CS21
Decidability and Tractability

Lecture 5

January 17, 2018

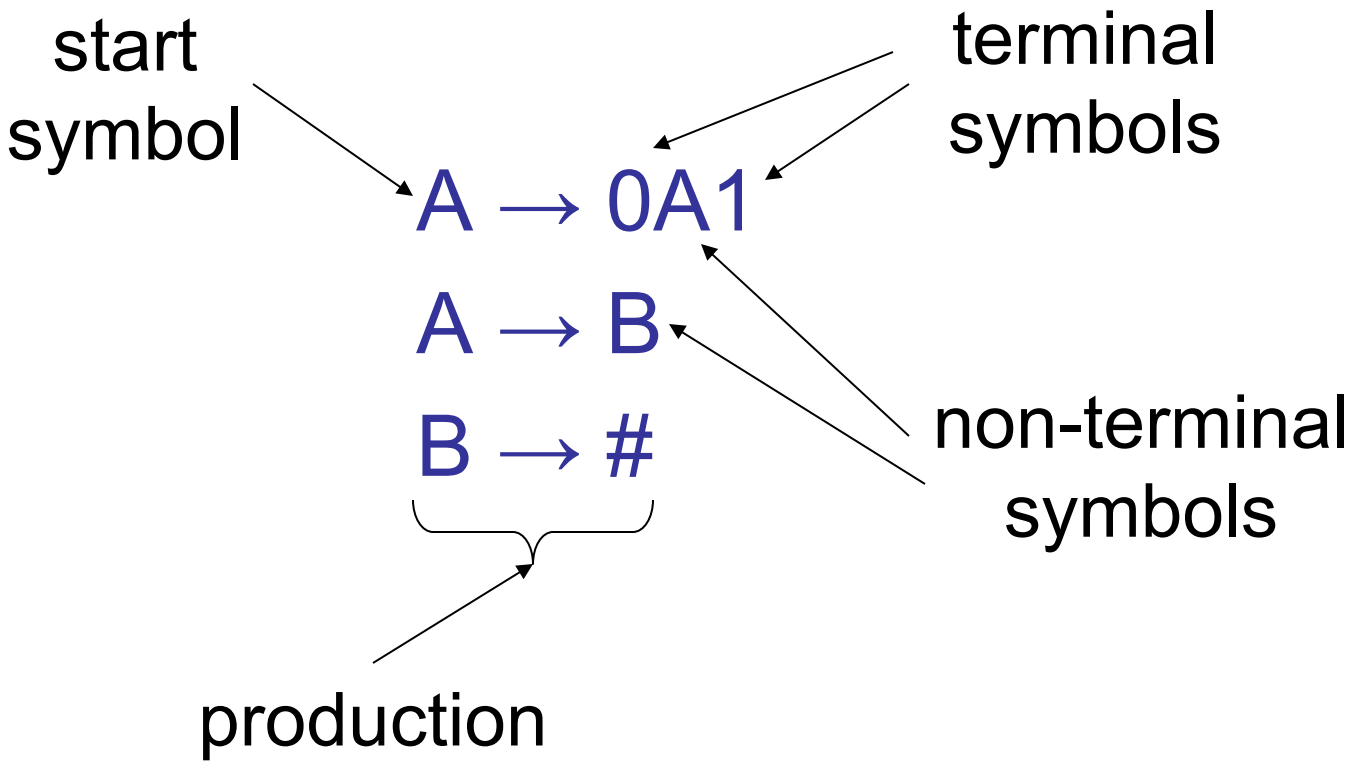
Outline

- Context-Free Grammars and Languages
- equivalence of NPDAs and CFGs
- non context-free languages

Context-free grammars and languages

- languages recognized by a (N)FA are exactly the languages described by regular expressions, and they are called the regular languages
- languages recognized by a NPDA are exactly the languages described by context-free grammars, and they are called the context-free languages

Context-Free Grammars



Context-Free Grammars

- generate strings by repeated replacement of non-terminals with string of terminals and non-terminals
  - write down start symbol (non-terminal)
  - replace a non-terminal with the right-hand-side of a rule that has that non-terminal as its left-hand-side.
  - repeat above until no more non-terminals

Context-Free Grammars

Example:

$A \Rightarrow 0A1 \Rightarrow 00A11 \Rightarrow$
$000A111 \Rightarrow 000B111 \Rightarrow$
$000\#111$

$A \rightarrow 0A1$
$A \rightarrow B$
$B \rightarrow \#$

- a derivation of the string 000#111
- set of all strings generated in this way is the language of the grammar L(G)
- called a Context-Free Language

Context-Free Grammars

- Natural languages (e.g. English) have structure:

<sentence> → <noun-phrase><verb-phrase>
<noun-phrase> → <cpx-noun> | <cpx-noun><prep-phrase>
<verb-phrase> → <cpx-verb> | <cpx-verb><prep-phrase>
<prep-phrase> → <prep><cpx-noun>
<cpx-noun> → <article><noun>
<cpx-verb> → <verb>|<verb><noun-phrase>
<article> → a | the
<noun> → dog | cat | flower
<verb> → eats | sees
<prep> → with

shorthand for multiple rules with same lhs

Generate a string in the language of this grammar.

