UNIT 4 MODELS FOR EXECUTING ALGORITHMS-II: PDFA & CFG

Structure **Page Nos.**

4.0 INTRODUCTION

We have mentioned earlier that not every problem can be solved algorithmically and that good sources of examples of such problems are provided by formal models of computation viz., FA, PDFA and TA. In the previous unit, we discussed FA. In this unit, we discuss PDFA, CFG and related topics.

4.1 OBJECTIVES

After going through this unit, you should be able to:

- explain and create context free grammar and language;
- define the pushdown automata;
- find the equivalence of context free grammar and Pushdown Automata, and
- create a grammar from language and vice versa.

4.2 FORMAL LANGUAGE & GRAMMAR

In our day-to-day life, we often use the common words such as grammar and language. Let us discuss it through one example.

Example 1: If we talk about a sentence in English language, "Ram reads", this sentence is made up of Ram and reads. Ram and reads are replaced for <noun> and <verb>. We can say simply that a sentence is changed by noun and verb and is written as

<sentence> $\rightarrow$ <noun> <verb>

where noun can be replaced with many such values as Ram, Sam, Gita…. and also <verb> can be replaced with many other values such as read, write, go …. As noun and verb are replaced, we easily write

<noun> $\rightarrow$ I
<noun> $\rightarrow$ Ram
<noun> $\rightarrow$ Sam
<verb> $\rightarrow$ reads
<verb> $\rightarrow$ writes

From the above, we can collect all the values in two categories. One is with the parameter changing its values further, and another is with termination. These

collections are called variables and terminals, respectively. In the above discussion variables are, <sentence>, <noun> and <verb>, and terminals are I, Ram, Sam, read, write. As the sentence formation is started with <sentence>, this symbol is special symbol and is called **start symbol**.

Now **formally**, a Grammar G = (V, Σ, P, S) where,

- V is called the set of variables. e.g., {S, A, B, C}
- Σ is the set of terminals, e.g., {a, b}
- P is a set of production rules
  (- Rules of the form $A \rightarrow \alpha$ where $A \in (VU\Sigma)^+$ and $\alpha \in (VU\Sigma)^+$ e.g., $S \rightarrow aA$).
- S is a special variable called the start symbol $S \in V$.

Structure of grammar: If L is a language over an alphabet A, then a grammar for L consists of a set of grammar rules of the form

$$x \rightarrow y$$

where x and y denote strings of symbols taken from A and from a set of grammar symbols disjoint from A. The grammar rule $x \rightarrow y$ is called a production rule, and application of production rule (x is replaced by y), is called derivation.

Every grammar has a special grammar symbol called the start symbol and there must be at least one production with the left side consisting of only the start symbol. For example, if S is the start symbol for a grammar, then there must be at least one production of the form $S \rightarrow y$.

Example 2: Suppose A = {a, b, c} then a grammar for the language A^* can be described by the following four productions:

$S \rightarrow \wedge$ (i)
$S \rightarrow aS$ (ii)
$S \rightarrow bS$ (iii)
$S \rightarrow cS$ (iv)

$S \Rightarrow aS \Rightarrow aaS \Rightarrow aacS \Rightarrow aacbS \Rightarrow aacb = aacb$
using prod.(u) using prod.(ii) using prod.(iv) using prod.(iii) using prod.(i)

The desired derivation of the string is aacb. Each step in a derivation corresponds to a branch of a tree and this true is called **parse tree**, whose root is the start symbol. The completed derivation and parse tree are shown in the *Figure 1,2,3*:

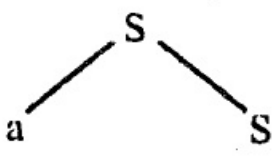

Figure 1: S ⇒ aS

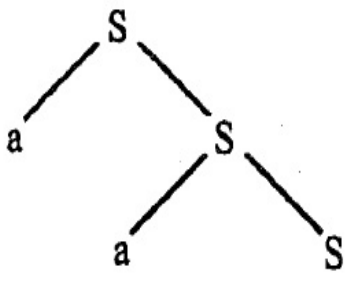

Figure 2: S ⇒ aS ⇒ aaS

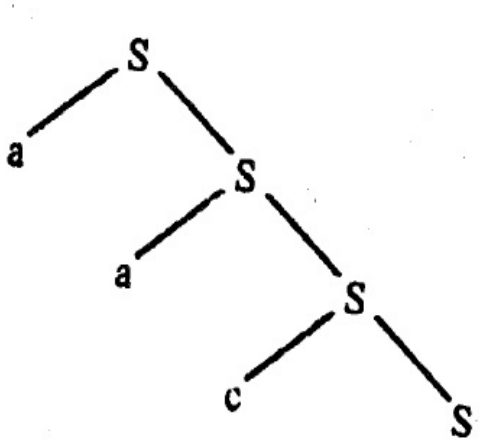

Figure 3: S ⇒ aS ⇒ aaS ⇒ aacS

Let us derive the string aacb, its parse tree is shown in *Figure 4*.

