Parsing
Top-Down Parsing

Laura Kallmeyer

Heinrich-Heine-Universität Düsseldorf

Winter 2017/18
Table of contents

1 Introduction

2 The recognizer

3 The parser

4 Control structures

5 Parser generators

6 Conclusion
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 $i$th 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). Grune and Jacobs (2008)
The recognizer (1)

Assume CFG without left recursion $A \xrightarrow{\dagger} A\alpha$.

Function `top-down` with arguments

- $w$: remaining input;
- $\alpha$: remaining sentential form (a stack).

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

Initial call:
$\text{top-down}(w, S)$
def top-down(w, α):
    out = False
    if w = α = ε:
        out = True
    elif w = aw' and α = aα':
        out = top-down(w', α')  # scan
    elif α = Xα' with X ∈ N:
        for X → X₁...Xₖ in P:
            if top-down(w, X₁...Xₖα'):
                predict
                out = True
    return out
This is exactly what the following PDA-construction for a CFG does:

- start with stack $Z_0$ and $q_0$.
- $\delta(q_0, \epsilon, Z_0) = \{ \langle q_1, SZ_0 \rangle \}$
- $\langle q_1, \alpha \rangle \in \delta(q_1, \epsilon, A)$ for all $A \rightarrow \alpha$
- $\langle q_1, \epsilon \rangle \in \delta(q_1, a, a)$ for all $a \in T$. 
- $\delta(q_1, \epsilon, Z_0) = \{ \langle q_f, \epsilon \rangle \}$

(LL-PDA construction in JFLAP)
**The recognizer (4)**

**Example: Top-down recognizer**

\[ G = \langle N, T, P, S \rangle, \quad N = \{A, B\}, \quad T = \{a, b, c\} \]

\[ P = \{S \rightarrow ASB \mid AASB \mid c, A \rightarrow a, B \rightarrow b\} \]

**Input** \( w = aacb \). Calls of top-down (order is depth-first)

<table>
<thead>
<tr>
<th>Stack ( \alpha )</th>
<th>Input ( w )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. S</td>
<td>aacb</td>
</tr>
<tr>
<td>2. ASB</td>
<td>aacb</td>
</tr>
<tr>
<td>3. aSB</td>
<td>aacb</td>
</tr>
<tr>
<td>4. SB</td>
<td>acb</td>
</tr>
<tr>
<td>5. ASBB</td>
<td>acb</td>
</tr>
<tr>
<td>6. aSBB</td>
<td>acb</td>
</tr>
<tr>
<td>7. SBB</td>
<td>cb</td>
</tr>
</tbody>
</table>

4 unsuccessful predicts

<table>
<thead>
<tr>
<th>Stack ( \alpha )</th>
<th>Input ( w )</th>
</tr>
</thead>
<tbody>
<tr>
<td>9. b</td>
<td>--</td>
</tr>
<tr>
<td>10. AASB</td>
<td>aacb</td>
</tr>
<tr>
<td>11. aASB</td>
<td>aacb</td>
</tr>
<tr>
<td>12. SB</td>
<td>cb</td>
</tr>
</tbody>
</table>

4 unsuccessful predicts

<table>
<thead>
<tr>
<th>Stack ( \alpha )</th>
<th>Input ( w )</th>
</tr>
</thead>
<tbody>
<tr>
<td>13. cB</td>
<td>cb</td>
</tr>
<tr>
<td>14. B</td>
<td>b</td>
</tr>
<tr>
<td>15. b</td>
<td>b</td>
</tr>
<tr>
<td>16. --</td>
<td>--</td>
</tr>
</tbody>
</table>

Scan – predict – scan
The parser (1)

How to turn the recognizer into a parser:
Add an 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 parser (2)

