DD1362 Programming Paradigms

Formal Languages and Syntactic Analysis
Lecture 4

Philipp Haller

April 26th, 2021

Review of Lecture 3

- Lexical analysis
- Derivations and parse trees
- Recursive descent parsing
- Eliminating ambiguity

Simplify parsing

Also builds parse tree

Today’s Lecture

- Stack automata
- Different classes of languages and grammars
- Parser generators
- Summary

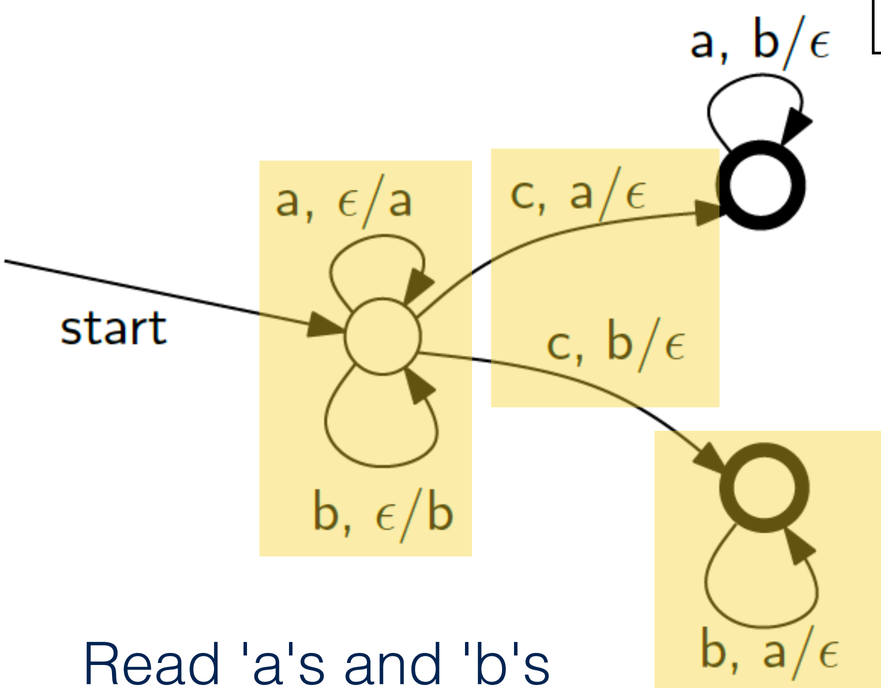
Today's Lecture

- Stack automata
- Different classes of languages and grammars
- Parser generators
- Summary

Stack Automata

A ***stack automaton*** (DPDA, Deterministic Push-Down Automaton) is like a DFA but with an ***unbounded memory*** in the form of a stack

Label "x, y/z" on edge:
Read x, pop y from stack, push z



When reading a 'c', jump to one of the accepting states depending on whether the top-most character on the stack is 'a' or 'b'

Read 'a's and 'b's and push to stack

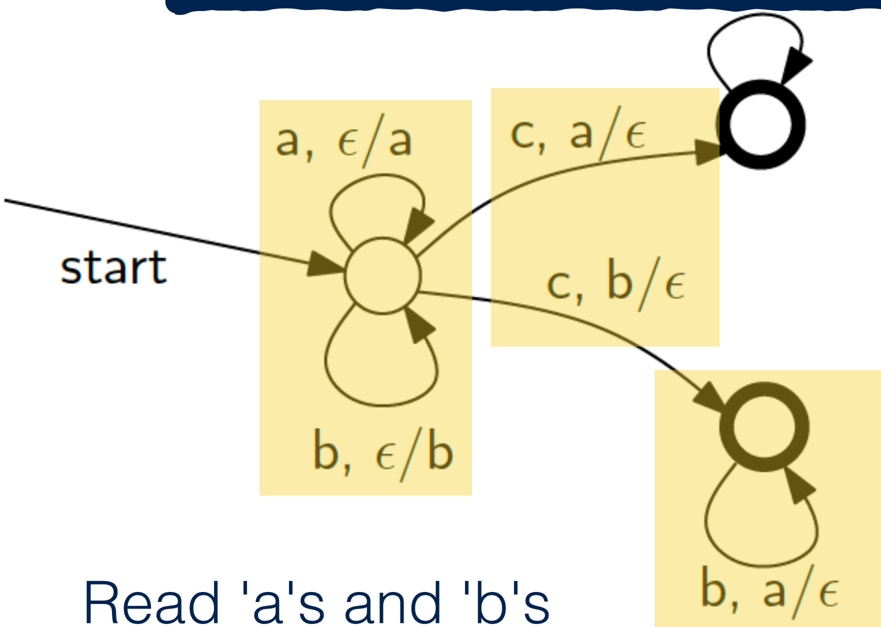
Continue to read 'b's and check that the same number of 'a's are on the stack

Stack Automata

A ***sta***
Autom ***nory***
in the

, push z

Stack automaton accepts input if it is in an accepting state *and* the stack is empty when the input is finished.



