Introduction

CFG parser that is
• a top-down parser: we start with S and subsequently replace lefthand sides of productions with righthand sides.
• a directional parser: the expanding of non-terminals (with appropriate righthand sides) is ordered; we start with the leftmost non-terminal and go through the righthand sides of productions from left to right.
In particular: we determine the start position of the span of the ith symbol in a rhs only after having processed the i − 1 preceding symbols.
• a LL-parser: we process the input from left to right while constructing a leftmost derivation.

First proposed by Sheila Greibach (for CFGs in GNF).

The recognizer (1)

Assume CFG without left recursion $A \Rightarrow A_0$.

Function top-down with arguments
• $w$: remaining input;
• $\alpha$: remaining sentential form (a stack).

top-down($w, \alpha$) iff $\alpha \Rightarrow w$ (for $\alpha \in (N \cup T)^*, w \in T^*$)

Initial call:
top-down($w, S$)
The recognizer (2)

function top-down(w, α):
    out := false;
    if w = α = ε, then out := true;
    else if w = aw' and α = aa',
        then out := top-down(w', α')  \text{ scan}
    else if α = Xα' with X ∈ N,
        then for all X → X₁...Xₖ:
            if top-down(w, X₁...Xₖα') \text{ predict}
            then out := true;
    return out

The parser (1)

How to turn the recognizer into a parser:
Add an \textit{analysis stack} to the parser that allows you to construct the parse tree.
Assume that for each \( A \in N \), the righthand sides of \( A \)-productions are numbered (have indices).
Whenever
• a production is applied (prediction step), the lefthand side is pushed on the analysis stack together with the index of the righthand side;
• a terminal \( a \) is scanned, \( a \) is pushed on the analysis stack.
(This is needed for backtracking in a depth-first strategy.)

The recognizer (3)

This is exactly what the following PDA-construction for a CFG does:
• start with stack \( Z₀ \) and \( q₀ \).
• \( \delta(q₀, ε, Z₀) = \{(q₁, SZ₀)\} \)
• \( (q₁, α) \in \delta(q₁, ε, A) \) for all \( A \rightarrow α \)
• \( (q₁, ε) \in \delta(q₁, a, a) \) for all \( a \in T \).
• \( \delta(q₁, ε, Z₀) = \{(q₁, ε)\} \)

(LL-PDA construction in JFLAP)
The parser (3)

Problematic grammars for this parser: CFGs that allow for left-recursions. Solutions:

- Eliminate the left-recursion.
  
  Drawback: derivation trees change considerably.

- Make sure, grammar does not contain $\epsilon$-productions or loops.
  
  Then do an additional check:

  \[
  \text{then for all } X \rightarrow X_1 \ldots X_k:\
  \]

  \[
  \text{if } |w| \geq |X_1 \ldots X_k\alpha'|\quad \text{and top-down}(w, X_1 \ldots X_k\alpha')
  \]

  \[
  \text{then } \text{out} := \text{true};
  \]

  This check is useful even for grammars that are not left-recursive.

An example (1)

<table>
<thead>
<tr>
<th>Grammar</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S \rightarrow AB$</td>
</tr>
<tr>
<td>$A \rightarrow aAB \mid a$</td>
</tr>
<tr>
<td>$B \rightarrow b$</td>
</tr>
</tbody>
</table>

| $aABB \Rightarrow abb$ | $aB \Rightarrow abb$ |
| $ABB \Rightarrow abb$ | $B \Rightarrow abb$ |
| $AB \Rightarrow aabb$ | $aB \Rightarrow aabb$ |
| $ABBB \Rightarrow bb$ | $BB \Rightarrow bb$ |
| $aABBBB \Rightarrow bb$ | $aBBB \Rightarrow bb$ |
| $BB \Rightarrow bb$ | $B \Rightarrow b$ |
| $bB \Rightarrow bb$ | $b \Rightarrow b$ |

(basic algorithm)

An example (2)

| $S \Rightarrow aabb$ |
| $AB \Rightarrow aabb$ |

| $aABB \Rightarrow aabb$ | $aB \Rightarrow aabb$ |
| $ABB \Rightarrow abb$ | $B \Rightarrow abb$ |
| $aB \Rightarrow abb$ | $b \Rightarrow abb$ |
| $BB \Rightarrow bb$ |
| $bB \Rightarrow bb$ | $b \Rightarrow b$ |
| $B \Rightarrow b$ |
| $b \Rightarrow b$ |
| $\epsilon \Rightarrow \epsilon$ |

Control structures (1)

In general, directional top-down parsing is non-deterministic because of multiple right-hand sides for single non-terminals.

Two different control strategies: You can

- either proceed depth-first (proceed the right-hand sides one after the other, each time pursuing the possible derivation tree up to the moment where we either find a parse tree or fail)
- or proceed breadth-first (try all right-hand sides in parallel)

Usually, all possible predicts are done before scanning the next input symbol.

These are different control structures, they are not part of the general top-down parsing algorithm.
Control structures (2)
Advantages and disadvantages:

Breadth-first:
- Needs a lot of memory.

Depth-first (backtracking):
- Does not need much memory.
- If all parse trees are searched for and the grammar is known to be ambiguous, more time-consuming than breadth-first.

⇒ breadth-first probably preferable when needing an exhaustive parsing anyway. If only one parse tree needs to be found or if the grammar is not ambiguous, depth-first probably better.

Parser generators (1)
In general, we can
- either implement a general CFG parser (perhaps for a restricted type of CFG) that takes G and w as input

\[ w, G \rightarrow \text{Parser} \rightarrow \text{parse trees/no} \]

- or generate a specific parser for a given grammar. The new parser receives only w as input.

\[ G \rightarrow \text{Parser generator} \rightarrow \text{Parser} \rightarrow \text{parse trees/no} \]

Parser generators (2)
Parser generators for top-down (LL) parsers often use a technique called recursive descent:
- for each non-terminal X, a procedure is generated that tries all rhs of X-productions with calls for all non-terminals it encounters (top-down = descent)
- procedures can call each other, in particular, they can call (directly or via other intermediate calls) itself again (recursive)

Some recursive descent parser generators:
- JavaCC, Java Compiler Compiler: https://javacc.dev.java.net/
- ANTLR, ANother Tool for Language Recognition (generates C++, Java, Python, C#): http://www.antlr.org/

Conclusion
Important features of directional top-down parsing:
- LL-parsing: input processed from left to right, constructs a leftmost derivation;
- parsing steps prediction and scan;
- non-deterministic in general;
- different control structures (breadth-first, depth-first);
- does not work for left-recursive CFGs;
- parser generation with recursive descent.