$S \Rightarrow aS \Rightarrow aaS \Rightarrow aacS \Rightarrow aacbS \Rightarrow aacb\wedge = aacb$

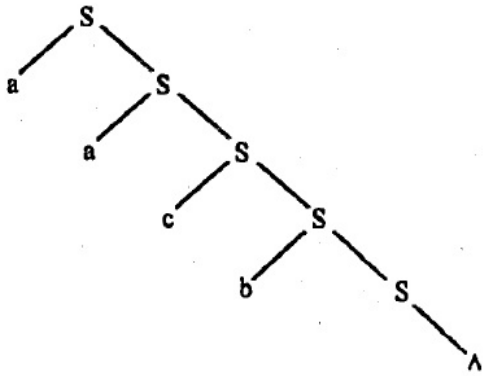

Figure 4: Parse tree deriving aacb

Sentential form: A string made up of terminals and/or non-terminals is called a sentential form.

In example 1, formally grammar is rewritten as

In G = (V, Σ, P, S) where
V = {<sentence>, <noun>, <verb>}
Σ = {Ram, reads,…}
P = <sentence> → <noun> <verb>
<noun> → Ram
<verb> → reads, and
S = <sentence>

If x and y are sentential forms and $\alpha \rightarrow \beta$ is a production, then the replacement of α by β in $x\alpha y$ is called a derivation, and we denote it by writing

$$x\alpha y \Rightarrow x\beta y$$

To the left hand side of the above production rule x is left context and y is right context. If the derivation is applied to left most variable of the right hand side of any production rule, then it is called leftmost derivation. And if applied to rightmost then is called rightmost derivation.

The language of a Grammar:

A language is generated from a grammar. If G is a grammar with start symbol S and set of terminals $\sum$, then the language of G is the set

$$L(G) = \{W \mid W \in \Sigma^* \text{ and } S \overset{*}{\underset{G}{\Rightarrow}} W\}.$$

Any derivation involves the application production Rules. If the production rule is applied once, then we write $\alpha \underset{G}{\Rightarrow} B$. When it is more than one, it is written as $\alpha \overset{*}{\underset{a}{\Rightarrow}} \beta$.

Recursive productions: A production is called recursive if its left side occurs on its right side. For example, the production $S \rightarrow aS$ is recursive. A production $A \rightarrow \alpha$ is indirectly recursive. If A derives a sentential form that contains A, Then, suppose we have the following grammar:

$$S \rightarrow b/aA$$
$$A \rightarrow c/bS$$

the productions $S \rightarrow aA$ and $A \rightarrow bs$ are both indirectly recursive because of the following derivations:

$$S \Rightarrow aA \Rightarrow abS,$$
$$A \Rightarrow bS \Rightarrow baA$$

A grammar is recursive if it contains either a recursive production or an indirectly recursive production.

A grammar for an infinite language must be recursive.

Example 3: Consider $\{\wedge, a, aa, \ldots, a^n, \ldots\} = \{a^n \mid n \geq 0\}$.

Notice that any string in this language is either $\wedge$ or of the form ax for some string x in the language. The following grammar will derive any of these strings:

$$S \rightarrow \wedge/aS.$$

Now, we shall derive the string aaa:

$$S \Rightarrow aS \Rightarrow aaS \Rightarrow aaaS \Rightarrow aaa.$$

Example 4: Consider $\{\wedge, ab, aabb, \ldots, a^n b^n, \ldots\} = \{a^n b^n \mid n \geq 0\}$.

Notice that any string in this language is either $\wedge$ or of the form axb for some string x in the language. The following grammar will derive any of the strings:

$$S \rightarrow \wedge/aSb.$$

For example, we will derive the string aaabbb;

$$S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aaaSbbb \Rightarrow aaabbb.$$

Example 5: Consider a language $\{\wedge, ab, abab, \ldots, (ab)^n, \ldots\} = \{(ab)^n \mid n \geq 0\}$.

Notice that any string in this language is either $\wedge$ or of the form abx for some string x in the language. The following grammar will derive any of these strings:

$$S \rightarrow \wedge/abS.$$

For example, we shall derive the string ababab:

$$S \Rightarrow abS \Rightarrow ababS \Rightarrow abababS \Rightarrow ababab.$$

Sometimes, a language can be written in terms of simpler languages, and a grammar can be constructed for the language in terms of the grammars for the simpler languages. We will now concentrate on operations of union, product and closure.

