Theory of Computation (CSC208)

Context-Free Languages

Ayan Das

Context-free languages

Context-free languages

- Just like the left-linear or right-linear grammar characterizes the regular languages.
- The *context-free grammars* characterizes the Context-free languages.
- In fact, the context-free languages are defined in terms of context-free grammars.

Context-free language

Suppose I have a string of the form $\alpha_1 X \alpha_2$ and $X \to \gamma$ is a production, then we get $\alpha_1 \gamma \alpha_2$. $G = (V, T, S, P)$ is a *context-free grammar* if each element of P is of the form $X \to \alpha$, where $X \in V$ and $\alpha \in (V \cup T)^*$

Formally, a language L is context-free if there exists a context-free grammar G such that $L = L(G)$.

Context-free grammar: Example

$G = (V, T, S, P)$

$V = \{S, A, B\}$, $T = \{a, b\}$

P:
$$
\begin{aligned}
S &\to aB \\
S &\to bA \\
A &\to aS \\
A &\to bAA \\
A &\to a \\
B &\to bS \\
B &\to aBB \\
B &\to b
\end{aligned}
$$

Context-free grammar: Example

$G = (V, T, S, P)$

$V = \{S, A, B\}$, $T = \{a, b\}$

P:
$S \to aB$
$S \to bA$
$A \to aS$
$A \to bAA$
$A \to a$
$B \to bS$
$B \to aBB$
$B \to b$

Rightmost derivation

$S \Rightarrow a\underline{B} \Rightarrow aaB\underline{B} \Rightarrow aaBb\underline{S}$

$\Rightarrow aaBbb\underline{A} \Rightarrow aa\underline{B}bba$

$\Rightarrow aabbba$

Context-free grammar: Example

$G = (V, T, S, P)$

$V = \{S, A, B\}, T = \{a, b\}$

P:
$S \to aB$
$S \to bA$
$A \to aS$
$A \to bAA$
$A \to a$
$B \to bS$
$B \to aBB$
$B \to b$

Leftmost derivation

(1) $S \Rightarrow aB \Rightarrow aa\underline{B}B \Rightarrow aab\underline{B}$
$\Rightarrow aabb\underline{S} \Rightarrow aabbb\underline{A} \Rightarrow aabbba$

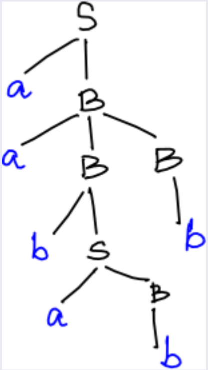
(2) $S \Rightarrow a\underline{B} \Rightarrow aa\underline{B}B \Rightarrow aab\underline{S}B$
$\Rightarrow aaba\underline{B}B \Rightarrow aabab\underline{B} \Rightarrow aababb$

Parse trees

Parse tree

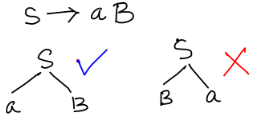
The graphical way of representing how a string is derived. It is rooted tree, in which a node can be a terminal or a non-terminal.

- The root of the tree is the *start variable*.
- If a node is labelled with a terminal symbol then it is a **leaf node**.
- Internal nodes are always labelled with **non-terminals**.
- An internal node will have as children the symbols of the right-hand side of a production whose left-hand side is that symbol.

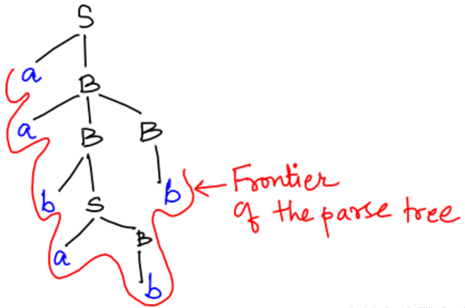


Parse tree

- The *ordering* is essential.



- The sequence should be the right-hand side of the production when read from left-to-right.
- **Frontier (yield) of a tree:** Sequence of leaf nodes from left to right.



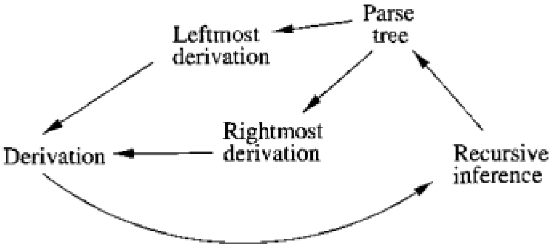
Parse tree

- $G_1 = (\{S\}, \{0, 1\}, S, \{S \rightarrow 0S1, S \rightarrow 01\})$
  $L(G_1) = \{0^n 1^n | n \geq 1\}$
  Show that
  1. $L(G_1) \subseteq \{0^n 1^n | n \geq 1\}$
  2. $\{0^n 1^n | n \geq 1\} \subseteq L(G_1)$
- **Proof:** File *gramlangeq1.pdf*

Equivalent statements

Given a grammar $G = (V, T, S, P)$ he following statements are equivalent

1. The recursive derivation procedure determines that the terminal string w is in the language of the variable A.
2. $A \overset{*}{\Rightarrow} w$
3. $A \underset{lm}{\overset{*}{\Longrightarrow}} w$
4. $A \underset{rm}{\overset{*}{\Longrightarrow}} w$
5. There is a parse tree with root at A and yield w.



Simplification of CFGs

- $G = (V, T, S, P)$, $G' = (V', T, S, P')$
- G' is called a **simplification** of G such that $L(G) = L(G')$ and G and G' are equivalent.
- The process of simplification will help us to transform a grammar to a standard form and that will help us to prove certain properties of CFGs which might otherwise be difficult.

Steps of simplification

1. Removal of useless symbols.
2. Identification and removal of non-generating non-terminals.
3. Removal of ϵ-productions.
4. Removal of unit productions.

Removal of useless symbols

Removal of useless symbols

- An **useless symbol** is a symbol that does not take part in the derivation of any string in the language.
- A non-terminal A can be useless in two ways
  1. If there is no $w \in \Sigma^*$ s.t. $A \stackrel{*}{\Rightarrow} w$ (*Non-generating symbols*)
  2. If the non-terminal or the terminal symbol cannot be reached from the start symbol S using the production rules of the grammar. (*Unreachable symbols*)
- **Case 1:** Assume that the derivation
  $S \stackrel{*}{\Rightarrow} \alpha A \beta \Rightarrow \cdots \Rightarrow x \in T^*$

  Eventually there has to be a derivation by which A can be converted to a sequence of terminals w i.e. $A \stackrel{*}{\Rightarrow} w \ \ w \in T^*$

  Example: $S \rightarrow aSb|\epsilon|A, A \rightarrow aA$
- **Case 2:** Suppose we never get the situation $S \stackrel{*}{\Rightarrow} \alpha A \beta$ for any $\alpha, \beta \in (V \cup T)^*$

  Example: $S \rightarrow aS|A, A \rightarrow a, B \rightarrow ab$

Identification of unreachable symbols

- $\mathcal{R} \rightarrow$ the set of reachable symbols
- We define $\mathcal{R}$ by induction.

1. **Base case:** $\mathcal{R} = \{S\}$
2. **Induction step:**
    1. Let $A \rightarrow \alpha \in P$ and A is reachable i.e. $A \in \mathcal{R}$. Then every symbol in α is also reachable.
3. If $B \in \alpha$, then update $\mathcal{R} = \mathcal{R} \cup B$.
4. Repeat the step over all productions starting from $\mathcal{R} = \{S\}$ until $\mathcal{R}$ stops growing.

Finally **Unreachable symbols**$=(V \cup T)^* \setminus \mathcal{R}$.

Removal of productions with unreachable symbols: $P' = P \setminus \{A \rightarrow \alpha |$ Any element of unreachable symbol occurs in $A \rightarrow \alpha$ i.e. A itself is unreachable or any element of α may be unreachable.$\}$

Identification of non-generating non-terminals

- **Step 1:** Identify the set of $\mathcal{G}$ generating non-terminals.

  A is generating if $A \Rightarrow w \in T^*$.

  $\mathcal{G}$ is defined inductively.

- **Base case:** Suppose there is a production of the form $A \to w$, $w \in T^*$.

  1. Put in $\mathcal{G}$, all $A \in V$ s.t. $A \to w$, $w \in T^*$.

- **Induction step:** Suppose we have a production $B \to \alpha$ where every non-terminal in α is already in $\mathcal{G}$, and if $B \notin \mathcal{G}$, then add B to $\mathcal{G}$.

$B \to b\,C\,A$ $\qquad x_1, x_2 \in T^*$

$x_1 \qquad x_2$

- Then the set of **non-generating** non-terminals will be $V \setminus \mathcal{G}$.

- Starting from $G = (V, T, S, P)$ after identification of the non-generating non-terminals, we can obtain a simplified grammar $G' = (V', T, S, P')$, assuming that S is generating, otherwise $L(G) = \phi$.
- V' will change if the non-generating non-terminals are removed.
- T would not change since the steps remove only non-terminals.
- P would change.

Sequence of removal

Example

$$S \to AB$$
$$S \to a$$
$$A \to a$$

Option 1

1. Remove unreachable symbols.
2. Remove non-generative non-terminals.

Sequence of removal

Example

$S \to AB$
$S \to a$
$A \to a$

Option 1

1. Remove unreachable symbols.
2. Remove non-generative non-terminals.

- Apply **Option 1**
  - $\{S, A, B\}$ are reachable.
  - B is found to be unproductive.
  - Remove $S \to AB$
  - A becomes unreachable.