Context-Free Grammars

- CFGs don't capture natural languages completely

- computer languages often defined by CFG
  - hierarchical structure
  - slightly different notation often used "Backus-Naur form"
  - see next slide for example

Example CFG

<stmt> → <if-stmt> | <while-stmt> | <begin-stmt>
| <asgn-stmt>
<if-stmt> → IF <bool-expr> THEN <stmt> ELSE <stmt>
<while-stmt> → WHILE <bool-expr> DO <stmt>
<begin-stmt> → BEGIN <stmt-list> END
<stmt-list> → <stmt> | <stmt>; <stmt-list>
<asgn-stmt> → <var> := <arith-expr>
<bool-expr> → <arith-expr><compare-op><arith-expr>
<compare-op> → < | > | ≤ | ≥ | =
<arith-expr> → <var> | <const>
| (<arith-expr><arith-op><arith-expr>)
<arith-op> → + | - | * | /
<const> → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<var> → a | b | c | … | x | y | z

CFG formal definition

- A context-free grammar is a 4-tuple

$$(V, \Sigma, R, S)$$

where

  - V is a finite set called the non-terminals
  - Σ is a finite set (disjoint from V) called the terminals
  - R is a finite set of productions where each production is a non-terminal and a string of terminals and non-terminals.
  - $S \in V$ is the start variable (or start non-terminal)

CFG formal definition

- u, v, w are strings of non-terminals and terminals, and $A \rightarrow w$ is a production:

  "uAv yields uwv" notation: $uAv \Rightarrow uwv$

  also: "yields in 1 step" notation: $uAv \Rightarrow^1 uwv$

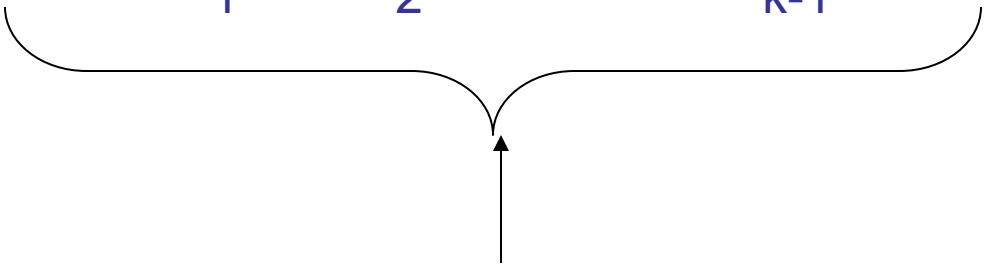
- in general:

  "yields in k steps" notation: $u \Rightarrow^k v$

  – meaning: there exists strings $u_1, u_2, \ldots u_{k-1}$ for which $u \Rightarrow u_1 \Rightarrow u_2 \Rightarrow \ldots \Rightarrow u_{k-1} \Rightarrow v$

CFG formal definition

- notation: $u \Rightarrow^* v$
  - meaning: $\exists\ k \geq 0$ and strings $u_1,\ldots,u_{k-1}$ for which $u \Rightarrow u_1 \Rightarrow u_2 \Rightarrow \ldots \Rightarrow u_{k-1} \Rightarrow v$



- if u = start symbol, this is a derivation of v
- The language of G, denoted L(G) is:

$$\{w \in \Sigma^* : S \Rightarrow^* w\}$$

CFG example

- Balanced parentheses:
  - ( )
  - (( ) (( ) ( ))))
- a string w in Σ* = { (,) }* is balanced iff:
  - # "("s equals # ")"s, and
  - for any prefix of w, # "("s ≥ # ")"s

Exercise: design a CFG for balanced parentheses.

