Deciding CFLs

• An algorithm: 
  \[
  \text{IsGenerated}(x, A)
  \]
  - if \(|x| = 1\), then return YES if \(A \rightarrow x\) is a production, else return NO
  - for all \(n\)-1 ways of splitting \(x = yz\)
    - for all \(m\) productions of form \(A \rightarrow BC\)
      - if \(\text{IsGenerated}(y, B)\) and \(\text{IsGenerated}(z, C)\), return YES
    return NO
  - worst case running time?

Deciding CFLs

• worst case running time \(\exp(n)\)
  - Idea: avoid recursive calls
    - build table of YES/NO answers to calls to \(\text{IsGenerated}\)
    - 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
    \(\{A: \text{A nonterminal such that } A \rightarrow x[i,j]\}\)

Deciding CFLs

\[
\text{IsGenerated}(x = x_1x_2x_3 \ldots x_n, G)
\]
for \(i = 1\) to \(n\)
  \(T[i, j] = \{A: “A \rightarrow x_i” \text{ is a production in } G\}\)
for \(k = 1\) to \(n - 1\)
  for \(i = 1\) to \(n - k\)
    for \(k\) splittings \(x[i, i+k] = x[i,i+j][i+j+1,i+k]\)
    \(T[i, i+k] = \{A: “A \rightarrow BC” \text{ is a production in } G \text{ and } B \in T[i,i+j] \text{ and } C \in T[i+j+1,i+k]\}\)
  output “YES” if \(S \in T[1, n]\), else output “NO”
**Deterministic PDA**

- A NPDA is a 6-tuple \((Q, \Sigma, \Gamma, \delta, q_0, F)\) where:
  - \(\delta: Q \times (\Sigma \cup \{\varepsilon\}) \times (\Gamma \cup \{\varepsilon\}) \rightarrow \mathcal{P}(Q \times (\Gamma \cup \{\varepsilon\}))\) is a function called the transition function.
- A deterministic PDA has only one option at every step:
  - for every state \(q \in Q\), \(a \in \Sigma\), and \(t \in \Gamma\), exactly 1 element in \(\delta(q, a, t)\), or
  - exactly 1 element in \(\delta(q, \varepsilon, t)\), and \(\delta(q, a, t)\) empty for all \(a \in \Sigma\).

**Example deterministic PDA**

\(\Sigma = \{0, 1\}\)  
\(\Gamma = \{0, 1, \square\}\)

\(L = \{0^n1^n : n \geq 0\}\)

(unpicted transitions go to a “reject” state and stay there)

**Theorem:** DCFLs are closed under complement (complement of \(L\) in \(\Sigma^*\) is \((\Sigma^* - L)\))

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

**Example of problem**

Language of this DPDA is \(0\Sigma^*\)

**Example of problem**

Language of this DPDA is \(\{\varepsilon\}\)
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

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

Deterministic PDA

step 2: eliminate infinite loops
– add new "reject" states

Deterministic PDA

step 2: eliminate infinite loops
– infinite seq. i_0 < i_1 < ... such that for all k, stack height never decreases below h(t(i_k)) after time i_k
– infinite subsequence j_0 < j_1 < j_2 < ... such that same transition is applied at each time j_k
  • never see any stack symbol below t from j_k
  • we are in a periodic, deterministic sequence of stack operations independent of the input
Deterministic PDA

step 2: eliminate infinite loops
– infinite subsequence \( j_1 < j_2 < \ldots \) such that same transition is applied at each time \( j_k \)
– safe to replace:
\[
\begin{align*}
p, t \rightarrow s \quad & (for \ all \ a, t) \\
p', t \rightarrow s' \quad & (for \ all \ a, t) \\
\varepsilon, t \rightarrow s \quad & (for \ all \ t) \\
\varepsilon, t \rightarrow s' \quad & (for \ all \ t)
\end{align*}
\]

or

– 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 \( L \)

Summary

• Nondeterministic Pushdown Automata (NPDA)
• Context-Free Grammars (CFGs) describe Context-Free Languages (CFLs)
  – terminals, non-terminals
  – productions
  – yields, derivations
  – parse trees

• NDPAs and CFGs are equivalent
• CFL Pumping Lemma is used to show certain languages are not CFLs

So far…

• several models of computation
  – finite automata
  – pushdown automata
• fail to capture our intuitive notion of what is computable
  – regular languages
  – context free languages
  – all languages

Summary

• deterministic PDAs recognize DCFLs
• DCFLs are closed under complement
• there is an efficient algorithm (based on dynamic programming) to determine if a string \( x \) is generated by a given grammar \( G \)
So far…

- We proved (using constructions of FA and NPDA and the two pumping lemmas):
  - all languages
  - context free languages
  - regular languages

  - \( \{ w : w \in \{a,b\}^* \text{ has an even # of b's} \} \)
  - \( \{ a^n b^n : n \geq 0 \} \)
  - \( \{ a^n b^n c^n : n \geq 0 \} \)

A more powerful machine

- limitation of NPDA related to fact that their memory is stack-based (last in, first out)

- What is the simplest alteration that adds general-purpose “memory” to our machine?

  - Should be able to recognize, e.g., \( \{ a^n b^n : n \geq 0 \} \)

Turing Machines

- New capabilities:
  - infinite tape
  - can read OR write to tape
  - read/write head can move left and right

Turing Machine

- Informal description:
  - input written on left-most squares of tape
  - rest of squares are blank
  - at each point, take a step determined by
    - current symbol being read
    - current state of finite control
  - a step consists of
    - writing new symbol
    - moving read/write head left or right
    - changing state

Example Turing Machine

language \( L = \{ w\#w : w \in \{0,1\}^* \} \)

Turing Machine diagrams

- Transition label: (tape symbol read \rightarrow tape symbol written, direction moved)
  - \( a \rightarrow R \) means “read a, move right”
  - \( a \rightarrow L \) means “read a, move left”
  - \( a \rightarrow b, R \) means “read a, write b, move right”

  - “\_” means blank tape square