Option 2

1. Remove non-generative non-terminals.
2. Remove unreachable symbols.

Sequence of removal

Example

$S \rightarrow AB$
$S \rightarrow a$
$A \rightarrow a$

Option 1

1. Remove unreachable symbols.
2. Remove non-generative non-terminals.

Option 2

1. Remove non-generative non-terminals.
2. Remove unreachable symbols.

- Apply **Option 1**
  - $\{S, A, B\}$ are reachable.
  - B is found to be unproductive.
  - Remove $S \rightarrow AB$
  - A becomes unreachable.
- Apply **Option 2**
  - Productive states $\{S, A\}$.
  - Remove $S \rightarrow AB$
  - A becomes unreachable from S.
  - Remove $A \rightarrow a$

Sequence of removal

Problem

Removal of non-generating states may introduce some unreachable states.

Why is the Option 2 correct?

1. **Choice 2** can go wrong if a state A is initially generating but after the removal of some unreachable states, A becomes non-generating.
2. Initially let, $A \Rightarrow \cdots \Rightarrow \alpha B \beta \Rightarrow \cdots \Rightarrow x \in T^*$
3. $\alpha B \beta$ can be any of the sentential forms.
4. Is it possible that B is unreachable?

Sequence of removal

Problem

Removal of non-generating states may introduce some unreachable states.

Why is the Option 2 correct?

1. **Choice 2** can go wrong if a state A is initially generating but after the removal of some unreachable states, A becomes non-generating.
2. Initially let, $A \Rightarrow \cdots \Rightarrow \alpha B \beta \Rightarrow \cdots \Rightarrow x \in T^*$
3. $\alpha B \beta$ can be any of the sentential forms.
4. Is it possible that B is unreachable?
5. Since $A \overset{*}{\Rightarrow} x$, so A is generating.
6. Since B is reachable from A so B cannot be unreachable.
7. Thus removal of unreachable states does not generate any non-generative state.

Remove ϵ-production

ϵ-production

- $A \rightarrow \epsilon$ is called an ϵ-production.
- In $G = (V, T, S, P)$, if $S \overset{*}{\Rightarrow} \epsilon$, it is clearly not possible to obtain a grammar G_1 without any ϵ-production such that $L(G) = L(G_1)$.
- Given a grammar G, to obtain G_1 such that $L(G_1) = L(G) \setminus \{\epsilon\}$.

Remove ϵ-production

Step 1: In the given grammar G identify all nullable non-terminals.

We say A is nullable if $A \stackrel{*}{\Rightarrow} \epsilon$. If $A \to \epsilon \in P$ then A is nullable.

Let $\mathcal{N}$ denote the set of all nullable non-terminals.

Base case: $\mathcal{N} = \{ A \in V \mid A \to \epsilon$ is in $P \}$

Inductive case: Suppose $A, B, C \in \mathcal{N}$ and $D \to ABC$

Then D is nullable.

$\therefore$ If $D \notin \mathcal{N}$, then add D to $\mathcal{N}$.

Repeat the inductive step until no more symbols are added to $\mathcal{N}$.

Removal of ϵ-production

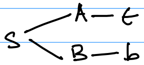
Removal of ϵ-production

Given $G = (V, T, S, P)$, $\mathcal{N}$ be the set of *nullable non-terminals* of G.
Goal: To obtain G_1 without ϵ-productions such that $L(G_1) = L(G) \setminus \{\epsilon\}$.

We eliminate from P all ϵ-productions.

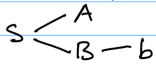
Example

$S \to AB$
$A \to \epsilon$
$B \to b$



If we remove $A \to \epsilon$ form P then A becomes unproductive and block generation of 'b'.

i.e;



This can be overcome by adding some productions.

If $A \to X_1 X_2 \cdots X_K$ is in P and of these X_i's some/all of them nullable, then we shall add productions of the form $A \to Y_1 Y_2 \cdots Y_K$ where $Y_i = \{X_i$ or ϵ if X_i is nullable$\}$

Remove ϵ-production

Example 1

$S \to AB$
$A \to \epsilon$
$B \to b$

$\Rightarrow$

$\mathcal{N} = \{A\}$

$S \to AB \mid B$
$A \to \epsilon$
$B \to b$

$\Rightarrow$

$S \to B$
$B \to b$

Example 2

$S \to ABaC$
$A \to BC$
$B \to b \mid \epsilon$
$C \to D \mid \epsilon$
$D \to d$

$\Rightarrow$

$\mathcal{N} = \{B, C, A\}$

$S \to ABaC \mid ABa \mid AaC \mid BaC \mid aC \mid Aa \mid Ba \mid a$
$A \to BC \mid B \mid C$
$B \to b$
$C \to D$
$D \to d$

Remove unit production

- Example

  $A \rightarrow B \;\; A, B \in V$

  $A \rightarrow A$ Can be easily removed

Remove unit productions

$A \to B$
$B \to A$
$A \overset{*}{\Rightarrow} B$ (Find all such derivations and mark them as unit production)

In the new grammar G_1,

① Add all non-unit productions in P

② For all productions of the type $A \overset{*}{\Rightarrow} B$ add to P_1, production of the form
$A \to y_1 | y_2 | \dots | y_n$
where,
$B \to y_1 | y_2 | \dots | y_n$

Example

$S \to Aa \mid B$

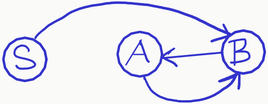
$B \to A \mid bb$

$A \to a \mid bc \mid B$

$S \to Aa \mid bb \mid a \mid bc$

$B \to bb \mid a \mid bc$

$A \to a \mid bc \mid bb$



$S \xRightarrow{*} B$

$B \xRightarrow{*} A$

$A \xRightarrow{*} B$

$S \xRightarrow{*} A$

Safe ordering

1. ϵ-production removal
2. Unit production removal
3. Useless production removal
   1. Removal of non-generative symbols.
   2. Removal of unreachable symbols.

Chomsky normal form (CNF)

Chomsky normal form

A context-free grammar G is said to be in CNF if every production of G is one of the two forms:

1. **Type I:** $A \rightarrow a \; [A \in V, a \in T]$
2. **Type II:** $A \rightarrow BC \; [A, B, C \in V]$

Further G does not have any useless symbols.

Every grammar G such that $L(G)$ is not empty and does not contain ϵ can be converted to CNF.

Chomsky normal form

- Assume G does not have any ϵ-production, unit production or useless symbol.
- I have productions of the form
  1. $A \rightarrow a$ $[A \in V, a \in T]$
  2. $A \rightarrow \alpha$ $[A \in V]$ and α has 2 or more symbols.
- Allowed productions
  1. $A \rightarrow a$ $[A \in V, a \in T]$
  2. $A \rightarrow \alpha$ $[A \in V]$ and α has *exactly* 2 non-terminals.
- Challenge
  - Converting the productions with *two or more symbols* in grammar G to the form $A \rightarrow BC$.

Conversion

Broad steps

1. **STEP 0**: REMOVE ALL ϵ-PRODUCTIONS, UNIT PRODUCTIONS AND USELESS SYMBOLS BEFORE PROCEEDING FURTHER
2. Proceed to the following steps.

Conversion

- **Step 1:** Ensure that the r.h.s. of every production should consist of
  - a single terminal, or,
  - two or more nonterminals
- **Procedure:** Whenever you find a terminal symbol in the r.h.s. of a production where the number of symbols in the r.h.s. is greater than 1 (e.g. $A \rightarrow Bc$)
  1. Introduce a new non-terminal (C_1)
  2. Replace the terminal with the new non-terminal (c by C_1 to get $A \rightarrow BC_1$)
  3. Add a new production of the form $C_1 \rightarrow c$
  4. Ensure that you use the same non-terminal (say C_1) to replace (say c) in future.

Conversion

- **Step 2:** Ensure that the r.h.s. of the productions consist of exactly two non-terminals except for the productions with a single terminal in the r.h.s.
  $A \rightarrow BC$ [Accepted]
- Consider the production
  $A \rightarrow C_1 C_2 \ldots C_k$ The problem is to
  1. eliminate all such productions where $k > 2$
  2. have equivalent productions of the form $A \rightarrow BD$.
  3. Solution: Introduce new non-terminals

$A \rightarrow C_1 C_2 C_3 C_4$

$A \rightarrow D_1 C_4$

$D_1 \rightarrow D_2 C_3$

$D_2 \rightarrow C_1 C_2$

OR,

$A \rightarrow C_1 D_1$

$D_1 \rightarrow C_2 D_2$

$D_2 \rightarrow C_3 C_4$

Conversion

$S \to ABa$

$A \to aab$

$B \to Ac$

Step 1

$S \to ABA_1$

$A \to A_1 A_1 B_1$

$B \to AC_1$

$A_1 \to a$

$B_1 \to b$

$C_1 \to c$

Step - 2

$S \to AD_1$

$D_1 \to BA_1$

$A \to A_1 D_2$

$D_2 \to A_1 B_1$

$B \to AC_1$

$A_1 \to a$

$B_1 \to b$

$C_1 \to c$

Application of CNF

1. Pumping lemma for context-free languages.
2. An efficient algorithm to prove whether a string belongs to the language of a grammar.

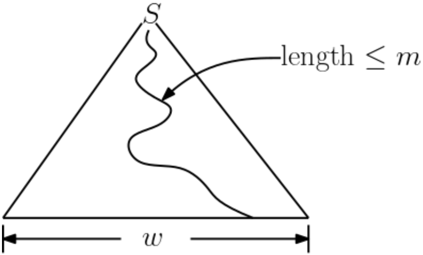
From now onwards we shall assume that the grammar we are talking about is in CNF.

