Context Free Languages (CFL)

Language Recognizer

A device that accepts valid strings. The FA are formalized types of language recognizer.

Language Generator:

Context free grammars are language generators, which are based on a more complete understanding of the structure of the strings belonging to the language.

Regular expression can be also viewed as language generator. E. g. a(a*∪ b*) b . In verbal form

- First output an *a* then
- Either output a number of *a's* or output number of *b's*
- Finally output a *b*.

Using context free grammar same language can be generated . For a(a*∪ b*) b

Let S → be a new symbol interpreted as "a string in the language"

M be a symbol standing for middle part.

Then ,we can express

S → aMb

Here, M can be either string of a's or string of b's .

So, M → A and M →B

strings of *a's* can be empty string so A→ e or it may consist of a leading a followed by a string of *a's*

A→ aA

Similarly

B→ e

B→ bB.

For Example : To generate a string aaab.

S → aMb
→aAb (M→A)
→aaAb (A→aA)
→ aaaAb (A→aA)
→aaab (A→e)

The Context free grammar is a language generator that operates like the one above ,with some such a set of rules.

Why it is called Context free?

Consider S → aaAb ,

↑ it is surrounded by *aa* and *b* , its called context of A, Now the rule A →aA says that we can replace A by the string aA no matter what the surrounding strings are i. e. independent of the context of A.

The following is a example of a context-free grammar, called G , which describes a fragment of the English language.

(SENTENCE) → (NOUN-PHRASE)(VERB-PHRASE)

(NOUN-PHRASE) → (CMPLX-NOUN) / (CMPLX-NOUN)(PREP-PHRASE)

(VERB-PHRASE) →(CMPLX-VERB) /(CMPLX-VERB) (PREP-PHRASE)

(PREP-PHRASE) →(PREP) (CMPLX-NOUN)

(CMPLX-NOUN) → (ARTICLE) (NOUN)

(CMPLX-VERB) →(VERB) / (VERB)(NOUN-PHRASE)

(ARTICLE) → a/ the

(NOUN) → boy/girl/flower

(VERB) → touches/likes/sees

(PREP) → with

Grammar G has 10 variables (the capitalized grammatical terms written inside brackets); 27 terminals (the standard English alphabet plus a space character); and 18 rules. Strings in L(G) include

a boy sees
the boy sees a flower
a girl with a flower likes the boy

Each of these strings has a derivation in grammar G. The following is a derivation of the first string on this list.

(SENTENCE) ⇒(NOUN-PHRASE) (VERB-PHRASE)
⇒ (CMPLX-NOUN)(VERB-PHRASE)
⇒ (ARTICLE) (NOUN) (VERB-PHRASE)
⇒ a (NOUN)(VERB-PHRASE)
⇒ a boy (VERB-PHRASE)
⇒ a boy (CMPLX-VERB)
⇒ a boy (VERB)
⇒ a boy sees

Formal Definition

A Context free grammar G is quadruple (V, ∑, R, S) , where

- V is an alphabet (finite set of variables)
- ∑ the set of terminals (is a subset of V)
- R the set of rules (is finite subset of V-∑ x V*)
- S the start symbol (is an element of V- ∑)

Here the member of V - ∑ is called non terminals

Derivation of string from CFG

A derivation of a string w in a given grammar G is a sequence of substitutions starting with the start symbol and resulting in w.

If *u*, *v*, and w are strings of variables and terminals, and A → w is a rule of the grammar, we say that *u*A*v* yields *u*w*v*, written *u*A*v* ⇒ *u*w*v*. Say that u derives v, written *u* ⇒* *v*, if *u* ⇒*v* or if a sequence $u_1, u_2, . ., u_k$ exists for k ≥ 0 and

u ⇒ u_1⇒ u_2 ⇒ .. ……. ⇒ u_k⇒ v.

e.g.

A→0A1

A → B

B→#

A ⇒0A1⇒00A11 ⇒ 000A111 ⇒ 000B111 ⇒ 000#111 is a derivation of the string 000#111 in the grammar defined above.

