Context Free Grammar

Unit 4

Introduction

Grammars are used to generate the words of a language and to determine whether a word is in a language

Formal languages, which are generated by grammars, provide models for both natural languages, such as english, and for programming languages, such as C, Java, etc

Introduction

Context free grammars are used to define syntax of almost all programming languages in context of computer science

Thus, an important application of context free grammars occurs in the specification and compilation of programming languages

Context Free Grammar

Formal description of CFG:

A context free grammar is defined by 4-tuples (V, T, P, S) where,

V = set of variables

T = set of terminal symbols

P = set of productions

S = start symbol; $S \in V$

Components of CFG

- There is a finite set of symbols that form the strings of language being defined. We call this alphabet the terminal symbols. In the above tuples, it is represented by T

- There is a finite set of variables, also called non-terminal symbols or syntactic categories. Each variable represents a language i.e a set of strings.

- One of the variables represents the language being defined. It is called the start symbol.

Components of CFG

- There is a finite set of productions or rules that represent the recursive definition of a language. Each production consists of :

(i) A variable that is being defined by the production. This is called head of production.

(ii) The production symbol "→"

(iii) A string of zero or more terminals and variables. This string is called the body of the production, and represents one way to form the string in the language of the head.

Components of CFG

The variable symbols are after represented by capital letters

The terminals are analogous to the input alphabet and are often represented by lowercase letters

One of the variables is designed as a start variable and it usually occurs on the left hand side of the topmost rule

Applications of CFG

- CFGs are used in compilers (like GCC) for parsing

- CFGs are used to define the high level structure of programming languages

- Document type definition in XML is a CFG which describes the HTML tags and the rules to use the tags in a nested fashion

Example

A CFG defining the grammar of all the strings with 0's followed by equal number of 1's:

$$S \rightarrow \varepsilon$$

$$S \rightarrow 0S1$$

The two rules define the production P,

ε, 0, 1 are the terminals defining T,

S is a variable symbol defining V, and

S is the start symbol from where production starts

Language of Grammar

Let G = (V, T, P, S) be a context free grammar

Then the language of G, denoted by L(G), is the set of terminal strings that have been derived from the start symbol in G

$$\text{i.e. } L(G) = \{x \in T^* \mid S \Rightarrow^* x\}$$

The language generated by a CFG is called **Context Free Language (CFL)**

CFG vs RE

CFGs are more powerful than regular expressions as they have more expressive power

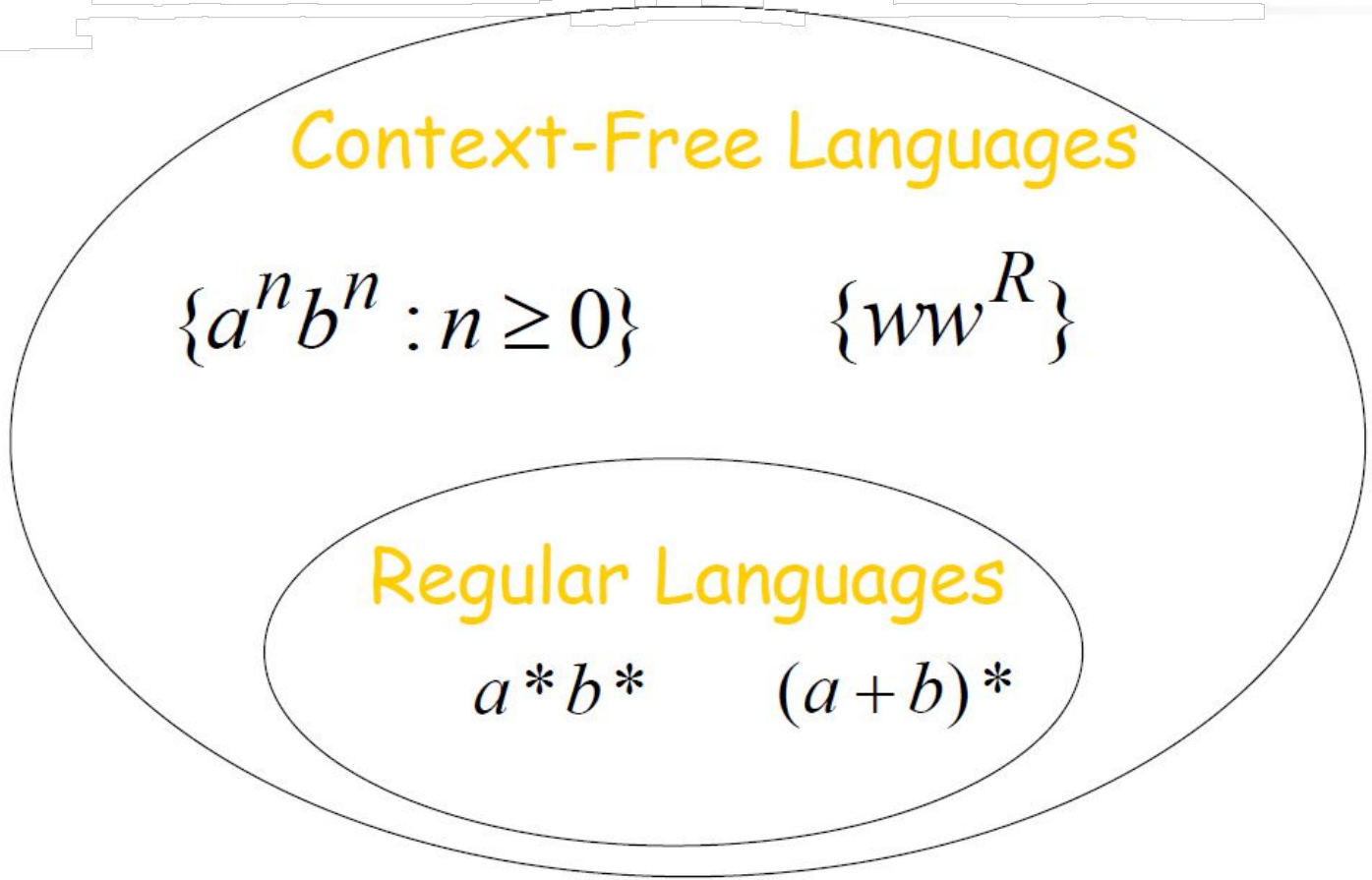
Generally REs are useful for describing the structure of lexical constructs as identical keywords, constants, etc

But, they do not have the capability to specify the recursive structure of programming constructs

CFG vs RE

CFGs are capable of defining recursive structures too

Thus, CFGs can define the languages that are regular as well as those languages that are not regular


Context-Free Languages
$\{a^n b^n : n \geq 0\}$
$\{ww^R\}$
Regular Languages
$a*b*$
$(a+b)*$

Compact Notation for Productions

We can think of a production as “belonging” to the variable of head

We may use terms like A-production or production of A to refer to the production whose head is A

We write the production for a grammar by listing each variable once and then listing all the bodies of the production for that variable, separated by |

This is called **compact notation**

Compact Notation for Productions

eg: CFG representing the language over Σ = {a, b} for palindromes:

$$S \rightarrow \varepsilon \mid a \mid b$$

$$S \rightarrow aSa \mid bSb$$

Multiple productions for a variable have been listed in a single line separated by |

Meaning of Context Free

Consider an example:

$S \rightarrow aMb$

$M \rightarrow A \mid B$

$A \rightarrow \varepsilon \mid aA$

$B \rightarrow \varepsilon \mid bB$

Meaning of Context Free

Consider the string aaAb, which is an intermediate stage in the generation of aaab

It is natural to call the strings “aa” and “b” that surround the symbol A, the “context” of A in this particular string

Now, the rule A → aA says that we can replace A by the string aA no matter what the surrounding strings are; in other words, independent of the context of A

Meaning of Context Free

When there is a production of the form $lw_1r \rightarrow lw_2r$ (but not of the form $w_1 \rightarrow w_2$), the grammar is **context sensitive** since w_1 can be replaced by w_2 only when it is surrounded by the strings “l” and”r”

Derivation using Grammar Rules

We apply the production rules of a CFG to infer that certain strings are in the language of a certain variable

This is the process of **deriving strings** by applying productions rules of the CFG

Derivation using Grammar Rules

General convention used in derivation:

Capital letters near the beginning of the alphabet such as A, B are used as variables

Capital letters near the end of the alphabet such as X, Y are used as either terminals or variables

Derivation using Grammar Rules

Lower case letters near the beginning of the alphabet such as a, b are used for terminals

Lower case letters near the end of alphabet, such as ‘w’ or ‘z’ are used for string of terminals

Lower case Greek letters such as α and β are used for string of terminal or variables

Derivation using Grammar Rules

A **derivation** of a CFG is a finite sequence of strings $\beta_0\beta_1\beta_2\ldots\ldots\beta_n$ such that:

- For $0 \leq i \leq n$, the string $\beta_i \in (V \cup T)^*$
- $\beta_0 = S$
- For $0 \leq i \leq n$, there is a production of P that applied to β_i yields β_{i+1}
- $\beta_n \in T^*$

Derivation using Grammar Rules

There are two possible approaches of derivation:

- Bottom-Up (body to head) approach

- Top-Down (head to body) approach

Bottom-Up Derivation

Here, we take strings known to be in the language of each of the variables of the body, concatenate them, in the proper order, with any terminals appearing in the body, and infer that the resulting string is the language of the variables in the head

Bottom-Up Derivation

eg: Consider grammar:

$S \rightarrow S + S$

$S \rightarrow S / S$

$S \rightarrow (S)$

$S \rightarrow S - S$

$S \rightarrow S * S$

$S \rightarrow a$

Bottom-Up Derivation

Given: a + (a * a) / a – a

SN	String Inferred	Variable(For Language of)	Production	String Used
1	a	S	S→a	
2	a*a	S	S→S*S	String 1
3	(a*a)	S	S→(S)	String 2
3	(a*a)/a	S	S→S/S	String 3 and string 1
4	a+(a*a)/a	S	S→S+S	String 1 and 3
5	a+(a*a)/a-a	S	S→S-S	String 4 and 1

We start with any terminal appearing in the body and use the available rules from body to head.

