Contents

1. Non-deterministic pushdown automata and context free languages

Section outline

1. **Non-deterministic pushdown automata and context free languages**
   - CGFs and CFLs
   - Derivations and parse trees
   - Ambiguity
   - CFG examples
   - Practice example of CFG
   - RGs are CF
   - PDA
   - PDA acceptance
   - PDA examples
   - Predictive parsing
   - PDA from CFG
   - CFG from PDA
   - CoNF
   - Pumping in CFLs
   - Pumping lemma
   - Beyond CFLs
   - Practice example of Pumping lemma
   - Closure properties
   - DCFL $\subseteq$ CFL
   - CKY parsing

CGFs and CFLs

- $E \to E + T$
- $E \to T$

CGFs and CFLs

- $E \rightarrow E + T | T$

CGFs and CFLs

- $E \rightarrow E + T | T$
- $T \rightarrow T + F | F$
- $F \rightarrow (E) | a$

CGFs and CFLs

- $E \rightarrow E + T | T$
- $T \rightarrow T + F | F$
- $F \rightarrow (E) | a$

- $S \rightarrow (S) | \epsilon$

CGFs and CFLs

- $E \rightarrow E + T | T$
- $T \rightarrow T + F | F$
- $F \rightarrow (E) | a$

- $S \rightarrow (S) | \epsilon$
- $S \rightarrow (S)S | \epsilon$

CGFs and CFLs

- $E \to E + T | T$
- $T \to T + F | F$
- $F \to (E) | a$

- $S \to (S) | \epsilon$
- $S \to (S)S | \epsilon$
- $S \to 0S1S | 1S0S | \epsilon$

CGFs and CFLs

- $E \to E + T | T$
- $T \to T + F | F$
- $F \to (E) | a$

- $S \to (S) | \epsilon$
- $S \to (S)S | \epsilon$
- $S \to 0S1S | 1S0S | \epsilon$

A context-free grammar (CFG) $G = (V, \Sigma, R, S)$ is defined by the 4-tuple:

V is a finite set of non-terminal symbols or non-terminals.

Σ is a finite set of terminal symbols or terminals, disjoint from V

R is a finite relation from $V \to (V \cup \Sigma)^*$

these are called production rules

S is the starting non-terminal (or start symbol)

CGFs and CFLs

- $E \to E + T | T$
- $T \to T + F | F$
- $F \to (E) | a$
- $S \to (S) | \epsilon$
- $S \to (S)S | \epsilon$
- $S \to 0S1S | 1S0S | \epsilon$

A context-free grammar (CFG) $G = (V, \Sigma, R, S)$ is defined by the 4-tuple:

V is a finite set of non-terminal symbols or non-terminals.

Σ is a finite set of terminal symbols or terminals, disjoint from V

R is a finite relation from $V \to (V \cup \Sigma)^*$

these are called production rules

S is the starting non-terminal (or start symbol)

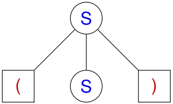
A context-free language (CFL) is a language generated by a CFG

Derivations and parse trees

- $S \rightarrow (S)|\epsilon$

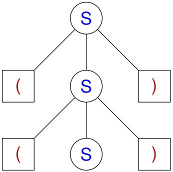
Derivations and parse trees

- $S \to (S)|\epsilon$
- $S \to (S)$



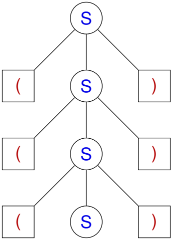
Derivations and parse trees

- $S \rightarrow (S)|\epsilon$
- $S \rightarrow (S) \rightarrow ((S))$



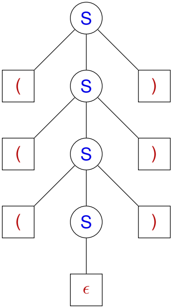
Derivations and parse trees

- $S \rightarrow (S)|\epsilon$
- $S \rightarrow (S) \rightarrow ((S)) \rightarrow (((S)))$



Derivations and parse trees

- $S \rightarrow (S)|\epsilon$
- $S \rightarrow (S) \rightarrow ((S)) \rightarrow (((S))) \rightarrow (((\epsilon)))$

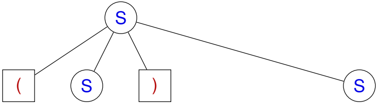


Derivations and parse trees (contd.)

- $S \rightarrow (S)S|\epsilon$

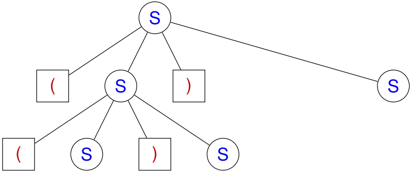
Derivations and parse trees (contd.)

- $S \rightarrow (S)S|\epsilon$
- $S \rightarrow (S)S$



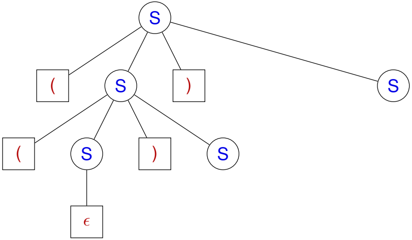
Derivations and parse trees (contd.)

- $S \rightarrow (S)S|\epsilon$
- $S \rightarrow (S)S \rightarrow ((S)S)S$



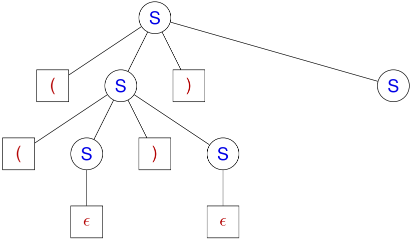
Derivations and parse trees (contd.)

- $S \rightarrow (S)S|\epsilon$
- $S \rightarrow (S)S \rightarrow ((S)S)S \rightarrow ((\epsilon)S)S$



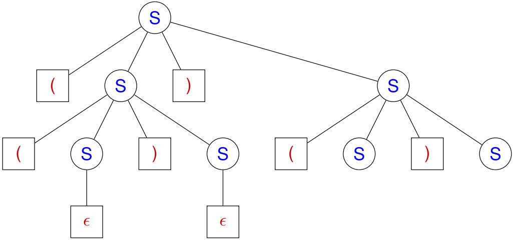
Derivations and parse trees (contd.)

- $S \rightarrow (S)S|\epsilon$
- $S \rightarrow (S)S \rightarrow ((S)S)S \rightarrow ((\epsilon)S)S \rightarrow ((\epsilon)\epsilon)S$



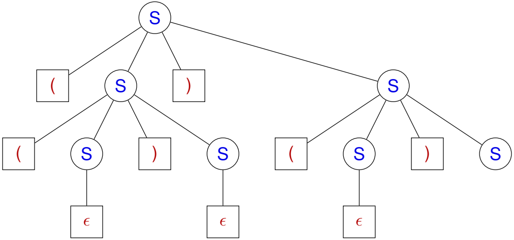
Derivations and parse trees (contd.)

- $S \rightarrow (S)S|\epsilon$
- $S \rightarrow (S)S \rightarrow ((S)S)S \rightarrow ((\epsilon)S)S \rightarrow ((\epsilon)\epsilon)S \rightarrow ((\epsilon)\epsilon)(S)S$



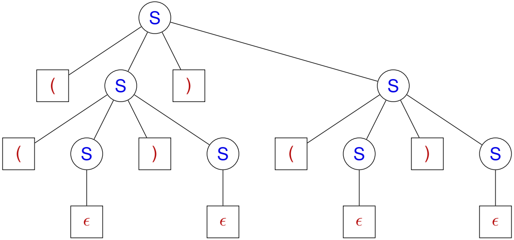
Derivations and parse trees (contd.)

- $S \rightarrow (S)S|\epsilon$
- $S \rightarrow (S)S \rightarrow ((S)S)S \rightarrow ((\epsilon)S)S \rightarrow ((\epsilon)\epsilon)S \rightarrow ((\epsilon)\epsilon)(S)S$
$\rightarrow ((\epsilon)\epsilon)(\epsilon)S$



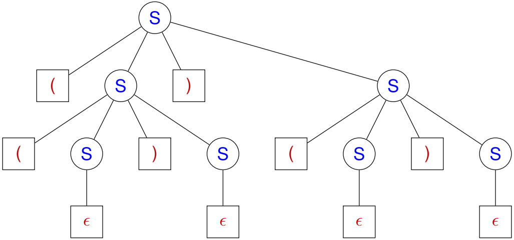
Derivations and parse trees (contd.)

- $S \rightarrow (S)S|\epsilon$
- $S \rightarrow (S)S \rightarrow ((S)S)S \rightarrow ((\epsilon)S)S \rightarrow ((\epsilon)\epsilon)S \rightarrow ((\epsilon)\epsilon)(S)S$
  $\rightarrow ((\epsilon)\epsilon)(\epsilon)S \rightarrow ((\epsilon)\epsilon)(\epsilon)\epsilon$



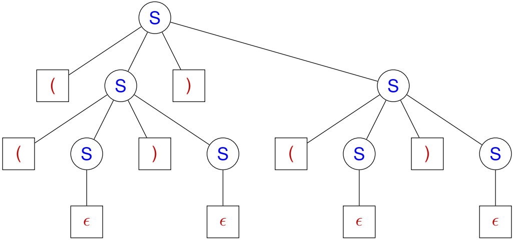
Derivations and parse trees (contd.)

- $S \rightarrow (S)S|\epsilon$
- $S \rightarrow (S)S \rightarrow ((S)S)S \rightarrow ((\epsilon)S)S \rightarrow ((\epsilon)\epsilon)S \rightarrow ((\epsilon)\epsilon)(S)S$ $\rightarrow ((\epsilon)\epsilon)(\epsilon)S \rightarrow ((\epsilon)\epsilon)(\epsilon)\epsilon$
- Leftmost derivaition involves expanding the leftmost non-terminal symbol in derivation



Derivations and parse trees (contd.)

