

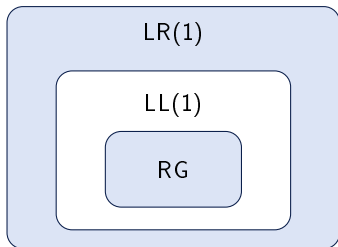
Top-Down Parsing

Compiler Construction, Lecture 7

K.R. Chowdhary, Professor of CS

August 8, 2019 | Fall 2019

Grammar Classes and Top-Down Parsing



$$LR(1) \supset LL(1) \supset RG$$

General CFG parsing needs more time, e.g. Earley's algorithm is $O(n^3)$.

- **LR(1)**: parsed bottom-up, one linear left-to-right scan, one word look-ahead
- **LL(1)**: parsed top-down, same scan and look-ahead; two common parsers: hand-coded *recursive descent* and table-driven *LL(1)*
- **RG**: scanners, implemented as DFAs

Top-down parsing

Build the parse tree from the root in preorder (depth-first), i.e. a **leftmost derivation**. Expand a non-terminal on the lower fringe; stop when the fringe is all terminals matching the input, or on a mismatch.

Leftmost Top-Down Parsing Algorithm

Expression grammar

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

Start: $word \leftarrow nextWord()$, $node \leftarrow$ start symbol.

Repeat:

- **Terminal matches** the word: advance $node$ along the fringe and fetch the next word
- **Terminal mismatches:** backtrack
- **Non-terminal:** pick a rule $node \rightarrow \beta$, extend the tree, move to the leftmost symbol of β
- **Fringe exhausted:** accept if the word is *eof*, else backtrack

Success is not guaranteed: on failure the algorithm may backtrack forever, and left recursion makes it expand forever.

Example: Parsing $x - 3 \times y$ with Backtracking

- Always choosing rule 1 expands $E \rightarrow E + T \rightarrow \dots$ forever with no progress
- An arbitrary choice
 $E \rightarrow T \rightarrow F \rightarrow id$ matches x , but the fringe ends while input remains $(-3 \times y)$
- The parser must **backtrack** through rules 9, 6, 3 and discover that rule 2 ($E \rightarrow E - T$) was needed
- After that, the sequence 3, 6, 9 gives id , and $-$ matches

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

$\rightarrow$ advances the input; the fringe is all terminals and input is *eof*, so accept.

Eliminating Left Recursion

$A \rightarrow Ab \mid b$ deriving $cdbb ef$: from $cd \uparrow Ab ef$, using $A \rightarrow Ab$ gives $cd \uparrow Ab bef$ and so on forever. There is no way to know how many times $A \rightarrow Ab$ is needed.

Transformation

$A \rightarrow A\alpha \mid \beta$ becomes

$$A \rightarrow \beta A'$$

$$A' \rightarrow \alpha A' \mid \varepsilon$$

Choosing $A' \rightarrow \varepsilon$ simply advances the node along the fringe.

Right-recursive expression grammar

$$E \rightarrow T E'$$

$$E' \rightarrow + T E' \mid - T E' \mid \varepsilon$$

$$T \rightarrow F T'$$

$$T' \rightarrow \times F T' \mid \div F T' \mid \varepsilon$$

$$F \rightarrow (E) \mid num \mid id$$

No left recursion and no ambiguity: each expression has exactly one parse tree. Compare the next input word with the alternatives to choose a rule, so **no backtracking**: LL(1), parsed in $O(n)$ time.

Recursive-Descent and Predictive Parsing

One procedure per non-terminal; execution starts at the start symbol's procedure.

Procedure for $A \rightarrow X_0 X_1 \cdots X_{k-1}$

for $i = 0$ **to** $k - 1$:

if X_i is a non-terminal: call X_i

else if X_i equals the input symbol: advance input
 else error

- General recursive descent may backtrack: try each A -production in turn, resetting the input pointer
- **Predictive parsing** is the special case with no backtracking, looking ahead k symbols: $LL(k)$

$S \rightarrow cAd, A \rightarrow ab \mid a, w = cad$

Step	Form	Action
0	S	start
1	$c A d$	expand S
2	$c a b d$	$A \rightarrow ab; b \neq d$
3	$c A d$	backtrack
4	$c a d$	$A \rightarrow a$; match

Two pointers are kept: *cptr*, the current symbol, and *pptr*, the position to backtrack to.

Review questions

1. Can a top-down parser handle left recursion? Why is it harmful?
2. Can a grammar be right recursive? Good or bad?
3. Which needs no backtracking: recursive descent, predictive, both or neither?
4. Predictive parsing is a special case of recursive descent: true or false?
5. Is recursive descent top-down, bottom-up, backtracking or not?

Exercises

1. Does the left-recursive expression grammar loop forever on $id + id$? Is there any input that terminates?
2. Parse $x \times 2 + y \times 2$ and x/y with the expression grammar and the LL grammar
3. Show $x + -y$ and $++x$ cannot be parsed
4. Drawbacks of Algorithm 1?
5. Make $E \rightarrow E + T \mid a$ (and the $\times$, $-$, ε variants) right recursive
6. Characteristics of right-recursive grammars?