Top-Down Derivation

We use productions from head to body

We expand the start symbol using a production whose head is the start symbol

Then we keep expanding the resulting string until a string of all terminals are obtained

Top-Down Derivation

There are two approaches:

(i) Leftmost Derivation: Leftmost symbol (variable) is replaced first in each step

(ii) Rightmost Derivation: Rightmost symbol is replaced first in each step

Top-Down Derivation

eg: Consider the previous example of deriving string

a + (a * a) / a – a

Leftmost derivation

for the given string is:

S→ S + S	rule → S + S
S→a + S	rule S→ a
S→a + S – S	rule S→S - S
S→a + S / S – S	rule S→ S / S
S→a + (S) /S – S	rule S→(S)
S→(S*S) / S – S;	rule S→S*S
S→ a + (a*S) / S – S	rule S→a
S→ a + (a*a) / S – S	" "
S→ a + (a*a) / a – S	" "
S→a + (a*a) / a – a	" "

Rightmost derivation

for the given string is:

S→ S – S;	rule S→S - S
S→ S – a;	rule S→ a
S→S + S – a;	rule S→ S + S
S→S + S / S – a;	rule S→S / S
S→S + S / a –a;	rule S→ a
S→S + (S) / a – a;	rule S→ (S)
S→S + (S*S) / a – a;	rule S→ S*S
S→S + (S*a) / a – a;	rule S→ a
S→S + (a*a) / a – a;	
S→ a + (a*a) / a – a;	

Direct Derivation

$\alpha 1 \Rightarrow \alpha 2$: If $\alpha 2$ can be derived directly from $\alpha 1$, then it is direct derivation

$\alpha 1 \Rightarrow^* \alpha 2$: If $\alpha 2$ can be derived from $\alpha 1$ with zero or more steps of the derivation, then it is just derivation

Direct Derivation

eg: S → aSa | ab | a | b | ε

Direct derivations: S ⇒ ab

⇒ aSa

⇒ aaSaa

⇒ aaabaa

Thus, S ⇒* aaabaa is a derivation of multiple steps

Sentential Forms

Derivations from the start symbol produce various intermediate strings

We call them "**sentential forms**". i.e. if G = (V, T, P, S) is a CFG, then any string $\alpha \in (V \cup T)^*$ such that $S \Rightarrow^* \alpha$ is a sentential form

If $S \Rightarrow^*_{lm} \alpha$, then α is a **left sentential form**, and if $S \Rightarrow^*_{rm} \alpha$, then α is a **right sentential form**

The language L(G) consists of sentential forms in T^* that consist solely of terminals (and they are called **sentences**)

Parse Tree (Derivation Tree)

A parse tree is a tree representation of strings of terminals using the productions defined by the grammar

A parse tree pictorially shows how the start symbol of a grammar derives a string in the language

Parse tree may be viewed as a graphical representation for a derivation that filters out the choice regarding the replacement order

Parse Tree

Formally, given a CFG, G = (V, T, P, S), a parse tree is a n-ary tree having the following properties:

- The root is labeled by the start symbol

- Each interior node of parse tree are variables

- Each leaf node of parse is labeled by a terminal symbol or ε

- If an interior node is labeled with a non-terminal A and its children are $x_1, x_2, \ldots\ldots, x_n$ from left to right, then there is a production P as:

$$A \rightarrow x_1\ x_2\ \ldots\ldots\ x_n \text{ for each } x_i \in T$$

Parse Tree

eg: Consider the grammar

$$S \rightarrow aSa \mid a \mid b \mid \varepsilon$$

For string aabaa,

$$S \Rightarrow aSa$$

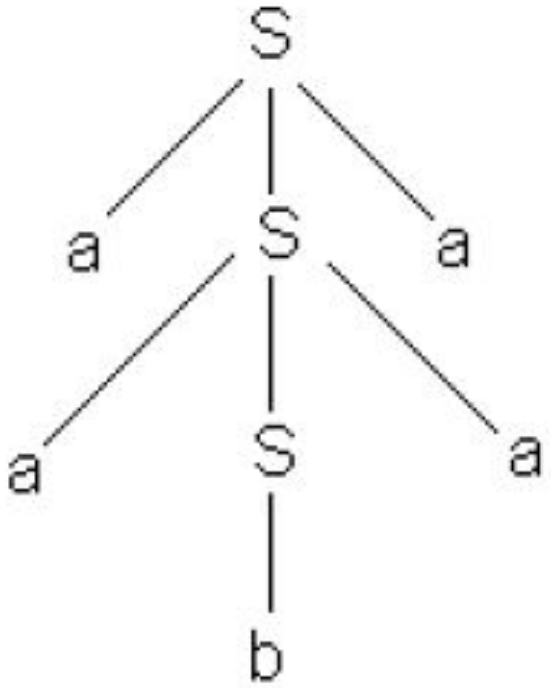
$$\Rightarrow aaSaa$$

$$\Rightarrow aabaa$$

Parse Tree

So, the parse tree

will be:



Ambiguity in Grammar

A grammar G = (V, T, P, S) is said to be **ambiguous** if there is a string w ∈ L(G) for which we can derive two or more distinct derivation trees rooted at S and yielding w

In other words, a grammar is ambiguous if it can produce more than one leftmost or more than one rightmost derivation for the same string in the language of the grammar

Ambiguity in Grammar

eg: Given grammar: $S \rightarrow AB \mid aaB$

$A \rightarrow a \mid Aa$

$B \rightarrow b$

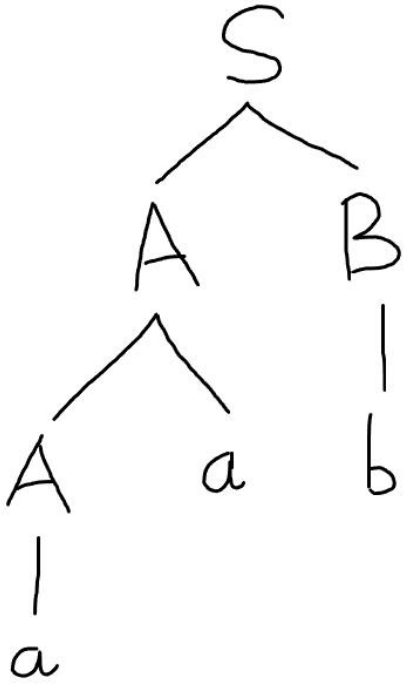
Consider the string “aab”

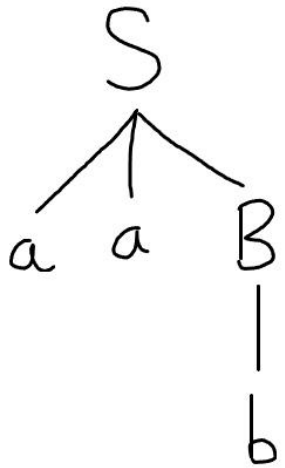
Ambiguity in Grammar

aab has two leftmost derivations:

(i) S ⇒ AB ⇒ AaB ⇒ aaB ⇒ aab

(ii) S ⇒ aaB ⇒ aab





Practice Questions

(Q1) Consider the Grammar G:

$S \rightarrow A1B$

$A \rightarrow 0A \mid \varepsilon$

$B \rightarrow 0B \mid 1B \mid \varepsilon$

Show leftmost and rightmost derivation and construct the parse tree for the strings: (a) 00101 (b) 1001 (c) 00011

Practice Questions

(Q2) Consider the grammar for arithmetic expressions:

E → E OP E | (E) | id

OP → + | - | * | / | ↑

(a) Construct the parse tree for id + id *id

(b) Construct the parse tree for (id +id)*(id + id)

Practice Questions

(Q3) Construct a CFG that generates language of balanced parentheses. Then, show the parse tree computation for

(a) ( ) ( )

(b) ((( ))) ( )

Practice Questions

Hint for Q3: The CFG is G=(V, T, P, S), where

$V = \{S\}$

$T = \{ (\ ,\ ) \}$

$P = \{ S \rightarrow SS \mid (S) \mid \varepsilon \}$

$S = \{S\}$

Simplification of CFG

The goal of this section is to show that every context free language (without ε) is generated by a CFG in which all productions are of the form $A \rightarrow BC$ or $A \rightarrow a$, where A, B and C are variables, and 'a' is a terminal

This form is called Chomsky Normal Form(CNF), first described by Noam Chomsky

Simplification of CFG

To get there, we need to make a number of preliminary simplifications, which are themselves useful in various ways:

(1) Elimination of useless symbols

(2) Elimination of ε-productions

(3) Elimination of unit productions

Eliminating Useless Symbols

We say a symbol x is **useful** for a grammar G = (V, T, P, S) if there is some derivation of the form $S \Rightarrow^* \alpha x \beta \Rightarrow^* w$, where $w \in T^*$

Here, x may either be a variable or a terminal and the sentential form αxβ might be the first or last in the derivation

If x is not useful, we say it is **useless**

Thus, useful symbols are those variables or terminals that appear in any derivation of a terminal string from the start symbol

Eliminating Useless Symbols

Eliminating a useless symbol includes identifying whether or not the symbol is "**generating**" and "**reachable**"

Generating Symbol: We say x is generating if $x \Rightarrow^* w$ for some terminal string w

Note that every terminal is generating since w can be that terminal itself, which is derived by zero steps

Eliminating Useless Symbols

Reachable symbol: We say x is reachable if there is derivation $S \Rightarrow^* \alpha x \beta$ for some α and β

So, if we eliminate non-generating symbols followed by non-reachable ones, we shall only have useful symbols left

Example

Consider a grammar defined by following productions:

$$S \rightarrow aB \mid bX$$

$$A \rightarrow Bad \mid bSX \mid a$$

$$B \rightarrow aSB \mid bBX$$

$$X \rightarrow SBd \mid aBX \mid ad$$

Now, eliminate useless symbols

Example

A and X can directly generate terminal symbols; so, A and X are generating symbols (as we have the productions $A \rightarrow a$ and $X \rightarrow ad$)

Also, $S \rightarrow bX$ and X generates a terminal string so S can also generate terminal string