Suppose M and N are languages whose grammars have disjoint sets of non-terminals. Suppose also that the start symbols for the grammars of M and N are A and B, respectively. Then, we use the following rules to find the new grammars generated from M and N:

Union Rule: The language $M \cup N$ starts with the two productions

$$S \rightarrow A/B.$$

Product Rule: The language MN starts with the production.

$$S \rightarrow AB$$

Closure Rule: The language M^* starts with the production

$$S \rightarrow AS/\wedge.$$

Example 6: Using the Union Rule:

Let's write a grammar for the following language:

$$L = \{\wedge, a, b, aa, bb, \ldots, a^n, b^n, \ldots\}.$$

L can be written as union.

$$L = M \cup N,$$

Where $M = \{a^n \mid n \geq 0\}$ and $N = \{b^n \mid n \geq 0\}$.

Thus, we can write the following grammar for L:

$S \rightarrow A \mid B$ union rule,
$A \rightarrow \wedge/aA$ grammar for M,
$B \rightarrow \wedge/bB$ grammar for N.

Example 7: Using the Product Rule:

We shall write a grammar for the following language:

$$L = \{a^m b^n \mid m,n \geq 0\}.$$

L can be written as a product L = MN, where $M = \{a^m \mid m \geq 0\}$ and $N = \{b^n \mid n \geq 0\}$. Thus we can write the following grammar for L:

$S \rightarrow AB$ product rule
$A \rightarrow \wedge/aA$ grammar for M,
$B \rightarrow \wedge/bB$ grammar for N,

Example 8: Using the Closure Rule: For the language L of all strings with zero or more occurrence of aa or bb. $L = \{aa, bb\}^*$. If we let $M = \{aa, bb\}$, then $L = M^*$. Thus, we can write the following grammar for L:

$S \rightarrow AS/\wedge$ closure rule,
$A \rightarrow aa/bb$ grammar for M.

We can simplify the grammar by substituting for A to obtain the following grammar:

$$S \rightarrow aaS/bbS/\wedge$$

Example 9: Let $\Sigma = \{a, b, c\}$. Let S be the start symbol. Then, the language of palindromes over the alphabet Σ has the grammar.

$$S \rightarrow aSa/bSb/cSc/a/b/c/\wedge.$$

For example, the palindrome abcba can be derived as follows:

$$S \Rightarrow aSa \Rightarrow abSba \Rightarrow abcba$$

Ambiguity: A grammar is said to be ambiguous if its language contains some string that has two different parse tree. This is equivalent to saying that some string has two distinct leftmost derivations or that some string has two distinct rightmost derivations.

Example 10: Suppose we define a set of arithmetic expressions by the grammar:

$$E \rightarrow a/b/E–E$$

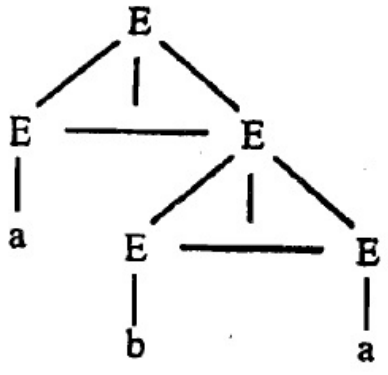

Figure 5: Parse tree

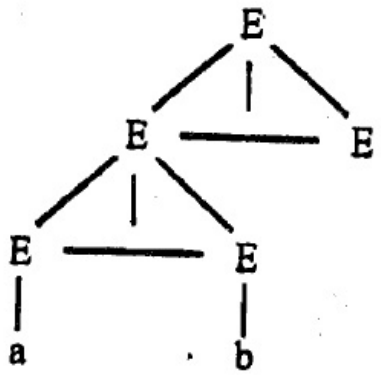

Figure 6: Parse tree showing ambiguity

This is the parse tree for an ambiguous string.

The language of the grammar $E \rightarrow a/b/E\text{-}E$ contains strings like a, b, b–a, a–b–a, and b–b–a–b. This grammar is ambiguous because it has a string, namely, a–b–a, that has two distinct parse trees.

Since having two distinct parse trees mean the same as having two distinct left most derivations.

$$E \Rightarrow E–E \Rightarrow a–E \Rightarrow a–E–E \Rightarrow a – b – E \Rightarrow a – b – a.$$
$$E \Rightarrow E–E \Rightarrow E–E–E \Rightarrow a–E–E \Rightarrow a – b – E \Rightarrow a – b – a.$$

The same is the case with rightmost derivation.

- A derivation is called a leftmost derivation if at each step the leftmost non-terminal of the sentential form is reduced by some production.
- A derivation is called a rightmost derivation if at each step the rightmost non-terminal of the sentential form is reduced by some production.

