UNIT - III

Context-Free Grammars

Definition of Context-Free Grammars, Derivations Using a Grammar, Leftmost and Rightmost Derivations, the Language of a Grammar, Sentential Forms, Parse Tress, Applications of Context-Free Grammars, Ambiguity in Grammars and Languages.

Push Down Automata

Definition of the Pushdown Automaton, the Languages of a PDA, Equivalence of PDA's and CFG's, Acceptance by final state, Acceptance by empty stack, Deterministic Pushdown Automata. From CFG to PDA, From PDA to CFG.

UNIT - III

K. ANJU ARAVIN

CONTEXT-FREE GRAMMARS

To define a Grammar formally, and introduce the process of "derivation," whereby it is determined which strings are in the language of the grammar.

A Context-free grammar is a formal notation for expressing such recursive definitions of languages.

A grammar consists of one or more variables that represent classes of strings, (ie) languages.

DEFINITION OF CONTEX-FREE GRAMMAR:-

CFG stands for Context free grammar. It is a formal grammar which is used to generate all possible patterns of strings in a given formal language.

Context-free grammar G by its can be defined four components or tuples.

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

Where V - is the set of variables
T - the terminals
P - the set of productions
S - the start symbol

There are four important components in a grammatical description of a language.

⇝ There is a finite set of symbols that forms the strings of the language being defined. This set {0, 1}, this alphabet the terminals or terminal symbols.

∴ T is the final set of a terminal symbols.

... denoted by lowercase letters.

⇝ There is a finite set of variables, also called sometimes nonterminals or syntactic categories. Each variable represents a language (ie) a set of strings.

∴ V is the final set of non-terminal symbol.
It is denoted by capital letters.

⇝ One of the variable represents the language bein defined; it is called the start symbol.
∴ S is the start symbol used to derive the string.

⇝ There is a finite set of productions or rules that represent the recursive definition of a language.

Each production consists of:

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

(b). The production symbol $\rightarrow$

(c). ~~The~~ A string of zero or more terminals and variables. This string, called the body of the production.

∴ P is the set of production rules, which is used ~~[illegible]~~ for replacing non-terminals symbols (on left side of production) in a string with other terminals or non-terminals symbols (on right side of production.

$$\left.\begin{cases} E \rightarrow E+E \\ S \rightarrow AS \\ S \rightarrow b/a \end{cases}\right\}$$

Left side should be capital letter

Right side should be capital / lower case letter

$V = \{\underbrace{A \ B \ S \ E}_{\text{Non-terminal}}\}$

$T = \{\underbrace{a, b, c, \ldots}_{\text{terminal}}\}$

∴ A set of productions that say essentially

Example: the same thing as this regular expression.

$E \rightarrow I$ $\quad$ $E \rightarrow E * E$

$E \rightarrow E + E$ $\quad$ $E \rightarrow (E)$

$I \to a \quad I \to Ia \quad I \to I0$
$I \to b \quad I \to Ib \quad I \to I1$

A context free grammar for simple expressions

Example.
① Construct CFG for language having any no. of a's over the set $\Sigma = \{a\}$

Solution:

$\because G = \{V, T, P, S\}$

$\Sigma = \{a\}$

derive i/p: aaaaaa

$L = \{\varepsilon, a, aa, aaa, aaaa \ldots\}$

derive i/p = "aaaaaa"

$R.E = a^*$

Production Rule:

$S \to aS \to ①$

$S \to \epsilon \to ②$

S
$\Rightarrow aS \quad S \to aS$
$\Rightarrow aaS \quad S \to aS$
$\Rightarrow aaaS \quad S \to aS$
$\Rightarrow aaaaS \quad S \to aS$
$\Rightarrow aaaaaS \quad S \to aS$
$\Rightarrow aaaaaaS \quad S \to aS$
$\Rightarrow aaaaaa \quad S \to \epsilon$

② Construct a CFG for language.

$L = \{\underset{\text{string}}{w} c \underset{\text{reverse string}}{w^R} \mid \text{where } w \in (a,b)^*\}$

$G = \{V, T, P, S\}$

$\because w = ab$
$w^R = ba$
$w = abb$
$w^R = bba$

Solution:

$L = \{aacaa, bcb, abcba, abbcbba, \ldots\ldots\}$

The grammar could be,

i/p = "abbcbba"

$S \to aSa \to ①$
$S \to bSb \to ②$
$S \to c \to ③$

$S \to aSa$
$S \to abSba$ - using rule ②
$S \to abbSbba$ - from rule ②
$S \to abbcbba$ - from rule ③

③ Construct a CFG for the language -

$L = a^n b^{2n}$ where $n >= 1$.

Solution:

$n=1 \quad n=2 \quad n=3$

$L = \{ \underline{abb}, aabbbb, aaabbbbbb, \dots \}$

$\underline{i/p}$: aaabbbbbb

The Grammar Could be,

$S \to aSbb \to ①$

$S \to abb \to ②$

$S \to aSbb$

$S \to aaSbbbb$ from ①

$S \to aaabbbbbb$ from ②

DERIVATIONS USING A GRAMMAR:-

The productions of a CFG to infer that certain strings are in the language of a certain variable.

There are two approaches to this inference.

① The more conventional approach is to use the rules from body to head.

②. To defining the language of a grammar, in which we use the production from head to body.

Derivation is a sequence of production rules. It is used to get i/p strings. During parsing, we have to take two decisions

- We have to decide the non-terminal which is to be replaced.
- We have to decide the production rule by which the non-terminal will be replaced.

We have two options to decide which non-terminal to be placed with production rule.

① Left Most Derivation → $S \to a\boxed{A}\boxed{B}b$ Left Right

②. Right most Derivation.

BASIS: For any string α of terminals and variables, we say $\alpha \overset{*}{\underset{G}{\Rightarrow}} \alpha$.

That is, any string derives itself.

INDUCTION: If $\alpha \overset{*}{\Rightarrow} \beta$ and $\beta \underset{G}{\Rightarrow} \gamma$, then $\alpha \overset{*}{\underset{G}{\Rightarrow}} \gamma$. (ie) If α can become β by zero or more steps and one more step takes β to γ, then α can become γ.

Put another way, $\alpha \overset{*}{\underset{G}{\Rightarrow}} \beta$ means that there is a sequence of strings $\gamma_1, \gamma_2, \ldots \gamma_n$, for some $n \geq 1$, such that

$E \Rightarrow E * E$
$E \Rightarrow (E) * E$
$\Rightarrow (E+E) * E$
$(I+E) * E$

1. $\alpha = \gamma_1$
2. $\beta = \gamma_n$, and
3. for $i = 1, 2, \ldots, n-1$, we have $\gamma_i = \gamma_{i+1}$.

LEFTMOST AND RIGHTMOST DERIVATIONS:-

In the left most derivation, the input is scanned and replaced with the production rule from left to right. So we have to read input string from left to right.

Ex: Production Rules

$E = E + E$
$E = E - E$
$E = a/b$.

Input :- $a - b + a$

The left most derivation is

$E = E + E$
$E = E - E + E$
$E = a - E + E$
$E = a - b + E$
$E = a - b + a$.

In the right most derivation, the input is scanned and replaced with the production rule from right to left. So, we have to read the input string from right to left.

Ex: Production Rule

$E = E + E$
$E = E - E$
$E = a/b$

Input : $a - b + a$

The right most derivation is

$E = E * E$
$E = E - E + E$
$E = E - E + a$
$E = E - b + a$
$E = a - b + a$.

Depart

Example.

Derive the String "abb" for left most Derivation and Right most derivation using CFG given by

$S \to AB | \epsilon$

$A \to aB$

$B \to Sb.$

Solution:

Left most Derivation:

$S \to AB$	S
$A \to aB$	A B
$B \to Sb$	aB B
$S \to \epsilon$	aSb B
$B \to Sb$	aϵb B
$S \to \epsilon$	ab B
	ab Sb
	ab ϵb
	ab b

Right Most Derivation:

S	$S \to AB$
A B	$B \to Sb$
A Sb	$S \to \epsilon$
A ϵ b	$A \to aB$
aB b	$B \to Sb$
aSb b	$S \to \epsilon$
aϵb b	
ab b.	

②. Derive the String "00101" for left most derivation and right most derivation using CFG given

$S \to A1B$

$A \to 0A | \epsilon$

$B \to 0B | 1B | \epsilon$

If a language L is the language of some Context-free grammar, then L is said to be a Context free language or CFL.

Theorem: $L(G_{pal})$, where G_{pal} is the grammar, the set of palindromes over $\{0,1\}$

Proof: We shall prove that a string w in $\{0,1\}^*$ is in $L(G_{pal})$ if and only if it is a palindrome; (ie) $w = w^R$.

(If) Suppose w is a palindrome. We show by induction on $|w|$ that w is in $L(G_{pal})$.