Applications of Chomsky normal form

Property of CNF

Claim

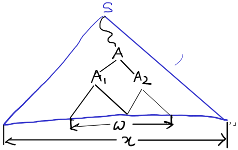
Suppose a grammar G is in CNF. Consider a derivation tree of G generating the terminal string w such that no path in the derivation tree has a length greater than m, then the length of $w \leq 2^{m-1}$



If we have a bound on the largest path in the derivation tree of a string in a CNF grammar then we can have a bound on the length of the string itself.

Proof is by induction on m.

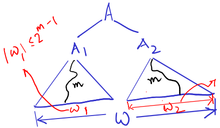
$|w| \leq 2^{m-1}$



Base: $m = 1$ $S \longrightarrow a \Rightarrow$ S | a A | a

Path length(m) = 1, so string length $|w| = 2^{1-1} = 1$

Induction step: The grammar is in CNF. The productions are of the form $A \to BC$ or $A \to a$. Derivation tree starts with A. Suppose we have two trees rooted at A_1 and A_2.



Let the longest path in either of the derivations with A_1 or A_2 as roots be m or less. So the lengths of the substrings w_1 and w_2

$|w_1| \leq 2^{m-1}$, $|w_2| \leq 2^{m-1}$

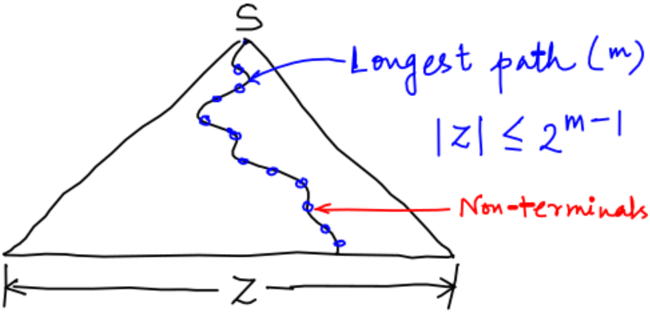
So the length of the longest path in the derivation with A is root will be m+1 or less. So, $|w| \leq 2^{m-1+1} = 2^m$

Pumping lemma for context-free languages

Pumping lemma for context-free language

Statement

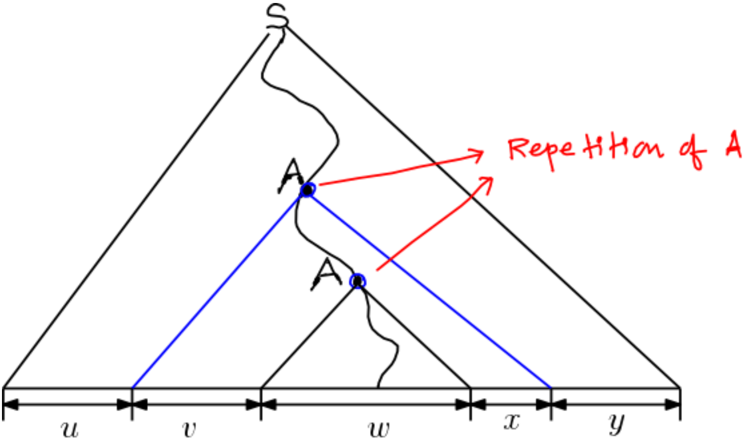
Suppose G is a CNF grammar. Let T be the derivation tree of G, such that T has no path longer than k (path from root to leaf). Then the string generated is of length $\leq 2^k - 1$



Pumping lemma for context-free language

- Let $L(G)$ be the language of G.
- Let G has m non-terminals.(We do not know m. It depends on the language. Assuming that a CNF grammar with m non-terminals exists for $L(G)$).
- Consider a derivation tree in G of string z in L of length 2^m or more.
- Then the length of the longest path in the derivation tree of z should be $m + 1$ or more.

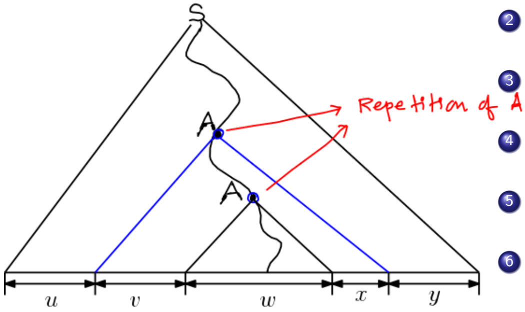
Pumping lemma for context-free language



- Maximum path length between repetitions is at most $m + 1$.

Pumping lemma for context-free language

- If we consider the longest path, start from the leaf, and stop when a non-terminal repeats for the first time, we have the following rules.



1. $A \xRightarrow{*} w$
2. $A \xRightarrow{*} vwx$
3. $S \xRightarrow{*} uvwxy$
4. $S \xRightarrow{*} uAy$
5. $S \xRightarrow{*} vAx$
6. $S \xRightarrow{*} uAy \xRightarrow{*} uvAxy \xRightarrow{*} uv^iwx^iy$

Pumping down means $S \xRightarrow{*} uAy \xRightarrow{*} uwy$

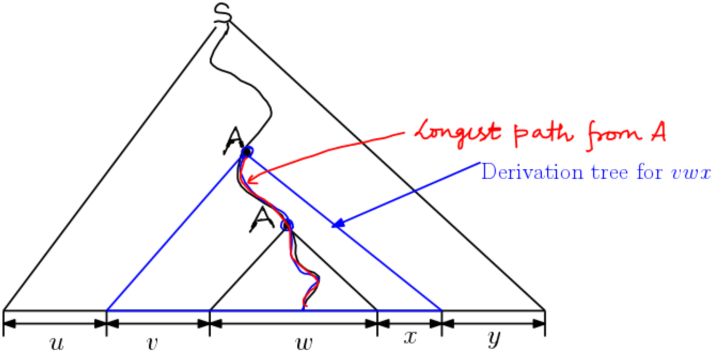
Pumping lemma for context-free language

Why vx is not empty?

- $|vx| = 0$ implies $A \stackrel{*}{\Rightarrow} A$ which is not possible because G is in CNF.
- For this to happen we need to have ϵ-productions, unit productions or both.

Pumping lemma for context-free language

Why $|vwx| \leq n$?



- The A generating vwx is the first repetition of A.
- The maximum path length after which repetition can occur is $m + 1$.
- The generated string cannot be of length greater than $n = 2^m$, where n is the *pumping constant*.

Pumping lemma for context-free languages

Statement

Let L be a context-free language. Then there is a constant n that depends only on L such that for all $z \in L$, $|z| \geq n$, there exists u, v, w, x, y satisfying:

1. $z = uvwxy$
2. $|vx| \neq \epsilon$ (Both v and x cannot be empty)
3. $|vwx| \leq n$
4. $\forall i \geq 0,\ uv^i wx^i y$

Example: Application of pumping lemma for CFL

$L = \{0^n 1^n 2^n | n \geq 0\}$
Prove that L is not context-free.

- We shall do this by adversarial game playing and contradiction.
- Let k be the constant for L (Given by the adversary).
- Choose a string $z = 0^k 1^k 2^k$ of length $3k$ (Chosen by you).
- The decomposition u, v, w, x, y will be given by the strong adversary.
- No matter whatever decomposition is given by the adversary we should be able to contradict the conditions of the PL for CFL.

Proof for $0^n1^n2^n$

1. Let k be the constant chosen by the adversary as *pumping constant*.

Proof for $0^n1^n2^n$

1. Let k be the constant chosen by the adversary as *pumping constant*.
2. We choose the string $0^k1^k2^k$.
3. What are the possible decompositions that the adversary can choose given that $|vwx| \leq k$ and $|vx| > 0$?

Proof for $0^n1^n2^n$

1. Let k be the constant chosen by the adversary as *pumping constant*.
2. We choose the string $0^k1^k2^k$.
3. What are the possible decompositions that the adversary can choose given that $|vwx| \leq k$ and $|vx| > 0$?
4. - It can be within 0s.
   - It can straddle some 0s and some 1s.
   - It can be within 1s.
   - It can straddle some 1s and some 2s.
   - It can be within 2s.

If $i = 0$, then $z = uwy$. However, the lemma says that both v and x cannot be simultaneously empty.

Proof for $0^n1^n2^n$: vwx is completely in $0, 1$ or 2

1. Suppose vwx is completely in 0.
2. $|vx| > 0$, so vx has some $0's$.
3. Upon removal of v and x, the number of 0s in uwy is less than k.

Same reasoning holds for vwx is completely in 1s or 2s.

Proof for $0^n1^n2^n$: vwx straddles 0s and 1s, or, 1s and 2s

1. Suppose vwx straddles 0s and 1s.
2. The number of 0s and 1s can be different in vwx but there are no 2s.
3. v and x cannot be simultaneously empty.
4. Now if you remove v and x, the number of 0s, 1s or both will be less in the string uwy but the number of 2s will be k.

Membership in CFL and Ambiguity

Parse trees

Membership problem

Given a string w, determine whether $w \in L(G)$ where G is a context-free language.

Top-down solution strategy

Start *deriving* from the start symbol S until the parse tree yields w.

Example: Consider the grammar G

$S \rightarrow SS|aSb|bSa|\epsilon$

String $w =$"aabb" (How do you derive this?)

Parse trees

Membership problem

Given a string w, determine whether $w \in L(G)$ where G is a context-free language.

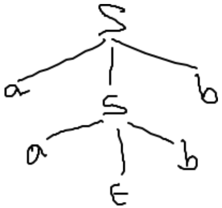
Top-down solution strategy

Start *deriving* from the start symbol S until the parse tree yields w.