Let us try some exercises.

Ex.8) Given the following grammar

$S \rightarrow S[S]/\wedge$

For each of the following strings, construct a leftmost derivation, a rightmost derivation and a parse tree.

(a) [ ] (b) [[ ]] (c) [ ] [ ] (d) [[] [[]]]

Ex.9) Find a grammar for each language

(a) $\{a^m b^n \mid m,n \in N, n>m\}$.
(b) $\{a^m bc^n \mid n \in N\}$.

Ex.10) Find a grammar for each language:

(a) The even palindromes over {a, b}.
(b) The odd palindromes over {a, b}.

Chomsky Classification for Grammar:

As you have seen earlier, there may be many kinds of production rules. So, on the basis of production rules we can classify a grammar. According to Chomsky classification, grammar is classified into the following types:

Type 0: This grammar is also called **unrestricted grammar**. As its name suggests, it is the grammar whose production rules are unrestricted.

All grammars are of type 0.

Type 1: This grammar is also called **context sensitive grammar**. A production of the form $xAy \rightarrow x\alpha y$ is called a type 1 production if $\alpha \neq \wedge$, which means length of the working string does not decrease.

In other words, $|xAy| \leq |x\alpha y|$ as $\alpha \neq \wedge$. Here, x is left context and y is right context.

A grammar is called type 1 grammar, if all of its productions are of type 1. For this, grammar $S \rightarrow \wedge$ is also allowed.

The language generated by a type 1 grammar is called a type 1 or **context sensitive language**.

Type 2: The grammar is also known as **context free grammar**. A grammar is called type 2 grammar if all the production rules are of type 2. A production is said to be of type 2 if it is of the form $A \rightarrow \alpha$ where $A \in V$ and $\alpha \in (V \cup \Sigma)^*$. In other words, the left hand side of production rule has no left and right context. The language generated by a type 2 grammar is called **context free language**.

Type 3: A grammar is called type 3 grammar if all of its production rules are of type 3. (A production rule is of type 3 if it is of form $A \rightarrow \wedge$, $A \rightarrow a$ or $A \rightarrow aB$ where $a \in \Sigma$ and $A,B \in V$), i.e., if a variable derives a terminal or a terminal with one variable. This type 3 grammar is also called **regular grammar**. The language generated by this grammar is called **regular language**.

Ex.11) Find the highest type number that can be applied to the following grammar:

(a) $S \rightarrow ASB/b$, $A \rightarrow aA$
(b) $S \rightarrow aSa/bSb/a/b/\wedge$
(c) $S \rightarrow Aa$, $A \rightarrow S/Ba$, $B \rightarrow abc$.

4.3 CONTEXT FREE GRAMMAR (CFG)

We know that there are non-regular languages. For example, $\{a^nb^n \mid n \geq 0\}$ is non-regular language. Therefore, we can't describe the language by any of the four representations of regular languages, regular expressions, DFAs, NFAs, and regular grammars.

Language $\{a^nb^n \mid n \geq 0\}$ can be easily described by the non-regular grammar:

$$S \rightarrow \wedge/aSb.$$

So, a context-free grammar is a grammar whose productions are of the form :

$$S \rightarrow x$$

Where S is a non-terminal and x is any string over the alphabet of terminals and non-terminals. Any regular grammar is context-free. A language is context-free language if it is generated by a context-free grammar.

A grammar that is not context-free must contain a production whose left side is a string of two or more symbols. For example, the production $Sc \rightarrow x$ is not part of any context-free grammar.

Most programming languages are context-free. For example, a grammar for some typical statements in an imperative language might look like the following, where the words in bold face are considered to be the single terminals:

$$S \rightarrow \text{while E do S/ if E then S else S/\{SL\}/I: = E}$$

$$L \rightarrow SL/\wedge$$

$$E \rightarrow \ldots\text{(description of an expression)}$$
$$I \rightarrow \ldots\text{(description of an identifier).}$$

We can combine context-free languages by union, language product, and closure to form new context-free languages.

Definition: A context-free grammar, called a CFG, consists of three components:

1. An alphabet Σ of letters called terminals from which we are going to make strings that will be the words of a language.
2. A set of symbols called non-terminals, one of which is the symbols, start symbol.
3. A finite set of productions of the form

   One non-terminal $\rightarrow$ finite string of terminals and/or non-terminals.

Where the strings of terminals and non-terminals can consist of only terminals or of only non-terminals, or any combination of terminals and non-terminals or even the empty string.

The language generated by a CFG is the set of all strings of terminals that can be produced from the start symbol S using the productions as substitutions. A language generated by a CFG is called a context-free language.