Basis:- We use length 0 and 1 as the basis. If $|w| = 0$ or $|w| = 1$, then w is ϵ, 0, or 1.

Since there are productions $P \to \epsilon$, $P \to 0$ and $P \to 1$, we conclude that $P \overset{*}{\Rightarrow} w$ in any of these basis cases.

Induction: Suppose $|w| \geq 2$. Since $w = w^R$, w must begin and end with the same symbol

(ie) $w = 0x0$ or $w = 1x1$.

$\therefore$ x ~~is~~ must be a palindrome

(ie) $x = x^R$.

If $w = 0x0$, $P \overset{*}{\Rightarrow} x$.

There is a derivation of w from P, namely $P \Rightarrow 0P0 \overset{*}{\Rightarrow} 0x0 = w$.

If $w = 1x1$,

The production $P \to 1P1$

(Only if) We assume that w is in $L(G_{pal})$

(ie) $P \overset{*}{\Rightarrow} w$

w is a palindrome.

An induction on the number of steps in a derivation of

Solution:

Left Most Derivation		Right Most Derivation	
S		S	
A1B	S → A1B	A1B	S → A1B
0A1B	A → 0A	A10B	B → 0B
00A1B	A → 0A	A101B	B → 1B
001B	A → ε	A101	B → ε
0010B	B → 0B	0A101	A → 0A
00101B	B → 1B	00A101	A → 0A
00101	B → ε	00101	A → ε

Right Most \ Derivation

③. Deeive the String "aabbabba" for leftmost derivation and rightmost derivation using a CFG.

$$S \rightarrow aB \mid bA$$
$$A \rightarrow a \mid aS \mid bAA$$
$$B \rightarrow b \mid bS \mid aBB$$

Solution:-

Left Most derivation		Right Most derivation	
S		S	
aB	S → aB	aB	S → aB
aaBB	B → aBB	aaBB	B → aBB
aabB	B → b	aaBbS	B → bS
aabbS	B → bS	aaBbbA	S → bA
aabbaB	S → aB	aaBbba	A → a
aabbabS	B → bS	aabSbba	B → bS
aabbabbA	S → bA	aabbAbba	S → bA
aabbabba	A → a	aabbabba	A → a

THE LANGUAGE OF A GRAMMAR:-

The language L(G) of a CFG G = (V, T, P, S) is

$$L(G) = \{w \mid w \in T, S \overset{*}{\underset{G}{\Rightarrow}} w\}.$$

The language of a CFG is called a Context-free Language (CFL).

BASIS:

Use one of the three productions that do not have P in the body.

(ie) The derivation is $P \Rightarrow \varepsilon$, $P \Rightarrow 0$ or $P \Rightarrow 1$.

$\because$ $\varepsilon, 0$, and 1 are all palindromes.

Induction: Suppose that the derivation takes $n+1$ steps, where $n \geq 1$, and the statement is true for all derivations of n steps.

(ie) If $P \overset{*}{\Rightarrow} x$ in n steps, then x is a palindrome.

Consider an $(n+1)$-step derivation of w, which must be of the form

$$P \Rightarrow 0P0 \overset{*}{\Rightarrow} 0x0 = w$$

(or)

$$P \Rightarrow 1P1 \overset{*}{\Rightarrow} 1x1 = w$$

$\because$ $n+1$ steps is at least two step, and the productions $P \rightarrow 0P0$ and $P \rightarrow 1P1$ are the only productions whose use allows additional steps of a derivation.

$$P \overset{*}{\Rightarrow} x \text{ in } n \text{ steps.}$$

W.K.T x is a palindrome; ie $x = x^R$.

then $0x0$ and $1x1$ are also palindrome.

$$(0x0)^R = 0x^R0 = 0x0.$$

$\therefore$ w is a palindrome

(※). Examples are in page No: 95

PARSE TREES:-

There is a tree representation for derivations that has proved extremely useful.

In a compiler, the tree structure of the source program facilitates the translation of the source program into executable

Departm

code by allowing natural, recursive functions to perform this translation process.

We introduce the parse tree and show that the existence of parse trees is tied closely to the existence of derivations and recursive inferences.

<u>Constructing Parse Trees</u>:-

Given a grammar $G=(V,T,P,S)$, the parse tree is defined as :

Each interior node is labeled by a variable in V.

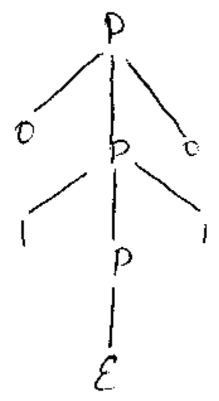
Each leaf is labeled by either a variable, a terminal or ε.

However, if the leaf is labeled ε, then it must be the only child of its parent.

If an interior node is labeled A, and its children are labeled $x_1, x_2, \ldots x_k$ respectively, from the left, then $A \rightarrow x_1 x_2 \ldots x_k$ is a production in P.

Note that the only time one of the x's can be ε is, when ε is the label of the only child, and $A \rightarrow \varepsilon$ is a production of G.

<u>Example</u>: A parse tree of derivation $P \overset{*}{\Rightarrow} 0110$ of the palindrome.



⇒ The production used at the root is $P \rightarrow 0P0$,

⇒ The middle child of the root it is $P \rightarrow 1P1$.

⇒ The bottom is a use of the production $P \rightarrow \varepsilon$.

A parse tree showing the derivation $P \overset{*}{\Rightarrow} 0110$

The Yield of a Parse Trees

The yield of a parse tree is the string obtained by concatenating all the leaves from the left, like 01ε10 = 0110 for the tree of the ~~least~~ last example.

1) The yield is a terminal string. ie all leaves are labeled either with a terminal or with ε.

2). The root is labeled by the start symbol.

Showing the yields of the parse trees of a grammar G is another way to describe the language of G (provable).

Inference, Derivations and Parse Trees :-

Given a grammar G = (V, T, P, S) the following facts are all equivalent :

1) The recursive inference procedure determines that terminal string w is in the language of variable A.

2). $A \overset{*}{\Rightarrow} w$

3). $A \underset{lm}{\overset{*}{\Rightarrow}} w$

4) $A \underset{rm}{\overset{*}{\Rightarrow}} w$

5). There is a parse tree with root A and yield w.

Equivalences of the facts in the previous page are proved by theorems in in a way as shown in the following diagram

$$\begin{tikzcd}
 & \text{Leftmost derivation} \arrow[ddl] & & & \\
 & & & \text{Parse tree} \arrow[ull] \arrow[dl] & \\
\text{Derivation} \arrow[drrrr, bend right] & & \text{Rightmost derivation} \arrow[ll] & & \\
 & & & & \text{Recursive Inference} \arrow[uul]
\end{tikzcd}$$

Proving the equivalent of certain statements about grammars.

From Inferences to Trees:-

Theorem: Let $G = (V, T, P, S)$ be a CFG.

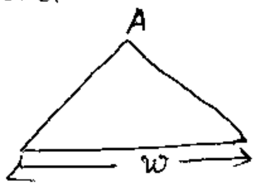
If the recursive inference procedure tells us that terminal string w is in the language of variable A, then there is a parse tree with root A and yield w.

Proof: The proof is an induction on the number of steps used to infer that w is in the language of A.

Basis: There must be a production $A \rightarrow w$.

There is one leaf for each position of w,

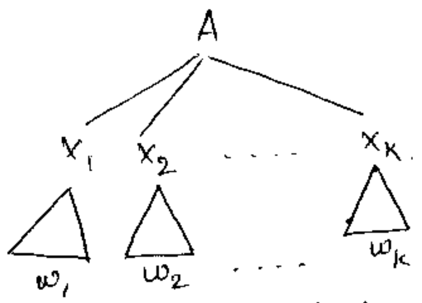
The conditions to be a parse tree for grammar G and it evidently has yield w and root A.



Tree constructed in the basis case.

1) If X_i is a terminal, then $w_i = X_i$; i.e., w_i consists of only this one terminal from the production.

2). If X_i is a variable, then w_i is a string



Tree used in the inductive part of the proof

From Trees to Derivations:-

To construct a leftmost derivation from a parse tree. The method for constructing a rightmost derivations uses the same ideas and not explore the rightmost-derivation case.

From Derivations to Recursive Inference:-

There is a derivation $A \overset{*}{\Rightarrow} w$ for some CFG, then the fact that w is in the language of A is discovered in the ~~in~~ recursive inference procedure.

We have a derivation $A \Rightarrow x_1 x_2 \dots x_k \overset{*}{\Rightarrow} w$.

If x_i is a variable, we can obtain the derivation of $x_i \overset{*}{\Rightarrow} w_i$ by starting with the derivation $A \overset{*}{\Rightarrow} w_i$

a). All the positions of the sentential forms that are either to the left or right of the positions that are derived from x_i and

b). All the steps that are not relevant to the derivation of w_i from x_i.