Example: Consider the grammar G

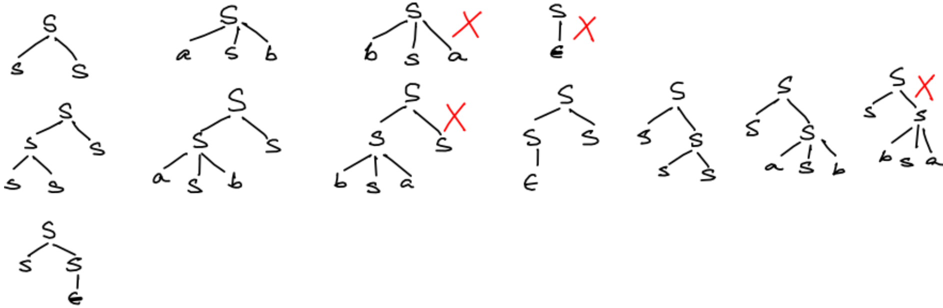
$S \rightarrow SS | aSb | bSa | \epsilon$

String $w =$ "aabb" (How do you derive this?)

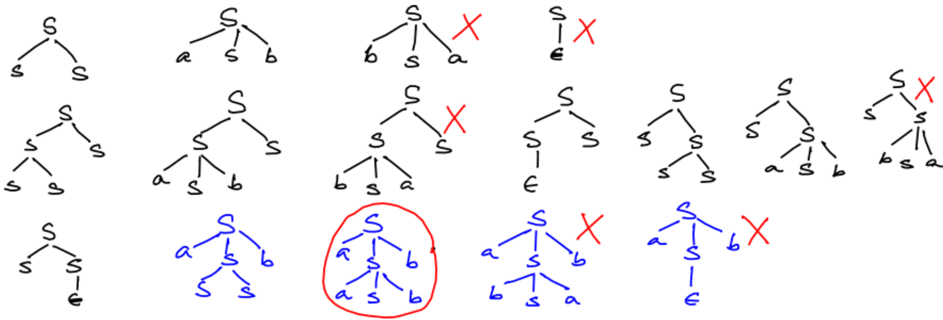


Top-down derivation

Top-down derivation



Top-down derivation

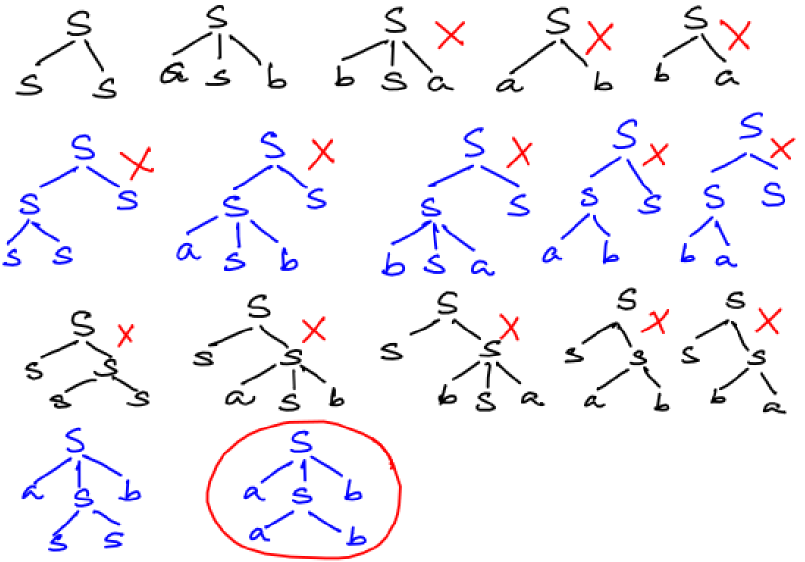


Problems with the top-down approach

1. **Tedious:** Cannot used for efficient parsing.
2. **Non-termination:** May not terminate for string $w \notin L(G)$.

   This problem can be largely eliminated by removing ϵ transition. This ensures that if the total number of symbols in the *frontier* of the derivation tree is greater than the length of w then the tree is not a valid parse tree for w.

Top-down derivation



Grammar in CNF (more restriction)

A grammar in CNF considers both the length and the number of terminals in the derivation.

$$S \rightarrow SS|AS_1|BS_2|AB|BA$$

$$S_1 \rightarrow SB$$

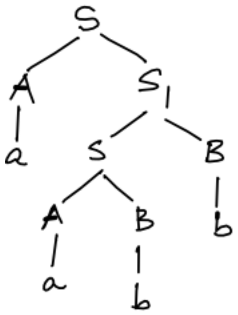
$$S_2 \rightarrow SA$$

$$A \rightarrow a$$

$$B \rightarrow b$$

- The derivation cannot exceed $2|w|$ derivation steps.
- Within that many rounds, either
  - the successful parse can be generated, or,
  - it will be clear that w cannot be generated by G or $w \notin L(G)$.
- Each derivation step in this case either
  - adds a non-terminal to the derivation or,
  - adds a terminal to the derivation.

Top-down derivation



$$
\begin{aligned}
& S \\
\Rightarrow\ & A S_1 \\
\Rightarrow\ & A S B \\
\Rightarrow\ & a S B \\
\Rightarrow\ & a A B B \\
\Rightarrow\ & a a B B \\
\Rightarrow\ & a a b B \\
\Rightarrow\ & a a b b
\end{aligned}
$$

- Adding a non-terminal increases the length of the *sentential form*.
- Adding a terminal for a non-terminal does not change the length.
- After $2|w|$ steps ($|w|$ for adding non-terminals and $|w|$ for adding terminals) either the length of the sentential form will exceed $|w|$.

Ambiguity

Number of derivation trees

Question

If a given string in a grammar has more than one derivation, then which one to take?

Ambiguous grammar

A context-free grammar G is said to be **ambiguous** if there exists some $w \in L(G)$ that has at least two **distinct derivation trees**.
Example:$S \rightarrow aSb|SS|\epsilon$ and $w =$"aabb"

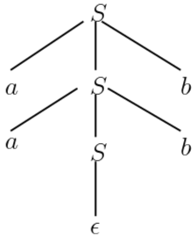
Number of derivation trees

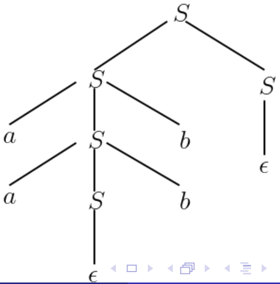
Question

If a given string in a grammar has more than one derivation, then which one to take?

Ambiguous grammar

A context-free grammar G is said to be **ambiguous** if there exists some $w \in L(G)$ that has at least two **distinct derivation trees**.
Example: $S \rightarrow aSb|SS|\epsilon$ and $w =$ "aabb"

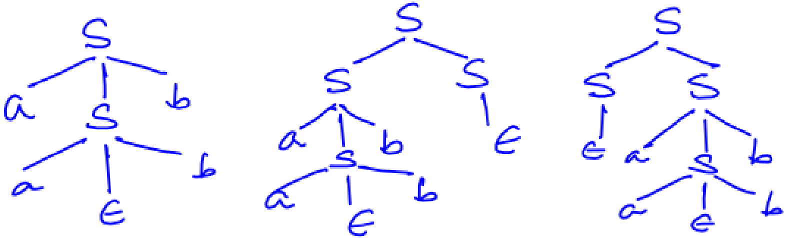




Ambiguous grammar

$S \rightarrow aSb \mid SS \mid \epsilon$

$w = aabb$



Ambiguous grammar

$$E \to E + E \mid E * E \mid (E) \mid I$$

$$I \to a \mid b \mid c$$

$$w = a + b$$

$$E \Rightarrow E + E \Rightarrow I + E \Rightarrow a + E$$
$$\Rightarrow a + I \Rightarrow a + b$$

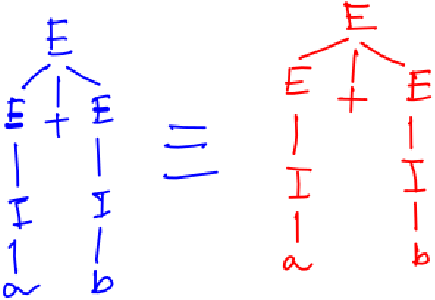
$$E \Rightarrow E + E \Rightarrow E + I \Rightarrow E + b$$
$$\Rightarrow I + b \Rightarrow a + b$$

Ambiguous grammar

$$E \rightarrow E + E \mid E * E \mid (E) \mid I$$

$$I \rightarrow a \mid b \mid c$$

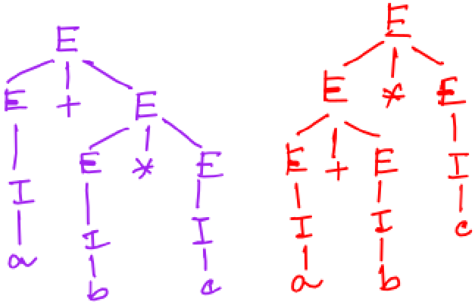
$$w = a + b$$



$$E \rightarrow E+E \mid E*E \mid (E) \mid I$$

$$I \rightarrow a \mid b \mid c$$

$$x = a + b * c$$



Causes of ambiguity

Ambiguity is due to

1. not respecting the precedence of operators.
2. A sequence of identical operators can group either from left to right or from right to left and order does not matter.

- There are no algorithms to check if context-free grammar is ambiguous.
- **Inherently ambiguous language:** A language for which all CFGs are ambiguous is called an **inherently ambiguous language**. (*Introduction to Automata Theory, Languages and Computation - Second Edition (Section 5.4.4)*)

Removal of ambiguity

- **Enforce precedence:** Introduce several different variables, such that each variable represents the expressions that share the same "binding strength".