Example 11: Find a grammar for the language of decimal numerals by observing that a decimal numeral is either a digit or a digit followed by a decimal numeral.

$$S \rightarrow D/DS$$
$$D \rightarrow 0/1/2/3/4/5/6/7/8/9$$

$S \Rightarrow DS \Rightarrow 7S \Rightarrow 7DS \Rightarrow 7DDS \Rightarrow 78DS \Rightarrow 780S \Rightarrow 780D \Rightarrow 780.$

Example 12: Let the set of alphabet A = {a, b, c}

Then, the language of palindromes over the alphabet A has the grammar:

$S \rightarrow aSa \mid bSb \mid cSc \mid a \mid b \mid c \mid \wedge$

For example, the palindrome abcba can be derived as follows:

$P \Rightarrow aPa \Rightarrow abPba \Rightarrow abcba$

Example 13: Let the CFG is $S \rightarrow L \mid LA$

$A \rightarrow LA \mid DA \mid \wedge$
$L \rightarrow a \mid b \mid \ldots \mid Z$
$D \rightarrow 0 \mid 1 \mid \ldots \mid 9$

The language generated by the grammar has all the strings formed by a, b,c ….z, 0, 1,…..9.

We shall give a derivation of string a2b to show that it is an identifier.

$S \Rightarrow LA \Rightarrow aA \Rightarrow aDA \Rightarrow a2A \Rightarrow a2LA \Rightarrow a2bA \Rightarrow a2b$

Context-Free Language: Since the set of regular language is closed under all the operations of union, concatenation, Kleen star, intersection and complement. The set of context free languages is closed under union, concatenation, Kleen star only.

Union

Theorem 1: if L_1 and L_2 are context-free languages, then L_1UL_2 is a context-free language.

Proof: If L_1 and L_2 are context-free languages, then each of them has a context-free grammar; call the grammars G_1 and G_2. Our proof requires that the grammars have no non-terminals in common. So we shall subscript all of G_1's non-terminals with a 1 and subscript all of G_2's non-terminals with a 2. Now, we combine the two grammars into one grammar that will generate the union of the two languages. To do this, we add one new non-terminal, S, and two new productions.

$S \rightarrow S_1$
$\mid S_2$

S is the starting non-terminal for the new union grammar and can be replaced either by the starting non-terminal for G_1 or for G_2, thereby generating either a string from L_1 or from L_2. Since the non-terminals of the two original languages are completely different, and once we begin using one of the original grammars, we must complete the derivation using only the rules from that original grammar. Note that there is no need for the alphabets of the two languages to be the same.

Concatenation

Theorem 2: If L_1 and L_2 are context-free languages, then L_1L_2 is a context-free language.

Proof : This proof is similar to the last one. We first subscript all of the non-terminals of G_1 with a 1 and all the non-terminals of G_2 with a 2. Then, we add a new nonterminal, S, and one new rule to the combined grammar:

$S \rightarrow S_1S_2$

S is the starting non-terminal for the concatenation grammar and is replaced by the concatenation of the two original starting non-terminals.

Kleene Star

Theorem 3: If L is a context-free language, then L^* is a context-free language.

Proof : Subscript the non-terminals of the grammar for L with a 1. Then add a new starting nonterminal, S, and the rules

$$S \rightarrow S_1S \mid \Lambda$$

The rule $S \rightarrow S_1S$ is used once for each string of L that we want in the string of L^*, then the rule $S \rightarrow \Lambda$ is used to kill off the S.

Intersection

Now, we will show that the set of context-free languages is not closed under intersection. Think about the two languages $L_1 = \{a^nb^nc^m \mid n,m \geq 0\}$ and $L_2 = \{a^mb^nc^n \mid n,m \geq 0\}$. These are both context-free languages and we can give a grammar for each one:

G_1:

$$S \rightarrow AB$$
$$A \rightarrow aAb \mid \Lambda$$
$$B \rightarrow cB \mid \Lambda$$

G_2:

$$S \rightarrow AB$$
$$A \rightarrow aA \mid \Lambda$$
$$B \rightarrow bBc \mid \Lambda$$

The strings in L_1 contain the same number of a's as b's, while the strings in L_2 contain the same number of b's as c's. Strings that have to be both in L_1 and in L_2, i.e., strings in the intersection, must have the same numbers of a's as b's and the same number of b's as c's.

Thus, $L_1 \cap L_2 = \{a^nb^nc^n \mid n \geq 0\}$. Using Pumping lemma for context-free languages it can be proved easily that $\{ a^n b^n c^n \mid n \geq 0\}$ is not context-free language. So, the class of context-free languages is **not closed** under intersection.