Easy to understand Parse tree :- [Short Explanation]

It is a graphical representation for the derivation of the given production rules for a given CFG. It is simple way to show how the derivation can be done to obtain some string from a given set of production rules. It is also called as Parse tree.

→ Parse tree follows the precendence of operates. the deepest subtree traversed first.

So, the operators in the parent node has less precedence over the operator in the subtree.

A parse tree contains the following properties:-

1) The root node is always a node indicating start symbols.

2). The derivation is read from left to right.

3). The leaf node is always terminal node.

4). The interior nodes are always the non-terminal node

Example:

E = E+E

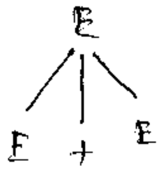
E = E*E

E = a|b|c

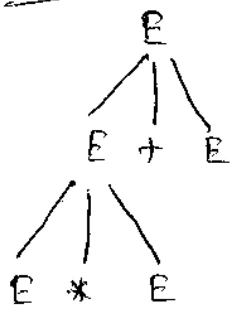
Input 1:

a*b+c

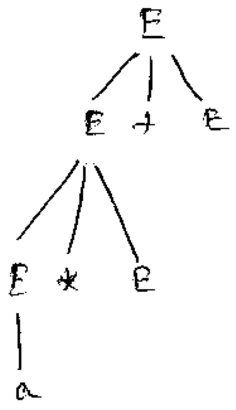
Step 1:



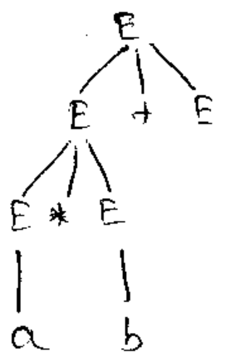
Step 2:



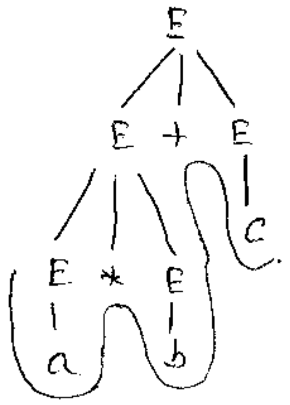
Step 3:



Step 4:



Step 5:



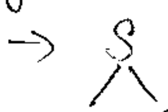
Example: 1:

Construct a derivation tree for the string "aabbabba

for C.F.G.
$S \rightarrow aB | bA$
$A \rightarrow a | aS | bAA$
$B \rightarrow b | bS | aBB$

Solution:

LeftMost Derivation:



$\rightarrow S$ $S \rightarrow aB$

$\rightarrow a\ aBB$ $B \rightarrow aBB$

$\rightarrow aabSB$ $B \rightarrow bS$

$\rightarrow aabbAB$ $S \rightarrow bA$

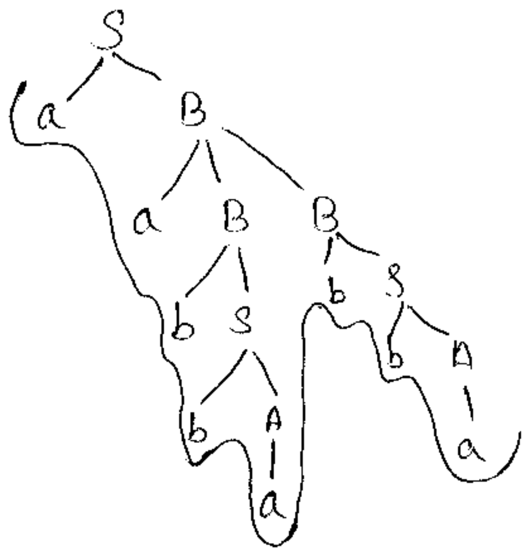
$\rightarrow aabbaB$ $A \rightarrow a$

$\rightarrow aabbabS$ $B \rightarrow bS$

$\rightarrow aabbabbA$ $S \rightarrow bA$

$\rightarrow aabbabba$ $A \rightarrow a$

derivation tree:

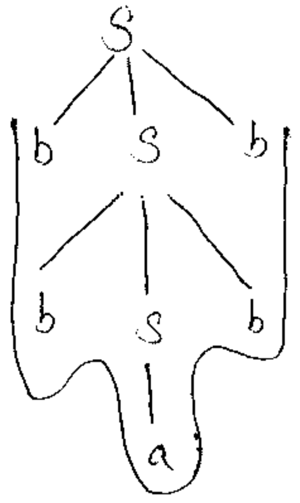


(2). Draw a derivative tree for the string "bbabb" from the C.F.G. Given by.

$S \to bSb \mid a \mid b$

~~$S \to$~~

Solution :-



$\Rightarrow$ bbabb

Example: Consider the grammar:
(3).

$E \Rightarrow E+E$
$E \to E*E$
$E \to (E)$
$E \to id.$

Obtain derivation tree of expression.
(i). id * id + id.
(ii). (id + id) * id.

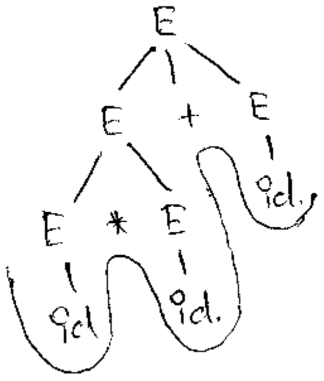
$G = \{V, T, P, S\}$
$V = \{E\}$
$T = \{id, +, *, (,)\}$
$S = E.$

Solution:
i). id * id + id.

E

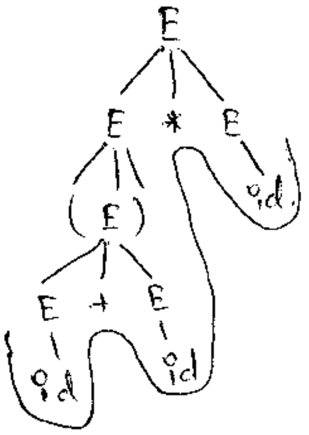
$E \to E+E$
$E \to E*E+E$
$E \to id*id+id.$

$\Rightarrow$



ii). (id + id) * id.

$E = E*E$
$E = (E)*E$
$E = (E+E)*E$
$= (id+id)*id.$



APPLICATIONS OF CONTEXT-FREE GRAMMAR :-

CFG's were originally conceived by N. Chomsky for describing natural languages. But not all natural languages can be so described.

Two Applications of CFG's :-

⇒ Grammars are used to describe programming languages.

There is a mechanical way for turning a CFG into a parser.

⇒ Describing DTD's in XML's -

DTD - Document type definition.

XML - Extensible markup language.

Parsers :-

A programming language have a structure that may be described by regular expressions.

There are components in programming languages which are not RL's.

Two examples of non-RL structures ...

Balanced structures like paranthesis "( )", "begin-end", pair, "if-else" pair, ...

Unbalanced structures like unbalanced "if else" pairs...

Eg:. A balanced if-else pair in C is

if (condition) Statement, else statement;

Example.

A Grammar $G_{bal} = (\{B\}, \{(,)\}, P, B)$ Generates all and only the strings of balanced parentheses, where P consis of the Productions :

$P : B \rightarrow BB \mid (B) \mid \varepsilon$

e.g., x = (())
()()
....

A grammar for unbalanced if-else pairs

$S \rightarrow \epsilon \mid SS \mid iS \mid iSeS$

where i = if..., e = else ....

The production $S \rightarrow iS$ is used to generate unbalance "if" (with no matching "else")

eg.. The following is a generated segment of a C Program.

```
if (condition) {
    ...
    if (condition) Statement;
    else Statement;
    ...
    if (Condition) Statement;
    else Statement;
    ...
}
```

The YACC Parser-Generator:

Input to YACC is a CFG with each production being associated additionally with an action ({...} below)

Example: A CFG in the YACC notation identical to that

```
1      E → I          Exp : Id           {...}
       E → E+E           | Exp '+' Exp   {...}
       E → E*E           | Exp '*' Exp   {...}
       E → (E)           |'(' Exp ')'    {...}
                         ;
```

```
I → a            Id : 'a'
I → b              | 'b'      {...}        EXP &
I → Ia             | Id 'a'   {...}        Id
I → Ib             | Id 'b'   {...}
I → I0             | Id '0'   {...}        E and I
I → I1             | Id '1'   {...}
                   ;
```

Markup Languages:-

The strings in a markup language are documents with "marks", called tags which specify semantics of the strings.

An example... HTML (HyperText Markup Language) for webpage design, including 2 functions.

Creating links between documents

Describing formats ("looks") of documents.

Example: HTML of A Webpage.

The thing I hate:

1). Moldy bready

2) People who drive too slow in the fast lane.

(a) The text as viewed on webpage.

```
<P> The things I <EM> hate </EM>:
<OL>
<LI> Moddy bread
<L1> People who drive too slow in the fast lane
</OL>
```

(b) The HTML Source.

