1

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

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

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

2

Chomsky Normal Form

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

Proof: exercise or in book…

3

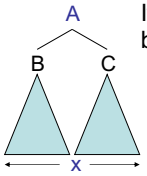
Deciding CFLs

- Useful to have an efficient algorithm to decide whether string x is in given CFL
  - e.g. programming language often described by CFG. Determine if string is valid program.
- If CFL recognized by deterministic PDA, just simulate the PDA.
  - but not all CFLs are (homework)…
- Can simulate NPDA, but this takes exponential time in the worst case.

4

Deciding CFLs

- Convert CFG into Chomsky Normal Form
- parse tree for string x generated by nonterminal A:



If $A \to^k x$ $(k > 1)$ then there must be a way to split x:

$x = yz$

- $A \to BC$ is a production and
- $B \to^i y$ and $C \Rightarrow^j z$ for $i, j < k$

5

Deciding CFLs

- An algorithm:

IsGenerated(x, A)

if $|x| = 1$, then return YES if $A \to x$ is a production, else return NO
for all n-1 ways of splitting x = yz
for all ≤ m productions of form $A \to BC$
if IsGenerated(y, B) and IsGenerated(z, C), return YES
return NO

- worst case running time?

6

Deciding CFLs

- worst case running time exp(n)
- Idea: avoid recursive calls
  - build table of YES/NO answers to calls to IsGenerated, in order of length of substring
  - example of general algorithmic strategy called dynamic programming
  - notation: x[i,j] = substring of x from i to j
  - table: T(i, j) contains
    {A: A nonterminal such that A →* x[i,j]}

7

Deciding CFLs

IsGenerated(x = $x_1x_2x_3…x_n$, G)
for i = 1 to n
T[i, i] = {A: "A → x_i" is a production in G}
for k = 1 to n - 1
for i = 1 to n - k
for k splittings x[i, i+k] = x[i,i+j]x[i+j+1, i+k]
T[i, i+k] = {A: "A → BC" is a production in G and B ∈ T[i,i+j] and C ∈ T[i+j+1,i+k] }
output "YES" if S ∈ T[1, n], else output "NO"

8

Deciding CFLs

IsGenerated(x = $x_1x_2x_3…x_n$, G)
for i = 1 to n
T[i, i] = {A: "A → x_i" is a production in G}
for k = 1 to n - 1
for i = 1 to n - k
for k splittings x[i, i+k] = x[i,i+j]x[i+j+1, i+k]
T[i, i+k] = {A: "A → BC" is a production in G and B ∈ T[i,i+j] and C ∈ T[i+j+1,i+k] }
output "YES" if S ∈ T[1, n], else output "NO"

O(nm) steps

$O(n^3m^3)$ steps

9

Deterministic PDA

- A NPDA is a 6-tuple (Q, Σ, Γ, δ, q_0, F) where:
  - δ:Q x (Σ ∪ {ε}) x (Γ ∪ {ε}) → $\mathcal{P}$ (Q x (Γ ∪ {ε})) is a function called the transition function
- A deterministic PDA has only one option at every step:
  - for every state q ∈ Q, a ∈ Σ, and t ∈ Γ, exactly 1 element in δ(q, a, t), or
  - exactly 1 element in δ(q, ε, t), and δ(q, a, t) empty for all a ∈ Σ

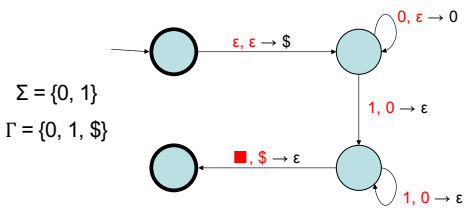
10

Deterministic PDA

- A technical detail:
  we will give our deterministic machine the ability to detect end of input string
  - add special symbol ■ to alphabet
  - require input tape to contain x■
- language recognized by a deterministic PDA is called a deterministic CFL (DCFL)

11

Example deterministic PDA



Σ = {0, 1}
Γ = {0, 1, $}

L = $\{0^n1^n : n \geq 0\}$

(unpictured transitions go to a "reject" state and stay there)