The symbol "⇒" reads "yields". We say *u* derives *v* and write *u* ⇒* *v* if there is a derivation (for some k ≥ 0): *u* ⇒ u_1 ⇒u_2 ⇒……⇒ u_k ⇒ *v*

When the grammar to which we refer is obvious, we write A → w and u ⇒ v instead of A →G w and *u* ⇒G *v*.

We call any sequence of the form

W_0 ⇒G W_I ⇒G ... ⇒GW_n is called a derivation in G of W_n from W_0. Here W_o,⋯, W_n may be any strings in V*, and *n*, the length of the derivation, may be any natural number, including zero. We also say that the derivation has n steps.

Examples

Design a context-free grammar (CFG) for the language $\{a^n b^n : n \geq O\}$.

Solution

Let required CFG G = (V,∑, R, S) where

V = {S, a, b},

∑ = {a, b}, and R consists of the rules

S → aSb (rule 1) and

S → e. (rule 2)

Derivation example

A possible derivation is

S => aSb (apply rule 1)

=> aaSbb (apply rule 1)
=> aabb (apply rule 2)

Example 2

Design a CFG for the language consists of all strings over the alphabet {(,), +, *, id} that represent syntactically correct arithmetic expressions involving + and *. id stands for any identifier, that is to say, variable name .Examples of such strings are id and id * (id * id + id), but not *id + (or + * id.

Solution

Let G = (V,∑, R, E) where V, ∑, and R are as follows.

V = {+,*, (,) , id, T,F , E},

∑= {+, *, (,), id},

R = {

$E \rightarrow E + T$,	(R1)
$E \rightarrow T$,	(R2)
$T \rightarrow T * F$,	(R3)
$T \rightarrow F$,	(R4)
$F \rightarrow (E)$,	(R5)
$F \rightarrow id$ }.	(R6)

The symbols E, T, and F are abbreviations for expression, term, and factor, respectively.

The grammar G generates the string (id * id + id) * (id + id) by the following derivation.

$E \Rightarrow T$	by R2
$\Rightarrow T*F$	by R3
$\Rightarrow T* (E)$	by R5
$\Rightarrow T* (E +T)$	by R1
$\Rightarrow T * (T + T)$	by R2
$\Rightarrow T* (F + T)$	by R4
$\Rightarrow T* (id + T)$	by R6
$\Rightarrow T* (id + F)$	by R4
$\Rightarrow T * (id + id)$	by R6
$\Rightarrow F * (id + id)$	by R4
$\Rightarrow (E) * (id + id)$	by R5
$\Rightarrow (E + T) * (id + id)$	by R1
$\Rightarrow (E+F)*(id+ id)$	by R4
$\Rightarrow (E + id) * (id + id)$	by R6
$\Rightarrow (T + id) * (id + id)$	by R2
$\Rightarrow (T * F + id) * (id + id)$	by R3
$\Rightarrow (F * F + id) * (id + id)$	by R4
$\Rightarrow (F * id + id) * (id + id)$	by R6
$\Rightarrow (id * id + id) * (id + id)$	by R6

Example 3
Design a CFG that generates all strings of properly balanced left and right parentheses: every left parenthesis can be paired with a unique subsequent right parenthesis, and every right parenthesis can be paired with a unique preceding left parenthesis.

Solution
Let required grammar G = (V, ∑, R, S), where
V = { S, (,) },
∑= { (,) },
R = {
S→e,
S → SS,
S → (S)
}.

To generate the string *()(())*
one possible derivation is
S ⇒ SS
⇒S(S)
⇒S((S))
⇒ S(())
⇒ ()(())

or we can generate by following way.

S ⇒ SS
⇒ (S)S
⇒ ()S
⇒ ()(S)
⇒()(())

Thus the same string may have several derivations in a context-free grammar;

Leftmost derivation
In leftmost derivation**,** when a step of a derivation can be a part of a leftmost derivation: the leftmost non-terminal must be replaced. Derivation of a string w in a grammar G is a leftmost derivation if at every step the leftmost remaining variable is the one replaced.

