Compiler design

Lecture 6: Bottom-Up Parsing
(EaC§3.4)

Christophe Dubach
Winter 2023
### Top-Down Parser
A Top-Down parser builds a derivation by working from the start symbol to the input sentence.

### Bottom-Up Parser
A Bottom-Up parser builds a derivation by working from the input sentence back to the start symbol.
Example: CFG

Goal ::= a A B e
A ::= A b c
A ::= b
B ::= d

Input: abbcde

Bottom-Up Parsing

abbcde
## Example: CFG

<table>
<thead>
<tr>
<th>Production</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Goal ::= a A B e</td>
<td>Goal is the starting point</td>
</tr>
<tr>
<td>A ::= A b c</td>
<td>Production for A</td>
</tr>
<tr>
<td>A ::= b</td>
<td>Direct production for A</td>
</tr>
<tr>
<td>B ::= d</td>
<td>Production for B</td>
</tr>
</tbody>
</table>

### Input: abbcde

**Bottom-Up Parsing**

1. abbcde
2. aAbcde
3. aABcde
Bottom-Up Parsing

Example: CFG

Goal ::= a A B e
A ::= A b c
A ::= b
B ::= d

Input: abbcde

Bottom-Up Parsing

abbcde
aAbcde
aAde
Bottom-Up Parsing

Example: CFG

Goal ::= a A B e
A ::= A b c
A ::= b
B ::= d

Input: abbcde

Bottom-Up Parsing

abbcde
aAbcde
aAde
aABe
# Bottom-Up Parsing

## Example: CFG

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Goal</strong> ::=</td>
<td>a A B e</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A ::=</td>
<td>A b c</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A ::=</td>
<td>b</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B ::=</td>
<td>d</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

## Input: abbcde

## Bottom-Up Parsing

<p>| |</p>
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>abbcde</td>
</tr>
<tr>
<td>aAbcde</td>
</tr>
<tr>
<td>aAde</td>
</tr>
<tr>
<td>aABe</td>
</tr>
<tr>
<td>Goal</td>
</tr>
</tbody>
</table>
Example: CFG

Goal ::= a A B e
A ::= A b c
A ::= b
B ::= d

Input: abbcde

Bottom-Up Parsing

Note that the production follows a rightmost derivation.
Leftmost vs Rightmost derivation
# Leftmost vs Rightmost derivation

## Example: CFG

<table>
<thead>
<tr>
<th>Production</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Goal ::= a A B e</td>
<td>Start symbol</td>
</tr>
<tr>
<td>A ::= A b c</td>
<td>Non-terminal symbol</td>
</tr>
<tr>
<td>A ::= b</td>
<td>Non-terminal symbol</td>
</tr>
<tr>
<td>B ::= d</td>
<td>Non-terminal symbol</td>
</tr>
</tbody>
</table>

### Leftmost derivation

1. Goal
2. aABe
3. aAbcBe
4. abbcBe
5. abbcde

### Rightmost derivation

1. Goal
2. aABe
3. aAde
4. aAbcde
5. abbcde

## LL parsers

- Goal
- aABe
- aAbcBe
- abbcBe
- abbcde

## LR parsers

- Goal
- aABe
- aAde
- aAbcde
- abbcde
Shift-Reduce Parser
Shift-reduce parser

- It consists of a stack and the input
- It uses four actions:
  1. **shift**: next symbol is shifted onto the stack
  2. **reduce**: pop the symbols $Y_n, \ldots, Y_1$ from the stack that form the right member of a production $X ::= Y_n, \ldots, Y_1$
  3. **accept**: stop parsing and report success
  4. **error**: error reporting routine

How does the parser know when to shift or when to reduce?

Similarly to a top-down parser, could back-track if wrong decision made or look ahead to decide.

Can build a DFA to decide when we should shift or reduce (will not see it in this course).
Example: CFG

Goal ::= a A B e
A ::= A b c | b
B ::= d

Operation:

<table>
<thead>
<tr>
<th>Input</th>
<th>Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>abbcde</td>
<td></td>
</tr>
</tbody>
</table>
Example: CFG

Goal ::= a A B e
A ::= A b c | b
B ::= d

Operation: shift

Input
bbcde

Stack
a
Example: CFG

Goal ::= a A B e
A ::= A b c | b
B ::= d

Operation: shift

<table>
<thead>
<tr>
<th>Input</th>
<th>Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>bcde</td>
<td>ab</td>
</tr>
</tbody>
</table>
Shift-reduce parser

Example: CFG

Goal ::= a A B e
A ::= A b c | b
B ::= d

Operation:

<table>
<thead>
<tr>
<th>Input</th>
<th>Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>bcde</td>
<td>ab</td>
</tr>
</tbody>
</table>

Choice here: shift or reduce?
Can lookahead one symbol to make decision.
(Knowing what to do is not explain here, need to analyse the grammar, see EaC§3.5)
**Example: CFG**

<table>
<thead>
<tr>
<th></th>
<th>Goal ::= a A B e</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>A ::= A b c</td>
</tr>
<tr>
<td></td>
<td>B ::= d</td>
</tr>
</tbody>
</table>

**Operation:** reduce

**Input**

<p>| |</p>
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>bcde</td>
</tr>
</tbody>
</table>

**Stack**

<p>| |</p>
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>aA</td>
</tr>
</tbody>
</table>
Example: CFG

Goal ::= a A B e
A ::= A b c | b
B ::= d

Operation: shift

<table>
<thead>
<tr>
<th>Input</th>
<th>Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>cde</td>
<td>aAb</td>
</tr>
</tbody>
</table>
Example: CFG

Goal ::= a A B e
A ::= A b c | b
B ::= d

Operation:

<table>
<thead>
<tr>
<th>Input</th>
<th>Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>cde</td>
<td>aAb</td>
</tr>
</tbody>
</table>

Choice here: shift or reduce?