- $S \rightarrow (S)S|\epsilon$
- $S \rightarrow (S)S \rightarrow ((S)S)S \rightarrow ((\epsilon)S)S \rightarrow ((\epsilon)\epsilon)S \rightarrow ((\epsilon)\epsilon)(S)S$
  $\rightarrow ((\epsilon)\epsilon)(\epsilon)S \rightarrow ((\epsilon)\epsilon)(\epsilon)\epsilon$
- Leftmost derivaition involves expanding the leftmost non-terminal symbol in derivation
- Derivation of string in zero or more steps: $S \xrightarrow{*} ((\epsilon)\epsilon)(\epsilon)\epsilon \equiv (())()$



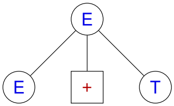
Derivations and parse trees (contd.)

- $E \rightarrow E + T | T$
  $T \rightarrow T \times F | F$
  $F \rightarrow (E) | a$
- Leaf nodes form the frontier of the parse tree; derivation over when all leaves are terminals

E

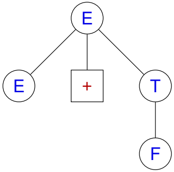
Derivations and parse trees (contd.)

- $E \to E + T | T$
  $T \to T \times F | F$
  $F \to (E) | a$
- Leaf nodes form the frontier of the parse tree; derivation over when all leaves are terminals
- $E \to E + T$



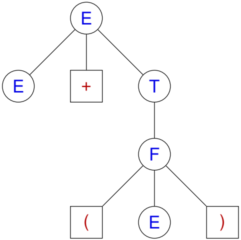
Derivations and parse trees (contd.)

- $E \rightarrow E + T | T$
  $T \rightarrow T \times F | F$
  $F \rightarrow (E) | a$
- Leaf nodes form the frontier of the parse tree; derivation over when all leaves are terminals
- $E \rightarrow E + T \rightarrow E + F$



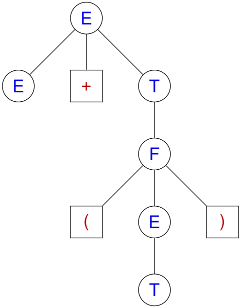
Derivations and parse trees (contd.)

- $E \rightarrow E + T | T$
  $T \rightarrow T \times F | F$
  $F \rightarrow (E) | a$
- Leaf nodes form the frontier of the parse tree; derivation over when all leaves are terminals
- $E \rightarrow E + T \rightarrow E + F$
  $\rightarrow E + (E)$



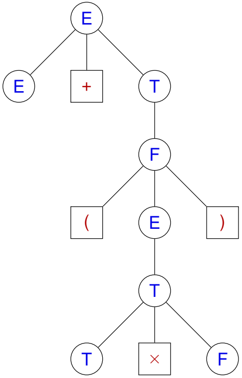
Derivations and parse trees (contd.)

- $E \to E + T | T$
  $T \to T \times F | F$
  $F \to (E) | a$
- Leaf nodes form the frontier of the parse tree; derivation over when all leaves are terminals
- $E \to E + T \to E + F$
  $\to E + (E) \to E + (T)$



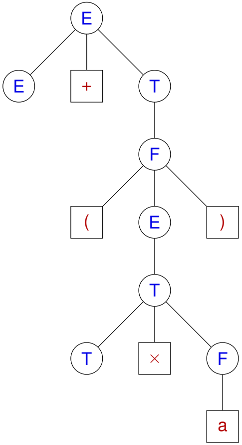
Derivations and parse trees (contd.)

- $E \rightarrow E + T | T$
  $T \rightarrow T \times F | F$
  $F \rightarrow (E) | a$
- Leaf nodes form the frontier of the parse tree; derivation over when all leaves are terminals
- $E \rightarrow E + T \rightarrow E + F$
  $\rightarrow E + (E) \rightarrow E + (T)$
  $\rightarrow E + (T \times F)$



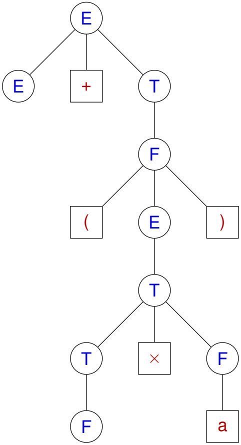
Derivations and parse trees (contd.)

- $E \to E + T | T$
  $T \to T \times F | F$
  $F \to (E) | a$
- Leaf nodes form the frontier of the parse tree; derivation over when all leaves are terminals
- $E \to E + T \to E + F$
  $\to E + (E) \to E + (T)$
  $\to E + (T \times F) \to E + (T \times a)$



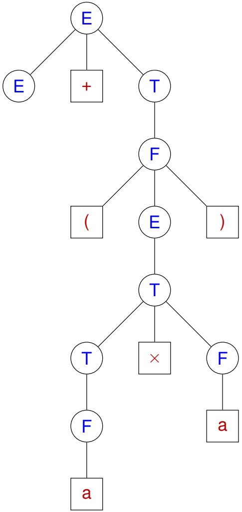
Derivations and parse trees (contd.)

- $E \rightarrow E + T|T$
  $T \rightarrow T \times F|F$
  $F \rightarrow (E)|a$
- Leaf nodes form the frontier of the parse tree; derivation over when all leaves are terminals
- $E \rightarrow E + T \rightarrow E + F$
  $\rightarrow E + (E) \rightarrow E + (T)$
  $\rightarrow E + (T \times F) \rightarrow E + (T \times a)$
  $\rightarrow E + (F \times a)$



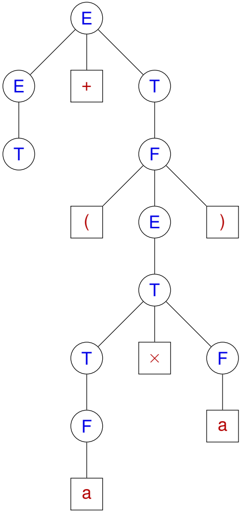
Derivations and parse trees (contd.)

- $E \to E + T | T$
  $T \to T \times F | F$
  $F \to (E) | a$
- Leaf nodes form the frontier of the parse tree; derivation over when all leaves are terminals
- $E \to E + T \to E + F$
  $\to E + (E) \to E + (T)$
  $\to E + (T \times F) \to E + (T \times a)$
  $\to E + (F \times a) \to E + (a \times a)$



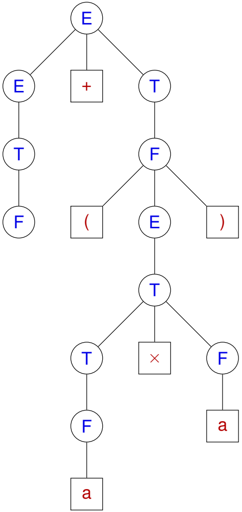
Derivations and parse trees (contd.)

- $E \rightarrow E + T | T$
  $T \rightarrow T \times F | F$
  $F \rightarrow (E) | a$
- Leaf nodes form the frontier of the parse tree; derivation over when all leaves are terminals
- $E \rightarrow E + T \rightarrow E + F$
  $\rightarrow E + (E) \rightarrow E + (T)$
  $\rightarrow E + (T \times F) \rightarrow E + (T \times a)$
  $\rightarrow E + (F \times a) \rightarrow E + (a \times a)$
  $\rightarrow T + (a \times a)$



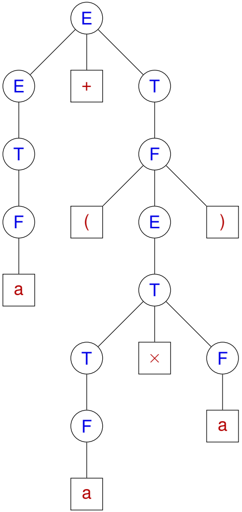
Derivations and parse trees (contd.)

- $E \to E + T | T$
  $T \to T \times F | F$
  $F \to (E) | a$
- Leaf nodes form the frontier of the parse tree; derivation over when all leaves are terminals
- $E \to E + T \to E + F$
  $\to E + (E) \to E + (T)$
  $\to E + (T \times F) \to E + (T \times a)$
  $\to E + (F \times a) \to E + (a \times a)$
  $\to T + (a \times a) \to F + (a \times a)$



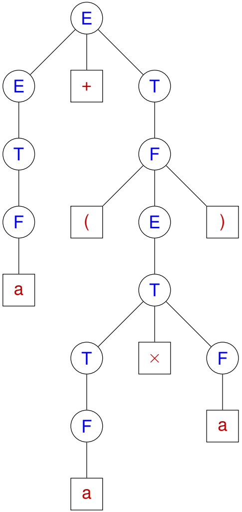
Derivations and parse trees (contd.)

- $E \to E + T | T$
  $T \to T \times F | F$
  $F \to (E) | a$
- Leaf nodes form the frontier of the parse tree; derivation over when all leaves are terminals
- $E \to E + T \to E + F$
  $\to E + (E) \to E + (T)$
  $\to E + (T \times F) \to E + (T \times a)$
  $\to E + (F \times a) \to E + (a \times a)$
  $\to T + (a \times a) \to F + (a \times a)$
  $\to a + (a \times a)$



Derivations and parse trees (contd.)