Hence, S is also a generating symbol

B cannot produce any terminal symbol, so it is non-generating

Example

Hence, the new grammar after removing non-generating symbols is:

$$S \rightarrow bX$$

$$A \rightarrow bSX \mid a$$

$$X \rightarrow ad$$

Now, A is non-reachable as there is no derivation of the form $S \Rightarrow^* \alpha A \beta$ in the grammar

Example

Thus, eliminating non-reachable symbol, the resulting grammar is:

$$S \rightarrow bX$$

$$X \rightarrow ad$$

This is the grammar with useful symbols only

Practice Questions

(Q1) Remove useless symbol from the following grammar:

$$S \rightarrow xyZ \mid XyzZ$$

$$X \rightarrow Xz \mid xYZ$$

$$Y \rightarrow yYy \mid XZ$$

$$Z \rightarrow Zy \mid z$$

Practice Questions

(Q2) Remove useless symbol from the following grammar:

$$S \rightarrow aC \mid SB$$

$$A \rightarrow bSCa$$

$$B \rightarrow aSB \mid bBC$$

$$C \rightarrow aBc \mid ad$$

Eliminating ε-productions

Here, the strategy is to begin by discovering the variables that are "**nullable**"

A variable 'A' is nullable if $A \Rightarrow^* \varepsilon$

Steps to remove ε-productions from grammar:

- If there is a production of the form $A \rightarrow \varepsilon$, then A is nullable

Eliminating ε-productions

- If there is production of the form B → X1 X2……Xn and each Xi's are nullable, then B is also nullable

- Find all the nullable variables

- If B → X1 X2……Xn is a production in P, then add all productions P' for all subsets formed after removing Xi's that are nullable

- Do not include B → ε if there is such production

Example

Consider the grammar:

$$S \rightarrow ABC$$

$$A \rightarrow BB \mid \varepsilon$$

$$B \rightarrow CC \mid a$$

$$C \rightarrow AA \mid b$$

Now, remove ε-productions

Example

Here,

$A \rightarrow \varepsilon$	So, A is nullable
$C \rightarrow AA$	Each A is nullable, so C is nullable
$B \rightarrow CC$	Each C is nullable, so B is nullable
$S \rightarrow ABC$	S is nullable since each of A, B and C are nullable

Example

Now, for removal of ε-production:

In the production S → ABC, A, B and C are all nullable

So, adding all combinations of subsets after striking out each nullable symbol gives new productions as:

S → ABC | AB | BC | AC | A | B | C

Example

Following this process for all productions gives the resulting final grammar as:

$$S \rightarrow ABC \mid AB \mid BC \mid AC \mid A \mid B \mid C$$

$$A \rightarrow BB \mid B$$

$$B \rightarrow CC \mid C \mid a$$

$$C \rightarrow AA \mid A \mid b$$

Practice Question

(Q) Remove ε-productions for the following grammar:

$$S \rightarrow AB$$

$$A \rightarrow aAA \mid \varepsilon$$

$$B \rightarrow bBB \mid \varepsilon$$

Eliminating Unit Productions

A unit production is a production of the form $A \rightarrow B$, where A and B are both variables

If $A \rightarrow B$, we say B is A-derivable; if $B \rightarrow C$, we say C is B-derivable

So, if both $A \rightarrow B$ and $B \rightarrow C$, then $A \Rightarrow^* C$; C is also A-derivable

Here pairs (A, B), (B, C) and (A, C) are called the unit pairs

To eliminate unit productions, we first find all of the unit pairs

Eliminating Unit Productions

The unit pairs are:

- (A, A) is a unit pair for any variable A as $A \Rightarrow^* A$

- If we have $A \rightarrow B$ then (A, B) is a unit pair

- If (A, B) is a unit pair i.e. $A \rightarrow B$, and if we have $B \rightarrow C$ then (A, C) is also a unit pair

Eliminating unit productions for a given grammar, say G = (V, T, P, S), is to find an equivalent grammar G' = (V, T, P', S) with no unit productions

Eliminating Unit Productions

Steps:

- Initialize P' = P

- For each A ∈ V, find a set of A-derivable variables

- For every pair (A, B) such that B is A-derivable and for every non-unit production B → α, we add production A → α in P' if it is not in P' already

- Delete all unit productions from P'

Example

Remove unit productions for grammar G defined by the following productions:

$$P = \{ \quad S \rightarrow S + T \mid T,$$

$$T \rightarrow T * F \mid F,$$

$$F \rightarrow (S) \mid a \quad \}$$

Example

(1) Initialize P' = { $S \rightarrow S + T \mid T,$

$T \rightarrow T * F \mid F,$

$F \rightarrow (S) \mid a$ }

(2) Now, find unit pairs:

$S \rightarrow T$	So, (S, T) is a unit pair
$T \rightarrow F$	So, (T, F) is a unit pair
$S \rightarrow T$ & $T \rightarrow F$	So, (S, F) is also a unit pair

Example

(3) Add each non-unit production of the form $B \rightarrow \alpha$ for each pair (A,B):

P' = { $S \rightarrow S + T \mid T \mid T * F \mid (S) \mid a$,

$T \rightarrow T * F \mid F \mid (S) \mid a$,

$F \rightarrow (S) \mid a$ }

Example

(4) Finally, delete all unit productions from the grammar:

P' = { $S \rightarrow S + T \mid T * F \mid (S) \mid a$,

$T \rightarrow T * F \mid (S) \mid a$,

$F \rightarrow (S) \mid a$ }

This is the required final grammar

Practice Questions

(Q1) Simplify the grammar G = (V, T, P, S) defined by the following productions:

S → ASB | ε

A → aAS | a

B → SbS | A | bb | ε [Simplify = remove useless symbols, ε-productions and unit productions]

Practice Questions

(Q2) Simplify the grammar defined by following productions:

$S \rightarrow 0A0 \mid 1B1 \mid BB$

$A \rightarrow C$

$B \rightarrow S \mid A$

$C \rightarrow S \mid \varepsilon$

Chomsky Normal Form

A context free grammar G = (V, T, P, S) is said to be in Chomsky Normal Form (CNF) if every production in G are in one of the two forms:

$$A \rightarrow BC \text{ or } A \rightarrow a$$

where A, B, C $\in$ V and a $\in$ T

A grammar in CNF is one which should not have:

(i) ε-productions (ii) unit productions (iii) useless symbols

Chomsky Normal Form

Theorem: Every context free language (CFL) without ε-productions can be generated by a grammar in CNF

Proof:

If all the productions are of the form $A \rightarrow a$ or $A \rightarrow BC$ with $A, B, C \in V$ and $a \in T$, we are done

Chomsky Normal Form

Otherwise, we require two tasks:

(1) Perform arrangement such that all bodies of length 2 or more consist only of variables

(2) Break bodies of length 3 or more into a cascade of productions, each with a body consisting of only two variables

Chomsky Normal Form

For task (1), if the productions are of the form: A → X1 X2 …… Xm, m>2 and if some Xi is terminal a, then we replace Xi by C, placing C→a where C is a variable

As a result we will have all productions in the form:

A → B1 B2 …… Bm, m>2; where all Bi's are non-terminals

Chomsky Normal Form

For task (2), we break all productions A → B1 B2 …… Bm for m>=3, into a group of productions with two variables in each body

We introduce m-2 new variables C1, C2 ……, Cm-2

The original production is replaced by the m-1 productions:

$A \rightarrow B1\ C1,$

$C1 \rightarrow B2\ C2,$

……

$Cm\text{-}2 \rightarrow Bm\text{-}1\ Bm$

Chomsky Normal Form

Hence, all productions are in the form

$$A \rightarrow BC \text{ or } A \rightarrow a$$

This is a grammar in CNF and generates a language without ε-productions

Example

Convert the following grammar into CNF:

$$S \rightarrow AAC$$

$$A \rightarrow aAb \mid \varepsilon$$

$$C \rightarrow aC \mid a$$

Example

(1) Removing ε-productions

A is a nullable symbol as $A \rightarrow \varepsilon$

So, eliminating ε-productions, we have:

$$S \rightarrow AAC \mid AC \mid C$$

$$A \rightarrow aAb \mid ab$$

$$C \rightarrow aC \mid a$$

Example

(2) Removing unit-productions

Here, (S, C) is a unit pair as $S \rightarrow C$

So, removing unit production, we have:

$$S \rightarrow AAC \mid AC \mid aC \mid a$$

$$A \rightarrow aAb \mid ab$$

$$C \rightarrow aC \mid a$$

(3) We do not have any useless symbols

Example

Now, we can convert the grammar to CNF:

(i) For productions with bodies of length 2 or more, replace terminals by variables and add corresponding productions:

$$S \rightarrow AAC \mid AC \mid C1\ C \mid a$$

$$C1 \rightarrow a$$

$$A \rightarrow C1\ A\ B1 \mid C1\ B1$$

$$B1 \rightarrow b$$

$$C \rightarrow C1\ C \mid a$$

Example

(ii) Replace the sequence of non-terminals by a variable and introduce new productions:

Here, replace S → AAC by S → A C2, C2 → AC

Similarly, replace A → C1 A B1 by A → C1 C3, C3 → A B1

Example

Thus the final grammar in CNF form will be as:

$$S \rightarrow A\ C2 \mid AC \mid C1\ C \mid a$$

$$A \rightarrow C1\ C3 \mid C1\ B1$$

$$C1 \rightarrow a$$

$$B1 \rightarrow b$$

$$C2 \rightarrow AC$$

$$C3 \rightarrow A\ B1$$

$$C \rightarrow C1\ C \mid a$$

Practice Questions

(Q) Simplify following grammars and convert to CNF:

(I) $S \rightarrow ASB \mid \varepsilon$

$A \rightarrow aAS \mid a$

$B \rightarrow SbS \mid A \mid bb$

(II) $S \rightarrow AACD$

$A \rightarrow aAb \mid \varepsilon$

$C \rightarrow aC \mid a$

$D \rightarrow aDa \mid bDa \mid \varepsilon$

(III) $S \rightarrow aaaaS$

$S \rightarrow aaaa$

Left Recursive Grammar

A grammar is said to be left recursive if it has a non-terminal A such that there is a derivation $A \Rightarrow^* A\alpha$ for some string x

