Compiler design

Lecture 6: Bottom-Up Parsing

Christophe Dubach

21 January 2021

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.

Bottom-Up Parser

Example: CFG

Goal ::= a A B e

A ::= A b c

A ::= b

B ::= d

Input: abbcde

Bottom-Up Parsing

abbcde

Bottom-Up Parser

Example: CFG

Goal ::= a A B e

A ::= A b c

A ::= b

B ::= d

Input: abbcde

Bottom-Up Parsing

abbcde

aAbcde

Bottom-Up Parser

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 Parser

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 Parser

Example: CFG

Goal ::= a A B e

A ::= A b c

A ::= b

B ::= d

Input: abbcde

Bottom-Up Parsing

abbcde

aAbcde

aAde

aABe

Goal

Bottom-Up Parser

Example: CFG

Goal ::= a A B e

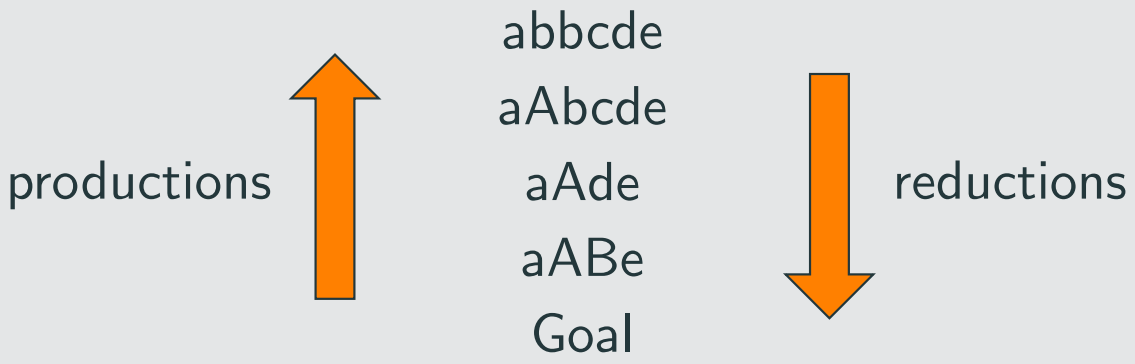
A ::= A b c

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

Goal ::= a A B e

A ::= A b c | b

B ::= d

Leftmost derivation

Goal

aABe

aAbcBe

abbcBe

abbcde

Rightmost derivation

Goal

aABe

aAde

aAbcde

abbcde

LL parsers

LR parsers

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.

Shift-reduce parser

Example: CFG

Goal ::= a A B e

A ::= A b c | b

B ::= d

Operation:

Input	Stack
abbcde	

Shift-reduce parser

Example: CFG

Goal ::= a A B e

A ::= A b c | b

B ::= d

Operation: shift

Input
bbcde

Stack
a

Shift-reduce parser

Example: CFG

Goal ::= a A B e

A ::= A b c | b

B ::= d

Operation: shift

Input	Stack
bcde	ab

Shift-reduce parser

Example: CFG

Goal ::= a A B e

A ::= A b c | b

B ::= d

Operation:

Input	Stack
bcde	ab

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)

Shift-reduce parser

Example: CFG

Goal ::= a A B e

A ::= A b c | b

B ::= d

Operation: reduce

Input	Stack
bcde	aA

Shift-reduce parser

Example: CFG

Goal ::= a A B e

A ::= A b c | b

B ::= d

Operation: shift

Input	Stack
cde	aAb

Shift-reduce parser

Example: CFG

Goal ::= a A B e

A ::= A b c | b

B ::= d

Operation:

Input	Stack
cde	aAb

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)