- $E \to E + T | T$
  $T \to T \times F | F$
  $F \to (E) | a$
- Leaf nodes form the frontier of the parse tree; derivation over when all leaves are terminals
- $E \to E + T \to E + F$
  $\to E + (E) \to E + (T)$
  $\to E + (T \times F) \to E + (T \times a)$
  $\to E + (F \times a) \to E + (a \times a)$
  $\to T + (a \times a) \to F + (a \times a)$
  $\to a + (a \times a)$
- This is a rightmost derivation
- Order of expansion of non-terminals does not affect the parse tree



Ambiguity

Grammar $E \to E + E | E \times E | (E) | a$

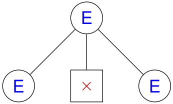
String $a + a \times a$

Ambiguity

Grammar $E \rightarrow E + E | E \times E | (E) | a$

String $a + a \times a$

- $E \rightarrow E \times E$

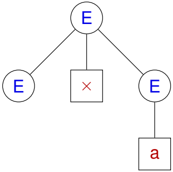


Ambiguity

Grammar $E \rightarrow E + E | E \times E | (E) | a$

String $a + a \times a$

- $E \rightarrow E \times E \rightarrow E \times a$

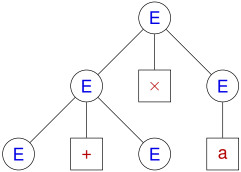


Ambiguity

Grammar $E \rightarrow E + E | E \times E | (E) | a$

String $a + a \times a$

- $E \rightarrow E \times E \rightarrow E \times a$
  $\rightarrow E + E \times a$

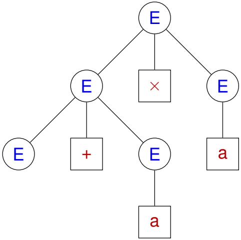


Ambiguity

Grammar $E \rightarrow E + E | E \times E | (E) | a$

String $a + a \times a$

- $E \rightarrow E \times E \rightarrow E \times a \rightarrow E + E \times a \rightarrow E + a \times a$

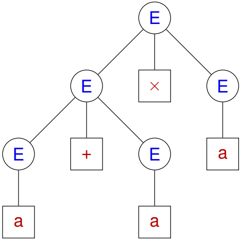


Ambiguity

Grammar $E \to E + E | E \times E | (E) | a$

String $a + a \times a$

- $E \to E \times E \to E \times a$
  $\to E + E \times a \to E + a \times a$
  $\to a + a \times a$

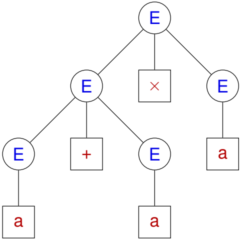


Ambiguity

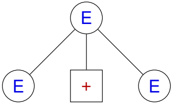
Grammar $E \rightarrow E + E | E \times E | (E) | a$

String $a + a \times a$

- $E \rightarrow E \times E \rightarrow E \times a$
  $\rightarrow E + E \times a \rightarrow E + a \times a$
  $\rightarrow a + a \times a$



- $E \rightarrow E + E$

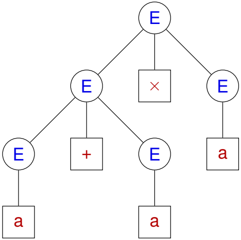


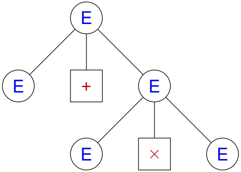
Ambiguity

Grammar $E \to E + E | E \times E | (E) | a$

String $a + a \times a$

- $E \to E \times E \to E \times a \to E + E \times a \to E + a \times a \to a + a \times a$
- $E \to E + E \to E + E \times E$



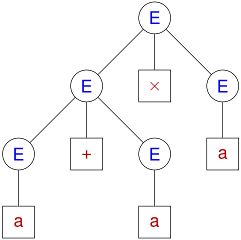


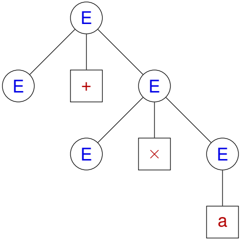
Ambiguity

Grammar $E \to E + E | E \times E | (E) | a$

String $a + a \times a$

- $E \to E \times E \to E \times a \to E + E \times a \to E + a \times a \to a + a \times a$
- $E \to E + E \to E + E \times E \to E + E \times a$



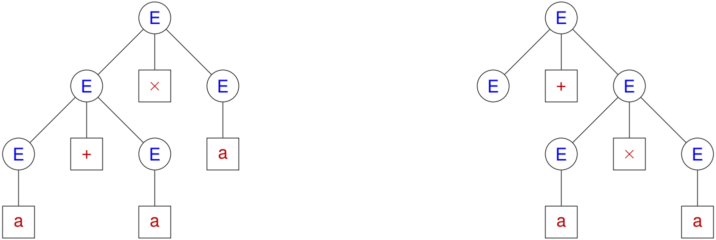


Ambiguity

Grammar $E \rightarrow E + E | E \times E | (E) | a$

String $a + a \times a$

- $E \rightarrow E \times E \rightarrow E \times a$
  $\rightarrow E + E \times a \rightarrow E + a \times a$
  $\rightarrow a + a \times a$
- $E \rightarrow E + E \rightarrow E + E \times E$
  $\rightarrow E + E \times a \rightarrow E + a \times a$

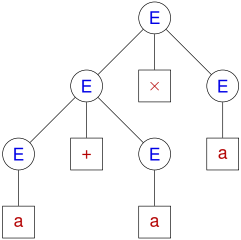


Ambiguity

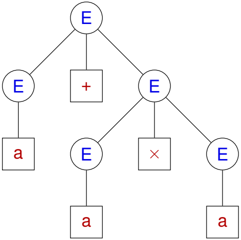
Grammar $E \to E+E|E\times E|(E)|a$

String $a+a\times a$

- $E \to E\times E \to E\times a \to E+E\times a \to E+a\times a \to a+a\times a$



- $E \to E+E \to E+E\times E \to E+E\times a \to E+a\times a \to a+a\times a$

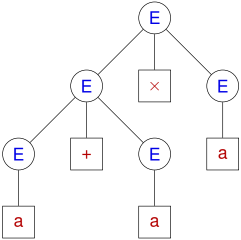


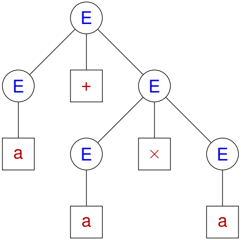
Ambiguity

Grammar $E \to E + E | E \times E | (E) | a$

String $a + a \times a$

- $E \to E \times E \to E \times a$
  $\to E + E \times a \to E + a \times a$
  $\to a + a \times a$
- $E \to E + E \to E + E \times E$
  $\to E + E \times a \to E + a \times a$
  $\to a + a \times a$





A grammar is *ambiguous* if some string can be derived in more than one way, i.e. a string has *multiple* parse trees

Ambiguous grammars and ambiguous languages

- $E \rightarrow E + E | E \times E | (E) | a$

Ambiguous grammars and ambiguous languages

- $E \rightarrow E + E | E \times E | (E) | a$
- $E \rightarrow E + T | T \qquad T \rightarrow T \times F | F \qquad F \rightarrow (E) | a$
  is an unambiguous grammar generating the same language

Ambiguous grammars and ambiguous languages

- $E \rightarrow E + E | E \times E | (E) | a$
- $E \rightarrow E + T | T \qquad T \rightarrow T \times F | F \qquad F \rightarrow (E) | a$
  is an unambiguous grammar generating the same language
- Sometimes it is possible to find an unambiguous grammar for a given ambiguous grammar

Ambiguous grammars and ambiguous languages

- $E \rightarrow E + E | E \times E | (E) | a$
- $E \rightarrow E + T | T \qquad T \rightarrow T \times F | F \qquad F \rightarrow (E) | a$
  is an unambiguous grammar generating the same language
- Sometimes it is possible to find an unambiguous grammar for a given ambiguous grammar
- An ambiguous language is one that is generated only by ambiguous grammars
  i.e. there is no unambiguous grammar that generates that language

Ambiguous grammars and ambiguous languages

- $E \to E + E | E \times E | (E) | a$
- $E \to E + T | T \qquad T \to T \times F | F \qquad F \to (E) | a$
  is an unambiguous grammar generating the same language
- Sometimes it is possible to find an unambiguous grammar for a given ambiguous grammar
- An ambiguous language is one that is generated only by ambiguous grammars
  i.e. there is no unambiguous grammar that generates that language
- Union of the lanugages $\{a^n b^m c^m | n, m > 0\}$ and $\{a^n b^n c^m | n, m > 0\}$ (also context free)

Ambiguous grammars and ambiguous languages

- $E \rightarrow E + E|E \times E|(E)|a$
- $E \rightarrow E + T|T \qquad T \rightarrow T \times F|F \qquad F \rightarrow (E)|a$
  is an unambiguous grammar generating the same language
- Sometimes it is possible to find an unambiguous grammar for a given ambiguous grammar
- An ambiguous language is one that is generated only by ambiguous grammars
  i.e. there is no unambiguous grammar that generates that language
- Union of the lanugages $\{a^n b^m c^m | n, m > 0\}$ and $\{a^n b^n c^m | n, m > 0\}$ (also context free)
- $S \rightarrow S_1 \mid S_2 \quad S_1 \rightarrow S_1 c \mid A \quad A \rightarrow aAb \mid \epsilon \; S_2 \rightarrow aS_2 \mid B$
  $B \rightarrow bBc \mid \epsilon$

Ambiguous grammars and ambiguous languages

- $E \to E + E | E \times E | (E) | a$
- $E \to E + T | T \qquad T \to T \times F | F \qquad F \to (E) | a$
  is an unambiguous grammar generating the same language
- Sometimes it is possible to find an unambiguous grammar for a given ambiguous grammar
- An ambiguous language is one that is generated only by ambiguous grammars
  i.e. there is no unambiguous grammar that generates that language
- Union of the lanugages $\{a^n b^m c^m | n, m > 0\}$ and $\{a^n b^n c^m | n, m > 0\}$ (also context free)
- $S \to S_1 \mid S_2 \quad S_1 \to S_1 c \mid A \quad A \to aAb \mid \epsilon \; S_2 \to aS_2 \mid B$
  $B \to bBc \mid \epsilon$
- $a^n b^n c^n$ will have two parse trees

CFG examples

- Language over $\{0, 1\}$ with equal number of 0's and 1's

CFG examples

- Language over $\{0, 1\}$ with equal number of 0's and 1's

  $S \rightarrow 0A|1B|\epsilon$ // equal number of 0's and 1's

  $A \rightarrow 1S|0AA$ // strings with one 1 more than 0's

  $B \rightarrow 0S|1BB$ // strings with one 0 more than 1's

CFG examples

- Language over $\{0, 1\}$ with equal number of 0's and 1's

  $S \rightarrow 0A|1B|\epsilon$ // equal number of 0's and 1's

  $A \rightarrow 1S|0AA$ // strings with one 1 more than 0's

  $B \rightarrow 0S|1BB$ // strings with one 0 more than 1's
- Another option

  $S \rightarrow SAB|\epsilon$ // to generate AB AB AB ...