We write $x \overset{L}{\Rightarrow} y$ if and only if x = wAβ, y = wαβ, where w ϵ ∑ *, 0:,,8 E V', A ϵ V*-I;, and A → 0:
is a rule of G. Thus, if Xl * X2 * ... * Xn is a leftmost derivation, then in
fact Xl i X2 i ... i Xn-

Rightmost derivation

Given a grammar G with production rules

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

Obtain the (i) leftmost derivation, and (ii) rightmost derivation for the string "*aaabbabbba*".

Solution

(i) *Leftmost derivation:*

$$S \Rightarrow aB \Rightarrow aaBB \Rightarrow aaaBBB \Rightarrow aaabBB \Rightarrow aaabbB$$
$$\Rightarrow aaabbabB \Rightarrow aaabbabbB \Rightarrow aaabbabbbS \Rightarrow aaabbabbba$$
$$\Rightarrow aaabbabb$$

(ii) *Rightmost derivation:*

$$S \Rightarrow aB \Rightarrow aaBB \Rightarrow aaBbS \Rightarrow aaBbbA \Rightarrow aaaBBbba$$
$$\Rightarrow aaabBbba \Rightarrow aaabbSbba \Rightarrow aaabbaBbba \Rightarrow aaabbabbba$$

Theorem

Let G = (V,∑, R, S) be a context-free grammar, and let A ϵ V - ∑ , and w ϵ ∑*. Then the following statements are equivalent:

(a) A ⇒* w.

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

(c) There is a leftmost derivation A $\overset{L}{\Rightarrow}^{*}$ w.

(d) There is a rightmost derivation A $\overset{R}{\Rightarrow}^{*}$ w.

Application of CFG

- An important application of context-free grammars occurs in the specification and compilation of programming languages.
- Most compilers and interpreters contain a component called a parser that extracts the meaning of a program prior to generating the compiled code or performing the interpreted execution.

Parse/ derivation Trees

Parsing

A compiler translates code written in a programming language into another form, usually one more suitable for execution. To do so the compiler extracts the meaning of the code to be compiled in a process called **parsing**. "Parsing" a string is finding a derivation (or a derivation tree) for that string. Parsing a string is like recognizing a string. The only realistic way to recognize a string of a context-free grammar is to parse it.

A parse tree is a convenient way to represent a derivation of a particular string.

A 'derivation tree' is an ordered tree which the nodes are labeled with the left sides of productions and in which the children of a node represent its corresponding right sides.

Formal Definition

Let G = (V, ∑, R, S) be a CFG. An ordered tree is a derivation tree for G if it has the following properties:

- The root of the derivation tree is S.
- Each and every leaf in the tree has a label from ∑ ∪ {e}.
- Each and every interior vertex (a vertex which is no a leaf) has a label from V.
- If a vertex has label A ϵ V, and its children are labeled (from left to right) a_1 , a_2 , …. a_n, then R must contain a production of the form
  $A \rightarrow a_1, a_2, \ldots. a_n$
- A leaf labeled l has no siblings, that is, a vertex with a child labeled e can have no other children.

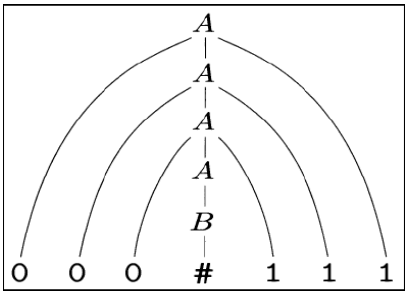
For example, in the grammar

$A \rightarrow 0A1$

$A \rightarrow B$

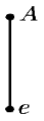
$B \rightarrow \#$

Derivation of the string 000#111 is given by the tree:



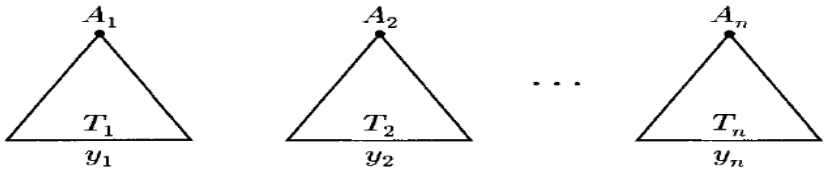
Formally, for an arbitrary context-free grammar G = (V, ∑, R, S), we define its parse trees and their roots, leaves, and yields, as follows.

