# CS21
# Decidability and Tractability

Lecture 5

January 15, 2014



# Outline

- Pushdown Automata
- Context-Free Grammars and Languages
  - parse trees
  - ambiguity
  - normal form
- equivalence of NPDAs and CFGs

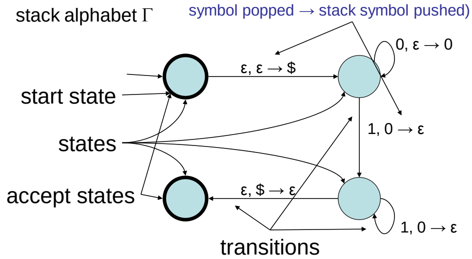


# NPDA diagram

tape alphabet $\Sigma$

stack alphabet $\Gamma$

transition label: (tape symbol read, stack symbol popped $\rightarrow$ stack symbol pushed)






# NPDA operation

- Taking a transition labeled:

$$a, b \rightarrow c$$

  - $a \in (\Sigma \cup \{\varepsilon\})$
  - $b, c \in (\Gamma \cup \{\varepsilon\})$

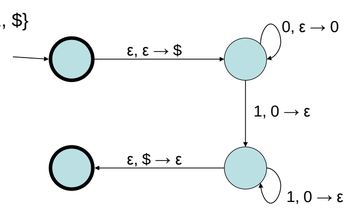
  - read $a$ from tape, or don't read from tape if $a = \varepsilon$
  - pop $b$ from stack, or don't pop from stack if $b = \varepsilon$
  - push $c$ onto stack, or don't push onto stack if $c = \varepsilon$



# Example NPDA

$\Sigma = \{0, 1\}$

$\Gamma = \{0, 1, \$\}$




- What language does this NPDA accept?



# Formal definition of NPDA

- A NPDA is a 6-tuple $(Q, \Sigma, \Gamma, \delta, q_0, F)$ where:
  - $Q$ is a finite set called the states
  - $\Sigma$ is a finite set called the tape alphabet
  - $\Gamma$ is a finite set called the stack alphabet
  - $\delta: Q \times (\Sigma \cup \{\varepsilon\}) \times (\Gamma \cup \{\varepsilon\}) \rightarrow \wp(Q \times (\Gamma \cup \{\varepsilon\}))$ is a function called the transition function
  - $q_0$ is an element of $Q$ called the start state
  - $F$ is a subset of $Q$ called the accept states

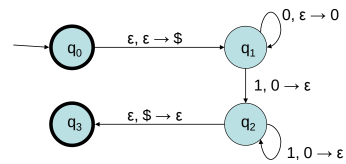
## Formal definition of NPDA

- NPDA M = $(Q, \Sigma, \Gamma, \delta, q_0, F)$ accepts string w $\in \Sigma^*$ if w can be written as

  $w_1w_2w_3\ldots w_m \in (\Sigma \cup \{\varepsilon\})^*$, and
- there exist states $r_0, r_1, r_2, \ldots, r_m$, and
- there exist strings $s_0, s_1, \ldots, s_m$ in $(\Gamma \cup \{\varepsilon\})^*$
  - $r_0 = q_0$ and $s_0 = \varepsilon$
  - $(r_{i+1}, b) \in \delta(r_i, w_{i+1}, a)$, where $s_i = at$, $s_{i+1} = bt$ for some $t \in \Gamma^*$
  - $r_m \in F$



## Example of formal definition




- $Q = \{q_0, q_1, q_2, q_3\}$
- $\Sigma = \{0,1\}$
- $\Gamma = \{0, 1, \$\}$
- $F = \{q_0, q_3\}$