Example (Imposition of precedence)

$x = a + b * c$

- Identifiers a, b and c are fundamental units.
- If two identifiers are connected by $*$ symbol ($b * c$) then the $+$ symbol cannot dissociate them i.e. $a+$ cannot take out b from the expression $b * c$.

Removal of ambiguity

1. A **factor** is an expression that cannot be broken apart by any adjacent operator. The only factors in our expression language are:
    1. **Identifier:** Letters of an identifier.
    2. **Any parenthesized expression:** Irrespective of whatever is appearing inside e.g.
    In $(a+b)*c$, $a+b$ will be operated first and not $b*c$.
2. A **term** is an expression that cannot be broken by a $+$ operator. Since in our case $+$ and * are the only operators. So a *term* corresponds is a product of several *factors*.

    $c*a*b = (c*a)*b = c*(a*b)$ (By left associativity)

    $c+a*b = c+(a*b) \neq (c+a)*b$

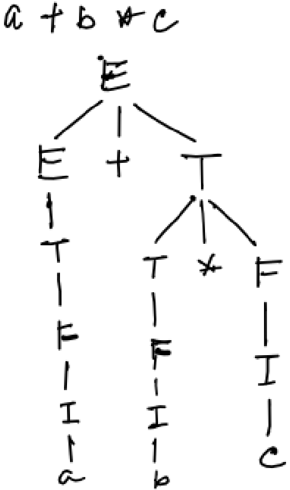
    $c+(a+b) = (c+a)+b$
3. An **expression** is any possible expression that can be broken by * or $+$ operator.

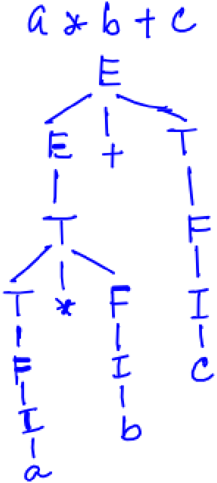
Removal of ambiguity

$$
\begin{aligned}
I &\rightarrow a|b|c \\
F &\rightarrow I|(E) \\
T &\rightarrow F|T * F \\
E &\rightarrow T|E + T
\end{aligned}
$$

1. The first production corresponds to the *identifiers*.
2. The second production corresponds to the *factors* (identifiers or parenthesized symbols).
3. The third production corresponds to *terms* that are factors or products of several factors.
4. The fourth production corresponds to the *expressions*.

Ambiguous grammar





Leftmost derivations as a way to express ambiguity

- In case of an *unambiguous grammar*, for a given string w, the *leftmost derivation* (and *rightmost derivation*) will be unique.
- If we can have two distinct leftmost (or rightmost) derivations for a string, then we can call the grammar *ambiguous*.

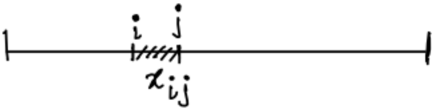
CYK Parsing

Testing membership in CFL (Bottom-up approach)

- A *dynamic programming*(DP)-based approach.
- We start from the string and build *upwards* by tabulation.
- *Cocke-Younger-Kasami*(CYK) algorithm.

CYK algorithm

$x \in T^*$



$x = a_1 a_2 a_3 \ldots a_p \ldots a_n$

$x_{ij} = a_i a_{i+1} \ldots a_j$

- If we have a way of determining the set of non-terminals deriving x_{ij}, $X_{ij} = \{A | A \overset{*}{\Rightarrow} x_{ij}\}$ for all i and j.

  then, we can say that $x \in L(G)$ iff $S \in X_{1n}$ where $|x| = n$.
- Complexity: $O(n^3)$ (Proof: Homework)

CYK algorithm

X_{15}				
X_{14}	X_{25}			
X_{13}	X_{24}	X_{35}		
X_{12}	X_{23}	X_{34}	X_{45}	
X_{11}	X_{22}	X_{33}	X_{44}	X_{55}
a_1	a_2	a_3	a_4	a_5

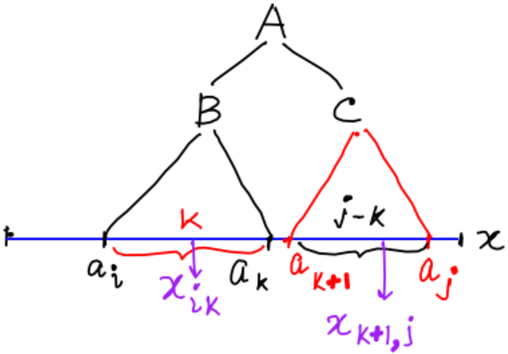
- **Initialization:** We define X_{ii} for each terminal symbol i.
  This corresponds to the productions of the form $A \rightarrow a$ in the CNF grammar.
  $X_{ii} = \{A | A \rightarrow a$ is a production where a is the i^{th} symbol$\}$

CYK algorithm

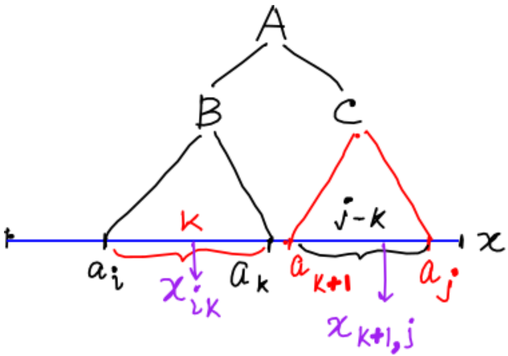
$A \in X_{ij}$ iff $A \stackrel{*}{\Rightarrow} x_{ij}$

1. We compute upward *row-by-row*.
2. Each row corresponds to a substring length. The length starts from 1 at the bottom row and increases by 1 at each row. The final row corresponds to the complete string x.
3. Computing X_{ij}
   - X_{ij} is in row $j - i + 1$.
   - Since we are building upward, we have information about all the Xs in the rows below (or the non-terminals generating substrings of $a_i, a_{i+1} \ldots a_j$).
     Particularly all possible prefixes and suffixes of that string.

- If length > 1, production is of the form $A \rightarrow BC$. B and C correspond to the prefix (left part) and suffix (right part) of the string respectively. $B \in X_{ik}$ and $C \in X_{k+1,j}$, where $i \leq k < j$
- Any derivation $A \overset{*}{\Rightarrow} a_1 a_2 \ldots a_k a_{k+1} \ldots a_j$ starts with a derivation $A \Rightarrow BC$



CYK algorithm



- **Objective:** Determine B, C and k, such that
  1. $i \leq k < j$
  2. B is in X_{ik}
  3. C is in $X_{i+1,j}$
  4. $A \rightarrow BC \in P$

CYK algorithm

for $i = 1$ to n **do**
  $X_{jj} = \{A|A \rightarrow a$ is in P a being $x_j\}$
end for
for $l = 2$ to n **do**
  for $i = 1$ to $n - l + 1$ **do**
    $j = i + l - 1$
    $X_{ij} = \phi$
    for $k = i$ to j **do**
      $X_{ij} = X_{ij} \cup \{A|A \rightarrow BC \in P$ such that $B \in X_{ik}$ and $C \in X_{k+1,j}\}$
    end for
  end for
end for

CYK algorithm: Example

Example 7.34: The following are the productions of a CNF grammar G:

$$
\begin{aligned}
S &\rightarrow AB \mid BC \\
A &\rightarrow BA \mid a \\
B &\rightarrow CC \mid b \\
C &\rightarrow AB \mid a
\end{aligned}
$$

We shall test for membership in $L(G)$ the string *baaba*. Figure 7.14 shows the table filled in for this string.

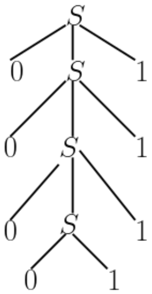
$\{S,A,C\}$				
-	$\{S,A,C\}$			
-	$\{B\}$	$\{B\}$		
$\{S,A\}$	$\{B\}$	$\{S,C\}$	$\{S,A\}$	
$\{B\}$	$\{A,C\}$	$\{A,C\}$	$\{B\}$	$\{A,C\}$
b	a	a	b	a

Pushdown Automata (PDA)

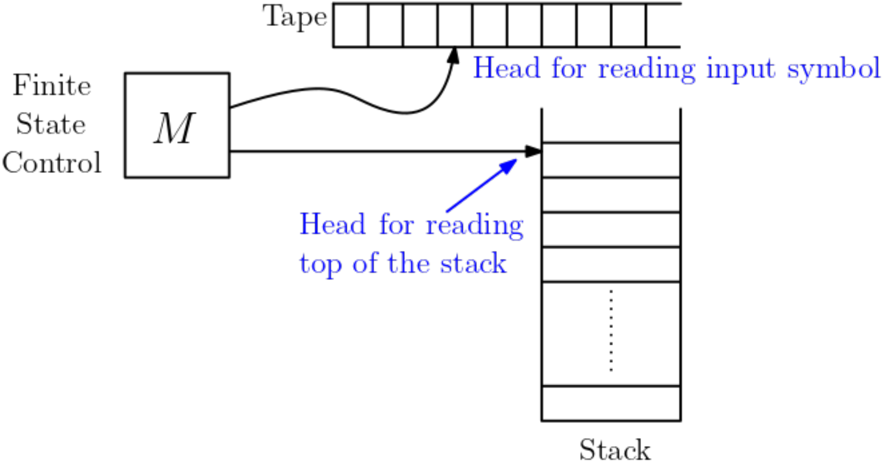
Pushdown Automata (PDA)

The class of finite-state machines that recognize exactly the context-free languages.