1. This is a parse tree for each a ϵ ∑. The single node of this parse tree is both the root and a leaf. The yield of this parse tree is a.
2. If A→e is a rule in R, then

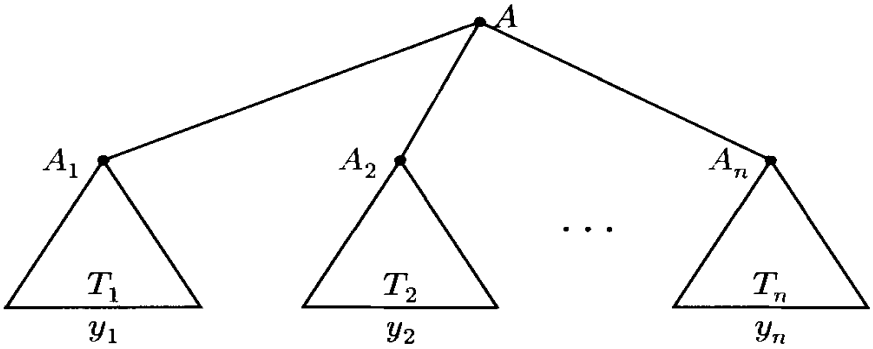


is a parse tree; its root is the node labeled A, its sole leaf is the node labeled e, and its yield is e.

3. If



Are parse trees, where $n \geq 1$, with roots labeled $A_1, \ldots, A_n$ respectively, and with yields $Y_1, \ldots, Y_n$, and $A \rightarrow A_1 \ldots A_n$ is a rule in R, then



is a parse tree. Its root is the new node labeled A, its leaves are the leaves of its constituent parse trees, and its yield is $Y_1 \ldots Y_n$.

4. Nothing else is a parse tree.

Example

For example, if G is the context-free grammar that generates the language of balanced parentheses , then the string ()() can be derived from S by at least two distinct derivations, namely,

$S \Rightarrow S\ S \Rightarrow (S)S \Rightarrow ()S \Rightarrow ()\ (S) \Rightarrow ()()$

and

$S \Rightarrow SS \Rightarrow S(S) \Rightarrow (S)(S) \Rightarrow (S() \Rightarrow ()\ ()$

However, these two derivations are in a sense "the same." The rules used are the same, and they are applied at the same places in the intermediate string. The only difference is in the order in which the rules are applied.

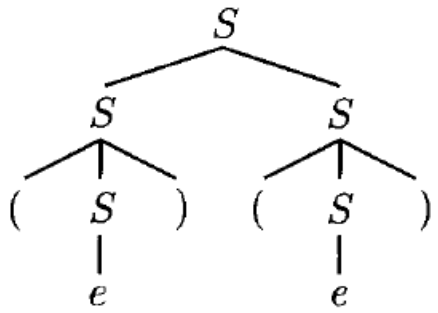
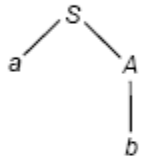


Fig: Parse tree for ()()

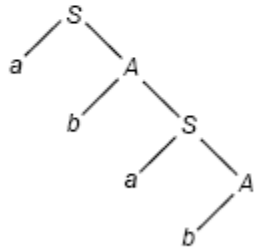
Example 2.2.9: Given a CFG with $P = \begin{Bmatrix} 1.\ S \to aA \\ 2.\ A \to bS \\ 3.\ A \to b \end{Bmatrix}$. Obtain the derivation tree and $L(G)$.

Solution

$S \Rightarrow aA \Rightarrow ab$



$S \Rightarrow aA \Rightarrow abS \Rightarrow abaA \Rightarrow abab \ldots \ldots$



The derivation trees suggest *ab*, *abab*, ...
Therefore the language generated

$$L(G) = \{(ab)^n \mid n \geq 1\}$$

Ambiguity in CFG:

Grammars such as G, with strings that have two or more distinct parse trees, are called ambiguous.