CFG example

- Arithmetic expressions over {+,*,(,),a}
  - (a + a) * a
  - a * a + a + a + a + a
- A CFG generating this language:

<expr> → <expr> * <expr>

<expr> → <expr> + <expr>

<expr> → (<expr>) | a

CFG example

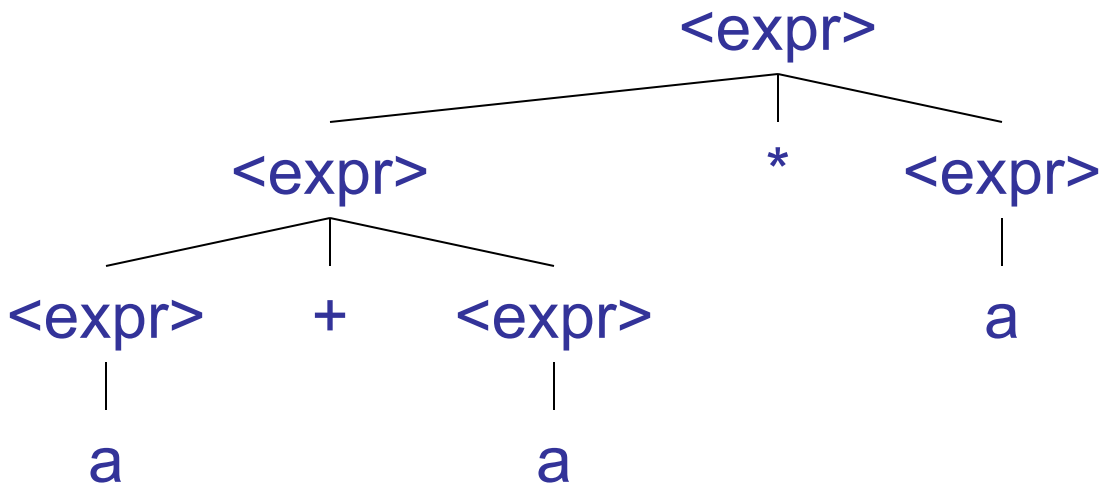
<expr> → <expr> * <expr>
<expr> → <expr> + <expr>
<expr> → (<expr>) | a

- A derivation of the string: a+a*a

<expr> ⇒ <expr> * <expr>
⇒ <expr> + <expr> * <expr>
⇒ a + <expr> * <expr>
⇒ a + a * <expr>
⇒ a + a * a

Parse Trees

- Easier way to picture derivation: parse tree



- grammar encodes grouping information; this is captured in the parse tree.

CFGs and parse trees

<expr> → <expr> * <expr>
<expr> → <expr> + <expr>
<expr> → (<expr>) | a

- Is this a good grammar for arithmetic expressions?
  - can group wrong way (+ precedence over *)

Solution to problem

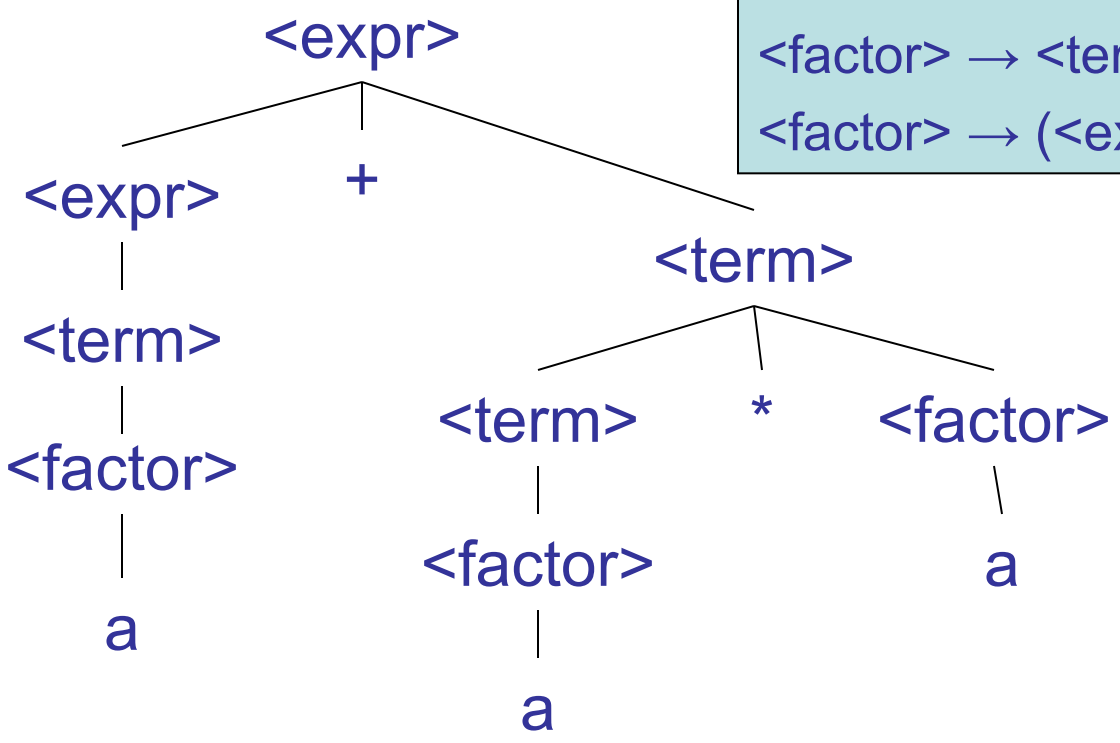
<expr> → <expr> + <term> | <term>
<term> → <term> * <factor> | <factor>
<factor> → <term> * <factor>
<factor> → (<expr>) | a

- forces correct precedence in parse tree grouping
  - within parentheses, * cannot occur as ancestor of + in the parse tree.