Meaning of the tags:-

<P> .... paragraph (unmatched single tag)

<EM> ... </EM> ... emphasizing text ... (matched tag pair)

<LI> ... list item. (unmatched single tag)

<OL> ... </OL> ... ordered list (matched tag pair)

<P> The thing I <EM> hate </EM>
<OL>
<LI> Moldy bread.
<LI> People who drive too slow in the fast lane.
</OL>

Part of an HTML grammar :-

$$
\begin{aligned}
\text{Char} &\rightarrow a \mid A \mid \ldots \\
\text{Text} &\rightarrow \varepsilon \mid \text{Char Text} \\
\text{Doc} &\rightarrow \varepsilon \mid \text{Element Doc} \\
\text{Element} &\rightarrow \text{Text} \mid \\
&\quad \text{<EM> Doc </EM>} \mid \\
&\quad \text{<P> Doc} \mid \\
&\quad \text{<OL> List </OL>} \mid \ldots \\
\text{ListItem} &\rightarrow \text{<LI> Doc} \\
\text{List} &\rightarrow \varepsilon \mid \text{ListItem List.}
\end{aligned}
$$

__XML and Document-Type Definitions (DTD's)__

An XML (extensible markup language) describes the semantics of the text in a document using DTD's, while an HTML describes the format of a document.

The DTD is itself described with a language which is essential in the form of a CFG mixed with regular expressions.

The form of a DTD is

<! DOCTYPE name-of-DTD [
list of element definitions
]>

The form of an element definition is

<! Element element-name (description of the element)>

Descriptions of elements are essentially Regular expressions.

The Regular expression may be described in the following way:

An element may appear in another element.

The special term # PCDATA stands for any text involving no tags.

The allowed Operators are

* A Vertical bar | means union
* A Comma, denotes Concatenation
* 3 Variants of the closure operator ... *, +, ?

as described before (* : zero or more Occurences of ;
+ : One or more Occurences of ;
? : Zero or more occurrence of)

<u>Example</u>: A DTD for describing Personal Computers (PC's)

```
<!DOCTYPE PcSpecs [
  <! ELEMENT PCS (PC*)>
   <! ELEMENT PC (MODEL, PRICE, PROCESSOR, RAM, DISK+)>
   <! ELEMENT MODEL (#PCDATA)>
   <! ELEMENT PRICE (#PCDATA)>
   <! ELEMENT
<!ELEMENT PROCESSOR (MANF, MODEL, SPEED)>
<!ELEMENT MANF(#PCDATA)>
<!ELEMENT MODEL(# PCDATA)>
<!ELEMENT SPEED(#PCDATA)>
<!ELEMENT RAM (#PCDATA)>
<!ELEMENT DISK(HARDDISK|CD | DVD)>
<!ELEMENT HARDDISK (MANF, MODEL, SIZE)>
<!ELEMENT SIZE (# PCDATA)>
<!ELEMENT CD( SPEED)>
<!ELEMENT DVD(SPEED)>
]>
```

Example: A DTD for describing personal Computers (PC's)

Each element is represented in the document by a tag with the name of the element and a matching at the end with an extra slash, just as in HTML.

for example:

```
<!Element MODEL (# PCDATA)> → <MODEL>4560</MODEL>
<!ELEMENT PCS(PC*)⇒
  <PCS>
<PC>
 ...
</PC>
<PC>
 ...
</PC>
  </PCS>
```

SENTENTIAL FORMS:

Derivations from the Start Symbol produce strings are called sentential forms.

That is, given a CFG $G = (V, T, P, S)$, if $S \overset{*}{\Rightarrow} \alpha$ where $\alpha \in (V \cup T)^*$, then α is a sentential form.

If $S \underset{lm}{\overset{*}{\Rightarrow}} \alpha$ where $\alpha \in (V \cup T)^*$, then α is a left-sentential form.

If $S \underset{rm}{\overset{*}{\Rightarrow}} \alpha$ where $\alpha \in (V \cup T)^*$, then α is a right-sentential form.

Example:

Consider the grammar for expressions. $E * (I + E)$ is a sentential form, since there is a derivation.

Solution:

$$\begin{aligned} E &\Rightarrow E * E \\ &\Rightarrow E * (E) \\ &\Rightarrow E * (E + E) \\ &\Rightarrow E * (I + E) \end{aligned}$$

This derivation is neither leftmost nor rightmost, since at the last step, the middle E is replaced.

A left-sentential form, consider $a * E$, with the leftmost derivation. $E \underset{lm}{\Rightarrow} E * E \underset{lm}{\Rightarrow} I * E \underset{lm}{\Rightarrow} a * E$

Additionally, the derivation
$E \underset{rm}{\Rightarrow} E * E \underset{rm}{\Rightarrow} E * (E) \underset{rm}{\Rightarrow} E * (E + E)$

Shows that $E * (E + E)$ is a right-sentential form.

② Consider the grammar $E \rightarrow id \mid E + E \mid E * E \mid (E)$
Left Sentential form.
Derivation for a string $id * id * id$.

$E \rightarrow E * E$
$\rightarrow id * E$
$\rightarrow id * E + E$
$\rightarrow id * id + E$
$\rightarrow id * id + id.$

Right sentential form
Derivation of $id * id + id$

$E \rightarrow E * E$
$\rightarrow E * E + E$
$\rightarrow E * E + id$
$\rightarrow E * id + id$
$\rightarrow id * id + id$

Derivation tree or Parse tree

Derivations in a CFG can be represented using trees called derivation trees.

A parse tree for a CFG $G = (V, T, P, S)$ is a tree satisfying following condition.

All the leaf nodes of the tree are labelled by the terminals of the grammar.

The root node of the tree labelled by the start symbol.

Interior nodes are labelled by non terminals

If an interior node is A and has n descendants with label $x_2, x_2, \ldots x_n$ from left to right, then there must exist a production rule.

$A \rightarrow x_1, x_2, \ldots x_n$ in the grammar.

<u>Example</u>:

Consider a grammar whose production rule is
$E \rightarrow E + E \mid E * E \mid id$

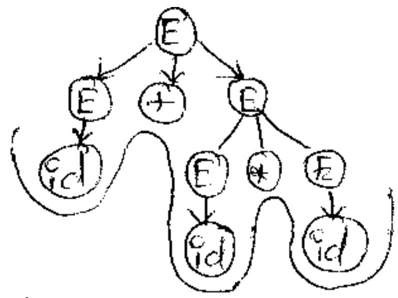
Derivation for a string $id + id * id$.

$E \rightarrow E + E$
$\rightarrow id + E$

$\rightarrow id + E * E$

$\rightarrow id + id * E$

$\rightarrow id + id * id$



Types of derivation tree

A derivation $A \Rightarrow w$ is called a leftmost derivation if we apply a production only to the leftmost non terminal at every step.

The tree representation of left most derivation is left most derivation tree

A derivation $A \Rightarrow w$ is called a right most derivation if we apply a production only to the rightmost non terminal at every step.

In a mixed derivation the string is obtained by applying productions to the leftmost variable and rightmost variable simultaneously as per the requirement in each successive step.

Example:

① Consider a CFG whose production is given below

$S \rightarrow aAS \mid a$

$A \rightarrow SbA \mid SS \mid ba$. Show that $S \Rightarrow aabbaa$ and construct a derivation tree.

Solution:

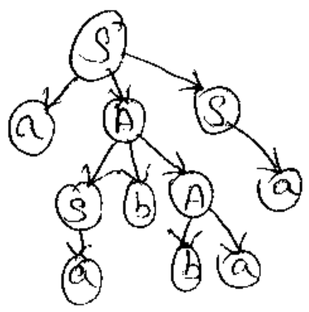
$S \rightarrow a\underline{A}S$

$\rightarrow a\underline{S}bAS$

$\rightarrow aab\underline{A}S$

$\rightarrow aabba\underline{S}$

$\rightarrow aabbaa$

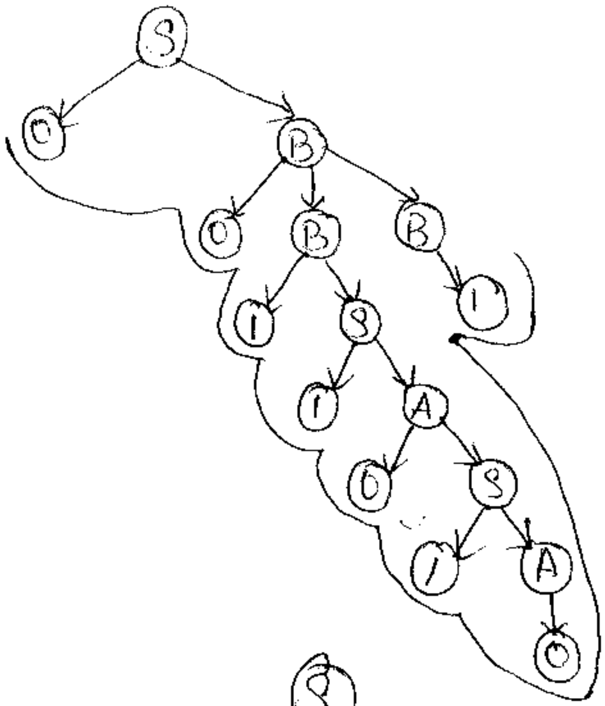


②. Consider a CFG $S \rightarrow 0B \mid 1A$, $A \rightarrow 0 \mid 0S \mid 1AA$, $B \rightarrow 1 \mid 1S \mid 0BB$. For the string 00110101. Find a) leftmost derivation b) rightmost derivation c) Parse tree.