- $\delta(q_0, \varepsilon, \varepsilon) = \{(q_1, \$)\}$
- $\delta(q_1, 0, \varepsilon) = \{(q_1, 0)\}$
- $\delta(q_1, 1, 0) = \{(q_2, \varepsilon)\}$
- $\delta(q_2, 1, 0) = \{(q_2, \varepsilon)\}$
- $\delta(q_2, \varepsilon, \$) = \{(q_3, \varepsilon)\}$

other values of $\delta(\bullet, \bullet, \bullet)$ equal {}



## Exercise

Design a NPDA for the language

$\{a^ib^jc^k : i, j, k \geq 0 \text{ and } i = j \text{ or } i = k\}$

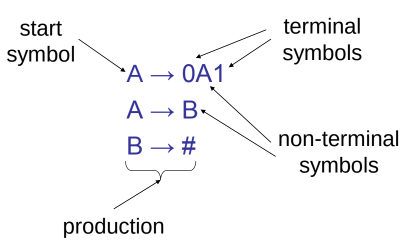


## Context-free grammars and languages

- 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) structure:

shorthand for multiple rules with same lhs

<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

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 → 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 ≥ 0 and strings $u_1,\ldots,u_{k-1}$ for which $\underbrace{u \Rightarrow u_1 \Rightarrow u_2 \Rightarrow \ldots \Rightarrow u_{k-1} \Rightarrow v}$
- 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

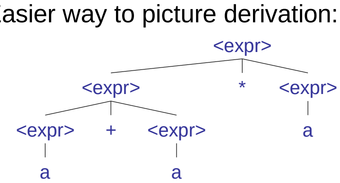
- A derivation of the string: a+a*a

  <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 *)
  - can also group correct way (ambiguous)

# Solution to first problem

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

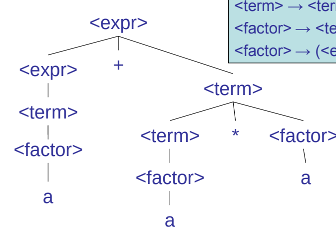
- forces correct precedence in parse tree grouping
  - within parentheses, * cannot occur as ancestor of + in the parse tree.



# 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






# Ambiguity

- Second problem: ambiguous grammar
- Definitions:
  - a string is derived ambiguously if it has two different parse trees
  - a grammar is ambiguous if its language contains an ambiguously derived string
- ambiguity sometimes undesirable
- some CFLS are inherently ambiguous



# Ambiguity

- Definition in terms of derivations (rather than parse trees):
  - order in which we replace terminals in shouldn't matter (often several orders possible)
  - define leftmost derivation to be one in which the leftmost non-terminal is always the one replaced
  - a string is ambiguously derived if it has 2 leftmost derivations



# Chomsky Normal Form

- Useful to deal only with CFGs in a simple normal form
- Most common: Chomsky Normal Form (CNF)
- Definition: every production has form

$A \rightarrow BC$ or $S \rightarrow \varepsilon$ or $A \rightarrow a$

where A, B, C are any non-terminals (and B, C are not S) and a is any terminal.



# Chomsky Normal Form

**Theorem**: Every CFL is generated by a CFG in Chomsky Normal Form.

**Proof**: Transform any CFG into an equivalent CFG in CNF. Four steps:
- add a new start symbol
- remove "ε-productions" $A \rightarrow \varepsilon$
- eliminate "unit productions" $A \rightarrow B$
- convert remaining rules into proper form

## Chomsky Normal Form

- add a new start symbol
  - add production $S_0 \to S$
- remove "ε-productions" $A \to \varepsilon$
  - for each production with A on rhs, add production with A's removed: e.g. for each rule $R \to uAv$, add $R \to uv$
- eliminate "unit productions" $A \to B$
  - for each production with B on lhs: $B \to u$, add rule $A \to u$



## Chomsky Normal Form

- convert remaining rules into proper form
  - replace production of form:

$$A \to u_1U_2u_3\ldots u_k$$

with:

$A \to U_1A_1$  $U_1 \to u_1$

$A_1 \to U_2A_2$

$A_2 \to U_3A_3$  $U_3 \to u_3$

:

$A_{k-2} \to U_{k-1}U_k$  $U_{k-1} \to u_{k-1}$  $U_k \to u_k$

$U_2$ already a non-terminal