12

Deterministic PDA

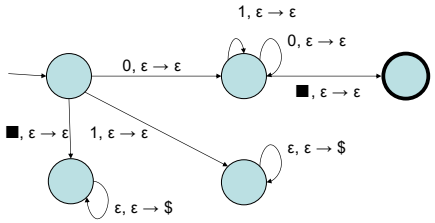
Theorem: DCFLs are closed under complement
(complement of L in Σ* is (Σ* - L))

Proof attempt:

- swap accept/non-accept states
- problem: might enter infinite loop before reading entire string
- machine for complement must accept in these cases, and read to end of string

13

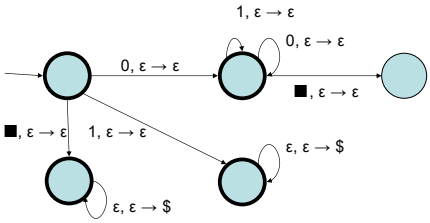
Example of problem



Language of this DPDA is $0\Sigma^*$

14

Example of problem



Language of this DPDA is $\{\epsilon\}$

15

Deterministic PDA

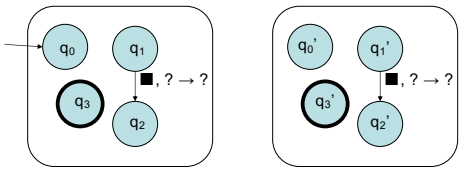
Proof:

- convert machine into "normal form"
  - always reads to end of input
  - always enters either an accept state or single distinguished "reject" state, and stay there
- step 1: keep track of when we have read to end of input
- step 2: eliminate infinite loops

16

Deterministic PDA

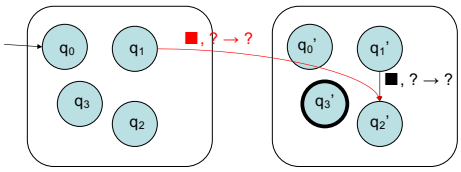
step 1: keep track of when we have read to end of input



17

Deterministic PDA

step 1: keep track of when we have read to end of input



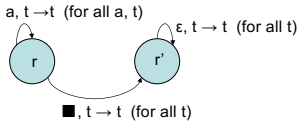
for accept state q': replace outgoing "ε, ? → ?" transition with self-loop with same label

18

Deterministic PDA

step 2: eliminate infinite loops

– add new "reject" states

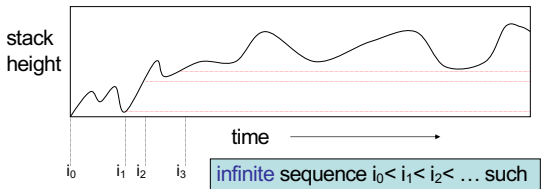


19

Deterministic PDA

step 2: eliminate infinite loops

– on input x, if infinite loop, then:



infinite sequence $i_0 < i_1 < i_2 < \ldots$ such that for all k, stack height never decreases below $ht(i_k)$ after time i_k

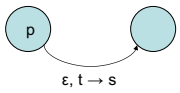
20

Deterministic PDA

step 2: eliminate infinite loops

– infinite seq. $i_0 < i_1 < \ldots$ such that for all k, stack height never decreases below $ht(i_k)$ after time i_k

– infinite subsequence $j_0 < j_1 < j_2 < \ldots$ such that same transition is applied at each time j_k



• never see any stack symbol below t from j_k on

• we are in a periodic, deterministic sequence of stack operations independent of the input

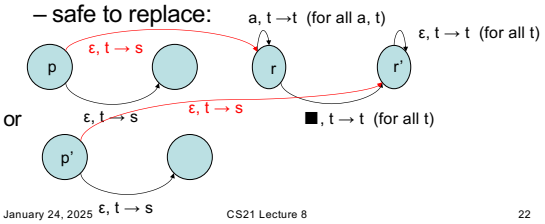
21

Deterministic PDA

step 2: eliminate infinite loops

– infinite subsequence $j_0 < j_1 < j_2 < \ldots$ such that same transition is applied at each time j_k

– safe to replace:



22

Deterministic PDA

– finishing up…

– have a machine M with no infinite loops

– therefore it always reads to end of input

– either enters an accept state q', or enters "reject" state r'

– now, can swap: make r' unique accept state to get a machine recognizing complement of L

23