  $A \rightarrow 0S1|\epsilon$ // #0 = #1, starting with 0 ending with 1 if non-empty

  $B \rightarrow 1S0|\epsilon$ // #0 = #1, starting with 1 ending with 0 if non-empty

CFG examples

- Language over $\{0, 1\}$ with equal number of 0's and 1's

  $S \rightarrow 0A|1B|\epsilon$ // equal number of 0's and 1's

  $A \rightarrow 1S|0AA$ // strings with one 1 more than 0's

  $B \rightarrow 0S|1BB$ // strings with one 0 more than 1's
- Another option

  $S \rightarrow SAB|\epsilon$ // to generate AB AB AB ...

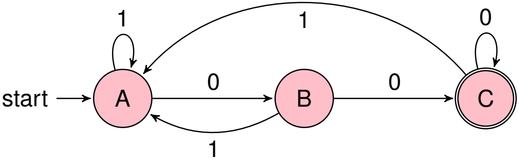
  $A \rightarrow 0S1|\epsilon$ // #0 = #1, starting with 0 ending with 1 if non-empty

  $B \rightarrow 1S0|\epsilon$ // #0 = #1, starting with 1 ending with 0 if non-empty
- Are they really equivalent?
  - Check whether for a string generated by G_1 has a parse tree for G_2
  - If not, they are not equivalent
  - Vice-versa
  - Any issues with this scheme?

Practice example of CFG

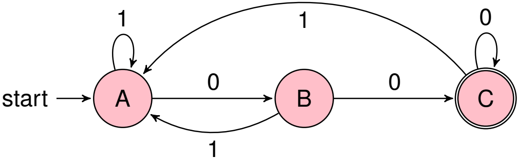
1. For each of the following languages, give a context-free grammar.
   1. $L_1 = \{xy \mid |x| = |y| \text{ and } x \neq y\}$ $\Sigma = \{a, b\}$.
   2. $L_2 = \{w | w$ has the same number of $a's$ as $b's$ and $c's$ together$\}$, $\Sigma = \{a, b, c\}$.
   3. $L_3 = \left\{a^i b^j c^k d^l | i + k = j + l, i, k, j, l \geq 0\right\}.\Sigma = \{a, b, c, d\}$.
   4. $L_4 = \left\{w\#x | w^R \text{ is a substring of } x \text{ for } w, x \in \{0, 1\}^\star\right\}$.
   5. $L_5 = \{w | w$ has twice as many $a's$ as $b's\}$.
2. What is the language defined by the following grammar:
   1. $S \rightarrow AS \mid \epsilon$
      $A \rightarrow 0A1 \mid A1 \mid 01$
   2. $S \rightarrow A1B$
      $A \rightarrow 0A \mid \epsilon$
      $B \rightarrow 0B \mid 1B \mid \epsilon$.

Regular grammars are context free



- $S \to A$
- $A \to 1A$
- $A \to 0B$
- $B \to 1A$
- $B \to 0C$
- $C \to 1A$
- $C \to 0C$
- $C \to \epsilon$

Regular grammars are context free



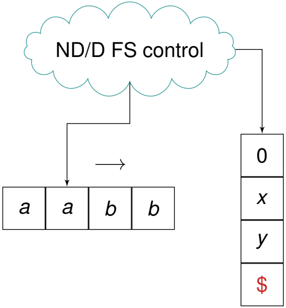
- $S \to A$
- $A \to 1A$
- $A \to 0B$
- $B \to 1A$
- $B \to 0C$
- $C \to 1A$
- $C \to 0C$
- $C \to \epsilon$

- Introduce a non-terminal A for a state A and also the start symbol S
- For the start state A, introduce the production $S \to A$
- For a transition from state A to state B on a symbol a, introduce the production $A \to aB$
- For the accepting state F, introduce the production $F \to \epsilon$

PDA

$M = \langle Q, \Sigma, \Gamma, \delta, q_0, F \rangle$ where

- Q is a finite set of states
- Σ is a finite set which is called the input alphabet
- Γ is a finite set which is called the stack alphabet
- δ is a **finite** subset of $\left(Q \times \Sigma_\epsilon \times \Gamma^*\right) \times \left(Q \times \Gamma^*\right)$, the transition relation, also
  $\delta : \left(Q \times \Sigma_\epsilon \times \Gamma^*\right) \to 2^{\left(Q\times\Gamma^*\right)}$
- $q_I \in Q$ is the start state
- $F \subseteq Q$ is the set of accepting states
- $\$ \in \Gamma$ is the initial stack symbol



In each possible transition:

- 0/1 symbol of input read
- 0/1 symbol from stack popped or 0/more pushed
- deterministic transitions for DPDA

PDA (contd.)

Testing empty stack:

- PDA has no explicit mechanism for testing the empty stack
- PDA may test empty stack by initially placing a special symbol $, on the stack
- If $ is seen again on the stack, it knows that the stack is effectively empty

PDA (contd.)

Testing empty stack:

- PDA has no explicit mechanism for testing the empty stack
- PDA may test empty stack by initially placing a special symbol \$, on the stack
- If \$ is seen again on the stack, it knows that the stack is effectively empty

Testing end of input:

- PDA has no explicit mechanism for testing the end of input
- PDA may rely on transition out of the accepting state to failure state if more input is available

PDA (contd.)

Testing empty stack:

- PDA has no explicit mechanism for testing the empty stack
- PDA may test empty stack by initially placing a special symbol \$, on the stack
- If \$ is seen again on the stack, it knows that the stack is effectively empty

Testing end of input:

- PDA has no explicit mechanism for testing the end of input
- PDA may rely on transition out of the accepting state to failure state if more input is available

Final state in conjuction with empty stack determines acceptance

PDA acceptance

A m/c CmNFiguration is $\langle q, w, \gamma \rangle \in \{\langle Q \times \Sigma^* \times \Gamma^* \rangle\}$

p current state

w unprocessed input

γ stack content

The yields relationships are:

- $\langle p, aw, \alpha\gamma \rangle \vdash \langle q, w, \beta\gamma \rangle$ *if* $\langle \langle p, a, \alpha \rangle, \langle q, \beta \rangle \rangle \in \delta$
- $\langle p, w, \alpha\gamma \rangle \vdash \langle q, w, \beta\gamma \rangle$ *if* $\langle \langle p, \epsilon, \alpha \rangle, \langle q, \beta \rangle \rangle \in \delta$

PDA acceptance

A m/c CmNFiguration is $\langle q, w, \gamma \rangle \in \{\langle Q \times \Sigma^* \times \Gamma^* \rangle\}$

- p current state
- w unprocessed input
- γ stack content

The yields relationships are:

- $\langle p, aw, \alpha\gamma \rangle \vdash \langle q, w, \beta\gamma \rangle$ *if* $\langle \langle p, a, \alpha \rangle, \langle q, \beta \rangle \rangle \in \delta$
- $\langle p, w, \alpha\gamma \rangle \vdash \langle q, w, \beta\gamma \rangle$ *if* $\langle \langle p, \epsilon, \alpha \rangle, \langle q, \beta \rangle \rangle \in \delta$

A string w is accepted by the PDA if $\langle q_I, w, \epsilon \rangle \stackrel{*}{\vdash} \langle f, \epsilon, \epsilon \rangle, f \in F$

- start with empty stack
- consume all input with appropriate transitions
- reach a final state with the stack **empty**

Language $L(M)$ of PDA M is the set of all accepted strings

Equivalence of acceptence criteria

Acceptance by final state only If P is a PDA, then $L(P)$ is the set of strings such that $\langle q_0, w, \$\rangle \vdash^* \langle f, \epsilon, \alpha\rangle$, for any state α and $f \in F$

Acceptance by empty stack only If P is a PDA, then $N(P)$ is the set of strings such that $\langle q_0, w, \$\rangle \vdash^* \langle q, \epsilon, \epsilon\rangle$, for any state q

Equivalence of acceptence criteria

Acceptance by final state only If P is a PDA, then $L(P)$ is the set of strings such that $\langle q_0, w, \$\rangle \vdash^* \langle f, \epsilon, \alpha\rangle$, for any state α and $f \in F$

Acceptance by empty stack only If P is a PDA, then $N(P)$ is the set of strings such that $\langle q_0, w, \$\rangle \vdash^* \langle q, \epsilon, \epsilon\rangle$, for any state q

Equivalence of definitions

- If $L = L(P)$, then there is another PDA P_e such that $L = N(P_e)$
  - \$ is used to keep track of the bottom of stack; $\delta : \langle s, \epsilon, \epsilon\rangle \mapsto \langle q0, \$\rangle$
  - P_e will simulate P, if P accepts, P_e will empty its stack to accept; $\delta : \langle f, \epsilon, X\rangle \mapsto \langle e, \epsilon\rangle$, $\delta : \langle e, \epsilon, X\rangle \mapsto \langle e, \epsilon\rangle$

Equivalence of acceptence criteria

Acceptance by final state only If P is a PDA, then $L(P)$ is the set of strings such that $\langle q_0, w, \$\rangle \vdash^* \langle f, \epsilon, \alpha\rangle$, for any state α and $f \in F$

Acceptance by empty stack only If P is a PDA, then $N(P)$ is the set of strings such that $\langle q_0, w, \$\rangle \vdash^* \langle q, \epsilon, \epsilon\rangle$, for any state q