Solution:

Leftmost Derivation:-

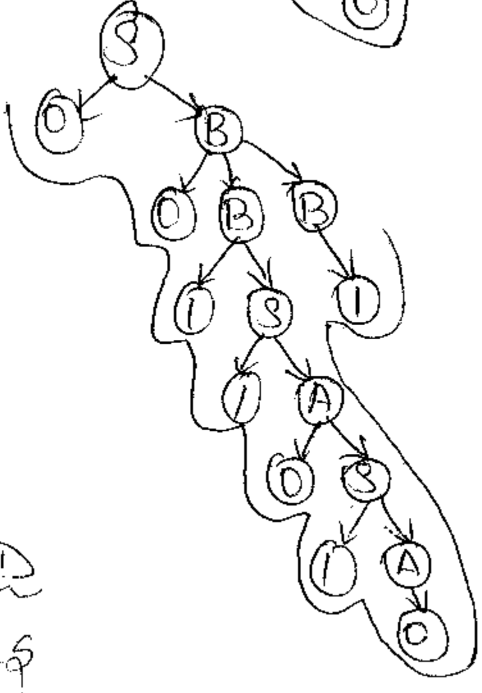
S → 0B
→ 00BB
→ 001SB
→ 0011AB
→ 00110SB
→ 001101AB
→ 0011010B
→ 00110101



Rightmost Derivation:-

S → 0B

→ 00BB	→ 00BB
→ 00B1S	→ 00B1
→ 00B10A	→ 001S1
→ 00B101	→ 001A1
→ 001S101	→ 0011A1
→ 001	→ 0011011
	→ 00110101



AMBIGUITY IN GRAMMARS AND LANGUAGES

A grammar is said to be ambiguous if there exists
more than one left most derivation (or)
more than one right most derivation (or)
more than one parse tree for given input string.

If the grammar is not ambiguous then we call unambiguous

If the grammar has ambiguity, then it is not good for compiler construction

No method can automatically detect and remove the ambiguity, but we can remove ambiguity by re-writing the whole grammar without ambiguity.

Ambiguous Grammars:

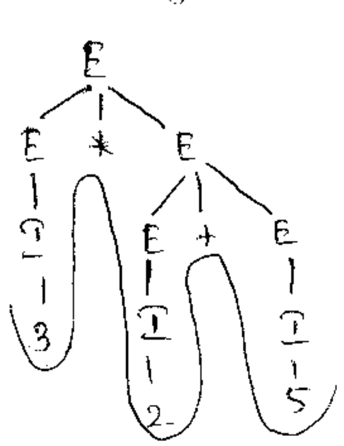
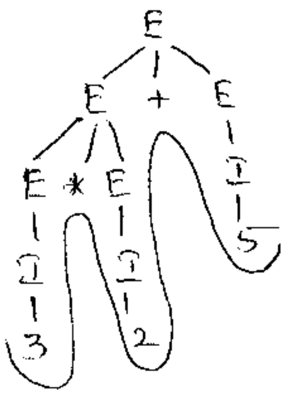
Grammar lets us generate expressions with any sequence of * and + operators and the productions E → E+E | E*E allows us to generate these expressions in any order.

Example: Let us consider a grammar with production rule

$E \to I$
$E \to E+E$
$E \to E*E$
$E \to (E)$
$I \to \epsilon|0|1|2|\dots 9.$

$G = \{V, T, P, S\}$
$V = \{I, E\}$
$T = \{+, *, (,), \epsilon, 0, 1, 2, \dots 9\}$

Solution: For the string $\boxed{\text{"}3*2+5\text{"}}$ the above grammar can generate two parse tree by leftmost derivation

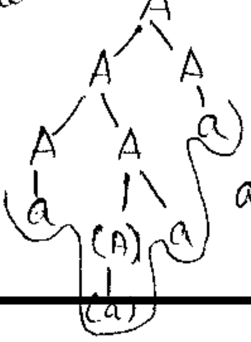



⇒ Since there are two parse tree for single string the grammar is ambiguous

Example: check whether the grammar is ambiguous or not

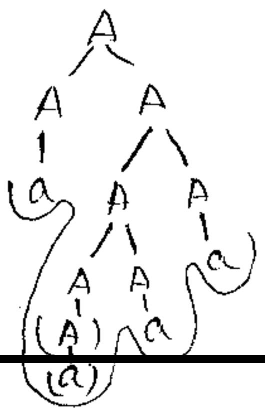
$A \to AA$
$A \to (A)$
$A \to a.$

Solution: For the string "a(a)aa", it can generate 2 parse tree



a(a)aa

$\longrightarrow$



118

③. Check whether the given grammar is ambiguous or not.

$E \rightarrow E + E$

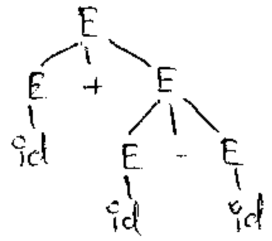
$E \rightarrow E - E$

$E \rightarrow id.$

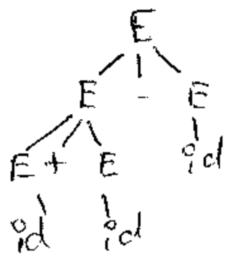
Solution:-

" id + id - id "

①. $E \rightarrow E + E$
$\rightarrow id + E$
$\rightarrow id + E - E$
$\rightarrow id + id - E$
$\rightarrow id + id - id.$



② $E \rightarrow E - E$
$\rightarrow E + E - E$
$\rightarrow id + E - E$
$\rightarrow id + id - E$
$\rightarrow id + id - id.$



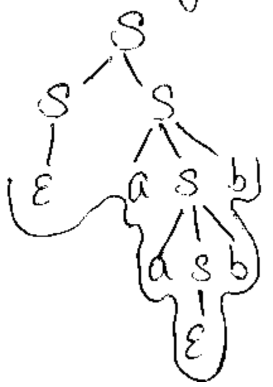
④. check whether the given grammar is ambiguous or not.

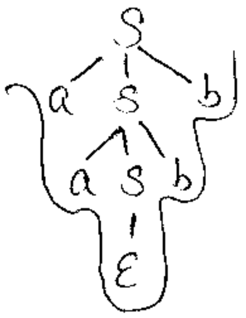
$S \rightarrow aSb \mid SS$

$S \rightarrow \varepsilon$

Solution:

String : aabb.





⇒ Two different types of Path string with the Same Input

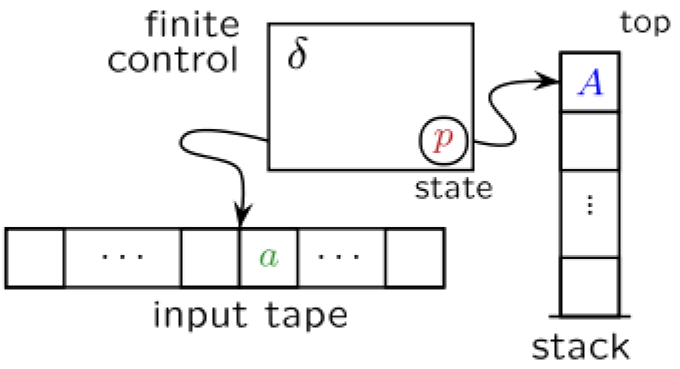
⇒ So the given grammar is an ambiguous.

Push Down Automata

pushdown automata can recognize all deterministic context-free languages while nondeterministic ones can recognize all context-free languages, with the former often used inparser design.

The term "pushdown" refers to the fact that the stack can be regarded as being "pushed down" like a tray dispenser at a cafeteria, since the operations never work on elements other than the topelement. A **stack automaton**, by contrast, does allow access to and operations on deeper elements. Stack automata can recognize a strictly larger set of languages than pushdown automata. A nested stack automaton allows full access, and also allows stacked values to be entire sub-stacks rather than just single finite symbols.