Although the set is not closed under intersection, there are cases in which the intersection of two context-free languages is context-free. Think about regular languages, for instance. All regular languages are context-free, and the intersection of two regular languages is regular. We have some other special cases in which an intersection of two context-free languages is context, free.

Suppose that L_1 and L_2 are context-free languages and that $L_1 \subseteq L_2$. Then $L_2 \cap L_1 = L_1$ which is a context-free language. An example is EQUAL $\cap \{a^nb^n\}$. Since strings in

$\{a^nb^n\}$ always have the same number of a's as b's, the intersection of these two languages is the set $\{a^nb^n\}$, which is context-free.

Another special case is the intersection of a regular language with a non-regular context-free language. In this case, the intersection will always be context-free. An example is the intersection of $L_1 = a^+b^+a^+$, which is regular, with L_2= PALINDROME. $L_1 \cap L_2 = \{a^nb^ma^n \mid m,n \geq 0\}$. This language is context-free.

Complement

The set of context-free languages is not closed under complement, although there are again cases in which the complement of a context-free language is context-free.

Theorem 4: The set of context-free languages is not closed under complement.
Proof: Suppose the set is closed under complement. Then, if L_1 and L_2 are context-free, so are L_1' and L_2'. Since the set is closed under union, $L_1' \cup L_2'$ is also context-free, as is $(L_1' \cup L_2')'$. But, this last expression is equivalent to $L_1 \cap L_2$ which is not guaranteed to be context-free. So, our assumption must be incorrect and the set is not closed under complement.

Here is an example of a context-free language whose complement is not context-free. The language $\{a^nb^nc^n \mid n \geq 1\}$ is not context-free, but the author proves that the complement of this language is the union of seven different context-free languages and is thus context-free. Strings that are not in $\{a^nb^nc^n \mid n \geq 1\}$ must be in one of the following languages:

1. $M_{pq} = \{a^pb^qc^r \mid p,q,r \geq 1 \text{ and } p>q\}$ (more a's than b's)
2. $M_{qp} = \{a^pb^qc^r \mid p,q,r \geq 1 \text{ and } q>p\}$ (more b's than a's)
3. $M_{pr} = \{a^pb^qc^r \mid p,q,r \geq 1 \text{ and } s>r\}$ (more a's than c's)
4. $M_{rp} = \{a^pb^qc^r \mid p,q,r \geq 1 \text{ and } r>p\}$ (more c's than a's)
5. M = the complement of $a^+b^+c^+$ (letters out of order)

Using Closure Properties

Sometimes, we can use closure properties to prove that a language is not context-free. Consider the language our author calls DOUBLEWORD = $\{ww \mid w \in (a+b)^*\}$. Is this language context-free? Assume that it is. Form the intersection of DOUBLEWORD with the regular language $a^+ b^+ a^+ b^+$, we know that the intersection of a context-free language and a regular language is always context-free. The intersection of DOUBLEWORD and is $a^nb^ma^nb^m \mid n,m \geq 1\}$. But, this language is not context-free, so DOUBLEWORD cannot be context-free.

Think carefully when doing unions and intersections of languages if one is a superset of the other. The union of PALINDROME and $(a+b)^*$ is $(a+b)^*$, which is regular. So, sometimes the union of a context-free language and a regular language is regular. The union of PALINDROME and a^* is PALINDROME, which is context-free but not regular.

Now try some exercises:

Ex.12) Find CFG for the language over $\Sigma = \{a,b\}$.

(a) All words of the form

$a^x b^y a^z$, where x, y, z = 1,2,3… and y = 5x+7z

(b) For any two positive integers p and q, the language of all words of the form $a^x b^y a^z$, where x, y, z = 1, 2, 3… and y = px + qz.

4.4 PUSHDOWN AUTOMATA (PDA)

Informally, a pushdown automata is a finite automata with stack. The corresponding acceptor of context-free grammar is pushdown automata. There is one start state and there is a possibly empty-set of final states. We can imagine a pushdown automata as a machine with the ability to read the letters of an input string, perform stack operations, and make state changes.

The execution of a PDA always begins with one symbol on the stack. We should always specify the initial symbol on the stack. We assume that a PDA always begins execution with a particular symbol on the stack. A PDA will use three stack operations as follows:

(i) The **pop** operation reads the top symbol and removes it from the stack.

(ii) The **push** operation writes a designated symbol onto the top of the stack. For example, push (x) means put x on top of the stack.

(iii) The **nop** does nothing to the stack.

We can represent a pushdown automata as a finite directed graph in which each state (i.e., node) emits zero or more labelled edges. Each edge from state i to state j labelled with three items as shown in the *Figure 7*, where L is either a letter of an alphabet or $\wedge$, S is a stack symbol, and 0 is the stack operation to be performed.