Parse Trees

- parse tree for a + a * a in new grammar:

<expr> → <expr> + <term> | <term>
<term> → <term> * <factor> | <factor>
<factor> → <term> * <factor>
<factor> → (<expr>) | a



Some facts about CFLs

- CFLs are closed under
  - union (proof?)
  - concatenation (proof?)
  - star (proof?)
- Every regular language is a CFL
  - proof?

NPDA, CFG equivalence

Theorem: a language L is recognized by a NPDA iff L is described by a CFG.

Must prove *two* directions:

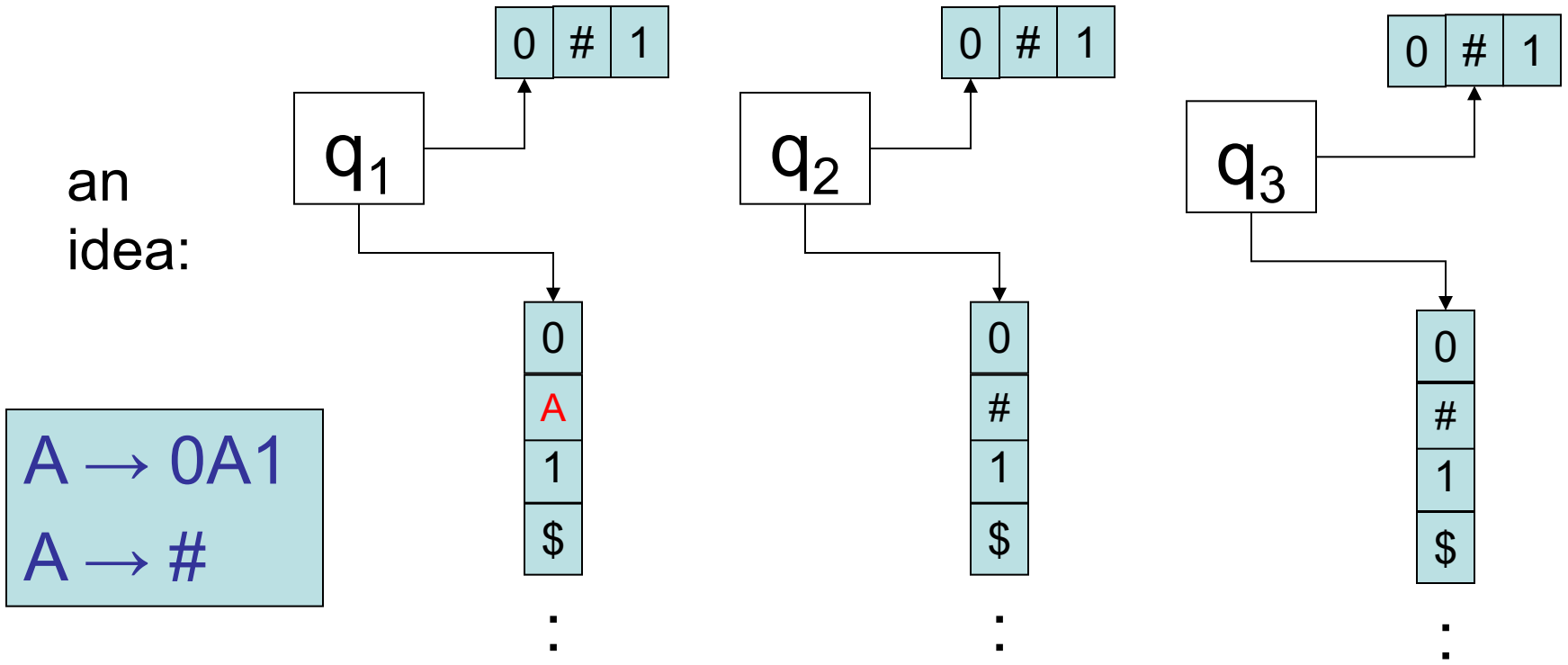
($\Rightarrow$) L is recognized by a NPDA implies L is described by a CFG.

($\Leftarrow$) L is described by a CFG implies L is recognized by a NPDA.

NPDA, CFG equivalence

Proof of (⇐): L is described by a CFG implies L is recognized by a NPDA.

an idea:



NPDA, CFG equivalence

1. we'd like to non-deterministically guess the derivation, forming it on the stack
2. then scan the input, popping matching symbol off the stack at each step
3. accept if we get to the bottom of the stack at the end of the input.

what is wrong with this approach?