- $L = \{0^n 1^n | n \geq 1\}$
- $00001111 \in L$
- $S \rightarrow 0S1|01$
- An FSM that accepts CFLs must be able to maintain the correspondences.



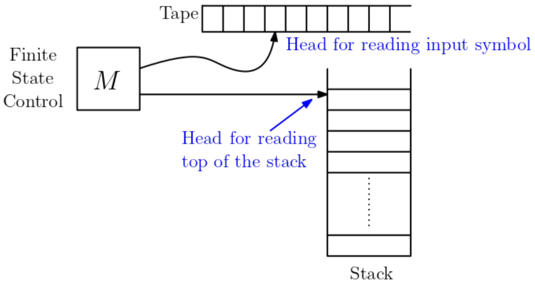
Pushdown Automata



Machine components

$M = (Q, \Sigma, \Gamma, \delta, q_0, z_0, F)$

- Q: finite set of states
- Σ: Tape alphabet/alphabet
- δ: Transition function
- Γ: Stack alphabet
- q_0: Initial state
- z_0: Initial stack symbol.
- F: Set of final states.



Working of a pushdown automata

At a time instance, the machine will use the current *configuration*,

1. The current symbol on the tape.
2. The current symbol on the top of the stack.
3. The state of the machine.

and then decides

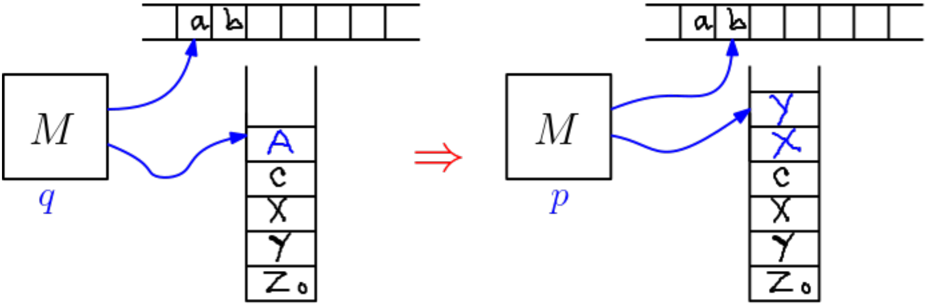
1. The next state.
2. what to do with the top symbol (push/pop).

The set of input symbols and the stack symbols may be the same, overlapping, or disjoint based on the problem.

Transition function

- The machine has to be in a particular state q.
- The tape is a read-only tape.
- The *writing* is done only on the *stack.*
- The tape is read from left to right and it never moves back.
- The stack head operates only on the top of the stack.
- $(q, a, A) \rightarrow (p, \gamma)$
  1. q: Current state
  2. a: Symbol on tape/ Input symbol
  3. A: Top element of the stack.
  4. p: Next state the machine will go to.
  5. The top of the stack will be manipulated by the string γ.

Transition function



- A is popped and $\gamma = YX$ is pushed.
- Note that when YX is pushed, X goes in first followed by Y, and Y the top of stack is now Y.

Non-deterministic transition function

Source of non-determinism in transition function

1. When there are multiple possible outcomes for a single current *configuration*.

   $\delta(q, a, A) = \{(p_1, \gamma_1), (p_2, \gamma_2), \ldots, (p_k, \gamma_k)\}$

2. ϵ-transition: This can result in a change of state and stack content without consuming an input symbol.

   $\delta(q, \epsilon, A) = \{(p_1, \gamma_1), (p_2, \gamma_2), \ldots, (p_k, \gamma_k)\}$

3. Absence of transition on some input and stack symbol pairs.

- If $M = (Q, \Sigma, \Gamma, \delta, q_0, z_0, F)$
  $\delta : Q \times (\Sigma \cup \{\epsilon\}) \times \Gamma \to \mathcal{P}(Q \times \Gamma^*)$

Example 1

Design a PDA for the language L_1
$L_1 = \{0^n 1^n | n \geq 1\}$

Pushdown Automata examples

Example 2

Design a PDA for the language L_1
$L_1 = \{xcx^R | x \in \{0,1\}^* \ and \ c \in \{a,b\}\}$

Example 3

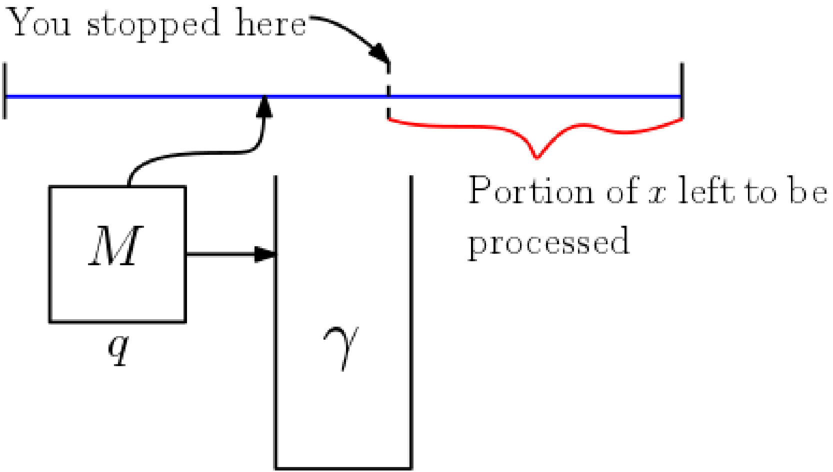
Design a PDA for the language L_1
$L_1 = \{xx^R | x \in \{0,1\}^* \ and \ |x| \geq 1\}$

Instantaneous description of a PDA

Situation

1. Suppose you and your friend know the description of a PDA.
2. Suppose you are running the PDA on a string x.
3. You are instructed to stop in the middle of x, call your friend, and tell her to complete the rest of the process.
4. What minimum information do you have to share to enable your friend to complete the process?

Instantaneous description of a PDA



The description that captures the overall state of the PDA at a certain instance of time is called the **instantaneous description** of the PDA.

Instantaneous description of a PDA

Instantaneous description is a triplet (q, w, γ) where

1. $q \rightarrow$ Current state of the PDA.
2. $w \rightarrow$ Portion of the input string yet to be processed.
3. $\gamma \rightarrow$ Stack content.

Instantaneous description of a PDA

Instantaneous description is a triplet (q, w, γ) where

1. $q \rightarrow$ Current state of the PDA.
2. $w \rightarrow$ Portion of the input string yet to be processed.
3. $\gamma \rightarrow$ Stack content.

The future of the PDA does **not** depend on the

1. input sequence it has processed.
2. the history through which the PDA has reached the current state or the current stack content.

Instantaneous description of a PDA

Instantaneous description (ID) of a PDA is (q, w, γ) where

- $q \in Q$
- $w \in \Sigma^*$
- $\gamma \in \Gamma^*$

Binary relation between two consecutive IDs

The *binary relation* between two IDs over the set of IDs of a given PDA.

$$(q, aw, X\gamma) \vdash (p, w, \beta\gamma)$$

holds if the transition function δ is such that $(p, \beta) \in (q, a, X)$ where $a \in \Sigma \cup \{\epsilon\}$ and $X \in \Gamma$, $\beta \in \Gamma^*$

- $\vdash$ implies " *derives in one step*".
- $ID_1 \vdash ID_2$ implies "ID_1 *derives* ID_2 *in one step*".
- $ID_1 \vdash^* ID_2$ implies " ID_1 *derives* ID_2 *in zero or more step*".

Language acceptance by a PDA

PDAs accept a language in two ways

1. Empty stack.
2. Machine in final state.

Language acceptance by a PDA

- PDA acceptance by **final state**.
  - Initial ID is (q_0, w, z_0)
  - If can go from initial ID to an ID where the state is one of the *final states* after consuming the entire input sequence irrespective of the content of the stack, then M is said to accept by final state.
  - If $L(M)$ is the language accepted by M by final state then $L(M) = \{w \in \Sigma^* | (q_0, w, z_0) \vdash^* (p, \epsilon, \beta)$ for some $p \in F$ and $\beta \in \Gamma^* \}$
- PDA acceptance by **empty stack**.
  - For the same machine M, let $N(M)$ be the language accepted by M by empty stack.
    $N(M) = \{w \in \Sigma^* | (q_0, w, z_0) \vdash^* (q, \epsilon, \epsilon)\}$ for any $q \in Q$.
  - We do not care the state in which the machine is so long as the input is consumed and the stack is empty.

A PDA accepts a language L by final state iff there exists a PDA that accepts L by empty stack.

Language acceptance by final state

Example 3

Design a PDA for the language L_1
$L_1 = \{xx^R | x \in \{0,1\}^* \textit{ and } |x| \geq 1\}$

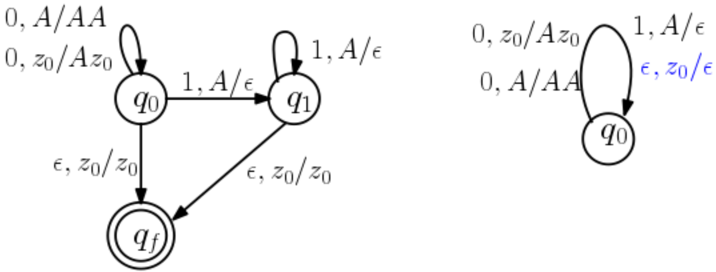
For the string $100001 \in L$, the ID derivation is

$$(q_0, 100001, z_0) \vdash (q_0, 00001, Bz_0) \vdash (q_0, 0001, ABz_0) \vdash$$
$$(q_0, 001, AABz_0) \vdash (q_1, 001, AABz_0) \vdash (q_0, 01, ABz_0) \vdash$$
$$(q_0, 1, Bz_0) \vdash (q_1, \epsilon, z_0) \vdash (q_2, \epsilon, z_0)$$

Can the string **011011** take the machine from q_0 to q_2?
Yes. But not by consuming the entire string! Verify.