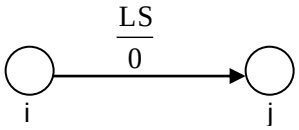

Figure 7: Directed graph

It takes fine pieces of information to describe a labelled edge. We can also represent it by the following 5-tuple, which is called a PDA instruction.

(i, L, S, 0, j)

An instruction of this form is executed as follows, where w is an input string whose letters are scanned from left to right.

If the PDA is in state i, and either L is the current letter of w being scanned or L = $\wedge$, and the symbol on top of the stack is S, then perform the following actions:

(1) execute the stack operation 0;
(2) move to the state j; and
(3) if L $\neq \wedge$, then scan right to the next letter of w.

A string is accepted by a PDA if there is some path (i.e., sequence of instructions) from the start state to the final state that consumes all letters of the string. Otherwise, the string is rejected by the PDA. The language of a PDA is the set of strings that it accepts.

Nondeterminism: A PDA is deterministic if there is at most one move possible from each state. Otherwise, the PDA is non-deterministic. There are two types of non-determinism that may occur. One kind of non-determinism occurs exactly when a state emits two or more edges labelled with the same input symbol and the same stack symbol. In other words, there are two 5-tuples with the same first three components. For example, the following two 5-tuples represent nondeterminism:

(i, b, c, pop, j)
(i, b, c, push(D), k).

The second kind of nondeterminism occurs when a state emits two edges labelled with the same stack symbol, where one input symbol is $\wedge$ and the other input symbol is not. For example, the following two 5-tuples represent non-determinism because the machine has the option of consuming the input letter b or cleaning it alone.

(i, $\wedge$, c, pop, j)
(i, b, c, push(D), k).

Example 14: The language $\{a^n b^n \mid n \geq 0\}$ can be accepted by a PDA. We will keep track of the number of a's in an input string by pushing the symbol Y onto the stack for each a. A second state will be used to pop the stack for each b encountered. The following PDA will do the job, where x is the initial symbol on the stack:

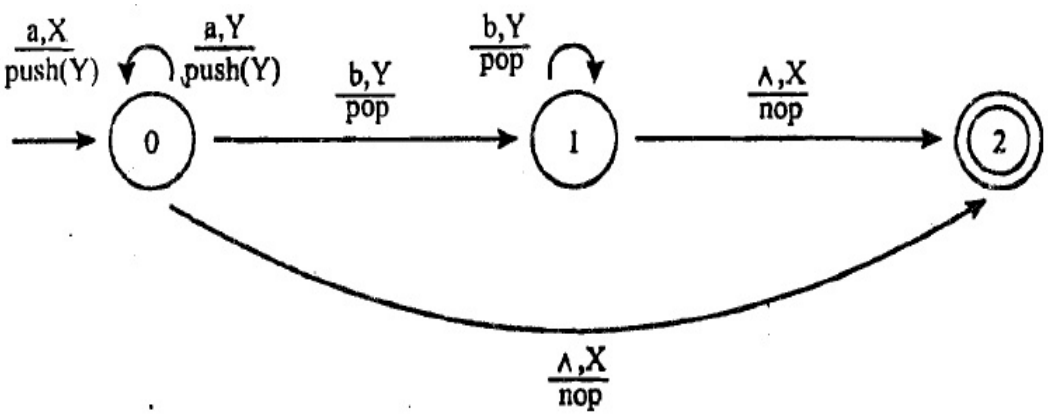

Figure 8: Pushdown automata

The PDA can be represented by the following six instructions:

(0, $\wedge$, X, nop, 2)
(0, a, X, push(Y), 0),
(0, a, Y, push(Y), 0),
(0, b, Y, pop,1),
(1, b, Y, pop,1),
(1, $\wedge$, X, nop,2).

This PDA is non-deterministic because either of the first two instructions in the list can be executed if the first input letter is a and X is on the top of the stack. A computation sequence for the input string aabb can be written as follows:

(0, aabb, X) start in state 0 with X on the stack,
(0, abb, YX) consume a and push Y,
(0, bb, YYX) consume a and push Y,
(1, b, YX) consume b and pop.
(0, $\wedge$, X) consume b and pop .
(2, $\wedge$, X) move to the final state.

Equivalent Forms of Acceptance:

Above, we defined acceptance of a string by a PDA in terms of final state acceptance. That is a string is accepted if it has been consumed and the PDA is in a final state. But, there is an alternative definition of acceptance called empty stack acceptance, which requires the input string to be consumed and the stock to be empty, with no requirement that the machine be in any particular state. The class of languages accepted by PDAs that use empty stack acceptance is the same class of languages accepted by PDAs that use final state acceptance.

