**COMPILER CONSTRUCTION (Top Down Parsing)** **Fall 2019**

Lecture 12:13: Aug. 08:09 2019

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



## 12.1 Context-free grammars and their parsers

We have discussed about CFG, LR(1) grammars, LL(1) grammar, and regular grammar (RG). These sets are nested: $LR(1) \supset LL(1) \supset RG$. The arbitrary context-free grammars require more time to parse than than the restricted sets LL(1) or LR(1). For example Earley's algorithm parse context-free grammar [1] in $O(n^3)$ time, worst case, where $n$ is number of words in the input stream.

The LR(1) grammars include a large subset of unambiguous context-free grammars. LR(1) grammars can be parsed, bottom up, in a linear scan from left to right, looking at most one word ahead of the current input symbol. Several tools exists, that automate the process of LR(1) parsing.

The LL(1) grammars are an important subset of the LR(1) grammars. The LL(1) grammars can be parsed in top-down mode, in a linear scan from left to right, with a single word look ahead. Two different parsers of are common for LL(1) grammars are:

1. hand coded recursive descent parsers, and,
2. table driven parsers called LL(1) parsers.

Many interesting properties of languages can be easily expressed with LL(1) grammars.

The regular grammars are for scanners, and can be implemented through DFA.

## 12.2 Top-Down Parsing

The top-down parsing can be viewed as the problem of constructing a parse tree for the input string, starting from the root and creating the nodes of the parse-tree in *preorder*, i.e., *depth-first order*. In other words, the top-down parsing can be considered as the leftmost derivation of an input string.

In the process of parsing, at each point the process considers a partially built parse tree. It selects a non-terminal symbol on the *lower fringe* of the tree and extends it by adding

children that corresponds to right-hand side of some production for that non-terminal. However, it cannot extend the frontier from a terminal. This process is continued until either:

1. all fringe of the parse tree contains only terminal symbols, and the input stream has been exhausted; or
2. a clear mismatch occur between the fringe of the partially built parse tree and the input stream.

In the first case, the parse succeeds. In the second case, two possibilities exists. The parser may have selected wrong production at some earlier step in the process. In this case backtracking will lead to correct choice of production. Other possibility is, the input string may not be a valid sentence in the language being parsed. In this case, the back tracking will fail, and the parser should report the syntactic error to the user.

One key factor makes the top-down parsing efficient: there exists a large subset of context-free grammars for which backtracking is never needed.

We will make use of classic *expression grammar* shown below.

Table 12.1: Expression Grammar

| RuleNo. | Production Rule |
|---|---|
| 1 | $E \rightarrow E + T$ |
| 2 | $\mid E - T$ |
| 3 | $\mid T$ |
| 4 | $T \rightarrow T \times F$ |
| 5 | $\mid T/F$ |
| 6 | $F$ |
| 7 | $F \rightarrow (E)$ |
| 8 | $\mid num$ |
| 9 | $\mid id$ |

The productions belong to the grammar $G = (V, \Sigma, S, P)$, where $V$ is set of non-terminals, $\Sigma$ is set of terminals, $S$ is start symbol, and $P$ is set of productions.

In the above productions, non-terminals $E, T, F \in V$, respectively, stand for *Expression*, *Term*, and *Factor*. The symbols *num* and *id* stand for number, and identifier, respectively, which are terminals.

## 12.3 Leftmost Top-down Parsing Algorithm

We consider the grammar, $G = (V, \Sigma, S, P)$, where $V$ is set of non-terminals,, and $\Sigma$ is set of terminal words, or tokens. Note that $S$ is special non-terminal, from where the derivation starts. The Algorithm 1 is top-down parsing algorithm. The process works on the lower fringe of the tree, that always corresponds to the sentential form. We will be expanding the left-most non-terminal in each step, which corresponds to the left-most derivation because parser considers words in the left-to-right order, the order in which scanner produces them.

**Example 12.1** *Demonstrating the working of Algorithm 1*

**Algorithm 1** Leftmost Top-down Parsing Algorithm

```
1: word ← nextWord()
2: root ← startSymbol
3: node ← root
4: while True do
5:    if node ∈ Σ and node matches input word then
6:       Advance node to next node on the fringe
7:       word ← nextWord()
8:    else
9:       if node ∈ Σ and node does not match input word then
10:         backtrack
11:      else
12:         if node ∈ V then
13:            Pick a rule "node → β"
14:            Extend tree from node by building β
15:            node ← left most symbol in β
16:         end if
17:      end if
18:   end if
19:   if node is empty and word is eof then
20:      accept and exit the loop
21:   else
22:      if node is empty and word is not eof then
23:         backtrack
24:      end if
25:   end if
26: end while
```

To understand the top-down parsing algorithm, assume that the parser goes through to recognize $x - 3 \times y$ using expression grammar shown in Table 11.1. The goal symbol of the grammar is $E$, thus parser begins with the tree rooted at $E$. To Show the parser's actions, we will expand the tabular representation of a derivation, as shown in Table 11.2. The left-most column shows the grammar rule used to reach each state (i.e., sentential form); the center column shows the lower fringe of the partially constructed parse tree, which is the most recently derived sentential form. On the right column is representation of the input stream of tokens. The "↑" shows the position of the scanner, which properly precedes the current input symbol. The actions "→" in the rule column represent advancing the input pointer, and "→" backtracking through a set of productions. The first several moves of the parser are shown in Table 11.2