If a grammar generates the same string in several different ways, we say that the string is derived ambiguously in that grammar. If a grammar generates some string ambiguously we say that the grammar is ambiguous.

Formal Definition

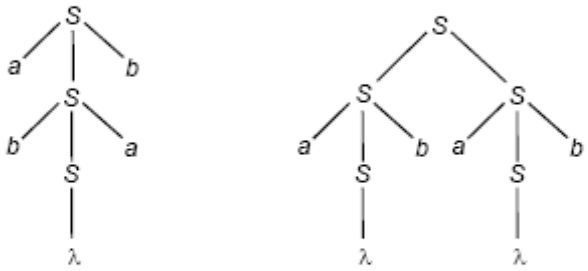
A string *w* is derived ambiguously in context-free grammar G if it has two or more different leftmost derivations. Grammar G is ambiguous if it generates some string ambiguously.

The grammar given by

$$G = (\{S\}, \{a, b\}, S, S \to aSb \mid bSa \mid SS \mid \lambda)$$

generates strings having an equal number of a's and b's.

The string "$abab$" can be generated from this grammar in two distinct ways, as shown in the following derivation trees:



Similarly, "$abab$" has two distinct leftmost derivations:

$$S \Rightarrow aSb \Rightarrow abSab \Rightarrow abab$$
$$S \Rightarrow SS \Rightarrow aSbS \Rightarrow abS \Rightarrow abaSb \Rightarrow abab.$$

Also, "$abab$" has two distinct rightmost derivations:

$$S \Rightarrow aSb \Rightarrow abSab \Rightarrow abab$$
$$S \Rightarrow SS \Rightarrow SaSb \Rightarrow Sab \Rightarrow aSbab \Rightarrow abab$$

Example 2.3.1: Prove that the grammar

$$S \to aB \mid ab,$$
$$A \to aAB \mid a,$$
$$B \to ABb \mid b$$

is ambiguous.

Solution

It is easy to see that "ab" has two different derivations as shown below.

Given the grammar G with production

1. $S \to aB$
2. $S \to ab$
3. $A \to aAB$
4. $A \to a$
5. $B \to ABb$
6. $B \to b$

Using (2), $S \Rightarrow ab$

Using (1), $S \Rightarrow aB \Rightarrow ab$ and then (6).

Example 2.3.3: Show that the grammar G with production

$$S \to a \mid aAb \mid abSb$$
$$A \to aAAb \mid bS$$

is ambiguous.

Solution

$$\begin{aligned} S &\Rightarrow abSb \quad (\because S \to abSb) \\ &\Rightarrow abab \quad (\because S \to a) \end{aligned}$$

Similarly,

$$\begin{aligned} S &\Rightarrow aAb \quad (\because S \to aAb) \\ &\Rightarrow abSb \quad (\because A \to bS) \\ &\Rightarrow abab \end{aligned}$$

Since '*abab*' has two different derivations, the grammar G is ambiguous.

Inherently ambiguous.

Sometimes when we have an ambiguous grammar we can find an unambiguous grammar that generates the same language. Some context-free languages, however, can be generated only by ambiguous grammars. Such languages are called inherently ambiguous.

Ambiguity is a property of a grammar, and it is usually, but not always possible to find an equivalent unambiguous grammar.
An "inherently ambiguous language" is a language for which no unambiguous grammar exists.

Simplification of CFG

Normal Forms

Two kinds of normal forms viz.,

- Chomsky Normal Form (CNF) and
- Greibach Normal Form (GNF).

Chomsky Nor mal Form (CNF)

Any context-free language L without any e -production is generated by a grammar is which productions are of the form A $\to$ BC or A$\to$ a, where
A, B ϵ V (, and a $\epsilon \sum$.
NT $\to$ NT x NT or
NT$\to$ T

NT : non terminal
T : Terminal

Procedure to find Equivalent Grammar in CNF

(i) Eliminate the unit productions, and l-productions if any,
(ii) Eliminate the terminals on the right hand side of length two or more.
(iii) Restrict the number of variables on the right hand side of productions to two.