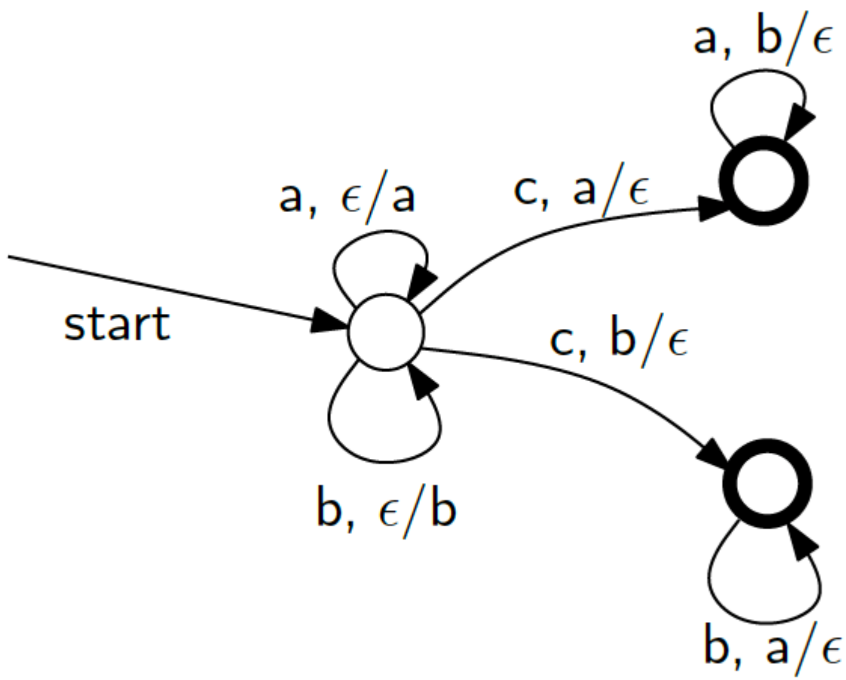
When reading a 'c', jump to one of the accepting states depending on whether the top-most character on the stack is 'a' or 'b'

Read 'a's and 'b's and push to stack

Continue to read 'b's and check that the same number of 'a's are on the stack

Example Run of Stack Automaton

Label "x, y/z" on edge:
Read x, pop y from stack, push z

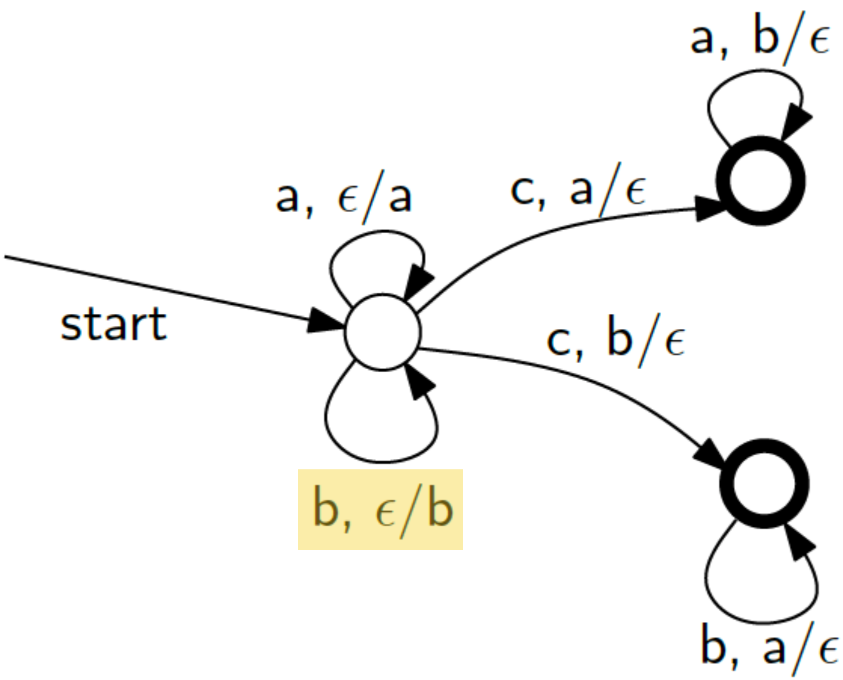


Stack:

Input:
baabcbbb

Example Run of Stack Automaton

Label "x, y/z" on edge:
Read x, pop y from stack, push z



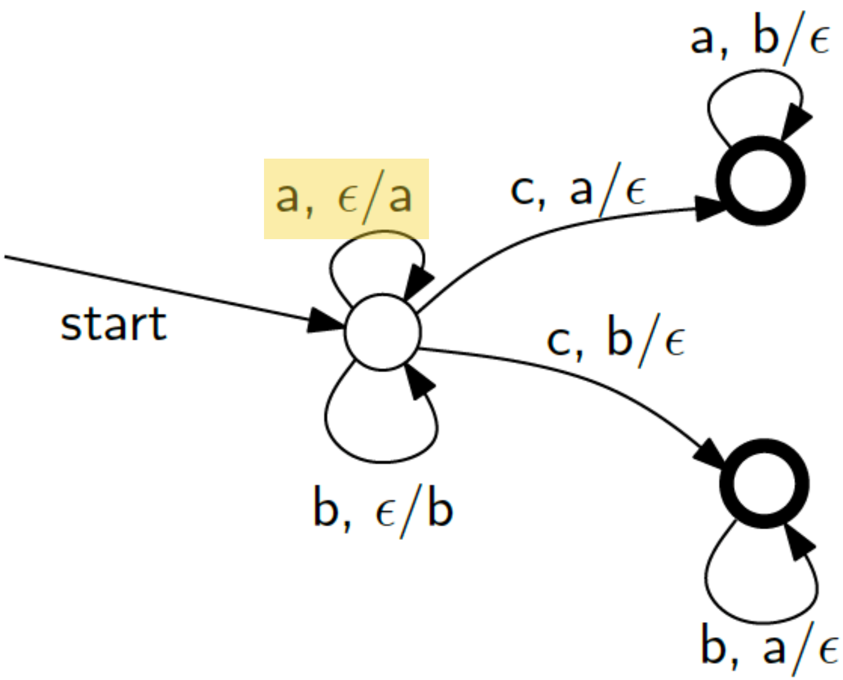
Input:
baabcbbb

Stack:

b pushed

Example Run of Stack Automaton

Label "x, y/z" on edge:
Read x, pop y from stack, push z



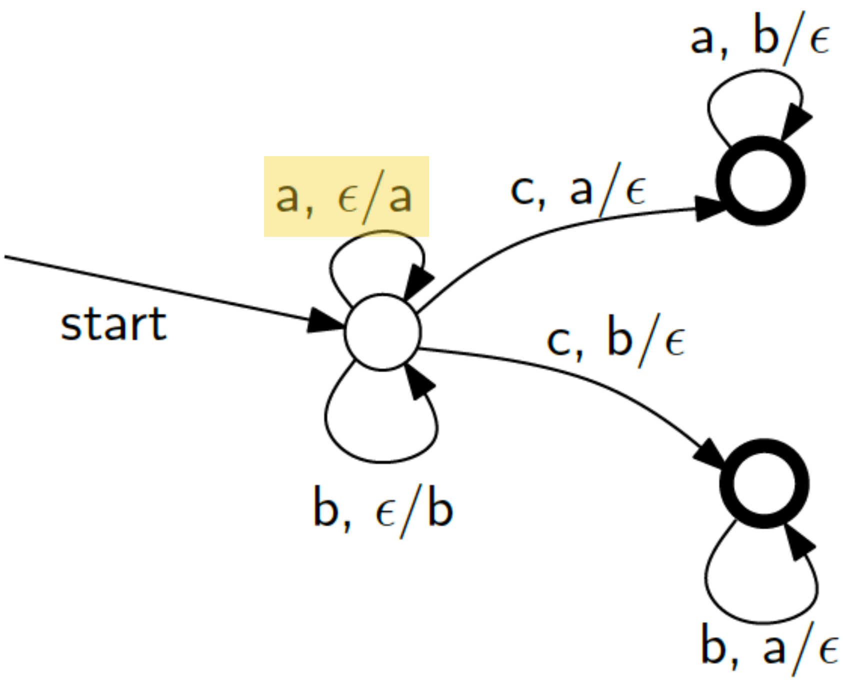
Input:
baabcbbb

Stack:

a pushed
b

Example Run of Stack Automaton

Label "x, y/z" on edge:
Read x, pop y from stack, push z



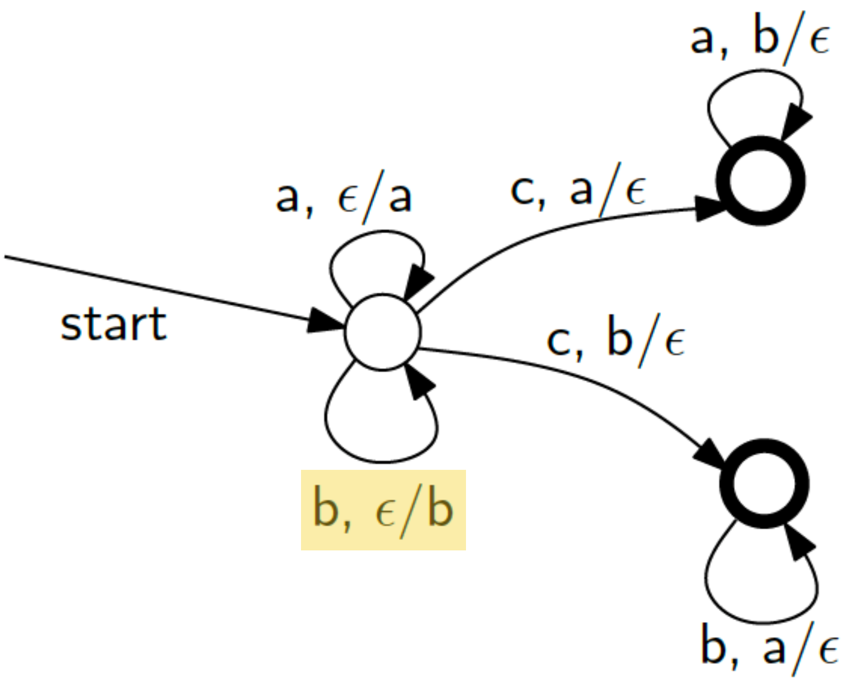
Input:
baabcbbb

Stack:

a pushed
a
b

Example Run of Stack Automaton

Label "x, y/z" on edge:
Read x, pop y from stack, push z

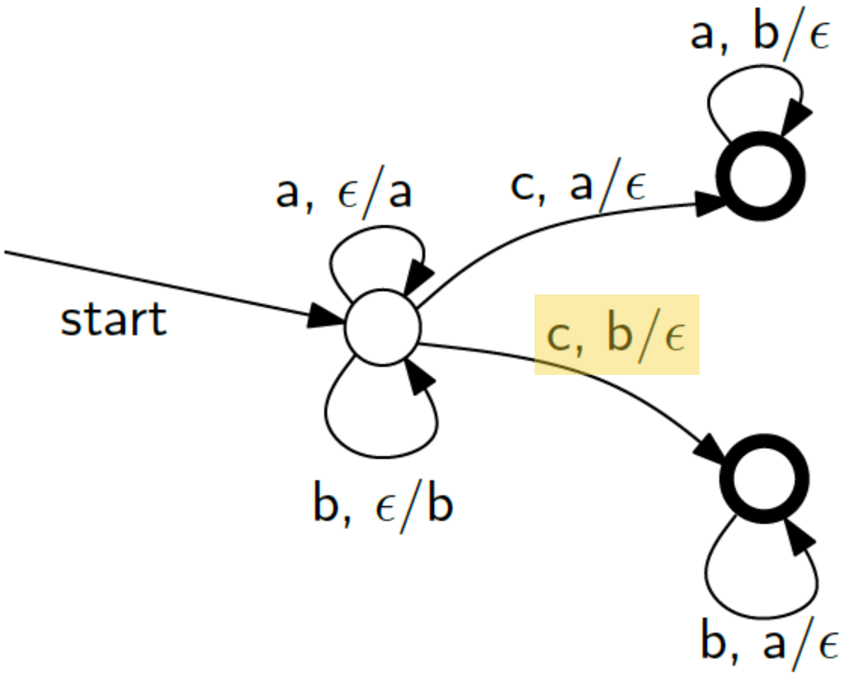


Input:
baabcbbb

Stack:
b pushed
a
a
b

Example Run of Stack Automaton

Label "x, y/z" on edge:
Read x, pop y from stack, push z

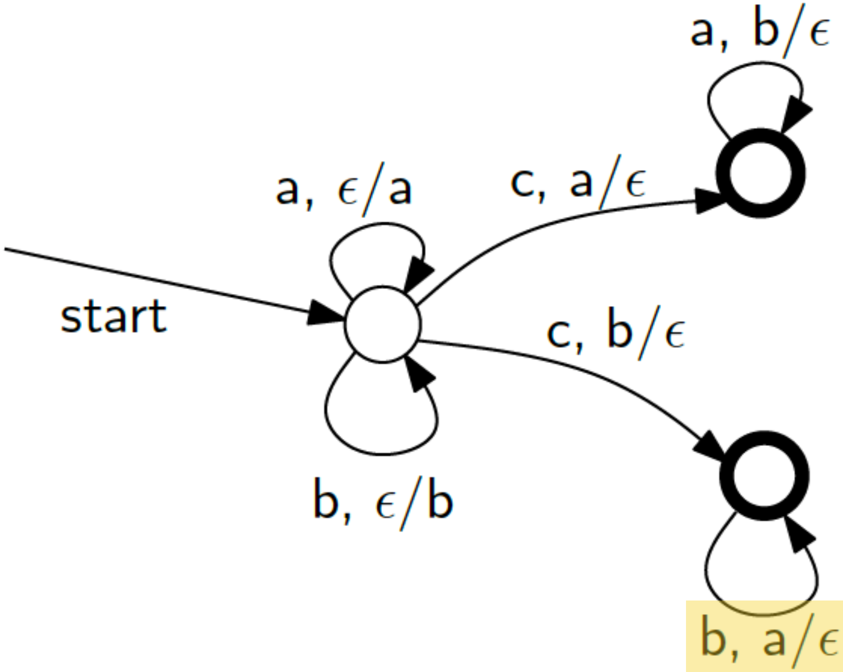


Input:
baabcbbb

Stack:
b popped
a
a
b

Example Run of Stack Automaton

Label "x, y/z" on edge:
Read x, pop y from stack, push z



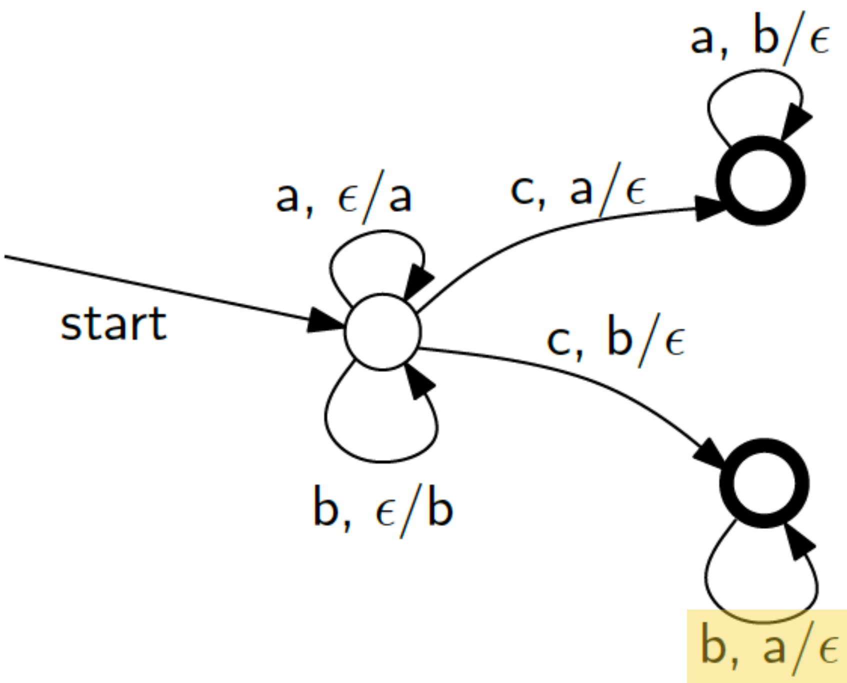
Input:
baabcbbb

Stack:

a popped
a
b

Example Run of Stack Automaton

Label "x, y/z" on edge:
Read x, pop y from stack, push z



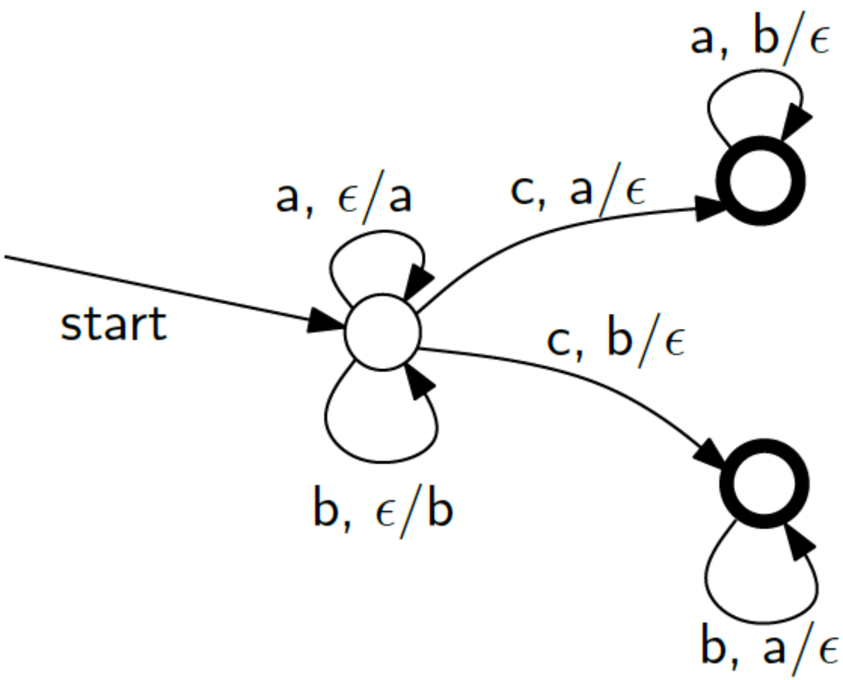
Input:
baabcbbb

Stack:

a popped
b

Example Run of Stack Automaton

Label "x, y/z" on edge:
Read x, pop y from stack, push z



Input:
baabcbbb

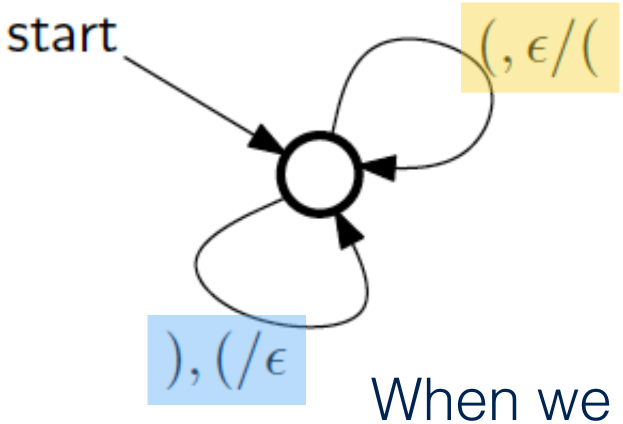
Transition missing: reading 'b' requires an 'a' on top of the stack → automaton does not accept input

Stack:
b

Stack Automaton for Strings of Balanced Parentheses

- A grammar for strings of balanced parentheses:
  Expr → ε | (Expr) Expr
- It is equally simple to construct a DPDA for this language
- The DPDA only requires a single state!

(In addition to the implicit fail-state)



When we see a left parenthesis, push left parenthesis to the stack

When we see a right parenthesis, pop a left parenthesis off the stack

From Grammars to Stack Automata

- Can we always convert a grammar to a stack automaton?
- ***No, that is not always possible!***
- ***Example:*** the set of palindromes over {a, b}
  - Simple to express as a grammar:
    Palin → ε | a | b | a Palin a | b Palin b
  - However, there is no DPDA that recognizes this language
- ***Intuition:*** when DPDA has read exactly half of the input, it would have to match the remaining input against those characters that it has already seen. However, there is no way for the automaton to know when it has read exactly half of the input.

It is possible to prove this using a "pumping-lemma" for DPDAs.

Grammars vs Stack Automata

If stack automata are not sufficiently powerful to handle all context-free grammars, ***what is their point?***

Considerations:

- Stack automata ***can be applied to a large number of grammars***, for example, even most grammars for general-purpose programming languages
- The construction of stack automata from grammars ***can be automated in most cases***

Today’s Lecture

- Stack automata
- Different classes of languages and grammars
- Parser generators
- Summary

Classes of Languages and Grammars

The grammars that we have used are so-called ***context-free grammars***.

- (There are so-called context-sensitive grammars which are more general.)

We have seen two tools for parsing context-free grammars:

- Recursive descent parsing
- Stack automata

Neither of them is sufficiently powerful to handle ***all*** context-free grammars, but they are sufficient to handle most languages that we want to write parsers for.

Classes of Languages and Grammars

The grammars that we have used are so-called ***context-free grammars***.