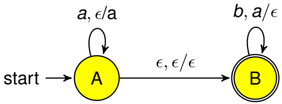
Equivalence of definitions

- If $L = L(P)$, then there is another PDA P_e such that $L = N(P_e)$
  - \$ is used to keep track of the bottom of stack; $\delta : \langle s, \epsilon, \epsilon\rangle \mapsto \langle q0, \$\rangle$
  - P_e will simulate P, if P accepts, P_e will empty its stack to accept; $\delta : \langle f, \epsilon, X\rangle \mapsto \langle e, \epsilon\rangle$, $\delta : \langle e, \epsilon, X\rangle \mapsto \langle e, \epsilon\rangle$
- If $L = N(P)$, then there is another PDA P_f such that $L = L(P_f)$
  - P_f will simulate P, using \$ to keep track of the bottom of stack; $\delta : \langle s, \epsilon, \epsilon\rangle \mapsto \langle q0, \$\rangle$
  - Make a transition to a designated final state to accept; $\delta : \langle q, \epsilon, \$\rangle \mapsto \langle f, \epsilon\rangle$,

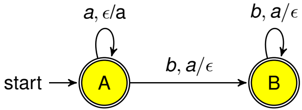
PDA examples

$L = \{w \in a, b^* : w = a^n b^n, n \geq 0\}$

- NPDA



- DPDA

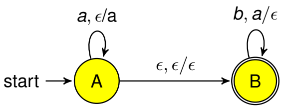


- ϵ on stack top means *not consulting* the stack
- For acceptance, stack needs to be empty after exhausing input

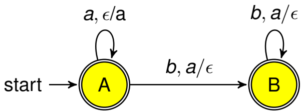
PDA examples

$L = \{w \in a,b^* : w = a^n b^n, n \geq 0\}$

- NPDA



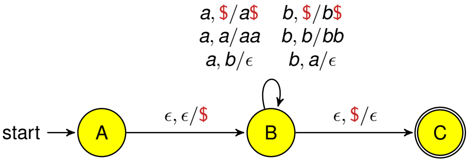
- DPDA



- ϵ on stack top means *not consulting* the stack
- For acceptance, stack needs to be empty after exhausing input

$L = \{w \in a,b^* : \#(a) = \#(b)\}$

- DPDA/NPDA?



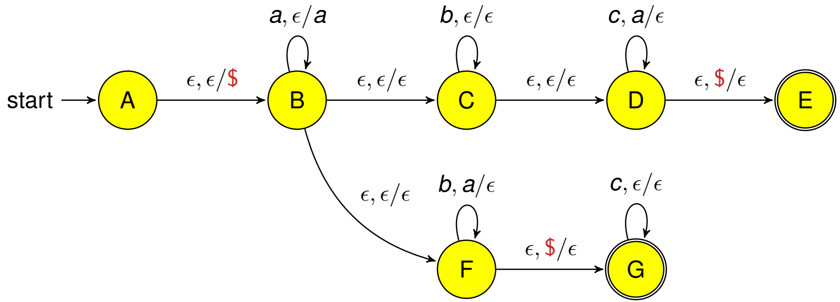
- Push \$ (mark bottom of stack)
- Handle inputs
- Emptiness of stack is checked using \$, pushed at start state with no other outgoing transitions
- Never removed except by a transition to (final) state with no other outgoing transitions

PDA examples (contd.)

$$L = \{a^i b^j c^k | i, j, k \geq 0 \wedge (i = j \vee i = k)\}$$

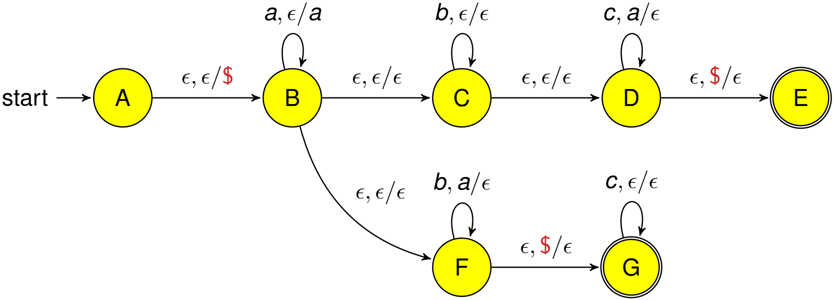
PDA examples (contd.)

$L = \{a^i b^j c^k | i, j, k \geq 0 \wedge (i = j \vee i = k)\}$



PDA examples (contd.)

$$L = \{a^i b^j c^k | i, j, k \geq 0 \wedge (i = j \vee i = k)\}$$



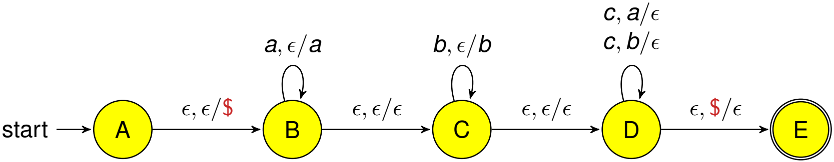
How to make it more “robust”?

PDA examples (contd.)

$L = \{a^i b^j c^{i+j} | i, j \geq 0)\}$

PDA examples (contd.)

$L = \{a^i b^j c^{i+j} | i, j \geq 0)\}$



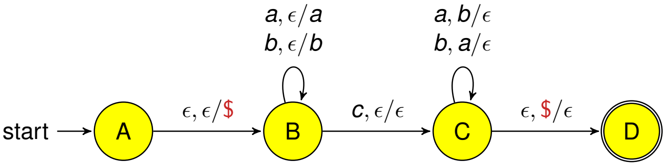
PDA examples (contd.)

- Palindrome with mid marker

PDA examples (contd.)

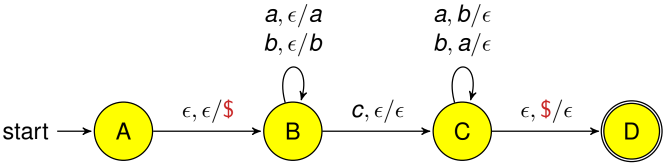
- Palindrome with mid marker

$$L = \left\{ x \in \{a, b, c\}^* \,|\, x = wcw^R, w \in \{a, b\}^* \right\}$$



PDA examples (contd.)

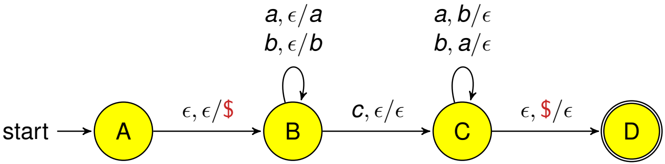
- Palindrome with mid marker
  $L = \left\{ x \in \{a, b, c\}^* \,|x = wcw^R, w \in \{a, b\}^* \right\}$



- Palindrome without mid marker

PDA examples (contd.)

- Palindrome with mid marker
  $L = \left\{ x \in \{a, b, c\}^* \,|x = wcw^R, w \in \{a, b\}^* \right\}$

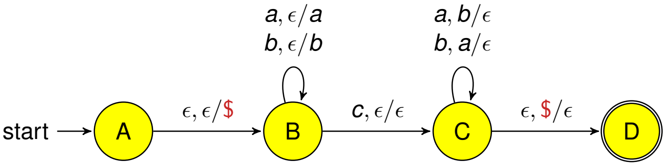


- Palindrome without mid marker
  $L = \left\{ x \in \{a, b\}^* \,|x = ww^R, w \in \{a, b\}^* \right\}$

PDA examples (contd.)

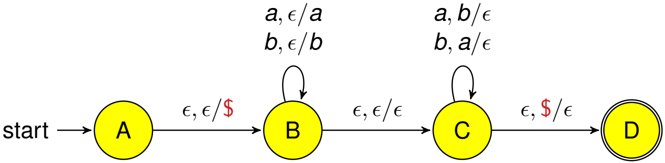
- Palindrome with mid marker

$$L = \left\{ x \in \{a, b, c\}^* \,|\, x = wcw^R, w \in \{a, b\}^* \right\}$$



- Palindrome without mid marker

$$L = \left\{ x \in \{a, b\}^* \,|\, x = ww^R, w \in \{a, b\}^* \right\}$$



Predictive parsing

- Consider a grammar with productions $S \to aSb|bSa|\epsilon$
- This grammar generates palindromes over $\Sigma = \{a, b\}$
- How to recognise such a plaindrome using a suitable PDA?

Predictive parsing

- Consider a grammar with productions $S \rightarrow aSb|bSa|\epsilon$
- This grammar generates palindromes over $\Sigma = \{a, b\}$
- How to recognise such a plaindrome using a suitable PDA?
- Start with the stack as $S\$$
- Non-deterministically predict which rule should be applied to the non-terminal at the top of the stack [leftmost derivation] and push the right of it on to the stack

Predictive parsing

- Consider a grammar with productions $S \rightarrow aSb|bSa|\epsilon$
- This grammar generates palindromes over $\Sigma = \{a, b\}$
- How to recognise such a plaindrome using a suitable PDA?
- Start with the stack as $S\$$
- Non-deterministically predict which rule should be applied to the non-terminal at the top of the stack [leftmost derivation] and push the right of it on to the stack
- If $S \rightarrow aSb$ is chosen, then push b, then S and finally a
- If $S \rightarrow bSa$ is chosen, then push a, then S and finally b

Predictive parsing

- Consider a grammar with productions $S \to aSb|bSa|\epsilon$
- This grammar generates palindromes over $\Sigma = \{a, b\}$
- How to recognise such a plaindrome using a suitable PDA?
- Start with the stack as $S\$$
- Non-deterministically predict which rule should be applied to the non-terminal at the top of the stack [leftmost derivation] and push the right of it on to the stack
- If $S \to aSb$ is chosen, then push b, then S and finally a
- If $S \to bSa$ is chosen, then push a, then S and finally b
- If the top of stack is a terminal, pop if it matches with the terminal in the input stream, otherwise fail

PDA from CFG

Given a CFG $G = (V, \Sigma, R, S)$, a PDA $\langle Q, \Sigma, \Gamma, \delta, q_I, F\rangle$ is constructed as follows:

- $Q = \{q_0, q_1, q_2\}$
- $F = \{q_2\}$
- $q_I = q_1$
- $\Gamma = V \cup \Sigma$
- $\delta : \left(Q \times \Sigma_\epsilon \times \Gamma^*\right) \to 2^{\left(Q \times \Gamma^*\right)}$
  - $\langle q_0, \epsilon, \epsilon\rangle \mapsto \langle q_1, S\$\rangle$ – push the end of stack marker and the start symbol
  - $\langle q_1, A, \epsilon\rangle \mapsto \langle q_1, \alpha\rangle$, where $\langle A \to \alpha\rangle \in R$ – non-deterministically predict that a left most derivation step should be done using $\langle A \to \alpha\rangle$, when A is at the top of the stack
  - $\langle q_1, a, a\rangle \mapsto \langle q_1, \epsilon\rangle$ – pop a terminal at the top of the stack matching with the input
  - $\langle q_1, \epsilon, \$\rangle \mapsto \langle q_2, \epsilon\rangle$ – on end of input move to the final state

CFG from PDA

- For any PDA, let consider the language $\langle A, B, t \rangle =$ $\left\{ x \in \Sigma^* \;\middle|\; \begin{array}{l} x \text{ is consumed in moving from state } A \text{ to state } B \text{ in} \\ \text{the machine, with the symbol } t \text{ being taken from} \\ \text{the top of the stack in the process} \end{array} \right\}$

CFG from PDA

- For any PDA, let consider the language $\langle A, B, t \rangle = \left\{ x \in \Sigma^* \;\middle|\; \begin{array}{l} x \text{ is consumed in moving from state } A \text{ to state } B \text{ in} \\ \text{the machine, with the symbol } t \text{ being taken from} \\ \text{the top of the stack in the process} \end{array} \right\}$
- Now, given any PDA, construct a CFG which accepts the same language as follows:

CFG from PDA

- For any PDA, let consider the language $\langle A, B, t\rangle = \left\{ x \in \Sigma^* \;\middle|\; \begin{array}{l} x \text{ is consumed in moving from state } A \text{ to state } B \text{ in} \\ \text{the machine, with the symbol } t \text{ being taken from} \\ \text{the top of the stack in the process} \end{array} \right\}$
- Now, given any PDA, construct a CFG which accepts the same language as follows:
  - the terminal symbols are just the input symbols of the PDA

CFG from PDA

- For any PDA, let consider the language $\langle A, B, t\rangle = \left\{ x \in \Sigma^* \;\middle|\; \begin{array}{l} x \text{ is consumed in moving from state } A \text{ to state } B \text{ in} \\ \text{the machine, with the symbol } t \text{ being taken from} \\ \text{the top of the stack in the process} \end{array} \right\}$
- Now, given any PDA, construct a CFG which accepts the same language as follows:
  - the terminal symbols are just the input symbols of the PDA
  - the non-terminal symbols are all triples of the form $\langle A, B, t\rangle$

CFG from PDA

- For any PDA, let consider the language $\langle A, B, t \rangle =$
$$\left\{ x \in \Sigma^* \;\middle|\; \begin{array}{l} x \text{ is consumed in moving from state } A \text{ to state } B \text{ in} \\ \text{the machine, with the symbol } t \text{ being taken from} \\ \text{the top of the stack in the process} \end{array} \right\}$$
- Now, given any PDA, construct a CFG which accepts the same language as follows:
  - the terminal symbols are just the input symbols of the PDA
  - the non-terminal symbols are all triples of the form $\langle A, B, t \rangle$
  - if S and F are the start and finish states respectively of the PDA, then the start symbol of the CFG is $\langle S, F, \epsilon \rangle$

CFG from PDA

- For any PDA, let consider the language $\langle A, B, t\rangle = \left\{ x \in \Sigma^* \;\middle|\; \begin{array}{l} x \text{ is consumed in moving from state } A \text{ to state } B \text{ in} \\ \text{the machine, with the symbol } t \text{ being taken from} \\ \text{the top of the stack in the process} \end{array} \right\}$
- Now, given any PDA, construct a CFG which accepts the same language as follows:
  - the terminal symbols are just the input symbols of the PDA
  - the non-terminal symbols are all triples of the form $\langle A, B, t\rangle$
  - if S and F are the start and finish states respectively of the PDA, then the start symbol of the CFG is $\langle S, F, \epsilon\rangle$
  - The production rules are as follows:

CFG from PDA

- For any PDA, let consider the language $\langle A, B, t \rangle = \left\{ x \in \Sigma^* \;\middle|\; \begin{array}{l} x \text{ is consumed in moving from state } A \text{ to state } B \text{ in} \\ \text{the machine, with the symbol } t \text{ being taken from} \\ \text{the top of the stack in the process} \end{array} \right\}$
- Now, given any PDA, construct a CFG which accepts the same language as follows:
  - the terminal symbols are just the input symbols of the PDA
  - the non-terminal symbols are all triples of the form $\langle A, B, t \rangle$
  - if S and F are the start and finish states respectively of the PDA, then the start symbol of the CFG is $\langle S, F, \epsilon \rangle$
  - The production rules are as follows:
    - For each transition of the form $\langle A, a, t \rangle \mapsto \langle C, \beta_1 \cdots \beta_j \rangle$, add all rules of the form $\langle A, B_j, t \rangle \to a. \langle C, B_1, \beta_1 \rangle . \langle B_1, B_2, \beta_2 \rangle \ldots \langle B_{j-1}, B_j, \beta_j \rangle$ where the B_i can be any state in the machine

CFG from PDA

- For any PDA, let consider the language $\langle A, B, t\rangle = \left\{ x \in \Sigma^* \;\middle|\; \begin{array}{l} x \text{ is consumed in moving from state } A \text{ to state } B \text{ in} \\ \text{the machine, with the symbol } t \text{ being taken from} \\ \text{the top of the stack in the process} \end{array} \right\}$
- Now, given any PDA, construct a CFG which accepts the same language as follows:
  - the terminal symbols are just the input symbols of the PDA
  - the non-terminal symbols are all triples of the form $\langle A, B, t\rangle$
  - if S and F are the start and finish states respectively of the PDA, then the start symbol of the CFG is $\langle S, F, \epsilon\rangle$
  - The production rules are as follows:
    - For each transition of the form $\langle A, a, t\rangle \mapsto \langle C, \beta_1 \cdots \beta_j\rangle$, add all rules of the form $\langle A, B_j, t\rangle \to a.\langle C, B_1, \beta_1\rangle . \langle B_1, B_2, \beta_2\rangle \ldots \langle B_{j-1}, B_j, \beta_j\rangle$ where the B_i can be any state in the machine
    - For each transition of the form $\langle A, a, t\rangle \mapsto \langle C, \epsilon\rangle$, add all rules of the form $\langle A, C, t\rangle \to a$

Chomsky normal form (CmNF)

Chomsky normal norm requires that each production rule in the grammar $G = (V, \Sigma, R, S)$ satisfies one of the following forms:

$A \rightarrow BC$ where A, B, C are all non-terminals and neither B nor C is the start symbol

$A \rightarrow a$ where A is non-terminal and a is a terminal

– exactly a single non-terminal on the right of a production

$S \rightarrow \epsilon$ where S is the start symbol

– Only the start symbol may derive an empty string

Chomsky normal form (CmNF)

Chomsky normal norm requires that each production rule in the grammar $G = (V, \Sigma, R, S)$ satisfies one of the following forms:

$A \to BC$ where A, B, C are all non-terminals and neither B nor C is the start symbol

$A \to a$ where A is non-terminal and a is a terminal

– exactly a single non-terminal on the right of a production

$S \to \epsilon$ where S is the start symbol

– Only the start symbol may derive an empty string

Algorithm outline to convert CFG to CmNF

- Create a new start symbol S_0 with new rule $S_0 \to S$
- Remove non-terminals that only generate ϵ
- Remove unit rules (with only a single non-terminal on the right)
- Restructure rules with long right sides

Useless non-terminals

Identify and drop non-terminals that do not generate strings of terminals

nonEmptyNT $(G = \langle V, \Sigma, R, S \rangle)$

$V_0 \leftarrow \varnothing$

$V_1 \leftarrow V_0$

do

$V_0 \leftarrow V_1$

for $X \in V$ **do**

for $\langle X \rightarrow w \rangle \in R$ **do**

if $w \in (\Sigma \cup V_0)^*$ **then**

$V_1 \leftarrow V_1 \cup \{X\}$

while $(V_0 \neq V_1)$

return V_1 // NTs in V_1 generate some word

Non-terminals generating ϵ

Identify non-terminals that generate ϵ

epsilonNT $(G = \langle V, \Sigma, R, S \rangle)$

$V_\epsilon \leftarrow \varnothing$

do

$V_0 \leftarrow V_\epsilon$

for $X \in V$ **do**

for $\langle X \rightarrow w \rangle \in R$ **do**

if $w = \epsilon$ or $w \in (V_\epsilon)^*$ **then**

$V_\epsilon \leftarrow V_\epsilon \cup \{X\}$

while $(V_0 \neq V_\epsilon)$

return V_ϵ

This algorithm identifies non-terminals that can generate ϵ

Removing ϵ-productions

Want to remove rules of the form $A \rightarrow \epsilon$

- Identify V_ϵ, the set of all non-terminals generating ϵ
- If $S \in V_\epsilon$, introduce the productions: $S' \rightarrow S|\epsilon$
- Remove all productions of the form $A \rightarrow \epsilon$
- Let the resulting grammar be $G' = \langle V', \Sigma, R', S' \rangle$

  Now, if $B \in V_\epsilon$ and $\langle A \rightarrow \alpha B \gamma \rangle \in R$, the production $\langle A \rightarrow \alpha\gamma \rangle$ is also needed because $B \stackrel{*}{\rightarrow} \epsilon$ is no longer possible
- So, for every rule $\langle A \rightarrow X_1 X_2 \ldots X_m \rangle \in R'$, add the rules of the form $A \rightarrow \alpha_1 \ldots \alpha_m$ to the grammar, so that
  - If $X_i \notin V_\epsilon$, then $\alpha_i = X_i$
  - If $X_i \in V_\epsilon$, then either $\alpha_i = X_i$ or $\alpha_i = \epsilon$
  - Not all α_i's are to set as ϵ
  - The resuling grammar may have useless non-terminals which may be removed
  - The new grammar may be placed as $G'' = \langle V'', \Sigma, R'', S' \rangle$