Example

$L_1 = \{x = 0^n 1^n | n \geq 0 \text{ and } x \in \{0,1\}^*\}$



- For acceptance by *empty stack* the description is $M = (\{q_0\}, \{0,1\}, \{A, z_0\}, \delta, q_0, z_0, \phi)$ (Final state not mandatory)
- $(\epsilon, z_0/\epsilon)$ is a special move. If z_0 is there in the stack, then further transition is possible. However, if z_0 is removed then no further transitions are possible.

Conversion from empty stack acceptance to final state acceptance

- $M = (Q, \Sigma, \Gamma, \delta, q_0, z_0, \phi)$
- In this machine, once the stack is empty there cannot be any further movement i.e. there cannot be any move to take M to the final state once the stack is empty.
- Because, by definition, every transition requires something in the stack for a transition to happen.

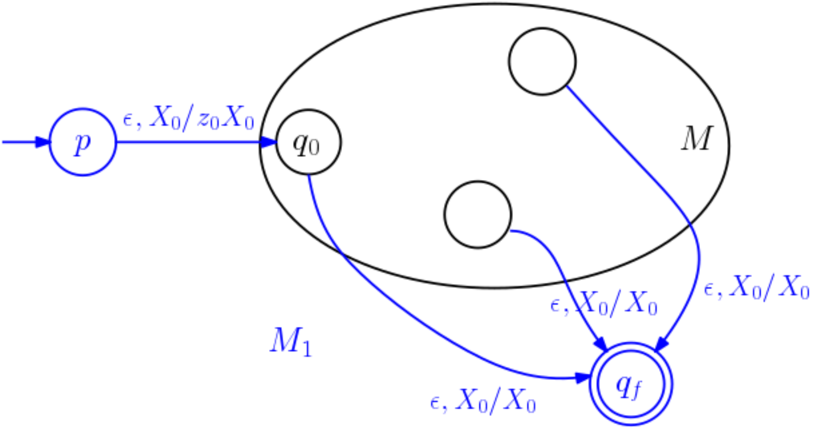
Conversion from empty stack acceptance to final state acceptance

Construction idea

Create a machine M_1 which when senses that M has emptied its stack, will transition to its final state. For the new machine M_1

- introduce a new initial stack symbol X_0.
- $M_1 = (Q \cup \{p, q_f\}, \Sigma, \Gamma \cup \{X_0\}, \delta_{M_1}, p, X_0, \{q_f\})$
- $\delta_{M_1} = \delta \cup \{(p, \epsilon, X_0) \vdash (q_0, z_0 X_0) \cup$ transitions that take M_1 from any state to the final state $q_f\}$
- No transition on X_0 is defined in M.
- When X_0 is exposed it implies that the stack of M is empty.

Conversion from empty stack acceptance to final state acceptance



Conversion from empty stack acceptance to final state acceptance

- Let at some point in time M has emptied its stack.
- Then M_1 is going to find the top of the stack to be X_0 because X_0 will not be in any transition of M.
- Now if M empties its stack (removes z_0 and exposes X_0) then M_1 moves to q_f.
- Note that X_0 is not removed. The presence of only X_0 in the stack indicates that the stack is empty for M. So no more transitions are allowed in M but transition can happen in M_1.

Conversion from empty stack acceptance to final state acceptance

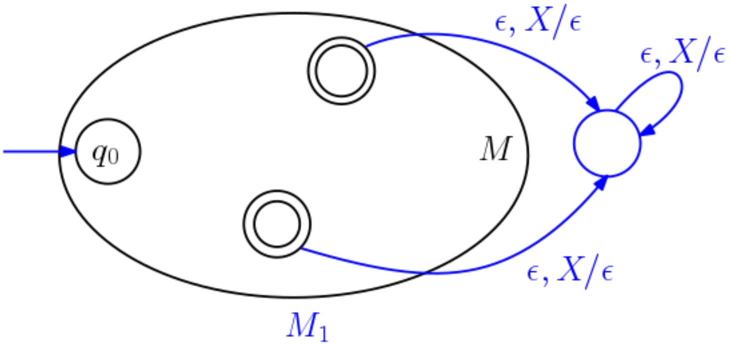
- What happens if the X_0 gets exposed in the middle of processing an input string?

  Answer: If it so happens, then it will imply the PDA is emptying up the stack in the middle of processing the string and halting the process. But if the PDA is designed properly then either

  1. it will empty the stack after reading the entire string.
  2. or it will not empty the stack under any other circumstance because if it empties in the middle then the PDA cannot proceed further otherwise it will imply *non-acceptance* by not emptying the stack.

  So it will imply a design flaw of M.

Conversion from final state acceptance to empty stack acceptance

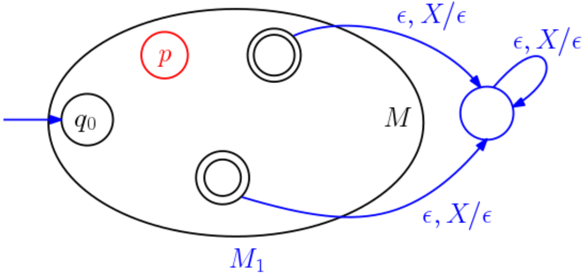


1. $M = (Q, \Sigma, \Gamma, \delta, q_0, z_0, F \neq \phi)$
2. From M construct M_1 that accepts by *empty stack*.
3. **Strategy:** Add a transition from each final state of M to another state in M_1 which empties the stack.

Conversion from final state acceptance to empty stack acceptance

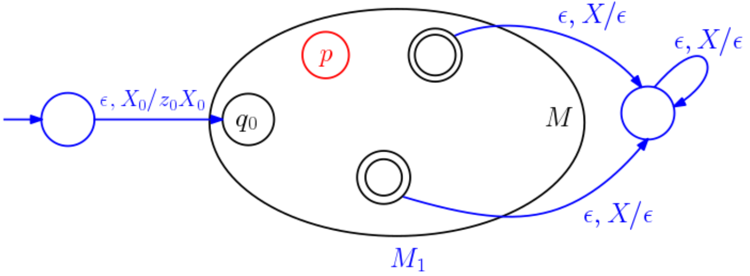
- This implies that once the string gets accepted by entering a final state after reading the entire input string then it will transition to a state that will simply empty the stack.
- M_1 will not have a final state because it accepts by emptying the stack, but it will accept the same language as M because a string the stack is being emptied after it is accepted by entering a final state of M.

Conversion from final state acceptance to empty stack acceptance



- Suppose there is a state p in M, that consumes the entire input and empties the stack.
- However, we do not care since M accepts only by the final state.
- But p can be a problem when acceptance is by *empty stack* because $(q_0, w, z_0) \vdash^* (p, \epsilon, \epsilon)$ and p is not a final state.
- Such a w will not be in $L(M)$ (the language of the old machine) but will get accepted by the new machine M_1.

Conversion from final state acceptance to empty stack acceptance



- We add a new start state, a new initial stack symbol X_0, and a transition for M_1 as shown.
- Since there is no transition defined on p for the stack symbol X_0, it cannot be eliminated from the stack from state p.
- The stack will now show the stack top X_0 here and will not be empty.
- However, the removal of X_0 shall be defined from the state in M_1 connected to the final states of M.

Closure properties of CFLS

Closure properties of CFLs

- The class of *context-free languages* is close under
  1. union
  2. Kleene closure
  3. concatenation
  4. reversal
  5. intersection with regular languages.
  6. difference with regular language.
  7. homomorphism, inverse homomorphism and substitution.
- The class of *context-free languages* is **not** closed under
  1. intersection
  2. complementation
  3. difference

Union

- Let L_1 and L_2 be two CFLs generated by G_1 and G_2 respectively.
- $G_1 = \{V_{N_1}, T, S_1, P_1\}$
- $G_2 = \{V_{N_2}, T, S_2, P_2\}$
- Assume $V_{N_1} \cap V_{N_2} = \phi$ (If there is overlap then it can be resolved by renaming the non-terminals).
- $G = \{\{S\} \cup V_{N_1} \cup V_{N_2}, T, S, P\}$
  $P = P_1 \cup P_2 \cup \{S \rightarrow S_1 | S_2\}$

Concatenation

- Let L_1 and L_2 be two CFLs generated by G_1 and G_2 respectively.
- $L = L_1.L_2 = \{wx|w \in Ł()\}$
- $G_1 = \{V_{N_1}, T, S_1, P_1\}$
- $G_2 = \{V_{N_2}, T, S_2, P_2\}$
- Assume $V_{N_1} \cap V_{N_2} = \phi$ (If there is overlap then it can be resolved by renaming the non-terminals).
- $G = \{\{S\} \cup V_{N_1} \cup V_{N_2}, T, S, P\}$
  $P = P_1 \cup P_2 \cup \{S \rightarrow S_1 S_2\}$

Kleene closure

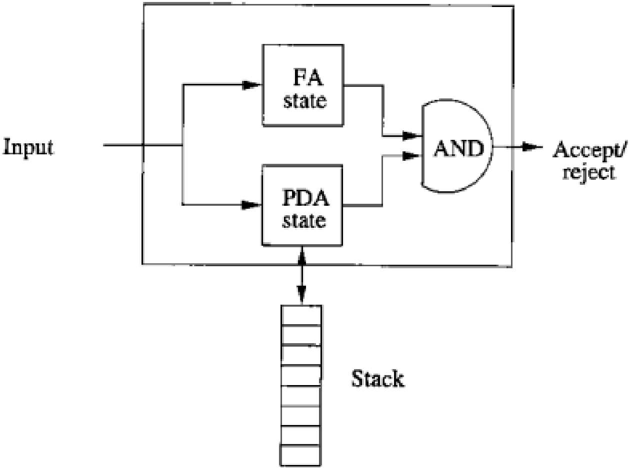
- Let L_1 be a CFL generated by G_1.
- $G_1 = \{V_N, T, S_1, P_1\}$
- $G = \{\{S\} \cup V_N, T, S, P\}$
  $P = P_1 \cup \{S \rightarrow SS|\epsilon\}$
- $L(G) = L_1^*$

Reversal

- Let L_1 be a CFL generated by G_1.
- $G_1 = \{V_N, T, S_1, P_1\}$
- Let $G = \{V_N, T, S, P\}$

  P contains all the productions of P_1 reversed i.e. for $A \to \alpha \in P_1$ there is a production $A \to \alpha^R \in P$

Intersection with regular languages



Create a PDA by combining a PDA and a DFA that run in parallel.