Table 12.2: Tabular representation of Parsing

| Rule | Sentential Form | Input Sentence |
|---|---|---|
| | $E$ | $\uparrow x - 3 \times y$ |
| 1 | $E + T$ | $\uparrow x - 3 \times y$ |
| 1 | $E + T + T$ | $\uparrow x - 3 \times y$ |
| 1 | $E + T + ...$ | $\uparrow x - 3 \times y$ |
| 1 | ... | $\uparrow x - 3 \times y$ |

The Table 11.2 begins with symbol $E$, and expands it with first right-hand side of first

rule in the the expression grammar, which produces the fringe $E + T$. To produce the leftmost derivation, the parser must expand the leftmost non-terminal on the fringe, which is still $E$. If it chooses the Rule 1 consistently, this leads to infinite sequence of expansions, each of that makes no progress. This suggest the problem with the expression grammar. A production is *left-recursive* if the first symbol on its right-hand side is the same as the symbol on the left-hand side. The grammars that exhibit left recursion can cause top-down parser to expand forever while making no progress.

To forget for a moment, the problem of left-recursion of top-down parser, and illustrate another potential problem, we will allow the parser to choose the productions in arbitrary manner. For example, we might try to rewrite $E$ into $id$. This produces the expansion rule 3 ($E \rightarrow T$), rule 6 ($T \rightarrow F$), and rule 9 ($F \rightarrow id$)(See Table 11.3).

Table 12.3: Tabular representation of Parsing

| Rule | Sentential Form | Input Sentence |
|---|---|---|
| | $E$ | $\uparrow x - 3 \times y$ |
| 3 | $T$ | $\uparrow x - 3 \times y$ |
| 6 | $F$ | $\uparrow x - 3 \times y$ |
| 9 | $id$ | $\uparrow x - 3 \times y$ |
| $\rightarrow$ | $id$ | $x \uparrow -3 \times y$ |

At this point, the left most symbol on the fringe is a terminal symbol, so the parser checks whether the symbol matches the current input symbol, held in the *word*. Since they match, it advances one symbol right-ward on the fringe and advances the input stream by calling *nextWord()*. Unfortunately, the parse trees fringe ends with $id$ even though the input stream continues, with "$-$". This mismatch shows that the steps taken so far do not leads to a valid parse. Either the parser made an incorrect choice at some earlier expansion, or the input string is not valid sentence for the expression grammar.

A parser handle this situation by backtracking. If the parser was making systematic choices, it would *retract* the most recent action, i.e., the expansion by rule number 9, and try the other possibilities for $F$. When these failed, it would retract the expansion by rule 6 and try the other possibilities for $T$. Finally it would try the alternatives for the expansion by rule 3 and discover that the first step should have been an expansion by rule 2, $E \rightarrow E - T$ (see expression grammar in Table 11.1).

From that point, the parser can work forward, applying the sequence $3, 6, 9$ to derive $id$ from $E$ in the first position on the fringe. Moving ahead on the fringe and in the input stream, it discovers that "$-$" sign in the input matches the "$-$" on the fringe.

Table 12.4: Tabular representation of Parsing

| Rule | Sentential Form | Input Sentence |
|---|---|---|
| | $E$ | $\uparrow x - 3 \times y$ |
| 2 | $E - T$ | $\uparrow x - 3 \times y$ |
| 3 | $T - T$ | $\uparrow x - 3 \times y$ |
| 6 | $F - T$ | $\uparrow x - 3 \times y$ |
| 9 | $id - T$ | $\uparrow x - 3 \times y$ |
| $\rightarrow$ | $id - T$ | $x \uparrow -3 \times y$ |
| $\rightarrow$ | $id - T$ | $x- \uparrow 3 \times y$ |

At this point, the parser should expand as per the sequence $4, 6, 8$ to match the *num* and

leave the appropriate correct context on the fringe (see Table 11.5). It can match *num* against the fringe, then advance and match "×". The final expansion uses rule 9 to place an *id* on the fringe to match the final input symbol.

Table 12.5: Tabular representation of Parsing

| Rule | Sentential Form | Input Sentence |
|---|---|---|
| 4 | $id - T \times F$ | $x- \uparrow 3 \times y$ |
| 6 | $id - F \times F$ | $x- \uparrow 3 \times y$ |
| 8 | $id - num \times F$ | $x- \uparrow 3 \times y$ |
| $\rightarrow$ | $id - num \times F$ | $x - 3 \uparrow \times y$ |
| $\rightarrow$ | $id - num \times F$ | $x - 3\times \uparrow y$ |
| 9 | $id - num \times id$ | $x - 3\times \uparrow y$ |
| $\rightarrow$ | $id - num \times id$ | $x - 3 \times y \uparrow$ |

At this point, the lower fringe contains only terminal symbols and the input has been exhausted. Now, the *node* is empty and *word* is $eof$, so it reports success and halts.

Note an important point about this algorithm. If the input matches with the lowest fringe of the top-down parse-tree, the algorithm terminates with success. However, if it does not, the algorithm keeps on backtracking, and it never terminates. Hence, the success is not guaranteed for this algorithm.

Note that, we could have made erroneous expansions at many points in the parse, triggering many back tracking. For the sake of clarity and brevity, we did not. □

# 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, url-http://dl.acm.org/citation.cfm?id=2387596.2387604.