Informal description



A diagram of a pushdown automaton

A finite state machine just looks at the input signal and the current state: it has no stack to work with. It chooses a new state, the result of following the transition. A **pushdown automaton (PDA)** differs from a finite state machine in two ways:

1. It can use the top of the stack to decide which transition to take.
2. It can manipulate the stack as part of performing a transition.

A pushdown automaton reads a given input string from left to right. In each step, it chooses a transition by indexing a table by input symbol, current state, and the symbol at the top of the stack. A pushdown automaton can also manipulate the stack, as part of performing a transition. The manipulation can be to push a particular symbol to the top of the stack, or to pop off the topof the stack. The automaton can alternatively ignore the stack, and leave it as it is.

Put together: Given an input symbol, current state, and stack symbol, the automaton can follow a transition to another state, and optionally manipulate (push or pop) the stack.

If, in every situation, at most one such transition action is possible, then the automaton is called a **deterministic pushdown automaton (DPDA)**. In general, if several actions are possible, then the automaton is called a **general**, or **nondeterministic**, **PDA**. A given input string may drive a nondeterministic pushdown automaton to one of several configuration sequences; if one of themleads to an accepting configuration after reading the complete input string, the latter is said to belong to the *language accepted by the automaton.*

Definition of the Pushdown Automaton:

Formal definition

We use standard formal language notation: denotes the set of strings over alphabet and denotes the empty string.

A PDA is formally defined as a 7-tuple: where

is a finite set of *states*

- is a finite set which is called the *input alphabet*
- is a finite set which is called the *stack alphabet*
- is a finite subset of , the *transition relation*.
- is the *start state*
- is the *initial stack symbol*
- is the set of *accepting states*

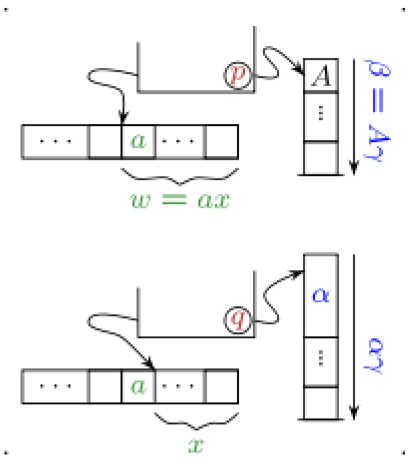
An element is a transition of . It has the intended meaning that , in state , on the input and with as topmost stack symbol, may read , change the state to , pop , replacing it by pushing . The component of the transition relation is used to formalize that the PDA can either read a letter from the input, or proceed leaving the input untouched.

In many texts the transition relation is replaced by an (equivalent) formalization, where

- is the *transition function*, mapping into finite subsets of

Here contains all possible actions in state with on the stack, while reading on the input. One writes for example precisely when because . Note that *finite* in this definition is essential.

Computations



A step of the pushdown automaton

In order to formalize the semantics of the pushdown automaton a description of the current situation is introduced. Any 3-tuple is called an instantaneous description (ID) of , which includes the current state, the part of the input tape that has not been read, and the contents of the stack (topmost symbol written first). The transition relation defines the step-relation of on instantaneous descriptions. For instruction there exists a step , for every and every .In general pushdown automata are nondeterministic meaning that in a given instantaneous description there may be several possible steps. Any of these steps can be chosen in a computation. With the above definition in each step always a single symbol (top of the stack) is popped, replacing it with as many symbols as necessary. As a consequence no step is defined when the stack is empty.Computations of the pushdown automaton are sequences of steps. The computation starts in the initial state with the initial stack symbol on the stack, and a string on the input tape, thus with initial description . There are two modes of accepting. The pushdown automaton either accepts by final state, which means after reading its input the automaton reaches an accepting state (in), or it accepts by empty stack (), which means after reading its input the automaton empties its stack. The first acceptance mode uses the internal memory (state), the second the external memory (stack).

Formally one defines

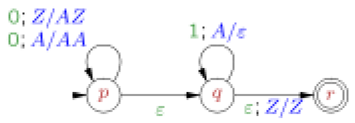
1. with and (final state)
2. with (empty stack)

Here represents the reflexive and transitive closure of the step relation meaning any number of consecutive steps (zero, one or more).For each single pushdown automaton these two languages need to have no relation: they may be equal but usually this is not the case. A specification of the automaton should also include the intended mode of acceptance. Taken over all pushdown automata both acceptance conditions define the same family of languages.

Theorem. For each pushdown automaton one may construct a pushdown automaton such that , and vice versa, for each pushdown automaton one may construct a pushdown automaton such that

Example

The following is the formal description of the PDA which recognizes the language by final state:



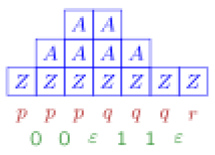
PDA for
(by final state), where

- **states:**
- **input alphabet:**
- **stack alphabet:**
- **start state:**
- **start stack symbol:** Z
- **accepting states:**

The transition relation consists of the following six instructions:,,,,, and.In words, the first two instructions say that in state *p* any time the symbol 0 is read, one *A* is pushed onto the stack.
Pushing symbol *A* on top of another *A* is formalized as replacing top *A* by *AA* (and similarly forpushing symbol *A* on top of a *Z*).The third and fourth instructions say that, at any moment the automaton may move from state *p* to state *q*.The fifth instruction says that in state *q*, for each symbol 1 read, one *A* is popped.

Finally, the sixth instruction says that the machine may move from state *q* to accepting state *r* only when the stack consists of a single *Z*.There seems to be no generally used representation forPDA. Here we have depicted the instruction by an edge from state *p* to state *q* labelled by (read *a*; replace *A* by).

Understanding the computation process



$$\begin{aligned}
&(p, 0011, Z) \vdash \\
&(p, 011, AZ) \vdash \\
&(p, 11, AAZ) \vdash \\
&(q, 11, AAZ) \vdash \\
&(q, 1, AZ) \vdash \\
&(q, \varepsilon, Z) \vdash \\
&(r, \varepsilon, Z)
\end{aligned}$$

Accepting computation for 0011

The following illustrates how the above PDA computes on different input strings. The subscript *M* from the step symbol is here omitted.

a. Input string = 0011. There are various computations, depending on the moment the move from state *p* to state *q* is made. Only one of these is accepting.
    i. The final state is accepting, but the input is not accepted this way as it has notbeen read.
    ii. No further steps possible.
    iii. Accepting computation: ends in accepting state, while complete input has beenread.
b. Input string = 00111. Again there are various computations. None of these is accepting.
    i. The final state is accepting, but the input is not accepted this way as it has notbeen read.
    ii. No further steps possible.
    iii. The final state is accepting, but the input is not accepted this way as it has notbeen (completely) read.

Description

A pushdown automaton (PDA) is a finite state machine which has an additional stack storage. The transitions a machine makes are based not only on the input and current state, but also on thestack. The formal definition (in our textbook) is that a PDA is this:

M = (K,Σ,Γ,Δ,s,F) where K = finite state set

- Σ = finite input alphabet
- Γ = finite stack alphabet
- s ∈ K: start state
- F ⊆ K: final states
- The transition relation, Δ is a **finite** subset of $(K\times(\Sigma\cup\{\varepsilon\})\times\Gamma^*) \times (K\times\Gamma^*)$

We have to have the **finite** qualifier because the full subset is infinite by virtue of the Γ* component. The meaning of the transition relation is that, for σ ∈ Σ, if ((p,σ,α),(q,β)) ∈ Δ:

- the current state is p
- the current input symbol is σ
- the string at the top of the stack is α then:

- the new state is q
- replace α on the top of the stack by β (pop the α and push the β)

Otherwise, if ((p,ε,α),(q,β)) ∈ Δ, this means that if

- the current state is p
- the string at the top of the stack is α then (not consulting the input symbol), we can

- change the state is q
- replace α on the top of the stack by β

Machine Configuration, yields, acceptance
A machine configuration is an element of $K\times\Sigma^*\times\Gamma^*$.

(p,w,γ) = current state, unprocessed input, stack content) We define the usual *yields* relationships:

(p,σw,αγ) ⊢ (q,w,βγ) if ((p,σ,α),(q,β)) ∈ Δ or (p,w,αγ) ⊢ (q,w,βγ) if ((p,ε,α),(q,β)) ∈ Δ As

expected, ⊢* is the reflexive, transitive closure of ⊢.

A string w is *accepted* by the PDA if

(s,w,ε) ⊢* (f,ε,ε)
Namely, from the start state with empty stack, we

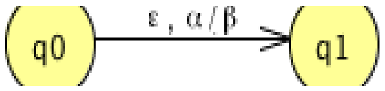
- process the entire string,
- end in a final state
- end with an empty stack.

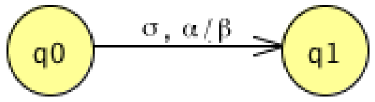
The language accepted by a PDA M, L(M), is the set of all accepted strings.
The empty stack is our key new requirement relative to finite state machines. The examples that we generate have very few states; in general, there is so much more control from using the stack memory. Acceptance by empty stack only or final state only is addressed in problems 3.3.3 and 3.3.4.