Example

Let us consider the grammar G= (V,Σ , R, S) where

V= {S, A, B}

Σ = {a, b},and the productions R are :

$S \rightarrow$ bA / aB

$A \rightarrow$ bAA / aS / a

$B \rightarrow$ aBB / bS / b

find an equivalent grammar in CNF

First, the only productions already in proper form are $A \rightarrow a$ and $B \rightarrow b$.

There are no unit productions, so we may begin by replacing terminals on the right by variables, except in the case of the productions $A \rightarrow a$ and $B \rightarrow b$.

$S \rightarrow bA$ is replaced by $S \rightarrow C_b A$ and $C_b \rightarrow b$.

Similarly, $A \rightarrow aS$ is replaced by $A \rightarrow C_a S$ and $C a \rightarrow a$ and $A \rightarrow bAA$ is replaced by $A \rightarrow C_b AA$;

$S \rightarrow aB$ is replaced by $S \rightarrow C_a B$;

$B \rightarrow bS$ is replaced by $B \rightarrow C_b S$, and $B \rightarrow aBB$ is replaced by $B \rightarrow C_a BB$.

In the next stage, the production $A \rightarrow C_b AA$ is replaced by $A \rightarrow C_b D1$ and

$D1 \rightarrow A A$, and the production $B \rightarrow CaBB$ is replaced by $B \rightarrow C_a D2$ and $D2 \rightarrow BB$

The productions for the grammar in CNF are shown below.

$S \rightarrow C_b A / C_a B$

$A \rightarrow C_a S / C_b D / a$

$B \rightarrow C_b S / C_a D2 / b$

$D1 \rightarrow AA$

$D2 \rightarrow BB$

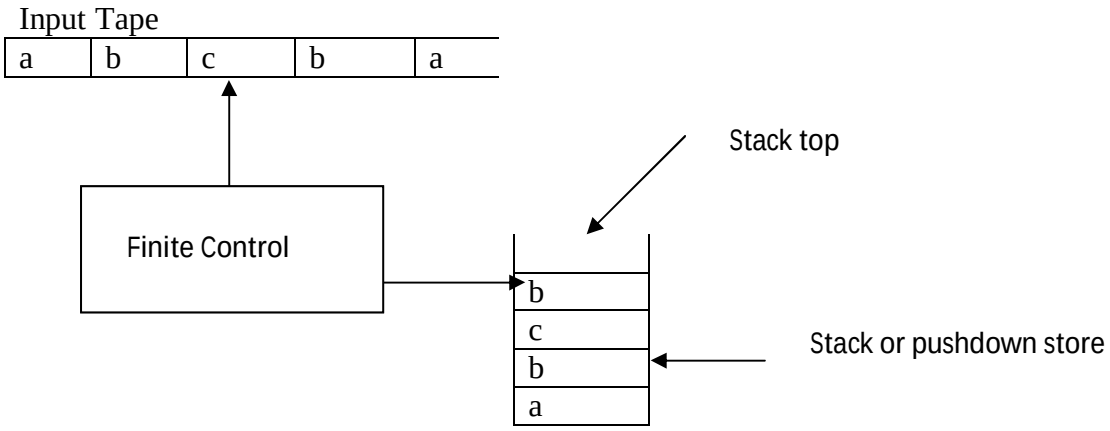
$C_a \rightarrow a$

$C_b \rightarrow b$

Pushdown Automata(PDA)

Consider the language L1 L1={ WCW^R : W ∈ $\sum^*$ or L2= { a^nb^n : n≥ 1 }, these languages are generated by CFG but can't be accepted by FA. So all CFG's are not accepted by FA. here we need to remember the strings before it goes to next part of the string and compare it with first half but FA can't do this because in FA don't have capability of remember anything. PDA is essentially a FA with control of both an input tape and a stack to store what it has read. A stack is a FIFO. PDA can accept all CFGs. PDA have three things

- input tape
- Finite control
- Stack



Definition of PDA:

A pushdown automata is a system which is mathematically defined by sixtuple

M= (K, ∑, Γ ,Δ , S , F)

where ,

K: Finite set of states

∑: is an alphabets (input symbols)

Γ : stack symbols

S ∈ K ,initial state

F⊆ K :set of final states

Δ : transition relation is a finite subset of Kx(∑ ∪ {e} x Γ *) x (K x Γ *)

i.e

Kx (∑ ∪ {e} x Γ *) → (Kx Γ *)

Transition :

if (p, a, β) , (q , γ) ∈ Δ , then M whenever it is in state p with ' β ' at the top of stack may read ' γ' from the input tape ,replace ' β' by ' γ' on the top of the stack and enter state q. such pair (p , a, β) , (q, γ) is called a transition of M.

Push/pop

Push : symbol input is to add to the top of stack . e.g (p, u, e) , (q, a) pushes a to the top of stack

Pop: symbol is to remove it from the top of the stack . e.g. (P, u, e) , (q , e) pops a.

Configuration of PDA

Configuration of a pushdown automaton is defined to be a member of K X ∑* x Γ * . The first component is the state of the machine (K), the second is the portion of the input yet to be read (∑*) , and the third is the contents of the pushdown store, read top-down (Γ *).

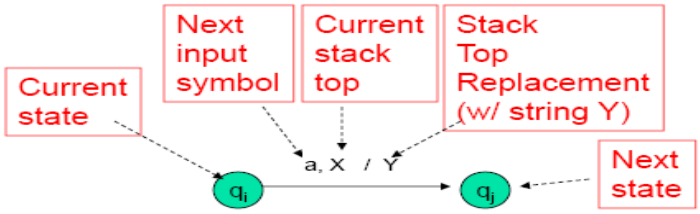
For example, if the configuration were (q, W, abc), the ' a ' would be on the top of the stack and the 'c' on the bottom. If (p, x, α) and (q, y, β) are configurations of M, we say that (p, x, α) yields in one step (q, y, β) . Thus a configuration of the machine is a triple (q, string, stack).

- When pushing symbol x, the configuration changes from (q, a.w, ε.t) to (qi, w, x.t).
- When popping symbol x, the configuration changes from(q, a.w, x.t) to (qi,w, t).
- When we don't wish the stack to change at all in a computation step, the machine moves from a configuration (q, a·w, ε·t) to (q_i,w, ε.t).
- Finally, on the occasion that we actually do wish to change the symbol c at the top of stack with symbol d, the configuration (q, a.w, c.t) changes to (qi,w, d.t).

An execution starts with the configuration (q_0, s, ε), i.e., the machine is in the start state, the input string s is on the tape, and the stack is empty. A successful execution is one which finishes in a configuration where s has been completely read, the final state of the machine is an accept state, and the stack is empty.

PDA as a state diagram:

$\delta(q_i, a, X)=\{(q_j, Y)\}$



Example 1

Design a pushdown automaton M to accept the language L = {WcW^R : w ∈{a, b } *}.

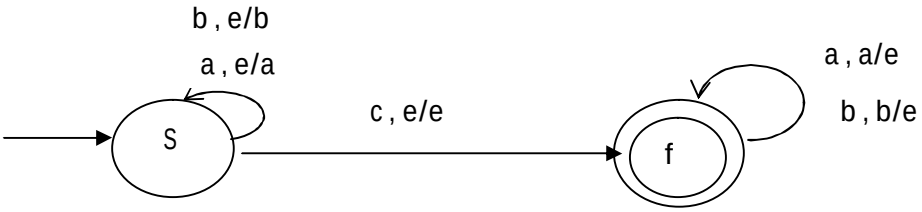
Solution : According to given language , ababcbaba ∈ L but abcab ϵ L. Let required PDA, M= (K, ∑, Γ ,Δ , S , F) where

K = {s , f}
∑= {a, b, c}
Γ= {a, b}
F = {f} and
Δ is given by following five transitions

1. ((s, a, e),(s, a) //push a
2. ((s, b, e),(s, b)) //push b
3. ((s, c, e), (f, e)) // change the state