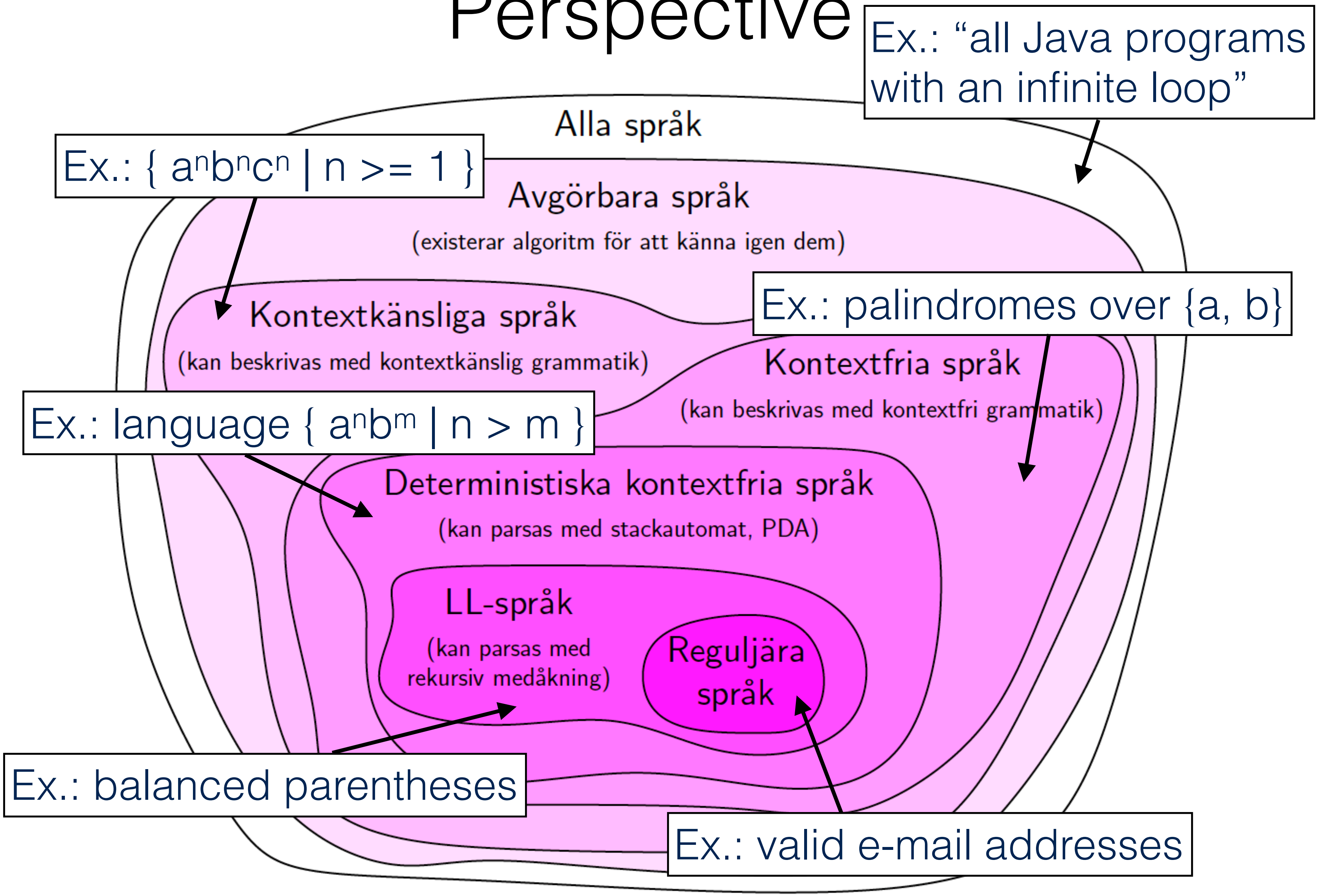
- (There are so-called context-sensitive grammars which are more general.)

We have seen two tools for parsing context-free grammars:

- Recursive descent parsing
  LL grammars → LL languages
- Stack automata
  Deterministic context-free grammars → Deterministic context-free languages

Neither of them is sufficiently powerful to handle ***all*** context-free grammars, but they are sufficient to handle most languages that we want to write parsers for.

Perspective

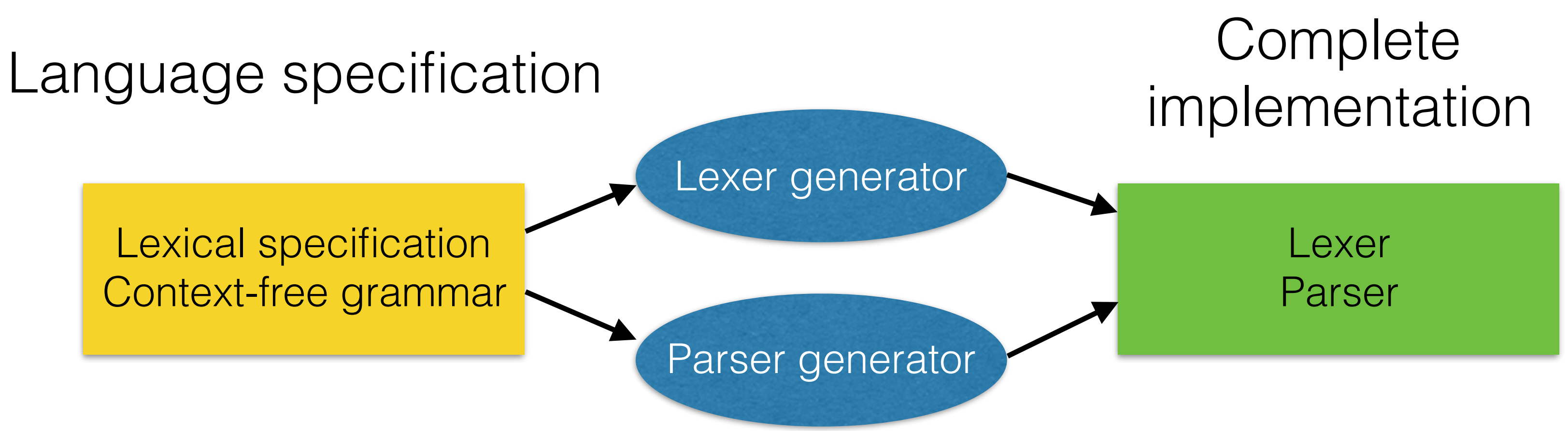


Today’s Lecture

- Stack automata
- Different classes of languages and grammars
- Parser generators
- Summary