Example 15: (An empty stack PDA): Let's consider the language $\{a^n b^n \mid n \geq 0\}$, the PDA that follows will accept this language by empty stack, where X is the initial symbol on the stack.

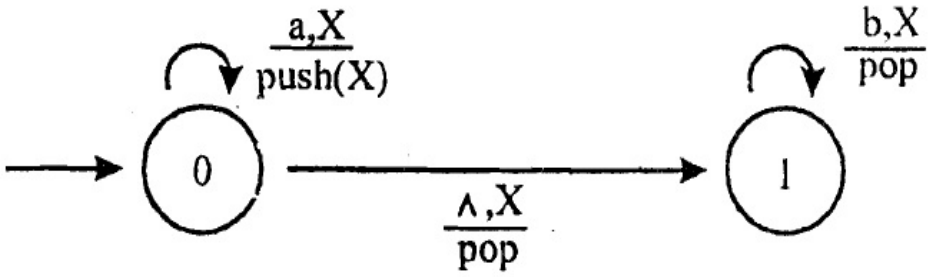

Figure 9: Pushdown automata

PDA shown in *Figure 9* can also be represented by the following three instructions:

(0, a, X, push (X), 0),
(0, ∧, X, pop, 1),
(1, b, X, pop, 1).

This PDA is non-determinstic. Let's see how a computation proceeds. For example, a computation sequence for the input string aabb can be as follows:

(0, aabb, X) start in state 0 with X on the stack
(0, abb, XX) consume a and push X
(0, bb, XXX) consume a and push X
(1, bb, XX) pop.
(1, b, X) consume b and pop
(1, ∧, ∧) consume b and pop (stack is empty)

Now, try some exercises.

Ex.13) Build a PDA that accepts the language odd palindrome.

Ex.14) Build a PDA that accepts the language even palindrome.

4.5 SUMMARY

In this unit we have considered the recognition problem and found out whether we can solve it for a larger class of languages. The corresponding accepter for the context-free languages are PDA's. There are some languages which are not context free. We can prove the non-context free languages by using the pumping lemma. Also in this unit we discussed about the equivalence two approaches, of getting a context free language. One approach is using context free grammar and other is Pushdown Automata.

4.6 SOLUTIONS/ANSWERS

Ex.1)

(a) $S \rightarrow S[S] \rightarrow [S] \rightarrow [\ ]$

(b) $S \rightarrow S[S] \rightarrow [S] \rightarrow [S[S\ ]] \rightarrow [[S] \rightarrow [[]]$.

Similarly rest part can be done.

Ex.2)

(a) $S \rightarrow aSb/aAb$
$A \rightarrow bA/b$

Ex.3)

(a) $S \rightarrow aSa/bSb/\wedge$

(b) $S \rightarrow aSa/bSb/a/b.$

Ex.4)

(a) $S \rightarrow ASB$ (type 2 production)
$S \rightarrow b$ (type 3 production)
$A \rightarrow aA$ (type 3 production)

So the grammar is of type 2.

(b) $S \rightarrow aSa$ (type 2 production)
$S \rightarrow bSb$ (type 2 production)
$S \rightarrow a$ (type 3 production)
$S \rightarrow b$ (type 3 production)
$S \rightarrow \wedge$ (type 3 production)

So the grammar is of type 2.

(c) Type 2.

Ex.5)

(a) $S \rightarrow AB$
$S \rightarrow aAb^5/\wedge$
$B \rightarrow b^7Ba/\wedge$

(b) $S \rightarrow AB$
$A \rightarrow aAb^p/\wedge$
$B \rightarrow b^qBa/\wedge$

Ex.6)

Suppose language is $\{wcw^T : w \in \{a,b\}^*\}$ then pda is

(0, a, x, push (a), 0), (0, b, x, push (b), 0),
(0, a, a, push (a), 0), (0, b, a, push (b), 0),
(0, a, b, push (a), 0), (0, b, b, push (b), 0),
(0, c, a, nop, 1), (0, c, b, nop, 1),
(0, c, x, nop, 1), (1, a, a, pop, 1),
(1, b, b, pop, 1), (1, $\wedge$, x, nop, 2),

Ex.7)

Language is $\{ww^T : w \in \{a,b\}^*\}$. Similarly as Ex 6.

4.7 FURTHER READINGS

1. *Elements of the Theory of Computation,* H.R. Lewis & C.H.Papadimitriou, PHI, (1981).

2. *Introduction to Automata Theory, Languages, and Computation* (*II Ed.)*, J.E. Hopcroft, R.Motwani & J.D.Ullman, Pearson Education Asia (2001).

3. *Introduction to Automata Theory, Language, and Computation,* J.E. Hopcroft and J.D. Ullman, Narosa Publishing House (1987).