Removing unit productions

- A unit production rule is of the form $A \to B$
- A pair of non-terminals A and B is a *unit pair* if $A \xrightarrow{*} B$
- At this stage ϵ-productions (where the producing non-terminal is not a start symbol) have been removed

  So, $A \xrightarrow{*} B \Rightarrow A \to C_1 \to C_2 \to \ldots \to C_m \to B$

unitPairs($G = \langle V, \Sigma, R, S \rangle$)

$R_u \leftarrow \{\langle A \to B \rangle \mid \langle A \to B \rangle \in R\}$

do

$R_0 \leftarrow R_U$

for $\langle A \to B \rangle \in R_u$ **do**

for $\langle B \to C \rangle \in R_u$ **do**

$R_u \leftarrow R_u \cup \{\langle A \to C \rangle\}$

while ($R_0 \neq R_u$)

return R_u

removeUnitRules($G = \langle V, \Sigma, R, S \rangle$)

$U \leftarrow$ unitPairs(G)

$R \leftarrow R \setminus U$

for $\langle A \to B \rangle \in U$ **do**

for $\langle B \to w \rangle \in R$ **do**

$R \leftarrow R \cup \{\langle A \to w \rangle\}$

return G

Further restructuring for CmNF

- Consider productions of the form $A_a \rightarrow a$ for each $a \in \Sigma$

Further restructuring for CmNF

- Consider productions of the form $A_a \rightarrow a$ for each $a \in \Sigma$
- Rewrite any production $A \rightarrow X_1 X_2 \ldots X_m, |m| > 1$ as replacing X_i with A_{a_i} if $X_i = a_i$

  Introduce $\langle A_{a_i} \rightarrow a_i \rangle$ to R (if not already present)

Further restructuring for CmNF

- Consider productions of the form $A_a \rightarrow a$ for each $a \in \Sigma$
- Rewrite any production $A \rightarrow X_1 X_2 \ldots X_m, |m| > 1$ as replacing X_i with A_{a_i} if $X_i = a_i$

  Introduce $\langle A_{a_i} \rightarrow a_i \rangle$ to R (if not already present)
- Rewrite any production $A \rightarrow B_1 B_2 \ldots B_k$ as

  $A \rightarrow B_1 C_1$

  $C_1 \rightarrow B_2 C_2$

  $\ldots$

  $C_{k-1} \rightarrow B_{k-1} C_k$

  update G appropriately

CmNF conversion example

Consider the CFG:

- $S \rightarrow aXbX$
- $X \rightarrow aY|bY|\epsilon$
- $Y \rightarrow X|c$

Nullable productions: X and also Y are nullable, so eliminate

- $S \rightarrow aXbX|abX|aXb|ab$
- $X \rightarrow aY|bY|a|b$
- $Y \rightarrow X|c$

Unit productions: Elimination $Y \rightarrow X$

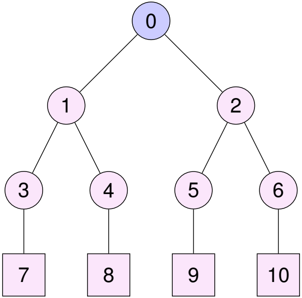
- $S \rightarrow aXbX|abX|aXb|ab$
- $X \rightarrow aY|bY|a|b$
- $Y \rightarrow aY|bY|a|b|c$

Later steps: Break up the RHSs of S, replace a by A, b by B and c by C where not as unit

- $S \rightarrow EF|AF|EB|AB$
- $X \rightarrow AY|BY|a|b$
- $Y \rightarrow AY|BY|a|b|c$
- $E \rightarrow AX$
- $F \rightarrow BX$
- $A \rightarrow a$
- $B \rightarrow b$

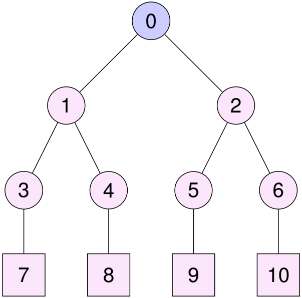
Also try: $S \rightarrow AbA$, $A \rightarrow Aa|\epsilon$

Pumping in CFLs



- CmNF parse tree of height m, has at most $3 \times 2^{m-1} - 1$ nodes
- Generated string length at most 2^{m-1}
- m non-terminals on any path

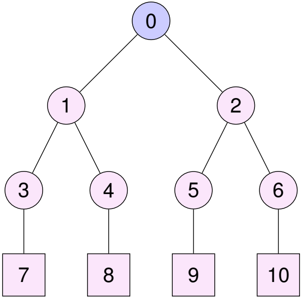
Pumping in CFLs



- CmNF parse tree of height m, has at most $3 \times 2^{m-1} - 1$ nodes
- Generated string length at most 2^{m-1}
- m non-terminals on any path

- Let G be a CmNF grammar with m non-terminal symbols

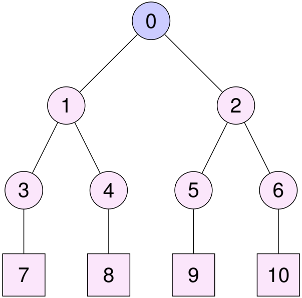
Pumping in CFLs



- CmNF parse tree of height m, has at most $3 \times 2^{m-1} - 1$ nodes
- Generated string length at most 2^{m-1}
- m non-terminals on any path

- Let G be a CmNF grammar with m non-terminal symbols
- consider a word $w \in L(G)$, such that $l = |w|$ and $l > 2^m$

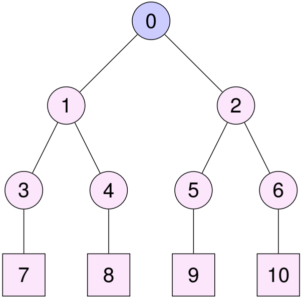
Pumping in CFLs



- CmNF parse tree of height m, has at most $3 \times 2^{m-1} - 1$ nodes
- Generated string length at most 2^{m-1}
- m non-terminals on any path

- Let G be a CmNF grammar with m non-terminal symbols
- consider a word $w \in L(G)$, such that $l = |w|$ and $l > 2^m$
- Any parse tree T for w (generated by G) must have a path from the root to some leaf with a repeated non-terminal on it

Pumping in CFLs



- CmNF parse tree of height m, has at most $3 \times 2^{m-1} - 1$ nodes
- Generated string length at most 2^{m-1}
- m non-terminals on any path

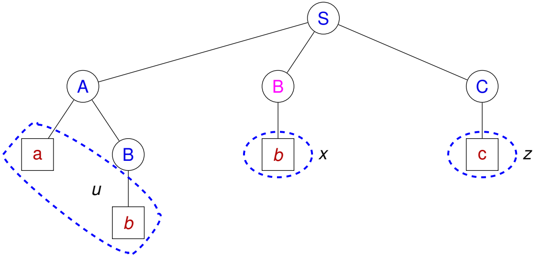
- Let G be a CmNF grammar with m non-terminal symbols
- consider a word $w \in L(G)$, such that $l = |w|$ and $l > 2^m$
- Any parse tree T for w (generated by G) must have a path from the root to some leaf with a repeated non-terminal on it
- Follows, because the height h will at least $m + 1$ ensuring that there is at least one path with more than m internal non-terminal nodes, ensuring a repetition

Pumping in CFLs (contd.)

- $S \to ABC \mid \epsilon$
  $A \to aB \mid a$
  $B \to bAS \mid b$
  $C \to c$

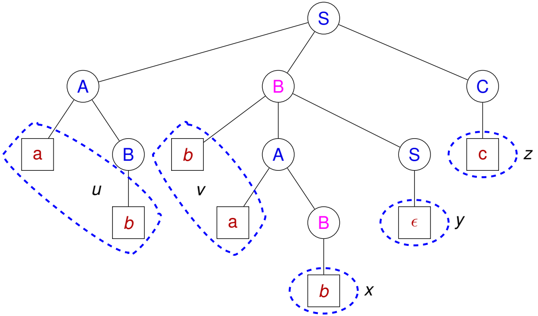
Pumping in CFLs (contd.)

- $S \rightarrow ABC \mid \epsilon$
  $A \rightarrow aB \mid a$
  $B \rightarrow bAS \mid b$
  $C \rightarrow c$
- Yield: $w_1 = uxz$, $w_1 \in L$



Pumping in CFLs (contd.)

- $S \rightarrow ABC \mid \epsilon$
  $A \rightarrow aB \mid a$
  $B \rightarrow bAS \mid b$
  $C \rightarrow c$
- Yield: $w_1 = uxz$, $w_1 \in L$
- Yield: $w_2 = uvxyz$, $w_2 \in L$
- B is repeated, further expansions of B can be pumped



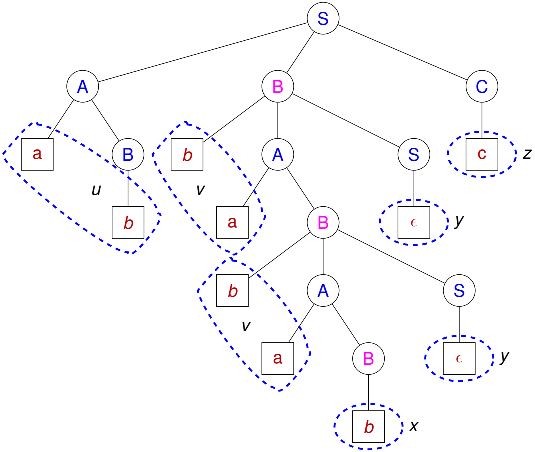
Pumping in CFLs (contd.)

