Compiler design

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

Timestamp: 2024/01/24 10:28:00

Christophe Dubach

Winter 2024

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 Parsing

Example: CFG

Goal ::= a A B e

A ::= A b c

A ::= b

B ::= d

Input: abbcde

Bottom-Up Parsing

abbcde

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

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

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 Parsing

Example: CFG

Goal ::= a A B e

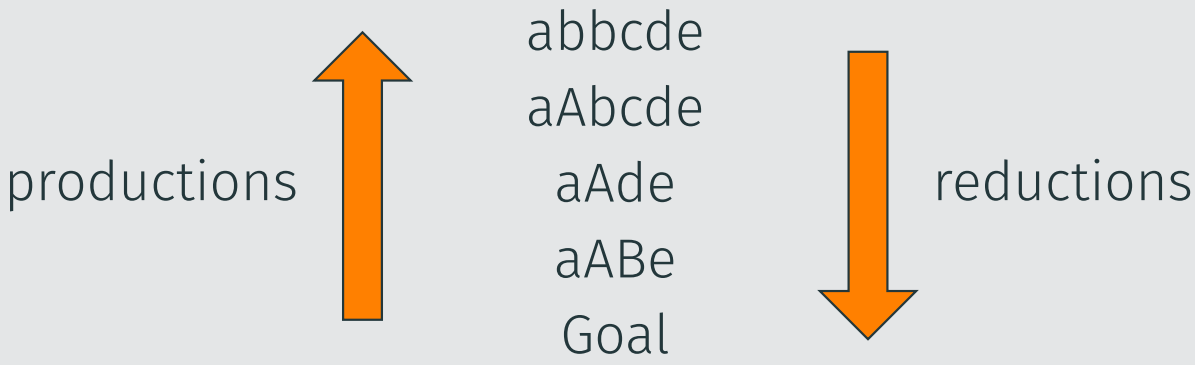
A ::= A b c

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
(will not see the construction process in this course).

The DFA recognizes handles on the stack: *i.e.* parts of inputs that can be reduced.

Handle-recognizer DFA

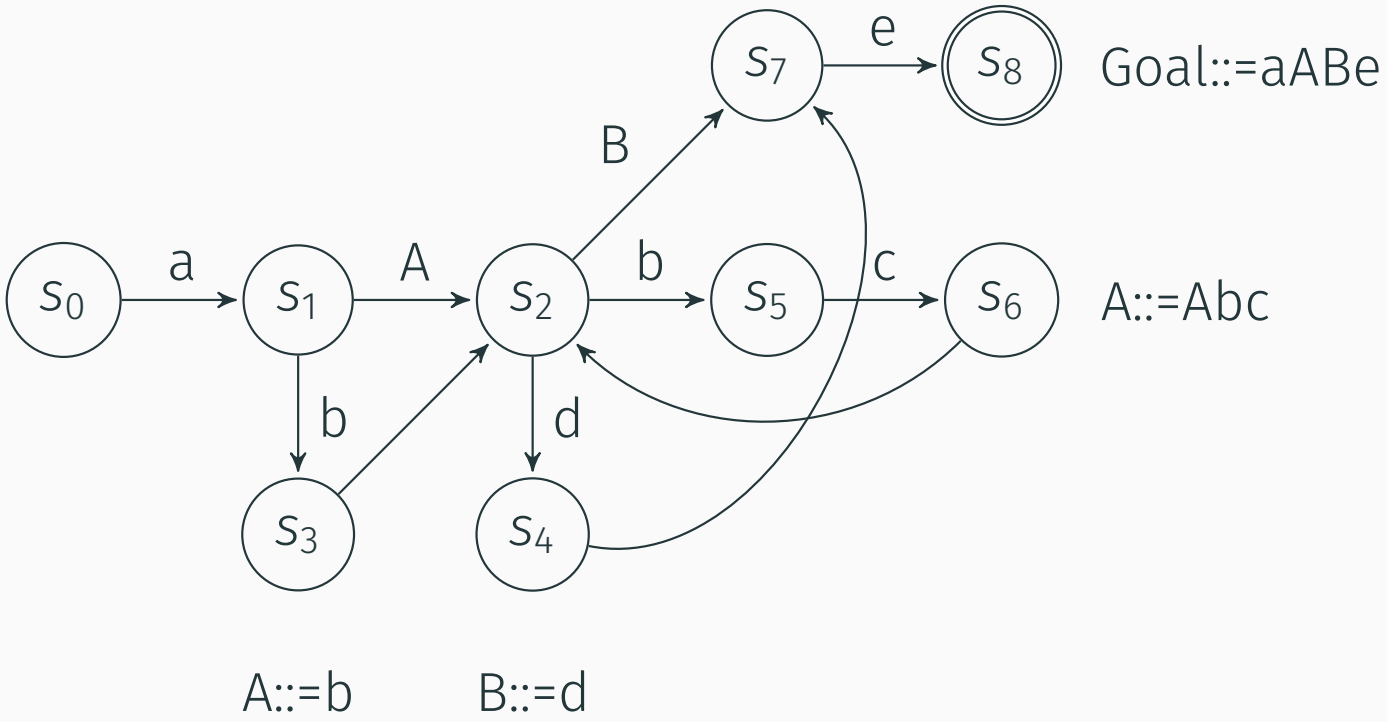
Example CFG

Goal ::= a A B e

A ::= A b c | b

B ::= d

Corresponding handle-recognizer DFA:



In states {s3,s4,s6,s8}, we reduce.