Top-down parsing methods cannot handle left recursive grammars, so a transformation that eliminates left recursion is needed

Left Recursive Grammar

Removal of immediate left recursion

Let $A \rightarrow A\alpha \mid \beta$, where β does not start with A

Then, the left recursive pair of productions could be replaced by the non-left recursive productions as:

$$A \rightarrow \beta A'$$

$$A' \rightarrow \alpha A' \mid \varepsilon$$

without changing the set of strings derivable from A

Left Recursive Grammar

Or equivalently, these production can be rewritten as with out Є-production

$A \rightarrow \beta A' | \beta$

$A' \rightarrow \alpha A' | \alpha$

Left Recursive Grammar

No matter how many A-productions there are, we can eliminate immediate left recursion from them

In general, if

$A \rightarrow A\,\alpha 1 \mid A\,\alpha 2 \mid \ldots\ldots \mid A\,\alpha m \mid \beta 1 \mid \beta 2 \mid \ldots\ldots \mid \beta n$

where βi does not start with A

Left Recursive Grammar

Then we can remove left recursion as:

$A \rightarrow \beta 1\ A' \mid \beta 2\ A' \mid \ldots\ldots \mid \beta n\ A'$

$A' \rightarrow \alpha 1\ A' \mid \alpha 2\ A' \mid \ldots\ldots \mid \alpha m\ A' \mid \varepsilon$

Equivalently, these productions can be rewritten as:

$A \rightarrow \beta 1\ A' \mid \beta 2\ A' \mid \ldots\ldots \mid \beta n\ A' \mid \beta 1 \mid \beta 2 \mid \ldots\ldots \mid \beta n$

$A' \rightarrow \alpha 1\ A' \mid \alpha 2\ A' \mid \ldots\ldots \mid \alpha m\ A' \mid \alpha 1 \mid \alpha 2 \mid \ldots\ldots \mid \alpha m$

Example

Consider a grammar:

$$S \rightarrow AA \mid 0$$

$$A \rightarrow AAS \mid 0S \mid 1$$

Make this grammar non-left recursive.

Example

Here, the production $A \rightarrow AAS$ is immediate left recursive

So, removing left recursion, we have:

$$S \rightarrow AA \mid 0$$

$$A \rightarrow 0SA' \mid 1A'$$

$$A' \rightarrow ASA' \mid \varepsilon$$

Example

Equivalently, we can write:

$$S \rightarrow AA \mid 0$$

$$A \rightarrow 0SA' \mid 1A' \mid 0S \mid 1$$

$$A' \rightarrow ASA' \mid AS$$

Practice Question

(Q) Remove left recursion from the following grammar:

$$E \rightarrow E + T \mid T$$

$$T \rightarrow T * F \mid F$$

$$F \rightarrow (E) \mid a$$

Greibach Normal Form

A grammar G = (V, T, P, S) is said to be in Greibach Normal Form (GNF) if all the productions of the grammar are of the form:

$$A \rightarrow a\alpha$$

where ‘a’ is a terminal i.e. $a \in T$ and α is a string of zero or more variables. i.e. $\alpha \in V^*$

This normal form is named after its creator Sheila Greibach

Greibach Normal Form

We can rewrite as:

$$A \rightarrow aV^* \qquad \text{with } a \in T$$

or,

$$A \rightarrow aV^+$$

$$A \rightarrow a \qquad \text{with } a \in T$$

Greibach Normal Form

Steps to convert a grammar in GNF:

- First, convert the grammar into CNF

- Remove any left recursions.

- Let the left recursion ordering is A_1, A_2, A_3, ..., A_p

- Let A_p is in GNF

- Substitute A_p in first symbol of A_{p-1}, if A_{p-1} contains A_p; then A_{p-1} is also in GNF

Greibach Normal Form

- Similarly, substitute first symbol of A_{p-2} by A_{p-1} productions and A_p productions

- Continue in a similar fashion until all productions are in GNF

The importance of GNF is that a grammar of this kind always tells us what the first terminal symbol to be derived using any given rule will be

Example

Consider an the following grammar:

$$S \rightarrow AA \mid 0$$

$$A \rightarrow SS \mid 1$$

Convert it into GNF.

Example

This grammar is already in CNF

Now, to remove left recursion, first replace symbol of A-production by S-production (since we do not have immediate left recursion) as:

$$S \rightarrow AA \mid 0$$

$$A \rightarrow AAS \mid 0S \mid 1$$

where, AS is of the form = α1, 0S of the form = β1 and 1 of the form = β2)

Now, removing the immediate left recursion:

$$S \rightarrow AA \mid 0$$

$$A \rightarrow 0SA' \mid 1A'$$

$$A' \rightarrow ASA' \mid \varepsilon$$

Equivalently, we can write it as:

$$S \rightarrow AA \mid 0$$

$$A \rightarrow 0SA' \mid 1A' \mid 0S \mid 1$$

$$A' \rightarrow ASA' \mid AS$$

Now we replace first symbol of S-production by A-productions as:

$$S \rightarrow 0SA'A \mid 1A'A \mid 0SA \mid 1A \mid 0$$

$$A \rightarrow 0SA' \mid 1A' \mid 0S \mid 1$$

$$A' \rightarrow ASA' \mid AS$$

Similarly, replacing first symbol of A'-production by A-productions, we get the grammar in GNF as:

$$S \rightarrow 0SA'A \mid 1A'A \mid 0SA \mid 1A \mid 0$$

$$A \rightarrow 0SA' \mid 1A' \mid 0S \mid 1$$

$$A' \rightarrow 0SA'SA' \mid 1A'SA' \mid 0SSA' \mid 1SA' \mid 0SA'S \mid 1A'S \mid 0SS \mid 1S$$

Practice Question

(Q) Convert into GNF:

$S \rightarrow XB \mid AA$

$A \rightarrow a \mid SA$

$B \rightarrow b$

$X \rightarrow a$

Practice Question

The given grammar is already in CNF and there is no left recursion

The production rule $A \rightarrow SA$ is not in GNF, so we substitute $S \rightarrow XB \mid AA$ in the production rule $A \rightarrow SA$ as:

$$S \rightarrow XB \mid AA$$

$$A \rightarrow a \mid XBA \mid AAA$$

$$B \rightarrow b$$

$$X \rightarrow a$$

Practice Question

The production rule S → XB and B → XBA is not in GNF, so we substitute X → a in the production rule S → XB and B → XBA as:

$$S \rightarrow aB \mid AA$$

$$A \rightarrow a \mid aBA \mid AAA$$

$$B \rightarrow b$$

$$X \rightarrow a$$

Practice Question

Now, we will remove left recursion (A → AAA) to get:

$$S \rightarrow aB \mid AA$$

$$A \rightarrow aC \mid aBAC$$

$$C \rightarrow AAC \mid \varepsilon$$

$$B \rightarrow b$$

$$X \rightarrow a$$

Practice Question

Now, we will remove null production C → ε to get:

$$S \rightarrow aB \mid AA$$

$$A \rightarrow aC \mid aBAC \mid a \mid aBA$$

$$C \rightarrow AAC \mid AA$$

$$B \rightarrow b$$

$$X \rightarrow a$$

Practice Question

The production rule S → AA is not in GNF, so we substitute A → aC | aBAC | a | aBA in production rule S → AA as:

$$S \rightarrow aB \mid aCA \mid aBACA \mid aA \mid aBAA$$

$$A \rightarrow aC \mid aBAC \mid a \mid aBA$$

$$C \rightarrow AAC$$

$$C \rightarrow aCA \mid aBACA \mid aA \mid aBAA$$

$$B \rightarrow b$$

$$X \rightarrow a$$

Practice Question

The production rule C → AAC is not in GNF, so we substitute A → aC | aBAC | a | aBA in production rule C → AAC as:

S → aB | aCA | aBACA | aA | aBAA

A → aC | aBAC | a | aBA

C → aCAC | aBACAC | aAC | aBAAC

C → aCA | aBACA | aA | aBAA

B → b

X → a

Backus-Naur Form

Backus-Naur Form(BNF) is a notation used to specify the CFG

It is named so after John Backus, who invented it, and Peter Naur, who refined it

It is a formal method for describing the syntax of programming languages

Its concept is similar to CFG, one difference is instead of using the symbol “→” in production, we use the symbol “::=”

Backus-Naur Form

Also, we enclose all non-terminals in angled brackets, “< >”

eg1: The BNF for identifiers is:

<identifier> ::= <letter or underscore> | <identifier> | <symbol>

<letter or underscore> ::= <letter> | <_>

<symbol> ::= <letter or underscore> | <digit>

<letter> ::= a | b | ..... | z

<digit> ::= 0 | 1| 2 | ..... |3

Backus-Naur Form

eg2:

<expr> ::= <term> "+" <expr> | <term>

<term> ::= <factor> " * " <term> | <factor>

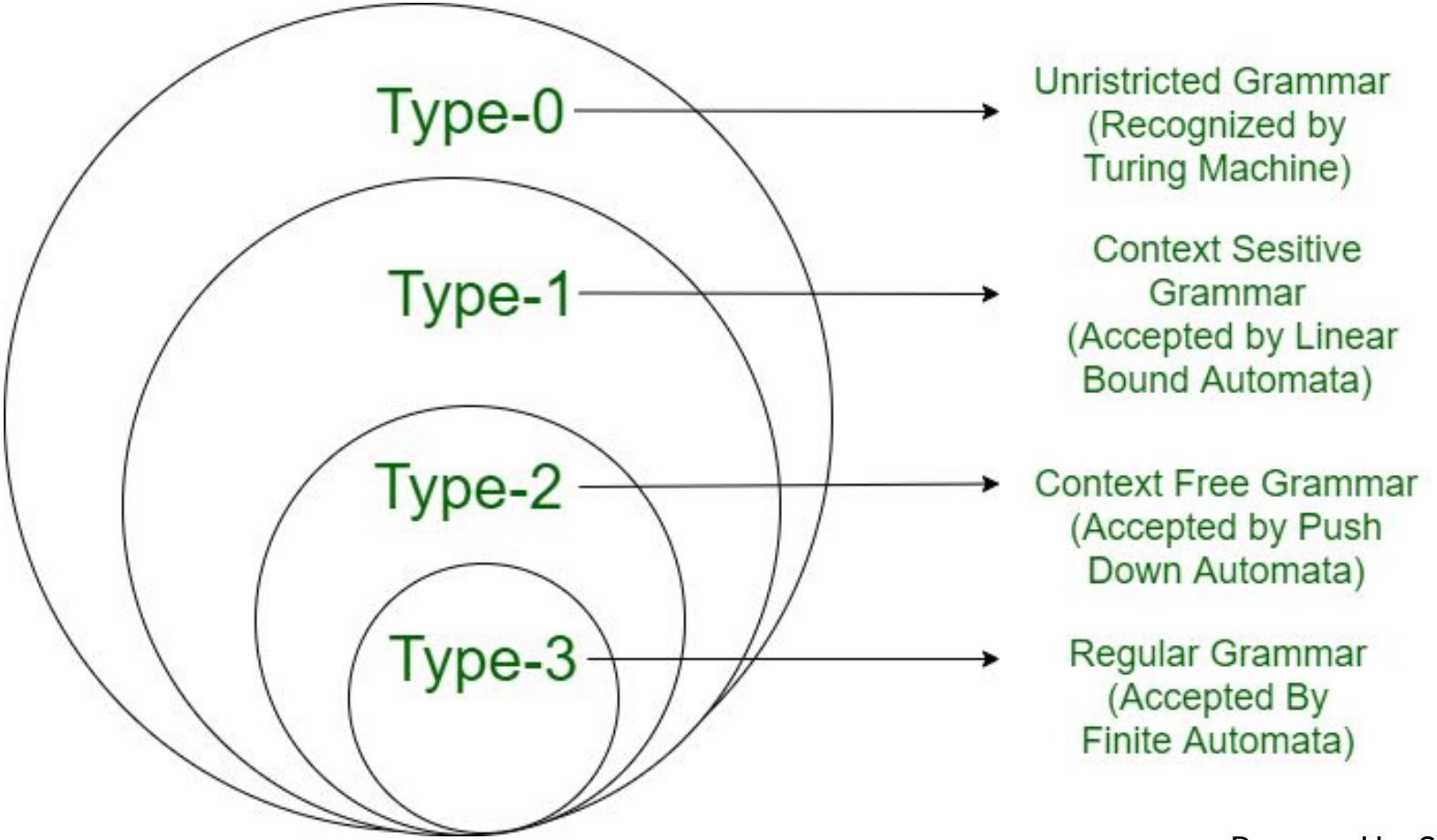
<factor> ::= "(" <expr> ")" | <const>

<const> ::= integer

Chomsky Hierarchy

According to Chomsky hierarchy, grammar is divided into 4 types:

- Type 0 - Unrestricted Grammar
- Type 1 - Context Sensitive Grammar
- Type 2 - Context Free Grammar
- Type 3 - Regular Grammar


Type-0
Unristricted Grammar
(Recognized by
Turing Machine)
Type-1
Context Sesitive
Grammar
(Accepted by Linear
Bound Automata)
Type-2
Context Free Grammar
(Accepted by Push
Down Automata)
Type-3
Regular Grammar
(Accepted By
Finite Automata)

Grammar Type	Grammar Accepted	Language Accepted	Automaton
Type 0	Unrestricted grammar	Recursively enumerable language	Turing Machine
Type 1	Context-sensitive grammar	Context-sensitive language	Linear-bounded automaton
Type 2	Context-free grammar	Context-free language	Pushdown automaton
Type 3	Regular grammar	Regular language	Finite state automaton

Unrestricted Grammar

Type 0 grammars include all formal grammar

Type 0 grammar languages are recognized by **turing machine**

These languages are also known as **recursively enumerable languages**

In type 0 grammars there must be at least one variable on the left side of production

Unrestricted Grammar

Grammar productions are of the form $\alpha \rightarrow \beta$

where, $\alpha \in (V + T)^* V (V + T)^*$ and $\beta \in (V + T)^*$

eg: $Sab \rightarrow ba$

$A \rightarrow S$

Context Sensitive Grammar

Type 1 grammars generate context-sensitive languages

The language generated by this type of grammar is recognized by **linear bounded automata**

For a grammar to be type 1,

- First of all the grammar should be type 0

- Grammar production are of the form of $\alpha A \beta \rightarrow \alpha \gamma \beta$

Context Sensitive Grammar

here, $\alpha, \beta, \gamma \in (V + T)^*$ and $A \in V$; also $|\alpha A \beta| \leq |\alpha \gamma \beta|$

- The strings α and β may be empty, but γ must be non-empty

- The rule $S \rightarrow \varepsilon$ is allowed if S does not appear on the right side of any rule

eg: $AB \rightarrow AbBc \mid bcA$

$B \rightarrow b$

Context Free Grammar

Type 2 grammars generate context free languages

The language generated by this type of grammar is recognized by a **pushdown automata**

The productions must be in the form $A \rightarrow \gamma$

where, $A \in V$ and $\gamma \in (V + T)^*$

Regular Grammar

Type 3 grammars generate regular languages

A regular grammar represents a language that is accepted by some finite automata

Regular grammar is a CFG which may be either **left linear** or **right linear**

Left Linear Regular Grammar

A grammar in which all of the productions are of the form

$$A \rightarrow Bw \quad \text{or} \quad A \rightarrow w$$

for $A, B \in V$ and $w \in T^*$ is called left linear

In this type of regular grammar, all the non-terminals of the right hand side of productions exist at the leftmost place

Left Linear Regular Grammar

eg1: $A \rightarrow a \mid Ba \mid \varepsilon$

eg2: $S \rightarrow B00 \mid S11$

$B \rightarrow B0 \mid B1 \mid 0 \mid 1$

Right Linear Regular Grammar

A grammar in which all of the productions are of the form

$$A \rightarrow wB \quad \text{or} \quad A \rightarrow w$$

for $A, B \in V$ and $w \in T^*$ is called right linear

In this type of regular grammar, all the non-terminals of the right hand side of productions exist at the rightmost place

Right Linear Regular Grammar

eg1: $A \rightarrow a \mid aB \mid \varepsilon$

eg2: $S \rightarrow 00B \mid 11S$

$B \rightarrow 0B \mid 1B \mid 0 \mid 1$

Equivalence of Regular Grammar and FA

Assume that a regular grammar is given in its right linear form

A right linear grammar, defined by G = (V, T, P, S), may be converted to a FA, defined by M = (Q, Σ, δ, q_0, F)

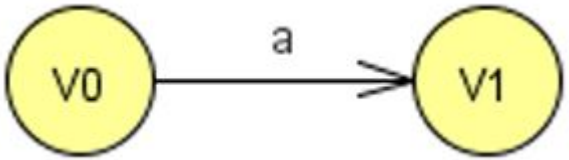
- Q = symbols α such that α is either S or a suffix from right hand side of a production in P plus the added final states

- Σ = terminal symbols of G i.e. T

Equivalence of Regular Grammar and FA

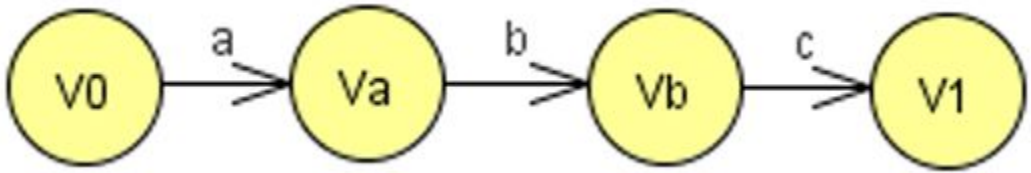
- q_0 = start symbol of G i.e. S

- If the production rule is of the form $V_i \rightarrow aV_j$, where $a \in T$, add the transition $\delta(V_i, a) = V_j$ to M. For example, $V_0 \rightarrow aV_1$ becomes:



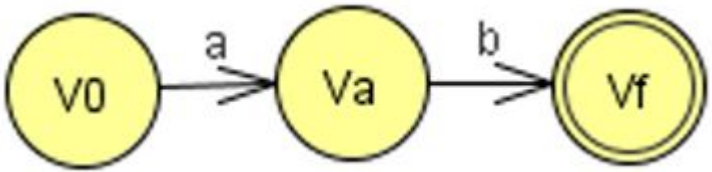
Equivalence of Regular Grammar and FA

- If the production rule is of the form $V_i \rightarrow wV_j$, where $w \in T^*$, create a series of states which derive w and end in V_j. Add the states in between to Q. For example, $V_0 \rightarrow abcV_1$ becomes:



Equivalence of Regular Grammar and FA

- If the production rule is of the form $V_i \rightarrow w$, where $w \in T^*$, create a series of states which derive w and end in a final state. For example, $V_0 \rightarrow ab$ becomes:



- For each production $V \rightarrow \varepsilon$, make the state V a final state

Example

Construct a FA for the following regular grammar:

$$S \rightarrow 0A \mid 1A$$

$$A \rightarrow 0A \mid 1A \mid +B \mid -B$$

$$B \rightarrow 0B \mid 1B \mid 0 \mid 1$$

Example

Firstly, make start symbol ‘S’ the start state of FA

Here, the transitions will be as follows:

For $S \rightarrow 0A$, $\delta(S, 0) = A$

For $S \rightarrow 1A$, $\delta(S, 1) = A$

For $A \rightarrow 0A$, $\delta(A, 0) = A$

For $A \rightarrow 1A$, $\delta(A, 1) = A$

For $A \rightarrow +B$, $\delta(A, +) = B$

For $A \rightarrow -B$, $\delta(A, -) = B$

Example

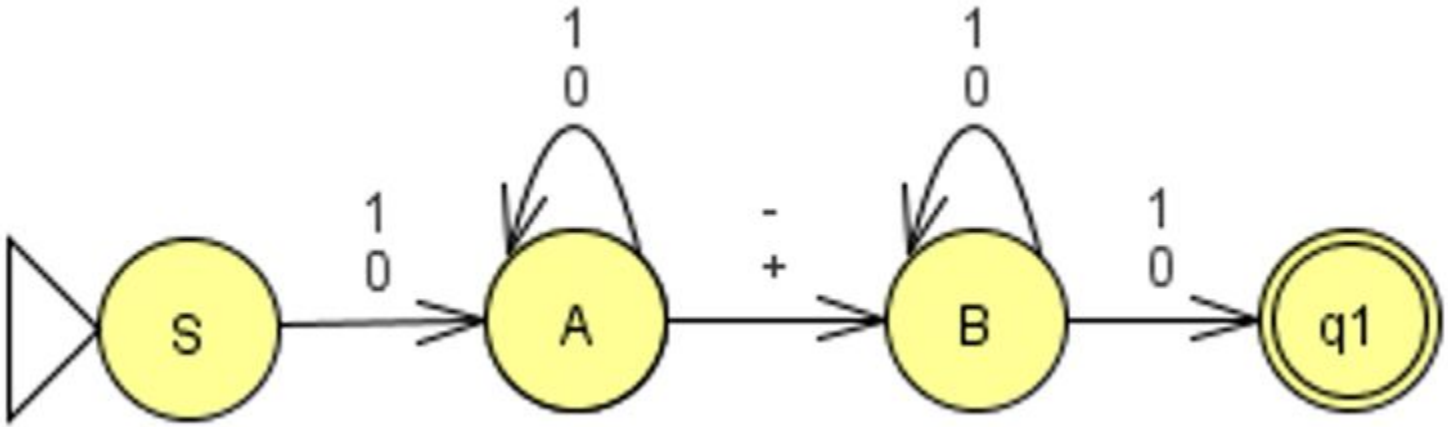
For $B \rightarrow 0, B, \delta(B, 0) = B$ $B \rightarrow 1, B\ \delta(B, 1) = B$

For $B \rightarrow 0$, $\delta(B, 0)$ = final state, let's call it q1

For $B \rightarrow 1$, $\delta(B, 1)$ = final state q1

Example

Hence, required FA is:



Practice Question

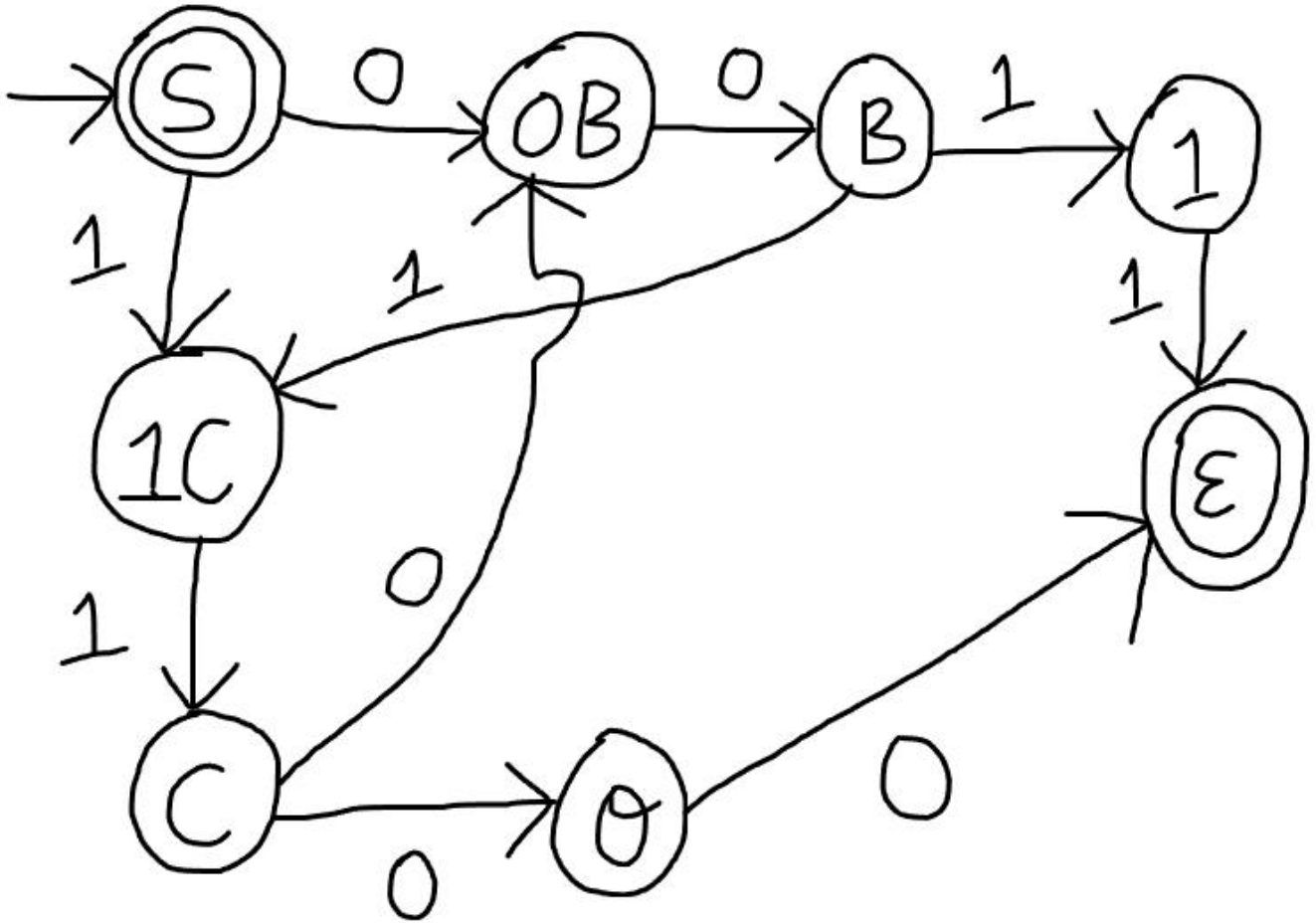
(Q) Convert the following regular grammar into a FA:

$$S \rightarrow 00B \mid 11C \mid \varepsilon$$

$$B \rightarrow 11C \mid 11$$

$$C \rightarrow 00B \mid 00$$

Ans:



Pumping Lemma for CFL

The pumping lemma for CFL says that in any sufficiently long string in a CFL, it is possible to find at most two short, nearby substrings that we can “pump” in one behind another

That is, we may repeat both of the strings i times, for any integer i, and the resulting string will still be in the language

We can use this lemma as a tool for showing that certain languages are not context free

Size of a Parse Tree

Our first step in pumping lemma for CFLs is to examine the size of parse trees

During the lemma, we will use grammar in CNF form

One of the uses of CNF is to turn parses into binary trees

Any grammar in CNF produces a parse tree for any string that is binary tree

These trees have some convenient properties, one of which can be exploited by the theorem given next

Yield of a CNF Parse Tree

Theorem: Let a terminal string w be a yield of parse tree generated by a grammar G = (V, T, P, S) in CNF. If the longest path in the tree is n, then $|w| \leq 2^{n-1}$.

Proof: We prove this theorem by simple induction on n

Basis Step: Let n = 1, then the tree consists of only the root and a leaf labeled with a terminal. So, string w is a single terminal.

Thus, $|w| = 1$ which is $\leq 2^{1-1} = 2^0$

Yield of a CNF Parse Tree

Inductive Step: Suppose n is the length of longest path and n>1. Then, the root of the tree uses a production of the form A→BC, since n>1.

No path in the subtree rooted at B and C can have length greater than n-1 since B & C are children of A. Thus, by inductive hypothesis, the yield of these subtrees are of length almost 2^{n-2}.

The yield of the entire tree is the concatenation of these two yields and therefore has length at most, $2^{n-2} + 2^{n-2} = 2^{n-1}$

Hence, $|w| \leq 2^{n-1}$. Proved.

Pumping Lemma for CFL

It states that every CFL has a special value called “pumping length” such that all strings longer than this length in the language can be pumped

The string can be divided into five parts such that the second and the fourth parts may be repeated together any number of times and the resulting string still remains in the language

Application: The pumping lemma gives us a technique to show that certain languages are not context free

Pumping Lemma for CFL

Theorem: Let L be a CFL. Then there exists a constant n (pumping length) such that if z is any string in L such that $|z| \geq n$, then we can write z = uvwxy, satisfying following conditions:

(i) $|vx| > 0$ i.e. one of v and x maybe empty, but not both

(ii) $|vwx| \leq n$ i.e. middle portion is not larger than n

(iii) for all $i \geq 0$, $uv^iwx^iy \in L$

i.e. the two strings v & x can be “pumped” zero or more times, and the resulting string will still be a member of L.

Pumping Lemma for CFL

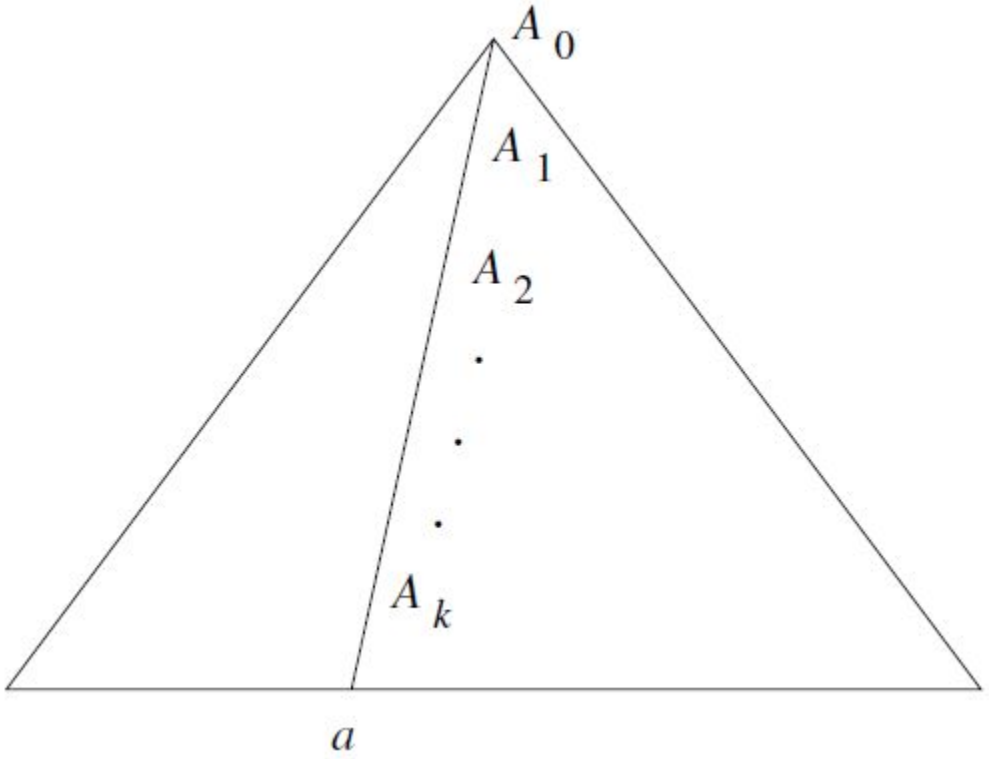
Proof: Given any context free grammar G, we can convert it to CNF. The parse tree creates a binary tree.

