$$E \rightarrow E + T \mid T$$
$$T \rightarrow T \times F \mid F$$
$$F \rightarrow (E) \mid id,$$

   (a) will the loop continue forever for deriving expression $id + id$? Justify.
   (b) Is there any possible finite expression for which the loop will terminate? If Yes, what is the expression? Justify.

2. Consider the following expressions:
   (a) $x \times 2 + y \times 2$
   (b) x/y

   Parse these expressions using the *expression grammar*, and top-down LL grammar. write down all the manual moves. Assume that the parser randomly selects the correct production rule.

3. Show that following expressions cannot be parsed using expression grammar:
   (a) $x + -y$
   (b) $+ + x$

4. What are the drawbacks of the Algorithm **??** (page no. **??**). Discuss all of them in detail.

5. Convert the following left-recursive grammars into right-recursive grammars.
   (a) $E \rightarrow E + T \mid a$
   (b) $E \rightarrow E \times T \mid a$
   (c) $E \rightarrow E - T \mid a$
   (d) $E \rightarrow E + T \mid \varepsilon$

6. What are the important characteristics of *right-recursive grammar*?

References

[1] Earley, Jay (1970), "An efficient context-free parsing algorithm" (PDF), Communications of the ACM, 13 (2): 94-102, doi:10.1145/362007.362035

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

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

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

[5] 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`.