Can lookahead one symbol to make decision.
(Knowing what to do is not explain here, need to analyse the grammar, see EaC§3.5)
Example: CFG

Goal ::= a A B e
A ::= A b c | b
B ::= d

Operation: shift

<table>
<thead>
<tr>
<th>Input</th>
<th>Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>de</td>
<td>aAbc</td>
</tr>
</tbody>
</table>
Example: CFG

<table>
<thead>
<tr>
<th>Goal</th>
<th>::=</th>
<th>a A B e</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>::=</td>
<td>A b c</td>
</tr>
<tr>
<td>B</td>
<td>::=</td>
<td>d</td>
</tr>
</tbody>
</table>

Operation: reduce

<table>
<thead>
<tr>
<th>Input</th>
<th>Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>de</td>
<td>aA</td>
</tr>
</tbody>
</table>
Example: CFG

Goal ::= a A B e
A ::= A b c | b
B ::= d

Operation: shift

Input
e

Stack
aAd
### Example: CFG

Goal ::= a A B e  
A ::= A b c | b  
B ::= d  

**Operation:** reduce

<table>
<thead>
<tr>
<th>Input</th>
<th>Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>e</td>
<td>aAB</td>
</tr>
</tbody>
</table>
Shift-reduce parser

Example: CFG

Goal ::= a A B e
A ::= A b c | b
B ::= d

Operation: shift

Input

Stack
aABe
Example: CFG

Goal ::= a A B e
A ::= A b c | b
B ::= d

Operation: reduce

Input

Stack
Goal
## Top-Down vs Bottom-Up Parsing

<table>
<thead>
<tr>
<th><strong>Top-Down</strong></th>
<th><strong>Bottom-Up</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>![thumb-up] Easy to write by hand</td>
<td><strong>thumbs-up</strong> Very efficient (no recursion)</td>
</tr>
<tr>
<td>![thumb-up] Easy to integrate with the compiler</td>
<td><strong>thumbs-up</strong> Supports a larger class of grammars</td>
</tr>
<tr>
<td>![thumb-down] Supports a smaller class of grammars</td>
<td>![thumb-up] Handles left/right recursion in the grammar</td>
</tr>
<tr>
<td>⇒ cannot handle left recursion in the grammar</td>
<td>⇒ Requires generation tools</td>
</tr>
<tr>
<td>![thumb-down] Recursion might lead to performance issues</td>
<td>![thumb-down] Rigid integration to compiler</td>
</tr>
<tr>
<td>![thumb-up] Table encoding possible for better performance</td>
<td></td>
</tr>
</tbody>
</table>
## Top-Down vs Bottom-Up Parsing

### Top-Down

<table>
<thead>
<tr>
<th>Thumbs-Up</th>
<th>Thumbs-Down</th>
</tr>
</thead>
<tbody>
<tr>
<td>Easy to write by hand</td>
<td>Supports a smaller class of grammars</td>
</tr>
<tr>
<td>Easy to integrate with the compiler</td>
<td>cannot handle left recursion in the grammar</td>
</tr>
<tr>
<td></td>
<td>Recursion might lead to performance issues</td>
</tr>
<tr>
<td></td>
<td>Table encoding possible for better performance</td>
</tr>
</tbody>
</table>

### Bottom-Up

<table>
<thead>
<tr>
<th>Thumbs-Up</th>
<th>Thumbs-Down</th>
</tr>
</thead>
<tbody>
<tr>
<td>Very efficient (no recursion)</td>
<td>Harder to write by hand</td>
</tr>
<tr>
<td>Supports a larger class of grammar</td>
<td>Requires generation tools</td>
</tr>
<tr>
<td>Handles left/right recursion in the grammar</td>
<td>Rigid integration to compiler</td>
</tr>
</tbody>
</table>
Expressive Power of Grammars

Context-Free Grammars

LL(k)  LR(k)

LL(1)  LR(1)

RG
Language \neq Grammar

- A language can be defined by more than one grammar
- These grammars might be of different “complexity” (LL(1), LL(k), LR(k))
- \Rightarrow Language complexity \neq grammar complexity
Real-world examples of parser technology used

Parser generators:

• YACC: bottom up (LR)
• ANTLR: recursive descent (LL)
• JavaCC: recursive descent (LL)

C compilers

• LLVM: hand-written recursive descent parser (LL)
• GCC: started with parser generator (YACC ⇒ LR), now uses hand-written recursive descent (LL)

Java compilers

• Eclipse compiler frontend: auto-generated using Jikes Parser Generator, bottom-up (LR)
• IntelliJ compiler frontend: hand-written recursive descent (LL)
• OpenJDK compiler frontend: hand-written recursive descent (LL)

Next lecture

- Parse tree and abstract syntax tree