- $S \rightarrow ABC \mid \epsilon$

  $A \rightarrow aB \mid a$

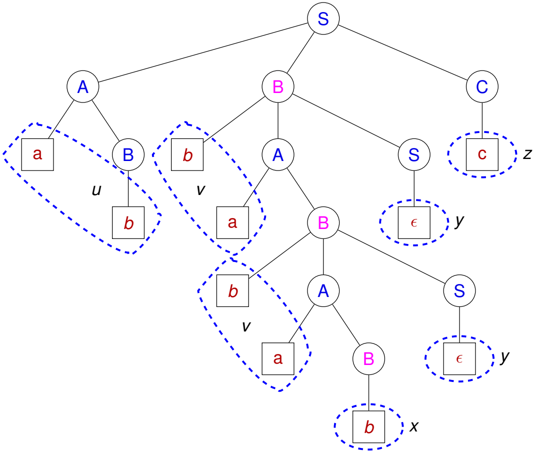
  $B \rightarrow bAS \mid b$

  $C \rightarrow c$
- Yield: $w_1 = uxz$, $w_1 \in L$
- Yield: $w_2 = uvxyz$, $w_2 \in L$
- B is repeated, further expansions of B can be pumped
- Yield: $w_3 = uv^2xy^2z$, $w_3 \in L$



Pumping in CFLs (contd.)

- $S \rightarrow ABC \,|\, \epsilon$
  $A \rightarrow aB \,|\, a$
  $B \rightarrow bAS \,|\, b$
  $C \rightarrow c$
- Yield: $w_1 = uxz$, $w_1 \in L$
- Yield: $w_2 = uvxyz$, $w_2 \in L$
- B is repeated, further expansions of B can be pumped
- Yield: $w_3 = uv^2xy^2z$, $w_3 \in L$
- $uv^ixy^iz \in L$, $i \geq 0$



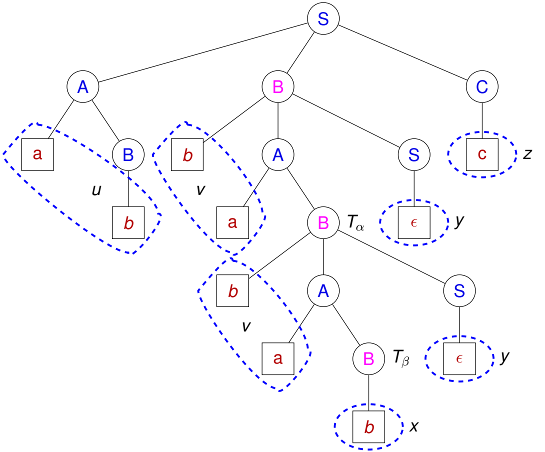
Pumping in CFLs (contd.)

- $S \rightarrow ABC \mid \epsilon$

  $A \rightarrow aB \mid a$

  $B \rightarrow bAS \mid b$

  $C \rightarrow c$
- Yield: $w_1 = uxz$, $w_1 \in L$
- Yield: $w_2 = uvxyz$, $w_2 \in L$
- B is repeated, further expansions of B can be pumped
- Yield: $w_3 = uv^2xy^2z$, $w_3 \in L$
- $uv^ixy^iz \in L$, $i \geq 0$



No repeated non-terminals beyond T_α, generated string of form: $w = qvxyr$

$h(T_\alpha) \leq m + 1$ and $|vxy| \leq 2^m$ (for CmNF)

Pumping lemma

Theorem (PL for CFG in CmNF)

Let G be a CmNF context-free grammar with m non-terminals, then given any word $w \in L(G)$ and $|w| > 2^m$, one can break w into five substrings $w = uvxyz$, such that for any $i \geq 0$, we have that $uv^ixy^iz \in L(G)$ with the following holding:

- *Strings v and y are not both empty (pumping leads to new words)*
- $|vxy| \leq 2^m$

Pumping lemma

Theorem (PL for CFG in CmNF)

Let G be a CmNF context-free grammar with m non-terminals, then given any word $w \in L(G)$ and $|w| > 2^m$, one can break w into five substrings $w = uvxyz$, such that for any $i \geq 0$, we have that $uv^ixy^iz \in L(G)$ with the following holding:

- *Strings v and y are not both empty (pumping leads to new words)*
- $|vxy| \leq 2^m$

Theorem (PL for CFG)

If L is a context-free language, then there is a number p (the pumping length) where if w is any string in L of length at least p, then w may be divided into five pieces $w = uvxyz$ satisfying the conditions:

- *for any $i \geq 0$, we have $uv^ixy^iw \in L$*
- $|vy| > 0$
- $|vxy| \leq p$

Pumping lemma (contd.)

Corollary (Long enough string of a CFL can be contracted)

If L is a CFL, then there is a number p such that, for any word $w \in L$, $|w| > p$, $w = utz$, where $|t| \leq p$, and there exists a strict contiguous substring t' of t such that $ut'z \in L$

Proof.

- Take p as the pumping length for the CFL
- Thus express w as $uvxyz$, st $|vy| > 0$, $|vxz| \leq p$, so that for any $i \geq 0$, $uv^ixy^iz \in L$
- In particular, for $i = 1$, $uvxyz \in L$
- Choose $t = vxy$ and $t' = x$
- Then $w = utz$, t' is a strict contiguous substring of t (since $|vy| > 0$), $|t| \leq p$, and $ut'z \in L$ □

Beyond CFLs

$L = \{a^n b^n c^n | n \geq 0\}$ is not context-free

- Let L be a CFL, so there is a $p > 0$, such that any word $w \in L, |w| > p$ can be pumped
- Consider $w = a^{p+1} b^{p+1} c^{p+1}$
- By PL, $w = uvxyz$, $|vxy| \leq p$
- By corollary, $w = a^{p+1} b^{p+1} c^{p+1} = utz$, $|t| \leq p$ and there is a strict contiguous substring t' of t such that $ut'z \in L$
- Since $|t| < p$, t cannot contain all the letters a, b, c
- Hence contraction to t' will reduce to at most two ($\{ab\}$, $\{bc\}$ or $\{ac\}$)
- Thus, $w' = ut'z \notin L$, contradticing the corollary
- Hence, L cannot be context free

Practice example of Pumping lemma

Show that the following languages are not context free.

1. $L_1 = \{a^i b^j c^k | k = max(i,j)\}$
2. $L_2 = \{w \in \{a,b,c\}^\star | n_a(w) = n_b(w) = n_c(w)\}$
3. $L_3 = \{xy | x, y \in \{a,b\}^\star, n_a(x) = n_a(y) \text{ and } n_b(x) = n_b(y)\}$
4. $L_4 =$
$\{t_1 \# t_2 \# \cdots \# t_k | k \geq 2, \text{ each } t_i \in \{a,b\}^\star, \text{ and } t_i = t_j \text{ for some } i \neq j\}$
5. $L_5 = \{w \in \Sigma^\star | \text{num}(1) = \text{num}(2) \wedge \text{num}(3) = \text{num}(4)\}$.

Closure properties

CFLs are not closed under intersection

$L_1 = \{a^n b^n c^m | m, n \geq 0\}$ and $L_2 = \{a^n b^m c^m | m, n \geq 0\}$ are CFL, however, $L_1 \cap L_2 = \{a^n b^n c^n | n \geq 0\}$ is not a CFL

CFLs are closed under union

Easily, shown by construction

CFLs are not closed under complementation

- For CFLs L_1 and L_2, $L_1 \cap L_2 = \overline{\overline{L_1} \cup \overline{L_2}}$
- If CFLs are closed under complementation, then for suitably chosen CFLs L_1 and L_2, $\overline{\overline{L_1} \cup \overline{L_2}}$ should be a CFL, but $L_1 \cap L_2$ will not be a CFL – a contradiction

Closure properties (contd.)

Although RLs are CFLs, their intersection with CFLs is closed under intersection

CFLs and RLs are closed under intersection

- Consider the FA that recognises the RL L_1
- Cross it with the controller of the PDA that recognises the CFL L_2
- Let the final states be those where both the automata accept their individual languages
- This automaton with accept the intersction of the two languages
- Intuitively, this is possible because the FA never needs to use the stack and so does not CmNFlict with the stack operations of the PDA

DCFL $\subseteq$ CFL

- Consider the CFL $\{x^n y^n \mid n \geq 0\} \cup \{x^n y^{2n} \mid n \geq 0\}$; is it deterministic?
- Suppose it is a DCFL then it has a corresponding DPDA M
- Create two copies of M as M_1 and M_2; call any two states "cousins" if they are copies of the same state in the original PDA. Now we construct a new PDA M_C as follows:
- The states of M_C is the union of the states in M_1 and M_2, q_I of M_C is q_I of M_1 F of M_C is F of M_2
- δ of M_1 and M_2 are altered as follows:
    - Change any transition originating from a final state in M_1 so that it now goes to its "cousin" state in M_2
    - Change all those 'y' transitions which cause a move into some state from M_2 into 'z' transitions
- This is a PDA over $\{x, y, z\}$

DCFL $\subseteq$ CFL (contd.)

- Consider its actions on an input of $x^k y^k z^k$ for some fixed $k \geq 0$
- Transitions will happen *deterministically* from q_l of M_1 while consuming $x^k y^k$
- M_1 can now also go on to accept k more copies of 'y'
- The next k more copies of 'z' will take the m/c through states of M_2
- Thus the constructed PDA accepts the language $\{x^n y^n z^n \mid n \geq 0\}$ – a contradiction
- So, M cannot be a DPDA, as assumed earlier

Cocke-Kasami-Younger parsing for CmNF grammar

A dynamic programming approach to parsing

- Consider every possible consecutive subsequence of terminals and non-terminals $K \in T[i,j]$ if the sequence of terminals from i to j can be produced by the non-terminal K
- No tracing through unit productions because of CmNF
- Once sequences of length 1 are considered, go on to sequences of length 2, and so on
- For subsequences of length 2 and greater, consider every possible partition of the subsequence into two parts and check to see if there is some production $A \rightarrow BC$ such that B matches the first part and C matches the second part
  if so, it record A as matching the whole subsequence
- Once this process is completed, the sentence is recognized by the grammar if the entire string is matched by the start symbol