Let m be the number of variables in this grammar, choose $n = 2^m$. Next suppose z in L is of length at least n. i.e. $|z| \geq n$.

Any parse tree for z must have height m+1.

If it would be less than m+1, say m, then by the theorem for size of parse tree, $|z| \leq 2^{m-1} = 2^m/2 = n/2$ which is contradicting. So, it should be m+1.

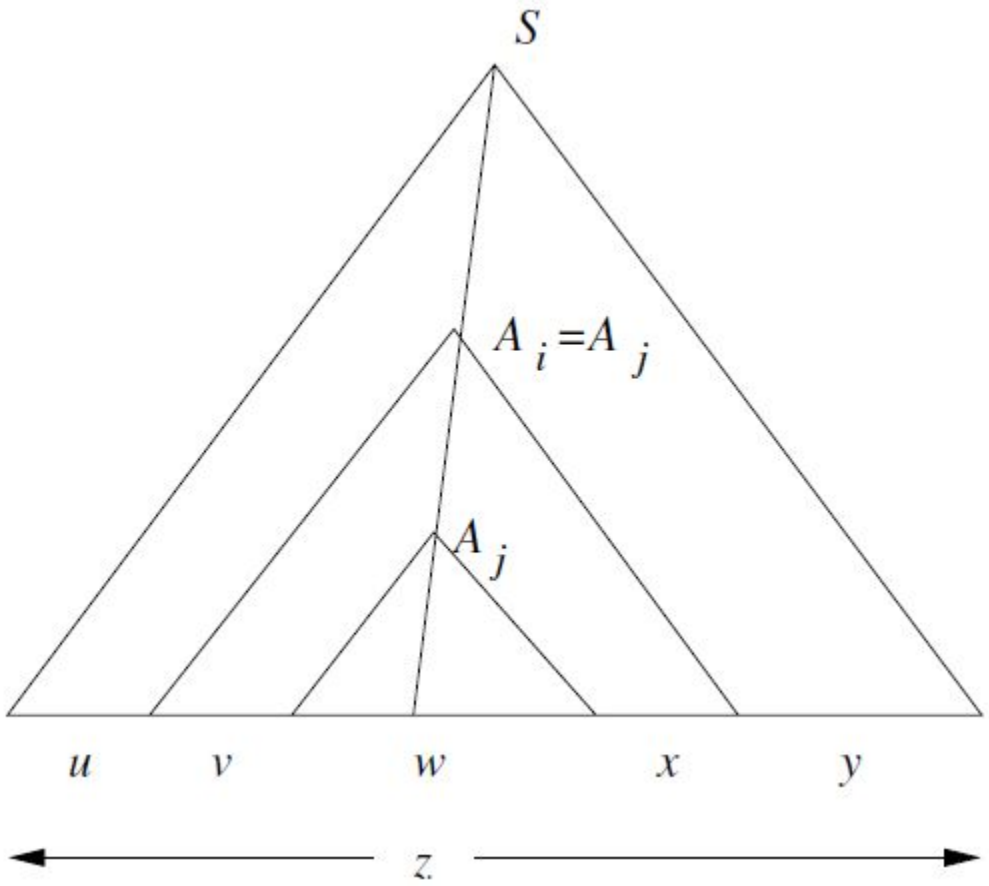
Let us consider a path of maximum length in tree for z, as shown below, where k is the least m and path is of length k.



Pumping Lemma for CFL

Since k ≥ m, there are at least m+1 occurrence of variables A_0, A_1, A_2,....., A_k on the path. But, there are only m variables in the grammar, so at least two of the m+1 variables on the path must be same(by pigeonhole principle).

Suppose $A_i = A_j$, then it is possible to divide tree.


S
A i =A j
A j
u
v
w
x
y
z

Pumping Lemma for CFL

String w is the yield of subtree rooted at A_j, string v and x are to the left and right of w in yield of larger subtree rooted at A_i.

If u and y represent beginning and end of z i.e. left and right of subtree rooted at A_i, then

$$Z = uvwxy$$

Pumping Lemma for CFL

We must use a production from A_i to A_j and it can't be a terminal or there would be no A_j.

Therefore, we must have two variables; one of these must lead to A_j and the other must lead to v or x or both.

This means v and x cannot both be empty but one might be empty.

Hence, $|vx| > 0$. [condition (i)]

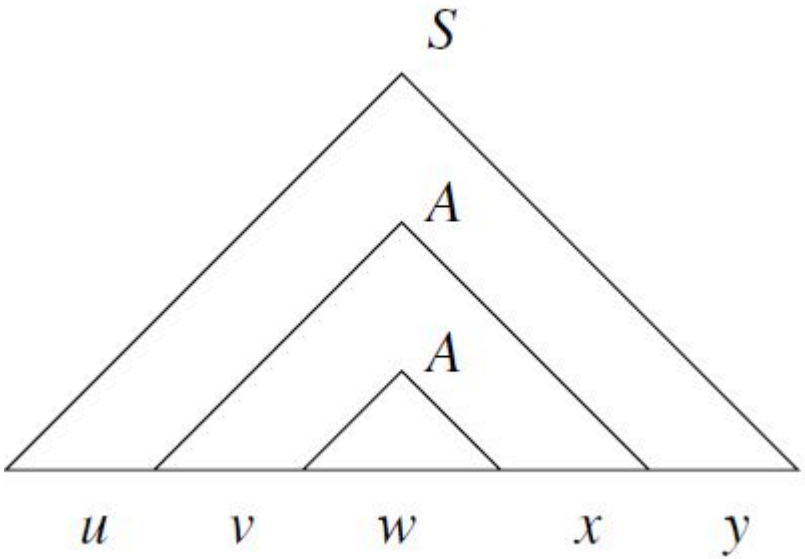
Pumping Lemma for CFL

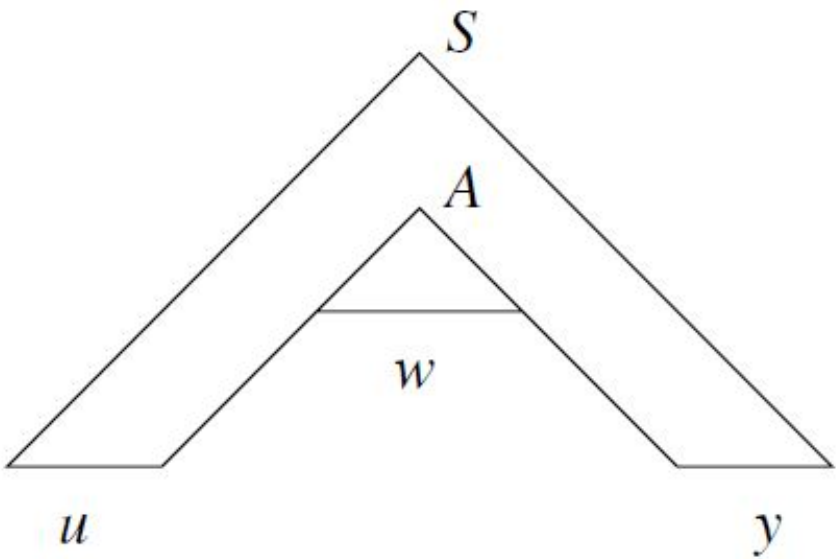
Now, we know $A_i = A_j = A$ say, as we found any two variables in the tree are same.

Then, we can construct new parse tree where we can replace subtree rooted at A_i, which has yield vwx, by subtree rooted at A_j, which has yield w.

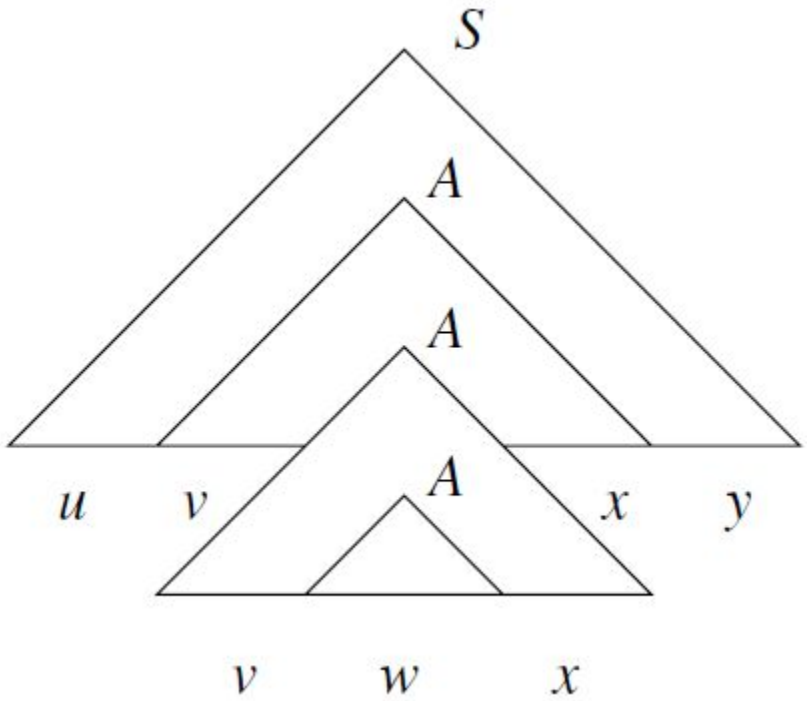
The reason we can do so is that both of these trees have root labeled A.