Parser Generators

- Writing the parser for an entire programming language by hand requires ***a lot of effort***, and can ***introduce bugs***
- Constructing a parser from a grammar is often a rather ***mechanical process*** → suitable for a computer to do instead!
- A ***parser generator*** generates a lexer and a parser from the syntactic specification of a language



Examples of Parser Generators

- ***Lex/Flex*** and ***Yacc/Bison***, classic Unix tools, generate C/C++ code
  - Handle most deterministic context-free grammars, specifically, a subset called LALR grammars
- ***JFlex*** and ***Cup***, Java-based versions of Flex and Yacc
- ***Definite Clause Grammar (DCG)*** rules in Prolog, built into the language
  - Handle all context-free grammars but use backtracking in Prolog, which can be slow
- ***ANTLR*** (ANother Tool for Language Recognition), generates LL-parsers in several languages (Java, C#, C++, JavaScript, Python, Swift, Go)

Grammar for Binary Trees, Again

```
<BinTree> ::= Leaf LPar Num RPar
  | Branch LPar <BinTree> Comma <BinTree> RPar
```

Terminal symbols:

- Leaf: `"leaf"`
- Branch: `"branch"`
- Num: `[0-9]+`
- LPar, RPar, Comma: parentheses and comma

Example:

```
branch(branch(leaf(17),leaf(42)),leaf(5))
```

Result of lexical analysis:

```
Branch LPar Branch LPar Leaf LPar Num RPar Comma Leaf
  LPar Num RPar RPar Comma Leaf LPar Num RPar RPar
```

Binary Trees in Cup

File "Parser.cup":

Declare terminal symbols

Declare nonterminal symbol

Productions

```
import java_cup.runtime.*;

terminal BRANCH;
terminal LEAF;
terminal LPAREN;
terminal RPAREN;
terminal COMMA;
terminal Integer NUM;

non terminal ParseTree BinTree;


BinTree ::= LEAF LPAREN NUM:t RPAREN {: RESULT = new LeafNode(t); :}
         | BRANCH LPAREN BinTree:left COMMA BinTree:right RPAREN {:
RESULT = new BranchNode(left, right); :}
         ;
```

Run Cup with "java -jar java-cup-11b.jar Parser.cup", generates "parser.java" and "sym.java"

Lexical analysis for binary trees in JFlex

File "Lexer.lex":

```
import java.lang.System;
import java_cup.runtime.Symbol;

%%
%cup
%class Lexer

%%

branch { return new Symbol(sym.BRANCH); }
leaf { return new Symbol(sym.LEAF); }
"(" { return new Symbol(sym.LPAREN); }
")" { return new Symbol(sym.RPAREN); }
, { return new Symbol(sym.COMMA); }
[0-9]+ { return new Symbol(sym.NUM, new Integer(yytext())); }
[ \t\n] { }
```

Terminal symbols from "Parser.cup"

Run with "jflex Lexer.lex", generates "Lexer.java"
Complete example with Main class on course website

Today's Lecture

- Stack automata
- Different classes of languages and grammars
- Parser generators
- Summary

Summary

- ***Formal languages***
  - Language classes (ex.: regular languages)
  - Formal language descriptions (ex. automata, regular expressions, grammars)
- ***Regular languages***
  - Equivalence of regular expressions and DFAs
  - Limitations: languages that are not regular
- ***Context-free languages***
  - Context-free grammars
  - Derivations, parse trees
  - Ambiguity
  - Stack automata, DPDA
- ***Lexical analysis***
- ***Recursive descent parsing***

Abbreviations

DFA Deterministic Finite Automaton

The simplest kind of automaton, same expressive power as regular expressions

DPDA Deterministic Push-Down Automaton, Stack Automaton

Like a DFA but with an unbounded stack as memory

LL Left-to-right, Leftmost-derivation

- LL-parser: read input from left to right and expand nonterminal symbols from left to right
- LL-grammar: grammar that can be parsed using an LL-parser

Follow-up Courses

- DD2350 Algorithms, Data Structures and Complexity
  More on language hierarchies/complexity
- DD2372 Automata and Languages
  More in-depth automata theory
- DD2481 Principles of Programming Languages
  Formal semantics, type systems, soundness, verification
- DD2488 Compiler Construction
  Write a complete compiler from scratch!
- ID2202 Compilers and Execution Environments
  Techniques for implementing programming languages

Good luck in the KS!