Graphical Representation and ε-transition
The book does not indicate so, but there is a graphical representation of PDAs. A transition

((p,x,α),(q,β)) where x = ε or x ∈ Σ would be depicted like this (respectively):

or





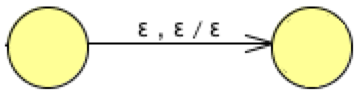
The stack usage represented byα/β
represents these actions:

- the top of the stack must **match** α
- if we make the transition, **pop** α and **push** β

A PDA is non-deterministic. There are several forms on non-deterministism in the description:

- Δ is a relation
- there are ε-transitions in terms of the input
- there are ε-transitions in terms of the stack contents

The true PDA ε-transition, in the sense of being equivalent to the NFA ε-transition is this:

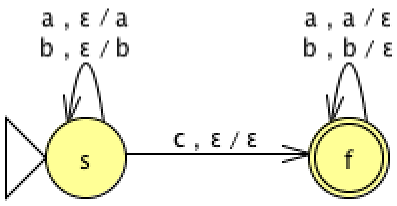


because it consults neither the input, nor the stack and will leave the previous configurationintact.

Palindrome examples

These are examples 3.3.1 and 3.3.2 in the textbook. The first is this:

$\{x \in \{a,b,c\}^* : x = wcw^R \text{ for } w \in \{a,b\}^*\}$

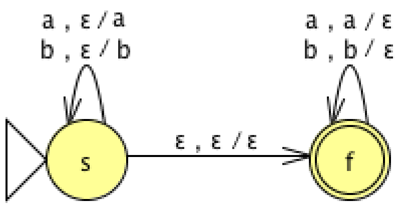


The machine pushes a's and b's in state s, makes a transition to f when it sees the middle marker, c, and then matches input symbols with those on the stack and pops the stack symbol. Non- accepting string examples are these:

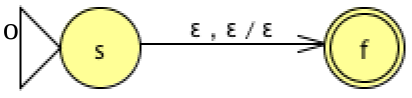
ε in state s
ab in state s with non-empty stack
abcab in state f with unconsumed input and non-empty stackabcbin
state f with non-empty stack
abcbab in state f with unconsumed input and empty stack
Observe that this PDA is deterministic in the sense that there are no choices in transitions. The

second example is:

$\{x \in \{a,b\}^* : x = ww^R \text{ for } w \in \{a,b\}^*\}$



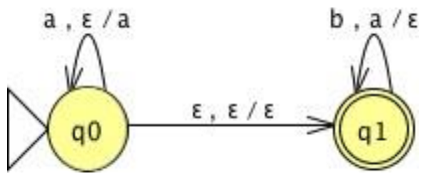
This PDA is identical to the previous o
except for the ε-transition



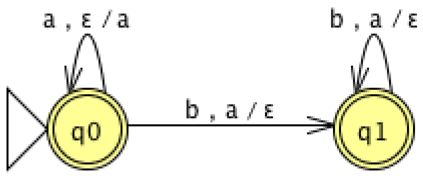
Nevertheless, there is a significant difference in that this PDA must **guess** when to stop pushing symbols, jump to the final state and start matching off of the stack. Therefore this machine is decidedly non-deterministic. In a general programming model (like Turing Machines), we have the luxury of preprocessing the string to determine its length and thereby knowing when themiddle is coming.

The $a^n b^n$ language

The language is L = { w ∈ {a,b}* : w = $a^n b^n$, n ≥ 0 }. Here are two PDAs for L:



and



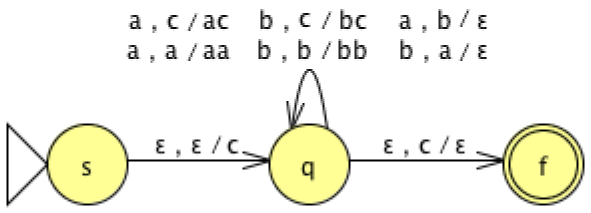
The idea in both of these machines is to stack the a's and match off the b's. The first one is non-deterministic in the sense that it could prematurely guess that the a's are done and start matching off b's. The second version is deterministic in that the first b acts as a trigger to start matching off. Note that we have to make both states final in the second version in order to accept ε.

The equal a's and b's language

This is example 3.3.3 in the textbook. Let us use the convenience notation:

#σ(w) = the number of occurrences of σ in w

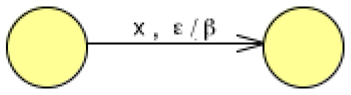
The language is L = {w ∈ {a,b}*: #a(w) = #b(w) }. Here is the PDA:



As you can see, most of the activity surrounds the behavior in state q. The idea is have the stack maintain the **excess** of one symbol over the other. In order to achieve our goal, we must know when the stack is empty.

Empty Stack Knowledge

There is no mechanism built into a PDA to determine whether the stack is empty or not. It's important to realize that the transition:



x = σ ∈ Σ or ε

means to do so **without consulting** the stack; it says nothing about whether the stack is empty ornot.

Nevertheless, one can maintain knowledge of an empty stack by using a dedicated stack symbol, c, representing the "stack bottom" with these properties:

- it is pushed onto an empty stack by a transition from the start state with no other outgoing or incoming transitions
- it is never removed except by a transition to state with no other outgoing transitions

Behavior of PDA
The three groups of loop transitions in state q represent these respective functions:

- input a with no b's on the stack: push a
- input b with no a's on the stack: push b
- input a with b's on the stack: pop b; or, input b with a's on the stack: pop a

For example if we have seen 5 b's and 3 a's in any order, then the stack should be "bbc". The transition to the final state represents the only non-determinism in the PDA in that it must guesswhen the input is empty in order to pop off the stack bottom.

DPDA/DCFL

The textbook defines DPDAs (Deterministic PDAs) and DCFLs (Deterministic CFLs) in the introductory part of section 3.7. According to the textbook's definition, a DPDA is a PDA inwhich no state p has two different outgoing transitions

$$((p,x,\alpha),(q,\beta)) \text{ and } ((p,x',\alpha'),(q',\beta'))$$

which are *compatible* in the sense that both could be applied. A DCFL is basically a languagewhich accepted by a DPDA, but we need to qualify this further.

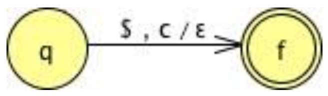
We want to argue that the language L = { w ∈ {a,b}* : #a(w) = #b(w)} is deterministic contextfree in the sense there is DPDA which accepts it.

In the above PDA, the only non-determinism is the issue of guessing the end of input; however this form of non-determinism is considered artificial. When one considers whether a language Lsupports a DPDA or not, a dedicated *end-of-input* symbol is always added to strings in the language.

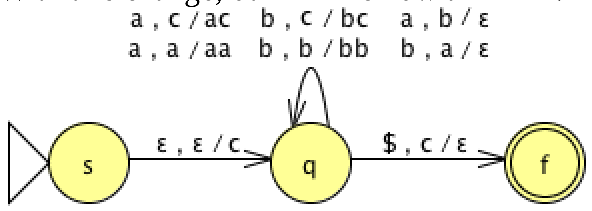
Formally, a language L over Σ is *deterministic context free*, or L is a DCFL , ifL**$** is

accepted by a DPDA M

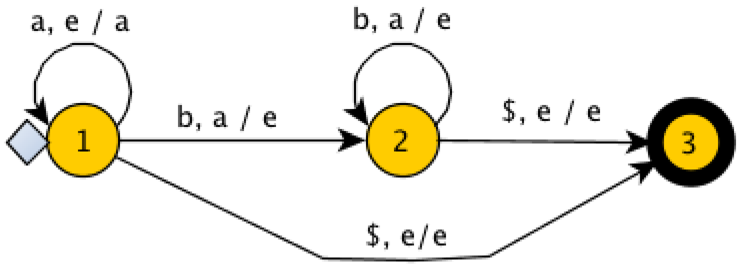
where **$** is a dedicated symbol not belonging to Σ. The significance is that we can make intelligent usage of the knowledge of the end of input to decide what to do about the stack. In ourcase, we would simply replace the transition into the final state by:



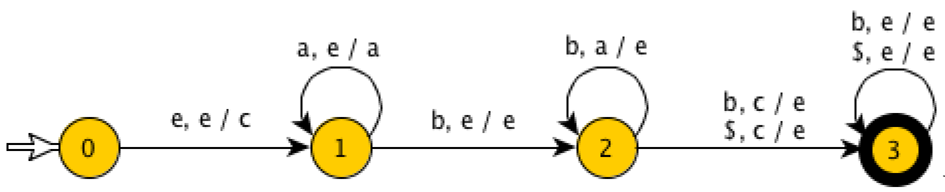
With this change, our PDA is now a DPDA:


a , c / ac   b , c / bc   a , b / ε
a , a / aa   b , b / bb   b , a / ε
s
ε , ε / c
q
$, c / ε
f

a*b* examples
Two common variations on a's followed by b's. When they're equal, no stack bottom is necessary. When they're unequal, you have to be prepared to recognize that the stacked a's havebeen completely matched or not.



a. { $a^n b^n : n \geq 0$ }



b. { $a^m b^n : 0 \leq m < n$ }

Let's look at a few sample runs of (b). The idea is that you cannot enter the final state with an "a" still on the stack. Once you get to the final state, you can consume remaining b's and end marker.