Shift-reduce parser

Example: CFG

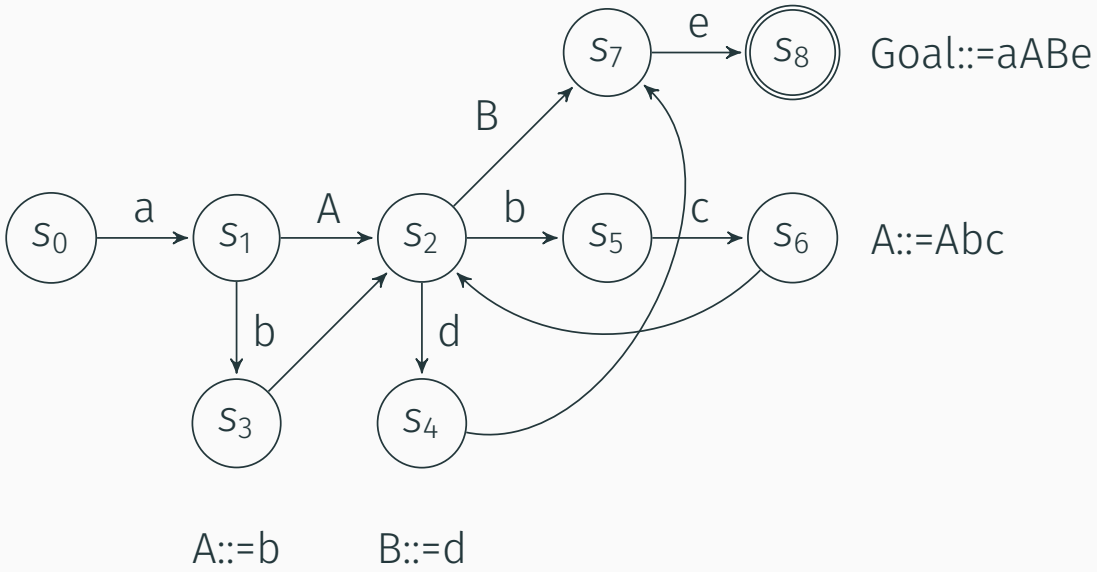
Goal ::= a A B e

A ::= A b c | b

B ::= d

Operation:

Input	Stack	State
abbcde		s0



Shift-reduce parser

Example: CFG

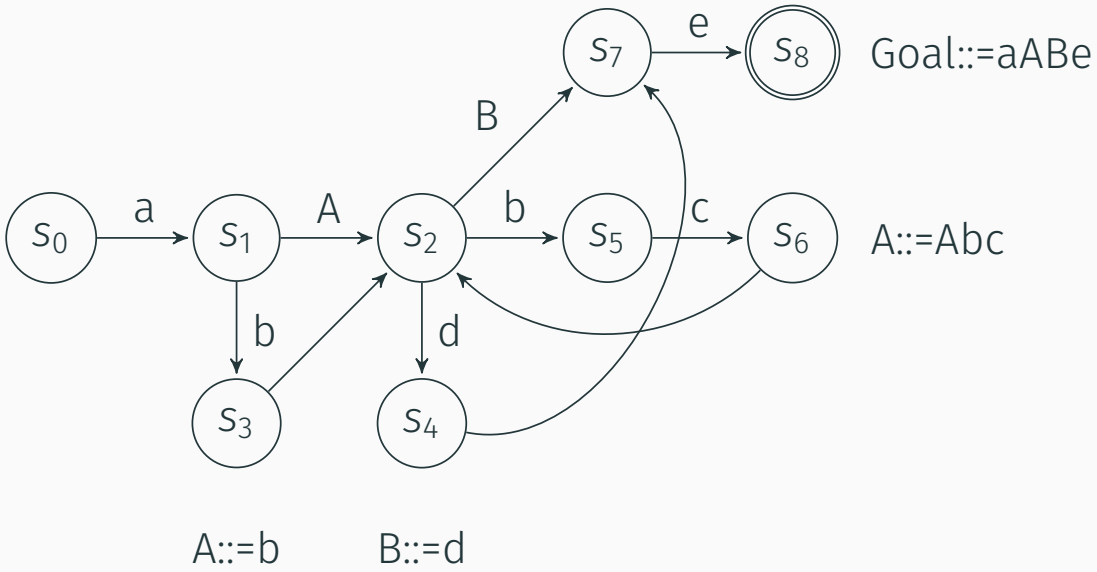
Goal ::= a A B e

A ::= A b c | b

B ::= d

Operation: shift

Input	Stack	State
bbcde	a	s1



Shift-reduce parser

Example: CFG

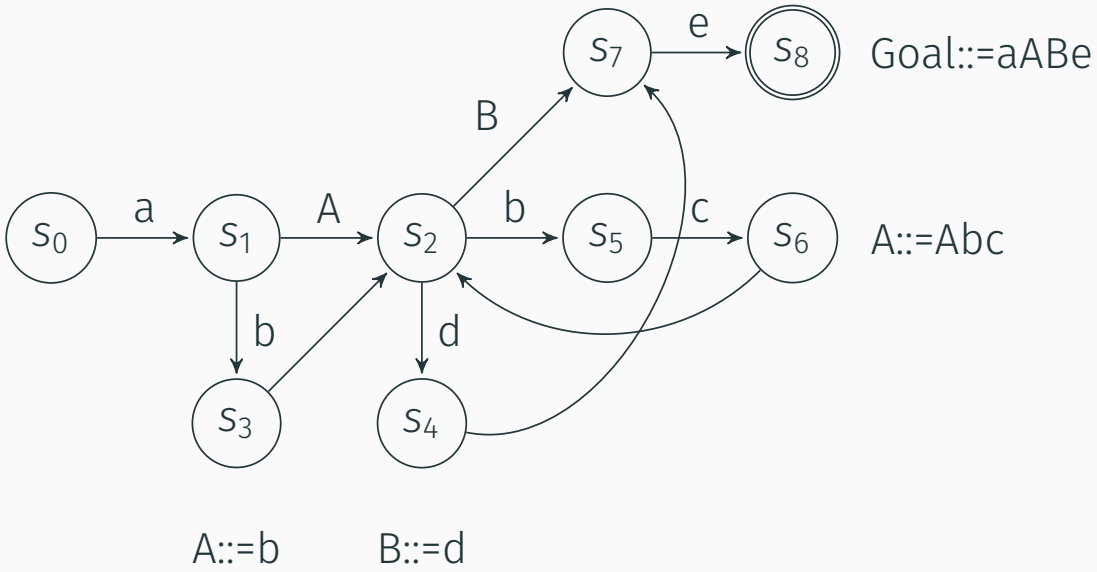
Goal ::= a A B e

A ::= A b c | b

B ::= d

Operation: shift

Input	Stack	State
bcde	ab	s3



Shift-reduce parser

Example: CFG

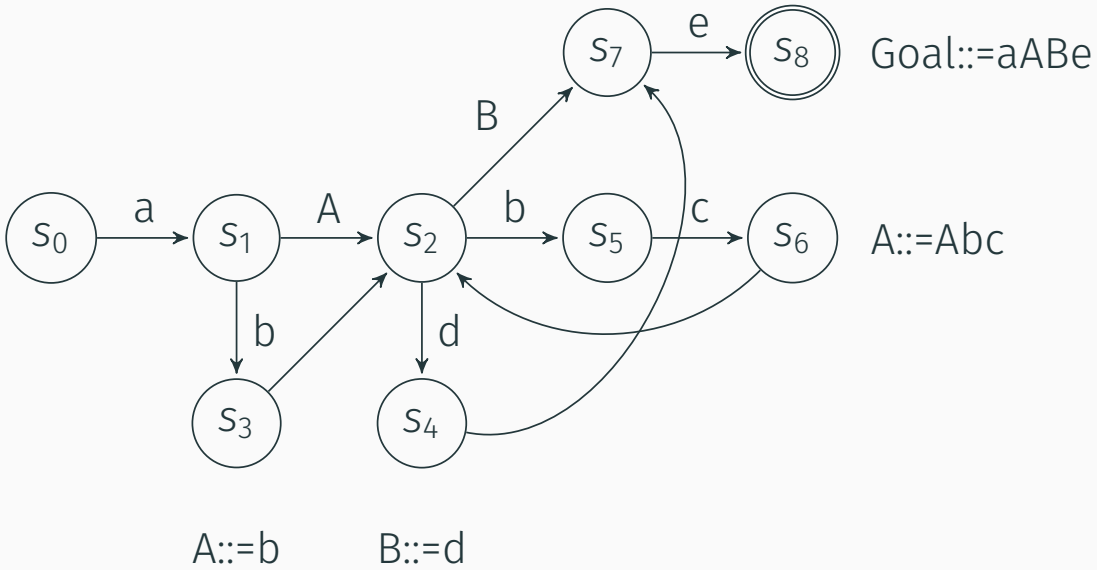
Goal ::= a A B e

A ::= A b c | b

B ::= d

Operation:

Input	Stack	State
bcde	ab	s3



Shift-reduce parser

Example: CFG

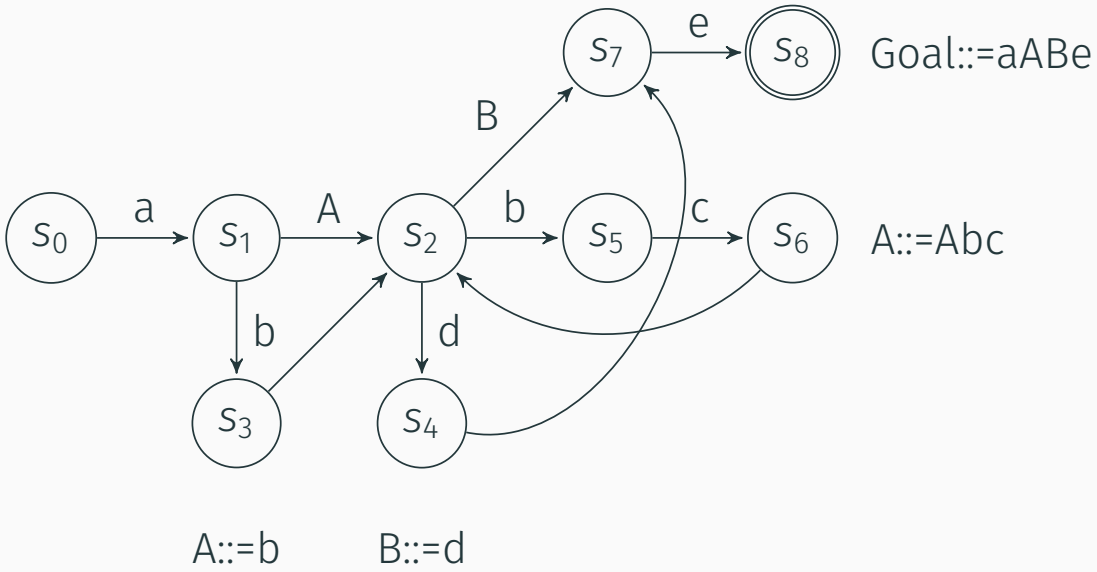
Goal ::= a A B e

A ::= A b c | b

B ::= d

Operation: reduce

Input	Stack	State
bcde	aA	s2



Shift-reduce parser

Example: CFG

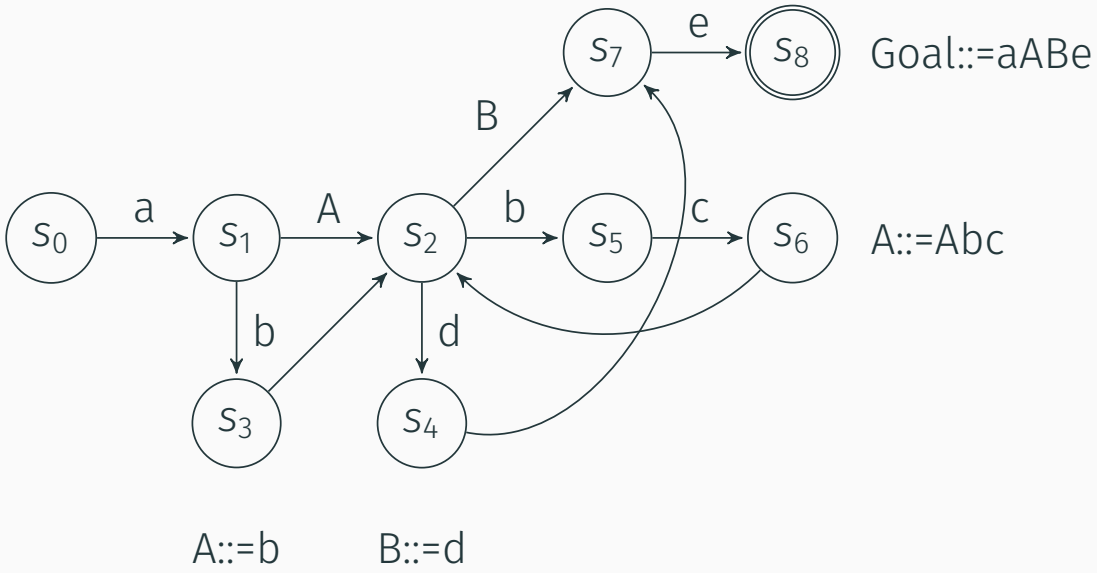
Goal ::= a A B e

A ::= A b c | b

B ::= d

Operation: shift

Input	Stack	State
cde	aAb	s5



Shift-reduce parser

Example: CFG

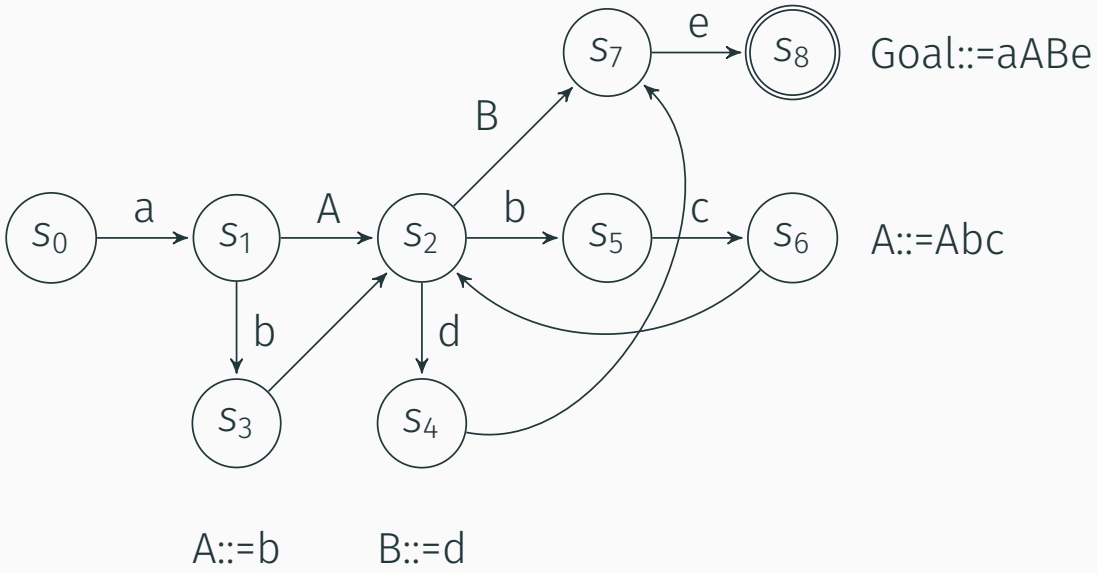
Goal ::= a A B e

A ::= A b c | b

B ::= d

Operation:

Input	Stack	State
cde	aAb	s5



Shift-reduce parser

Example: CFG

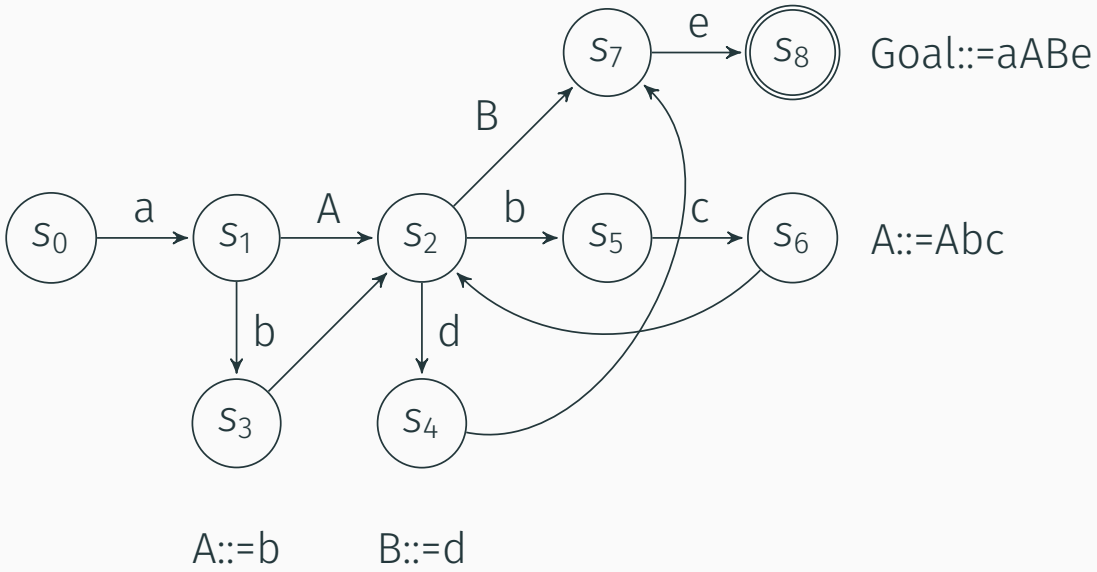
Goal ::= a A B e

A ::= A b c | b

B ::= d

Operation: shift

Input	Stack	State
de	aAbc	s6



Shift-reduce parser

Example: CFG

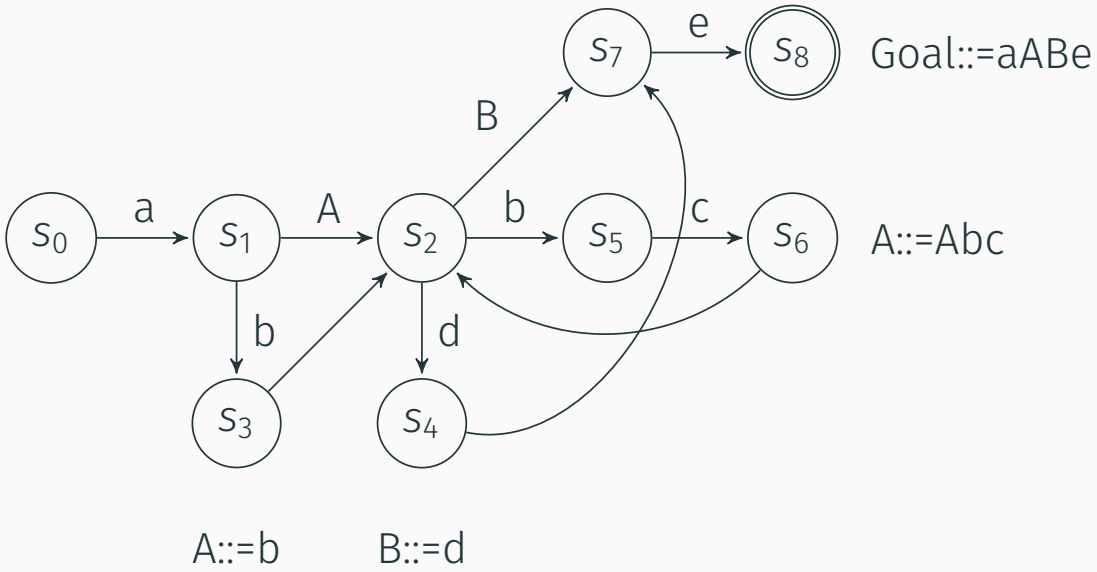
Goal ::= a A B e

A ::= A b c | b

B ::= d

Operation: reduce

Input	Stack	State
de	aA	s2



Shift-reduce parser

Example: CFG

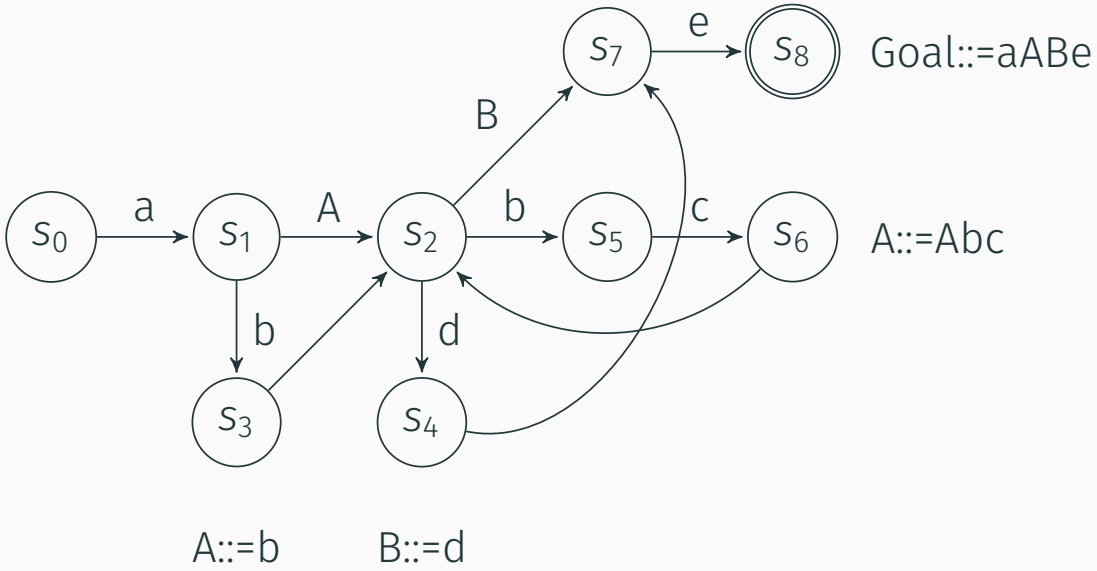
Goal ::= a A B e

A ::= A b c | b

B ::= d

Operation: shift