The resulting tree yields uv^0wx^0y as:





Another option is to replace subtree rooted at A_j by entire subtree rooted at A_i. Then, the yield will be of pattern uv^2wx^2y as:



Pumping Lemma for CFL

Were we to then replace the subtree of this tree with yield w by the larger subtree with yield vwx, we would have a tree with yield uv^3wx^3y, and so on, for any exponent i.

Hence, we can find any yield of the form $uv^iwx^iy \in L$ for any $i \geq 0$.

[condition (iii)]

Pumping Lemma for CFL

The A_i in the tree is the root of the portion which yields of vwx.

Since we begin with longest path length with height m+1, this subtree rooted at A_i has height of less than m+1.

By theorem for height of parse tree,

$$|vwx| \leq 2^{m+1-1} = 2^m = n \quad \text{i.e. } |vwx| \leq n \text{ [condition (ii)]}$$

This completes the proof.

Application of Pumping Lemma for CFL

To show a language is not context free we

- pick a language L to show that it is not a CFL

- then some n must exist, indicating the maximum yield and length of the parse tree

- we pick the string z and use n as a parameter

- break z into uvwxy subject to the pumping lemma constraints $|vwx| \leq n$, $|vx| > 0$

- We finish by picking i and showing that uv^iwx^iy is not in L, therefore L is not context free

Example

(Q) Show that the language L = $\{a^n b^n c^n \mid n \geq 1\}$ is is not context free

Ans:

Suppose that L is a CFL

Then some integer m exists and we pick z = $a^m b^m c^m$

Since z = uvwxy and |vwx| ≤ m, we know that the string vwx must consist of either:

Example

- all a’s
- all b’s
- all c’s
- combination of a’s & b’s
- combination of b’s & c’s

Example

The string vwx cannot contain a's, b's and c's because the string is not large enough to span all three symbols (because |vwx| ≤ m)

(i) If either v or x spans regions, i.e. contains a combination of two symbols, then let i = 2 (i.e., pump in once)

The resulting string will have letters out of order and thus not be in $a^m b^m c^m$

Example

(ii) If both v and x each contain only one distinct character, then set i=2

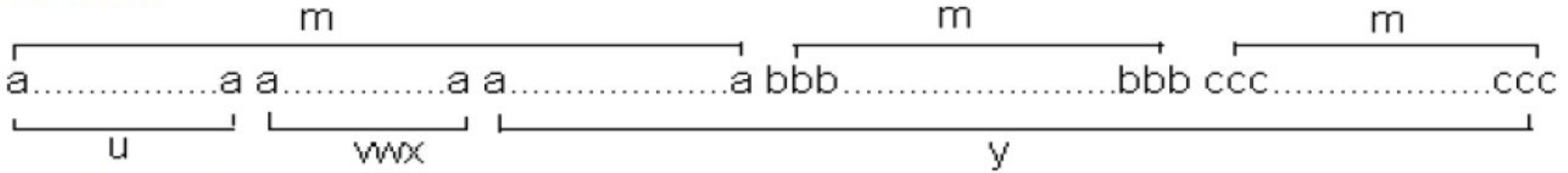
Additional copies of at most two different characters are added, leaving the third unchanged

There are no longer equal numbers of the three letters, so the resulting string is not in $a^m b^m c^m$

Hence, by contradiction, this language is not context free

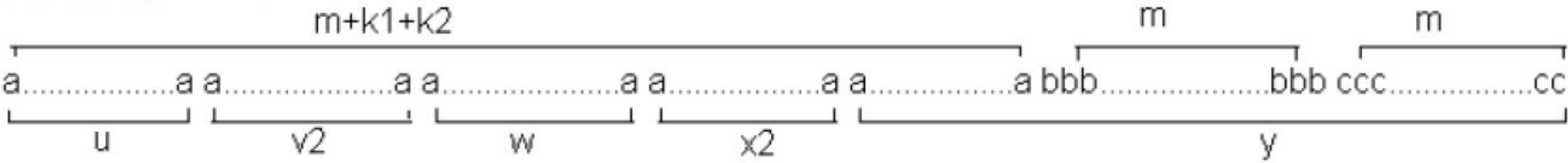
Visualization of one case: (other cases can be done similarly)

vwx is in a^m



$v = a^{k1}$, $x = a^{k2}$; $k1+k2 > 0$ i.e. $k1+k2 \geq 1$

Now pump v and x then



But, $uv^2xy^2z = a^{m+k1+k2}b^mc^m$ does not belong to L as $k1+k2 \geq 1$.
Hence, it shows contradiction to L is CFL.

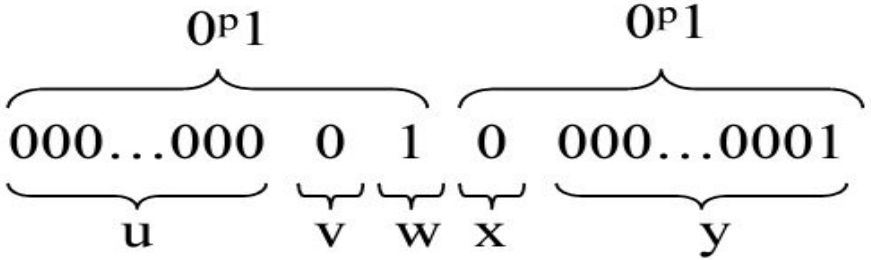
Prepared by Sherin Joshi

Example 2

- Let L be the language $\{ a^i b^j c^k \mid 0 \leq i \leq j \leq k \}$. Show that this language is not a CFL. This language is similar to the previous one, except proving that it is not context free requires the examination of more cases.
- Suppose that L is a CFL.
- Pick $z = a^p b^p c^p$ as we did with the previous language.
- As before, the string vwx cannot contain a's, b's, and c's. We then pump the string depending on the string vwx as follows:
  - There are no a's. Then we try pumping down to obtain the string uv^0wx^0y to get uwy. This contains the same number of a's, but fewer b'c or c's. Therefore it is not in L.
  - There are no b's but there are a's. Then we pump up to obtain the string uv^2wx^2y to give us more a's than b's and this is not in L.
  - There are no b's but there are c's. Then we pump down to obtain the string uwy. This string contains the same number of b's but fewer c's, therefore this is not in C.
  - There are no c's. Then we pump up to obtain the string uv^2wx^2y to give us more b's or more a's than there are c's, so this is not in C.
- Since we can come up with a contradiction for any case, this language is not a CFL language.

Example 3

- Let L be the language $\{ww \mid w \in \{0,1\}^*\}$. Show that this language is not a CFL.
- As before, assume that L is context-free and let p be the pumping length.
- This time choosing the string z is less obvious. One possibility is the string: 0^p10^p1. It is in L and has length greater than p, so it appears to be a good candidate.
- But this string can be pumped as follows so it is not adequate for our purposes:



Example 3

- This time lets try $z=0^p1^p0^p1^p$ instead. We can show that this string cannot be pumped.
- We know that $|vwx| \leq p$.
  - Let's say that the string $|vwx|$ consists of the first p 0's. If so, then if we pump this string to uv^2wx^2y then we'll have introduced more 0's in the first half and this is not in L.
  - We get a similar result if $|vwx|$ consists of all 0's or all 1's in either the first or second half.
  - If the string $|vwx|$ matches some sequence of 0's and 1's in the first half of z, then if we pump this string to uv^2wx^2y then we will have introduced more 1's on the left that move into the second half, so it cannot be of the form ww and be in L. Similarly, if $|vwx|$ occurs in the second half of z, them pumping z to uv^2wx^2y moves a 0 into the last position of the first half, so it cannot be of the form ww either.
  - This only leaves the possibility that $|vwx|$ somehow straddles the midpoint of z. But if this is the case, we can now try pumping the string down. $uv^0wx^0y = uwy$ has the form of $0^p1^i0^j1^p$ where i and j cannot both equal p. This string is not of the form ww and therefore the string cannot be pumped and L is therefore not a CFL.

Closure Properties of CFLs

Closure properties express the idea that given certain languages are context free, and a language L is formed from these languages by certain operations on them, then L is also context free

CFLs are closed under operations like union, concatenation, kleene star, reversal

CFLs are NOT closed under complement, intersection, difference

Closure of CFLs under Union

Let L and M be CFL's with grammars G and H, respectively

Assume G and H have no variables in common(names of variables do not affect the language)

Let S1 and S2 be the start symbols of G and H

Form a new grammar for L ∪ M by combining all the symbols and productions of G and H

Closure of CFLs under Union

Then, add a new start symbol S

Add productions S → S1 | S2

In the new grammar, all derivations start with S

The first step replaces S by either S1 or S2

In the first case, the result must be a string in L(G) = L, and in the second case a string in L(H) = M

Closure of CFLs under Concatenation

Let L and M be CFL's with grammars G and H, respectively

Assume G and H have no variables in common

Let S1 and S2 be the start symbols of G and H

Form a new grammar for LM by starting with all symbols and productions of G and H

Add a new start symbol S

Add production $S \rightarrow S1\ S2$

Every derivation from S results in a string in L followed by one in M

Closure of CFLs under Kleene Star

Let L have grammar G, with start symbol S1

Form a new grammar for L* by introducing to G a new start symbol S and the productions S → S1 S | ε

A rightmost derivation from S generates a sequence of zero or more S1's, each of which generates some string in L

Closure of CFLs Under Reversal

If L is a CFL with grammar G, form a grammar for L^R by reversing the right side of every production

Example:

Let G have $S \rightarrow 0S1 \mid 01$

The reversal of L(G) has grammar $S \rightarrow 1S0 \mid 10$