Intersection with regular languages

- $P = (Q_p, \Sigma, \Gamma, \delta_p, q_p, z_0, F_p)$
- $A = (Q_a, \Sigma, \delta_a, q_a, F_a)$
- P and A be a PDA and a DFA respectively.
- We shall reason in line with proving that the intersection of two regular languages is regular.
- We shall try to define a PDA that simulates the parallel execution of P and A on an input string w.
- If we are able to show that such a PDA exists, that can simulate the parallel execution of P and A on a string, and the new PDA after scanning the input string
  - ends up in the final state if both P and A end up in the respective final states.
  - does not end up in its final state otherwise.
- Then we shall be able to show that $L \cap A$ is a CFL.
- CFLs are closed under intersection with regular languages.

Intersection with regular languages

- Let the new PDA be

  $N = (Q_p \times Q_a, \Sigma, \Gamma, \delta, (q_p, q_a), z_0, F_p \times F_a)$

  where $\delta((q, p), a, X)$ is defined to be the set of all pairs $((r, s), \gamma)$

  1. $s = \delta_a(p, a)$
  2. (r, γ) is in $\delta_p(q, a, X)$

  - A state (q, p) in N is final state if $q \in F_p$ and $p \in F_a$.
  - The initial state is the state (q_p, q_a).
- $a \in \Sigma$ or $a = \epsilon$. If $a = \epsilon$, then $s = p$ i.e. P makes a move on ϵ while A remains in the same state.
- Thus, for the string w if
  - $(q_p, w, z_0) \overset{*}{\vdash} (q, \epsilon, \gamma)$
  - $\hat{\delta}(q_a, w) = p$

  $((q_p, q_a), w, z_0) \overset{*}{\vdash} ((q, p), \epsilon, \gamma)$
- If $q \in F_p$ and $p \in F_a$ then $(q, p) \in F_p \times F_a$. Then w will be accepted by N.

Difference with regular language

- If L is a CFL and R is a regular language.
- Then, $L - R$ is a CFL.
- $L - R = L \cap \overline{R}$
- Since, the class of regular languages is closed under complementation, so $\overline{R}$ is regular.
- Hence, $L - R$ is regular.

Intersection: Proof by counter-example

- $L_1 = \{a^i b^i c^j | i, j \geq 1\}$
- $L_2 = \{a^i b^j c^j | i, j \geq 1\}$
- Are L_1 and L_2 context-free?

Intersection: Proof by counter-example

- $L_1 = \{a^i b^i c^j | i, j \geq 1\}$
- $L_2 = \{a^i b^j c^j | i, j \geq 1\}$
- Are L_1 and L_2 context-free?

For L_1

$$S \rightarrow XY$$
$$X \rightarrow aXb | ab$$
$$Y \rightarrow cY | c$$

For L_2

$$S \rightarrow XY$$
$$X \rightarrow aX | a$$
$$Y \rightarrow bYc | bc$$

- $L = L_1 \cap L_2$. For w in L,
  - the number of as = number of bs
  - the number of bs = number of cs
- $L = \{a^i b^i c^i | i \geq 1\}$
- L is not a context-free language.

Complementation

- Let the class of CFLs be closed under complementation.
- $L_1 \cap L_2 = \overline{\overline{L_1} \cup \overline{L_2}}$
- The class of CFLs is closed under union.
- If the class of CFLs is closed under complementation, then it would be closed under intersection.
- We have already proved that the class of CFLs is not closed under the intersection.
- Contradiction!!

Difference

- Let the class of CFLs be closed under complementation.
- $L_1 \setminus L_2 = L_1 \cap \overline{L_2}$
- Since the class of CFLs is neither closed under intersection nor complementation. So it is not closed under the difference operation.

Decision problems for CFGs

1. Conversion from CFG to PDA and vice-versa.
2. Conversion of CFG to normal forms.
3. Testing the emptiness of a CFL.

Homework: Complexity of the above operations. Reference: Textbook 1.

Equivalence of PDA and CFGs

Conversion between PDA and CFG

CFG to PDA conversion

Objective

Convert a CFG to a PDA. More specifically, convert a CFG G to a PDA P that simulates the *leftmost derivation* of G.

- $S \Rightarrow \gamma_1 \Rightarrow \gamma_2 \cdots \Rightarrow \gamma_n = w$ where $w \in L(G)$.
- γ_i is a *left-sentential form*.
- Any left-sentential form can be written as $xA\alpha$.
- $x \in T^*$, $A \in V$ and $\alpha \in (V \cup T)^*$
- $A\alpha$ is called the **tail of the left-sentential form**.
- If the left-sentential form contains only terminals then the tail is ϵ.

CFG to PDA conversion: Idea

- The PDA should simulate the sequence of *left-sentential forms* (γ)*s* that the grammar uses to generate the terminal string w.
- The PDA is designed to be such that for a given $\gamma = xA\alpha$
  1. x is the portion of the string w already consumed. Let $w = xy$ where y is the portion yet to be processed.
  2. $A\alpha$ is the content of the stack with A as the top.
- Identify a transition $A \rightarrow \beta$ and replace A by β. Push β into the stack.

  $(q, y, A) \vdash (q, y, \beta)$
- In the new stack content $\beta\alpha$, we have to remove the terminals at the top so that the first non-terminal which corresponds to the leftmost non-terminal is exposed.
- The terminal so removed should match a prefix of y. Otherwise, that branch of the PDA reaches a dead end.

CFG to PDA conversion: Idea

- If the guess for the leftmost derivation of w is correct at each step then
  1. w is derived at the end.
  2. When w is formed, the content of the stack should be empty. The symbol on the stack is
     1. expanded if it is a non-terminal.
     2. matched against the input if it is a terminal.
- At this stage, when the stack is empty, we accept by the empty stack.

PDA equivalent to a CFG

- Let $G = (V, T, S, P)$ and $P = ()$ be a PDA that accepts by empty stack.
- $P = \{\{q\}, T, V \cup T, \delta, q, S\}$

where the transitions in δ are of the form

1. For each non-terminal A,

   $\delta(q, \epsilon, A) = \{(q, \beta) | A \rightarrow \beta \text{ is a production of } G\}$
2. For each terminal a, $\delta(q, a, a) = \{(q, \epsilon)\}$

Example

$$
\begin{aligned}
I &\rightarrow a \mid b \mid Ia \mid Ib \mid I0 \mid I1 \\
E &\rightarrow I \mid E * E \mid E + E \mid (E)
\end{aligned}
$$

Example

$$
\begin{array}{lcl}
I & \rightarrow & a \mid b \mid Ia \mid Ib \mid I0 \mid I1 \\
E & \rightarrow & I \mid E * E \mid E + E \mid (E)
\end{array}
$$

a) $\delta(q, \epsilon, I) = \{(q, a),\ (q, b),\ (q, Ia),\ (q, Ib),\ (q, I0),\ (q, I1)\}$.

b) $\delta(q, \epsilon, E) = \{(q, I),\ (q, E + E),\ (q, E * E),\ (q, (E))\}$.

c) $\delta(q, a, a) = \{(q, \epsilon)\}$; $\delta(q, b, b) = \{(q, \epsilon)\}$; $\delta(q, 0, 0) = \{(q, \epsilon)\}$; $\delta(q, 1, 1) = \{(q, \epsilon)\}$; $\delta(q, (, () = \{(q, \epsilon)\}$; $\delta(q,),)) = \{(q, \epsilon)\}$; $\delta(q, +, +) = \{(q, \epsilon)\}$; $\delta(q, *, *) = \{(q, \epsilon)\}$.

Conversion from PDA to CFG

Objective

Convert a PDA to a CFG. More specifically, convert a PDA P to a CFG G that denotes the working of P.

Conversion from PDA to CFG

Construction procedure

- $P = (Q, \Sigma, \Gamma, \delta, q_0, z_0)$ be a PDA.
- $G = (V, \Sigma, S, P)$ where V consists of
  1. The start symbol S.
  2. All symbols of the form (pXq), where p and q are states in Q and $X \in \Gamma$.
- The productions in P are as follows:
  1. For all states p, G has the productions $S \rightarrow (q_0 z_0 p)$
  2. Let $\delta(q, a, X)$ contain the pair $(r, Y_1 Y_2 \ldots Y_k)$ where
     1. a is either in Σ or $a = \epsilon$
     2. k can be any number (maybe 0). Then for all lists of states $r_1, r_2, \ldots, r_k$, G has the production $(qXr_k) \rightarrow a(rY_1r_1)(r_1Y_2r_2)\ldots(r_{k-1}Y_kr_k)$