Input	Stack	State
e	aAd	s4



Shift-reduce parser

Example: CFG

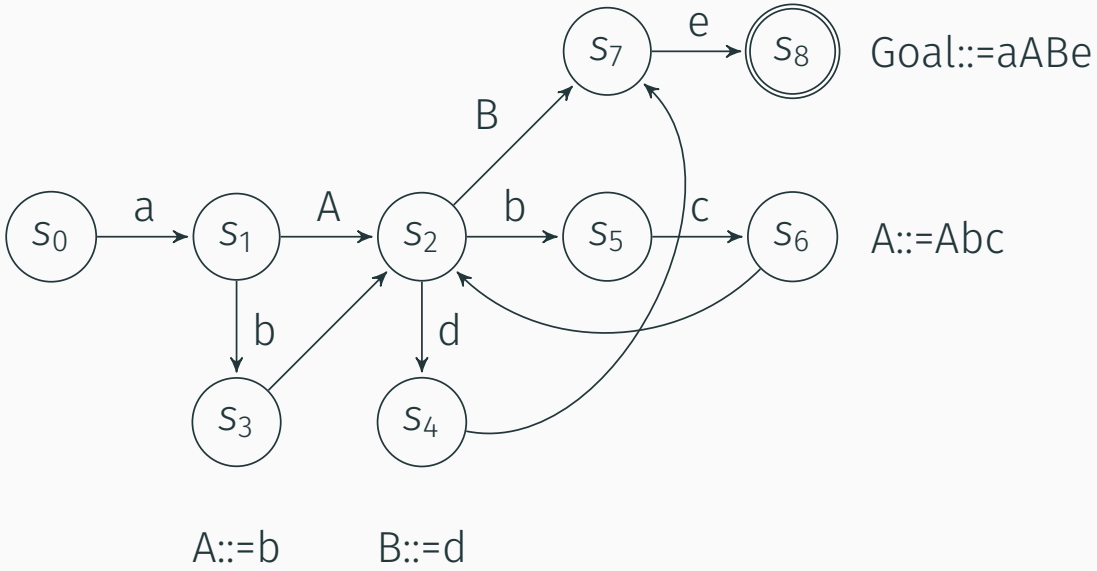
Goal ::= a A B e

A ::= A b c | b

B ::= d

Operation: reduce

Input	Stack	State
e	aAB	s7



Shift-reduce parser

Example: CFG

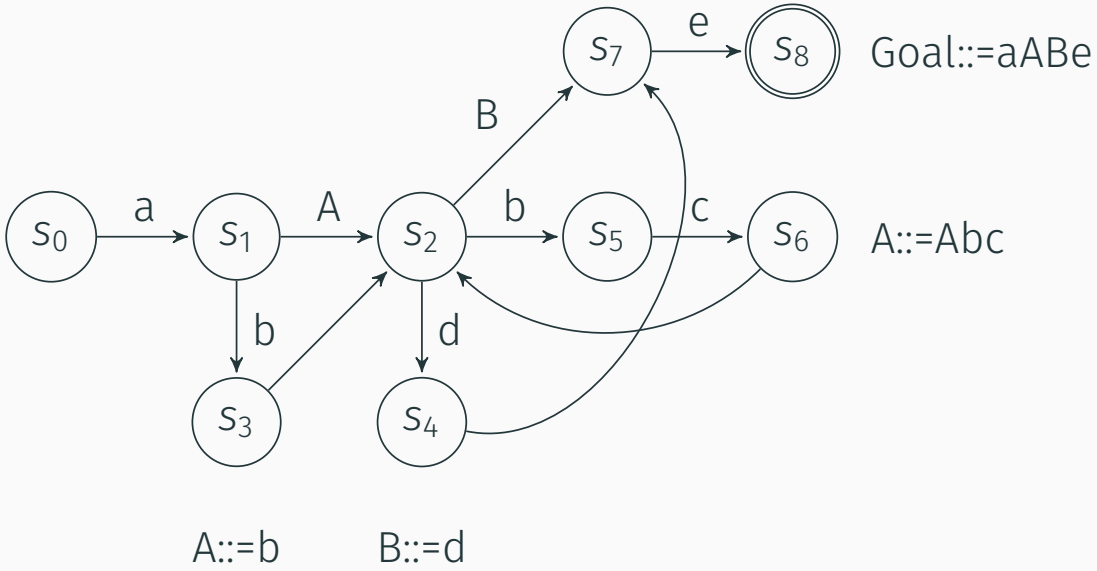
Goal ::= a A B e

A ::= A b c | b

B ::= d

Operation: shift

Input	Stack	State
	aABe	s8



Shift-reduce parser

Example: CFG

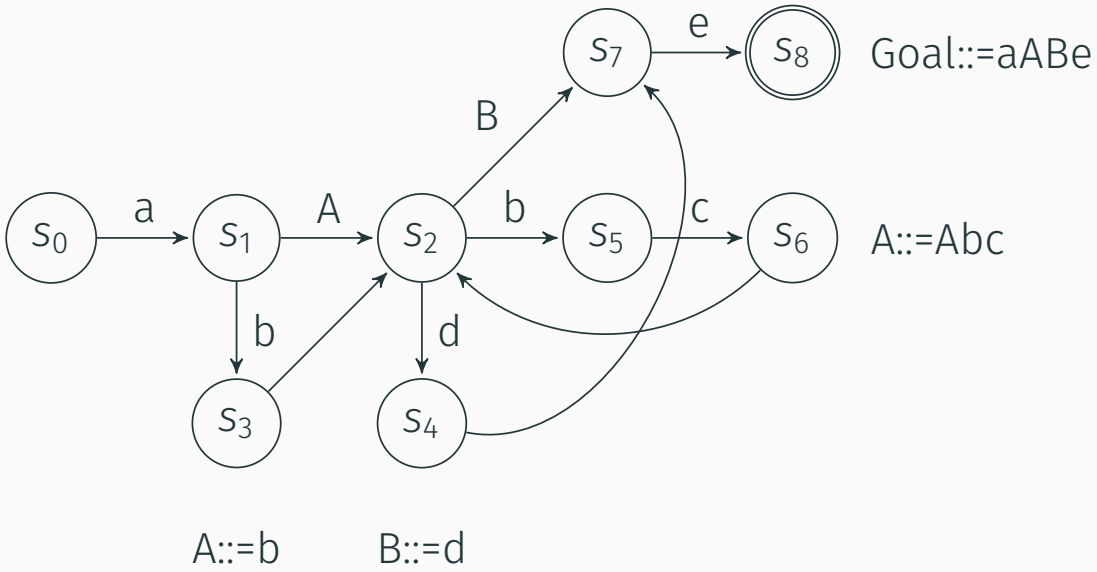
Goal ::= a A B e

A ::= A b c | b

B ::= d

Operation: reduce

Input	Stack	State
	Goal	s8



Top-Down vs Bottom-Up Parsing

Top-Down / LL parsers (*e.g.* recursrive descent parser)

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

Top-Down vs Bottom-Up Parsing

Top-Down / LL parsers (*e.g.* recursrive descent parser)

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

Bottom-Up / LR parsers (*e.g.* shift-reduce parser)

- 👍 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
- 👎 Hard to integrate in compiler

Real-world examples of parser technology used

Parser generators:

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

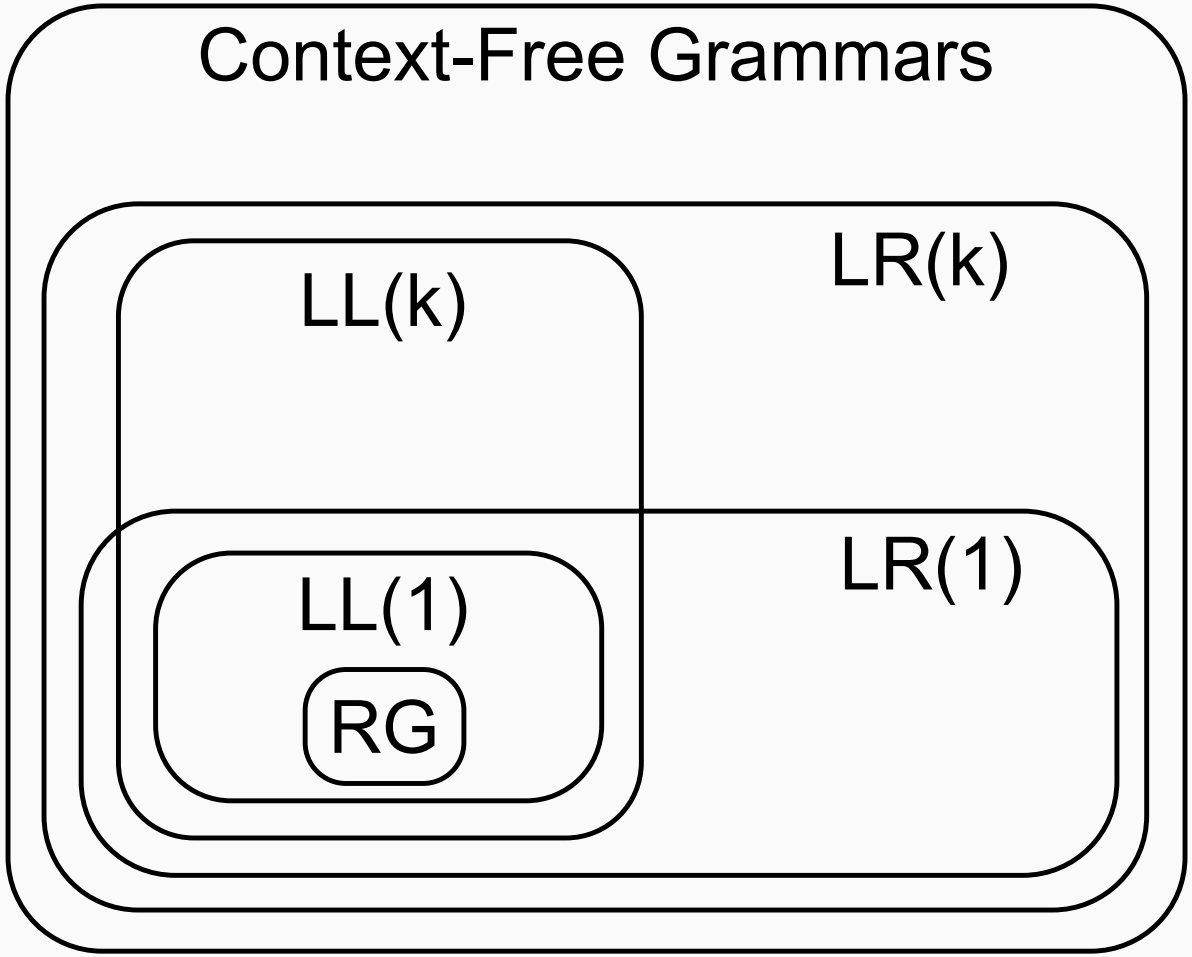
C compilers

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

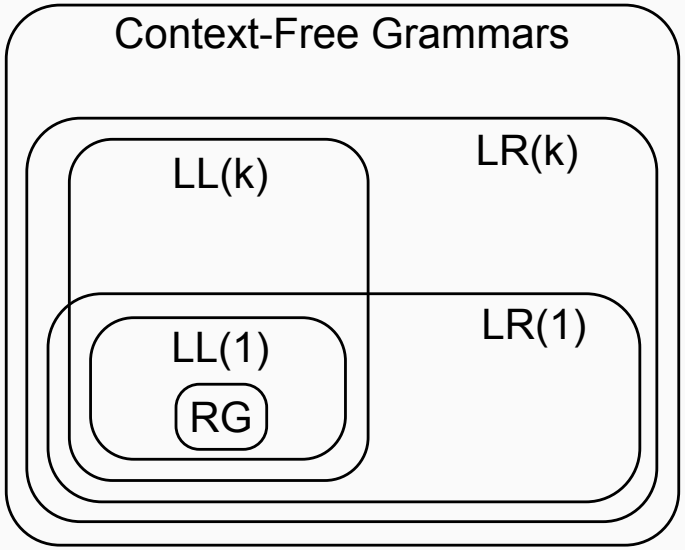
Java compilers

- 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)
  https://github.com/openjdk/jdk/blob/master/src/jdk.compiler/share/classes/com/sun/tools/javac/parser/JavacParser.java

Expressive Power of Grammars



Language vs. Grammar



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