Shift-reduce parser

Example: CFG

Goal ::= a A B e

A ::= A b c | b

B ::= d

Operation: shift

Input	Stack
de	aAbc

Shift-reduce parser

Example: CFG

Goal ::= a A B e

A ::= A b c | b

B ::= d

Operation: reduce

Input	Stack
de	aA

Shift-reduce parser

Example: CFG

Goal ::= a A B e

A ::= A b c | b

B ::= d

Operation: shift

Input	Stack
e	aAd

Shift-reduce parser

Example: CFG

Goal ::= a A B e

A ::= A b c | b

B ::= d

Operation: reduce

Input
e

Stack
aAB

Shift-reduce parser

Example: CFG

Goal ::= a A B e

A ::= A b c | b

B ::= d

Operation: shift

Input

Stack

aABe

Shift-reduce parser

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

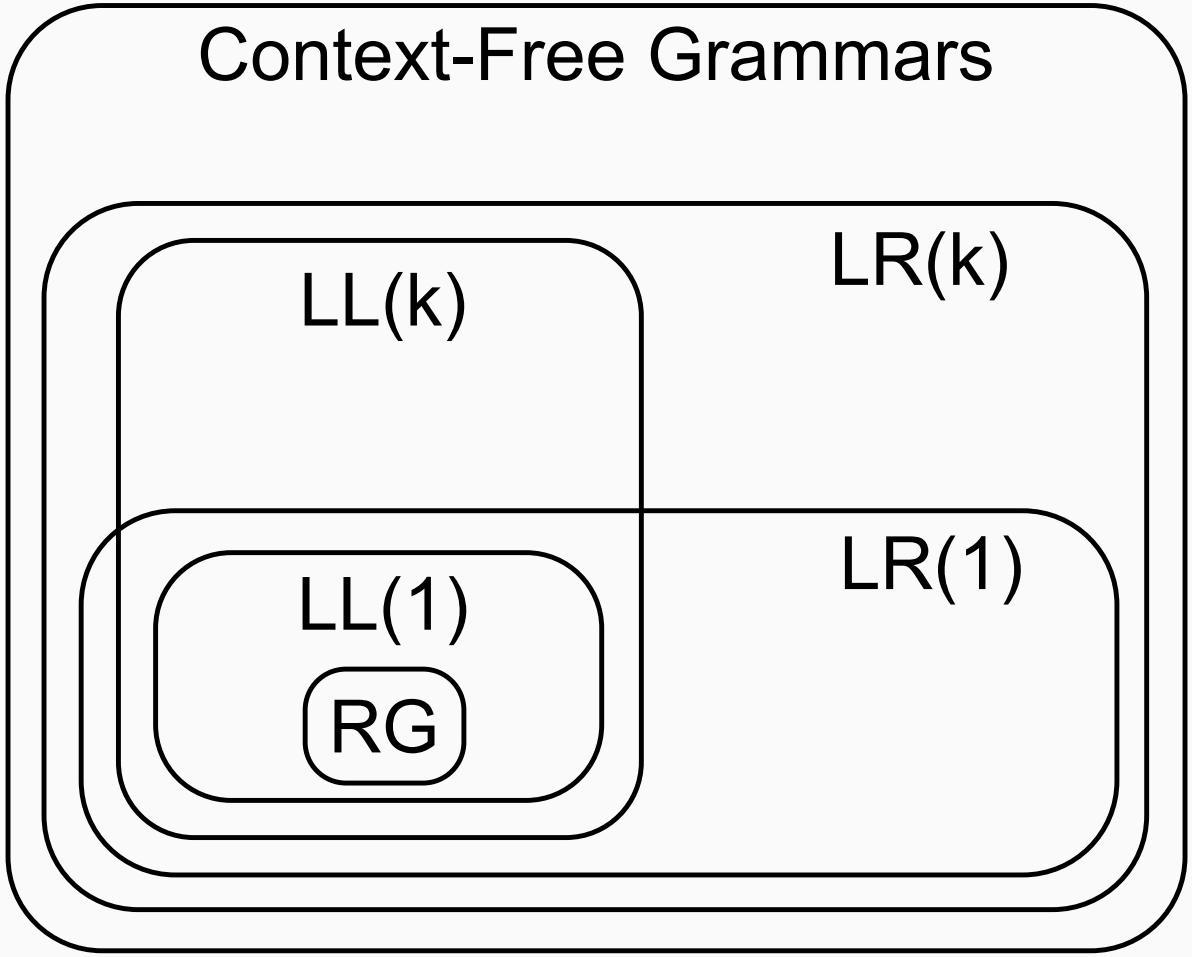
Top-Down

- 👍 Easy to write by hand
- 👍 Easy to integrate with the compiler
- 👎 Supports a smaller class of grammars
  - Cannot handle left recursion in the grammar
- 👎 Recursion might lead to performance issues
  - 👍 Table encoding possible for better performance

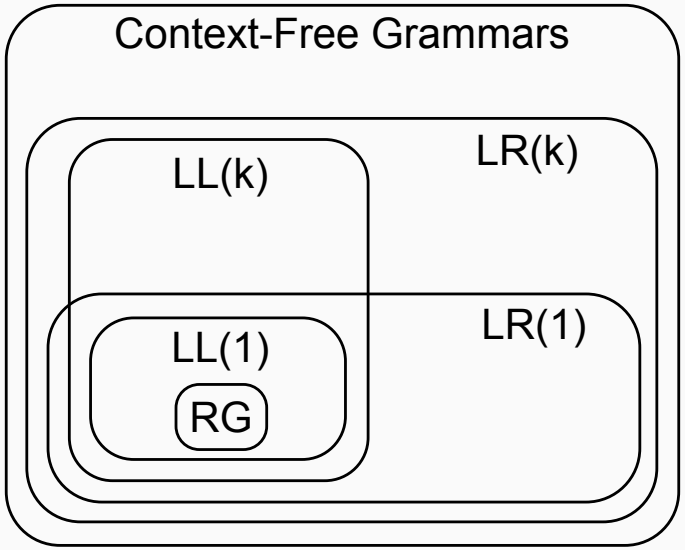
Bottom-Up

- 👍 Very efficient (no recursion)
- 👍 Supports a larger class of grammar
  - Handles left/right recursion in the grammar
- 👎 Harder to write by hand
  - $\Rightarrow$Requires generation tools
- 👎 Rigid integration to compiler

Expressive Power of Grammars



Last words



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

Next lecture

- Parse tree and abstract syntax tree