We can start from state 1 with the stack bottom pushed on:

success: abb

state	input	stack
1	abb$	c
1	bb$	ac
2	b$	ac
2	$	c
3	ε	ε

success: abbbb

state	input	stack
1	abbbb$	c
1	bbbb$	ac
2	bbb$	ac
2	bb$	c
3	b$	ε
3	$	ε

3	ε	ε	(fail:

ab)

state	input	stack
1	ab$	c
1	b$	ac
2	$	ac

(fail: ba)

state	input	stack
1	ba$	c
2	a$	c

Observe that a string like abbbaalso fails due to the inability to consume the very last a.

Equivalence of PDA's and CFG's:

If a grammar **G** is context-free, we can build an equivalent nondeterministic PDA which accepts the language that is produced by the context-free grammar **G**. A parser can be built for the grammar **G**.

Also, if **P** is a pushdown automaton, an equivalent context-free grammar G can be constructedwhere

L(G) = L(P)

In the next two topics, we will discuss how to convert from PDA to CFG and vice versa.

Algorithm to find PDA corresponding to a given CFG

Input − A CFG, G = (V, T, P, S)

Output − Equivalent PDA, P = (Q, ∑, S, δ, q0, I, F) **Step 1** −

Convert the productions of the CFG into GNF. **Step 2** − The

PDA will have only one state {q}.

Step 3 − The start symbol of CFG will be the start symbol in the PDA.

Step 4 − All non-terminals of the CFG will be the stack symbols of the PDA and all the terminals of the CFG will be the input symbols of the PDA.

Step 5 − For each production in the form **A → aX** where a is terminal and **A, X** are combination of terminal and non-terminals, make a transition **δ (q, a, A)**.

Problem

Construct a PDA from the following CFG.

G = ({S, X}, {a, b}, P, S)

where the productions are −

S → XS | ε , A → aXb | Ab | ab

Solution

Let the equivalent PDA,

P = ({q}, {a, b}, {a, b, X, S}, δ, q, S)

where δ −

δ(q, ε , S) = {(q, XS), (q, ε)}

δ(q, ε , X) = {(q, aXb), (q, Xb), (q, ab)} δ(q, a,

a) = {(q, ε)}

$\delta(q, 1, 1) = \{(q, \varepsilon)\}$

Algorithm to find CFG corresponding to a given PDA

Input − A CFG, G = (V, T, P, S)

Output − Equivalent PDA, P = (Q, ∑, S, δ, q0, I, F) such that the non- terminals of the grammarG will be $\{X_{wx} \mid w,x \in Q\}$ and the start state will be $A_{q0,F}$.

Step 1 − For every w, x, y, z ∈ Q, m ∈ S and a, b ∈ ∑, if δ (w, a, ε) contains (y, m) and (z, b, m)contains (x, ε), add the production rule $X_{wx} \rightarrow a X_{yz} b$ in grammar G.

Step 2 − For every w, x, y, z ∈ Q, add the production rule $X_{wx} \rightarrow X_{wy}X_{yx}$ in grammar G.

Step 3 − For w ∈ Q, add the production rule $X_{ww} \rightarrow \varepsilon$ in grammar G.

Parsing is used to derive a string using the production rules of a grammar. It is used to check the acceptability of a string. Compiler is used to check whether or not a string is syntactically correct. A parser takes the inputs and builds a parse tree.

A parser can be of two types −

- **Top-Down Parser** − Top-down parsing starts from the top with the start-symbol and derives a string using a parse tree.
- **Bottom-Up Parser** − Bottom-up parsing starts from the bottom with the string andcomes to the start symbol using a parse tree.

Design of Top-Down Parser

For top-down parsing, a PDA has the following four types of transitions −

- Pop the non-terminal on the left hand side of the production at the top of the stack andpush its right-hand side string.
- If the top symbol of the stack matches with the input symbol being read, pop it.
- Push the start symbol 'S' into the stack.
- If the input string is fully read and the stack is empty, go to the final state 'F'.

Example

Design a top-down parser for the expression "x+y*z" for the grammar G with the following production rules −

P: S → S+X | X, X → X*Y | Y, Y → (S) | id

Solution

If the PDA is (Q, ∑, S, δ, q0, I, F), then the top-down parsing is −(x+y*z,

I) ⊢(x +y*z, SI) ⊢ (x+y*z, S+XI) ⊢(x+y*z, X+XI)

⊢(x+y*z, Y+X I) ⊢(x+y*z, x+XI) ⊢(+y*z, +XI) ⊢ (y*z, XI)

⊢(y*z, X*YI) ⊢(y*z, y*YI) ⊢(*z,*YI) ⊢(z, YI) ⊢(z, zI) ⊢(ε, I)

Design of a Bottom-Up Parser

For bottom-up parsing, a PDA has the following four types of transitions −

- Push the current input symbol into the stack.
- Replace the right-hand side of a production at the top of the stack with its left-hand side.
- If the top of the stack element matches with the current input symbol, pop it.
- If the input string is fully read and only if the start symbol 'S' remains in the stack, pop itand go to the final state 'F'.

Example

Design a top-down parser for the expression "x+y*z" for the grammar G with the following production rules −

P: S → S+X | X, X → X*Y | Y, Y → (S) | id

Solution

If the PDA is (Q, ∑, S, δ, q0, I, F), then the bottom-up parsing is −(x+y*z,

I) ⊢ (+y*z, xI) ⊢ (+y*z, YI) ⊢ (+y*z, XI) ⊢ (+y*z, SI)

⊢(y*z, +SI) ⊢ (*z, y+SI) ⊢ (*z, Y+SI) ⊢ (*z, X+SI) ⊢ (z, *X+SI)

⊢ (ε, z*X+SI) ⊢ (ε, Y*X+SI) ⊢ (ε, X+SI) ⊢ (ε, SI)

Deterministic Pushdown Automata:

In automata theory, a **deterministic pushdown automaton** (**DPDA** or **DPA**) is a variation of the pushdown automaton. The class of deterministic pushdown automata accepts the deterministic context-free languages, a proper subset of context-free languages.

Machine transitions are based on the current state and input symbol, and also the current topmost symbol of the stack. Symbols lower in the stack are not visible and have no immediate effect. Machine actions include pushing, popping, or replacing the stack top. A deterministic pushdown automaton has at most one legal transition for the same combination of input symbol, state, and top stack symbol. This is where it differs from the nondeterministic pushdown automaton.

Formal definition

A (not necessarily deterministic) **PDA** can be defined as a 7-tuple:

Where is a finite set of states

- is a finite set of input symbols
- is a finite set of stack symbols
- is the start state
- is the starting stack symbol
- , where is the set of accepting states
- is a transition function, wherewhere is the Kleene star, meaning that is "the set of all finite strings (including the empty string) of elements of ", denotes the empty string, and is the power set of a set .*M* is *deterministic* if it satisfies both the following conditions:

- For any , the set has at most one element.

- For any , if , then for every

There are two possible acceptance criteria: acceptance by *empty stack* and acceptance by *final state*. The two are not equivalent for the deterministic pushdown automaton (although they are forthe non-deterministic pushdown automaton). The languages accepted by *empty stack* are those languages that are accepted by *final state* and are prefix-free: no word in the language is the prefixof another word in the language.

The usual acceptance criterion is *final state*, and it is this acceptance criterion which is used to define the deterministic context-free languages.

Languages recognized

If is a language accepted by a PDA , it can also be accepted by a DPDA if and only if there is a single computation from the initial configuration until an accepting one for all strings belonging to . If can be accepted by a PDA it is a context free language and if it can be accepted by a DPDA it is a deterministic context-free language.

Not all context-free languages are deterministic. This makes the DPDA a strictly weaker device than the PDA. For example, the language of even-length palindromes on the alphabet of 0 and 1 has the context-free grammar $S \rightarrow 0S0 \mid 1S1 \mid \varepsilon$. An arbitrary string of this language cannot be parsed without reading all its letters first which means that a pushdown automaton has to try alternative state transitions to accommodate for the different possible lengths of a semi-parsed string.

Restricting the DPDA to a single state reduces the class of languages accepted to the LL(1) languages. In the case of a PDA, this restriction has no effect on the class of languages accepted.