Top-down parser

```python
def top-down(w, α, Γ):
    out = false
    if w = α = ϵ:
        output Γ
        out = true
    elif w = aw' and α = aα':
        out = top-down(w', α', aΓ)
    elif α = Xα' with X ∈ N:
        for X → X₁...Xₖ in P with rhs-index i:
            if top-down(w, X₁...Xₖα', ⟨X, i⟩Γ):
                out = true
    return out
```
The recognizer (3)

Example: Top-down parser

\[ G = \langle N, T, P, S \rangle, \ N = \{ A, B \}, \ T = \{ a, b, c \} \]

\[ P = \{ S \rightarrow ASB \mid AASB \mid c, A \rightarrow a, B \rightarrow b \} \]

Input \( w = aacb \). Consider only the successful predicts and scans (\( X_i \) is a notation for \( \langle X, i \rangle \)):

<table>
<thead>
<tr>
<th>stack ( \alpha )</th>
<th>( w )</th>
<th>analysis stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>aacb</td>
<td></td>
</tr>
<tr>
<td>AASB</td>
<td>aacb</td>
<td>( S_2 )</td>
</tr>
<tr>
<td>aASB</td>
<td>aacb</td>
<td>( A_1S_2 )</td>
</tr>
<tr>
<td>ASB</td>
<td>acb</td>
<td>( aA_1S_2 )</td>
</tr>
<tr>
<td>aSB</td>
<td>acb</td>
<td>( A_1aA_1S_2 )</td>
</tr>
<tr>
<td>SB</td>
<td>cb</td>
<td>( aA_1aA_1S_2 )</td>
</tr>
<tr>
<td>cB</td>
<td>cb</td>
<td>( S_3aA_1aA_1S_2)</td>
</tr>
<tr>
<td>B</td>
<td>b</td>
<td>( cS_3aA_1aA_1S_2)</td>
</tr>
<tr>
<td>b</td>
<td>b</td>
<td>( B_1cS_3aA_1aA_1S_2)</td>
</tr>
<tr>
<td>–</td>
<td>–</td>
<td>( bB_1cS_3aA_1aA_1S_2)</td>
</tr>
</tbody>
</table>

the analysis stack gives a leftmost derivation in reverse order.

Leftmost derivation:

\( S_2A_1A_1S_3B_1 \)
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 (when predicting):

\[
\text{then for all } X \rightarrow X_1 \ldots X_k:\n\text{if } |w| \geq |X_1 \ldots X_k\alpha'|\n\text{and top-down}(w, X_1 \ldots X_k\alpha')
\text{then out := true;}
\]

This check is useful even for grammars that are not left-recursive.
An example (1)

**Grammar**

\[
S \rightarrow AB \\
A \rightarrow aAB | a \\
B \rightarrow b
\]

(basic algorithm)
An example (2)

(check that word length ≥ length of sentential form)

\[
S \Rightarrow aabb ? \\
AB \Rightarrow aabb ? \\
aABB \Rightarrow aabb ? \quad aB \Rightarrow aabb ? \\
ABB \Rightarrow abb ? \quad B \Rightarrow abb ? \\
aBB \Rightarrow abb ? \quad b \Rightarrow abb ? \\
BB \Rightarrow bb ? \\
bB \Rightarrow bb ? \\
B \Rightarrow b ? \\
b \Rightarrow b ? \\
\epsilon \Rightarrow \epsilon ?
\]
Control structures (1)

In general, directional top-down parsing is non-deterministic because of multiple righthand sides for single non-terminals. Two different control strategies: You can

- either proceed **depth-first** (proceed the righthand 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) If we fail, we have to go back and try the next possibility (**backtracking**). For this, we have to reverse the operations made on the stacks.

- or proceed **breadth-first** (try all righthand 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.
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.

⇒ No perfect solution. The best option depends on the grammar, the input, the task (exhaustive parsing or not), the programming language used...
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 (one procedure $\simeq$ one production)
- 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/](https://javacc.dev.java.net/)
- ANTLR, ANother Tool for Language Recognition (generates C++, Java, Python, C#):  
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.