4. ((f, a, a),(f, e)) // pop a on reading of input symbol a
5. ((f, b, b),(f, e)) // pop b on reading of input symbol b

State diagram



e.g Lets check for the string *abbcbba*

State	Unread input	stack content	Transition used
s	abbcbba	e	-
s	bbcbba	a	1
s	bcbba	ba	2
s	cbba	bba	2
f	bba	bba	3
f	ba	ba	5
f	a	a	5
f	e	e	4

- Here when machine sees a 'c' in input string , it switches from state s to f without operating on its stack.
- If the input symbol does not match the top stack symbol, no further operation is possible
- IF an automaton M reaches in configuration (f, e, e) , final state, end of input , empty stack , then the input was indeed of the form WcW^R

Example 2

Design a PDA M that accept a language given by L= { $a^n b^n$: $n \geq 1$ } .

Solution

Let required PDA, M= (K, ∑, Γ ,Δ , S , F) where

K = {s , q, f}

∑= {a,b}

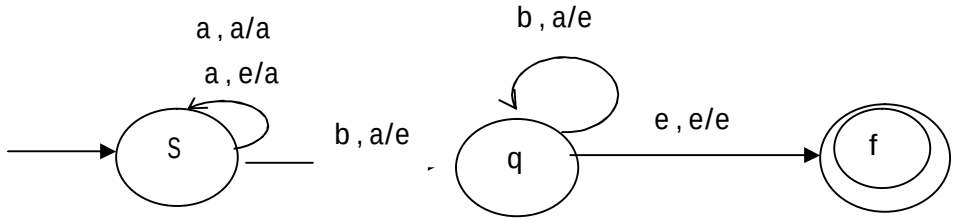
Γ= {a}

F = {f} and

Δ is given by following five transitions

1. ((s,a,e),(s,a) *//push a*
2. ((s,a,a),(s,a) *//push a*
3. ((s,b,a),(q,e)) *//pop a, when first b is encountered and also switch state from s to q*
4. ((q, b, a), (q, e)) *// continue pop a*
5. ((q,e,e),(f,e)) *// go to final state*

State diagram



e.g Lets check for the string *aabb*

State	Unread input	stack content	Transition used
s	aabb	e	-
s	abb	a	1
s	bb	aa	2
s	b	a	3
q	e	e	4
f	e	e	5

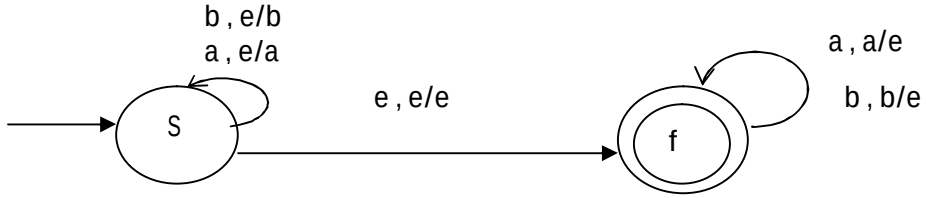
Example 3: Design a PDA that accept the following language L = {WW^R : w ∈ {a, b } * }.
Solution: By analysis of language the machine must guess when it has reached the middle of the input string and change from state s to state f in a non deterministic fashion.

Let required PDA, M= (K, ∑, Γ ,Δ , S , F) where

K = {s , q, f}
∑= {a, b}
Γ= {a, b}
F = {f} and
Δ is given by following five transitions

1. ((s, a, e),(s, a) *//push a*
2. ((s, b, e),(s, b) *//push b*
3. ((s, e, e),(f, e)) *//change the state non deterministically (middle of the input string is guessed by machine M itself)*
4. ((f, a, a), (f, e)) *// pop a*
5. ((f, b, b),(f, e)) *// pop b*

Here, ((s, e, e), (f, e)) means change to state f without reading /consuming any symbol. Clearly whenever the M is in state S it can non- deterministically choose either to push the next input symbol on the stack or to switch to state f without consuming any input.

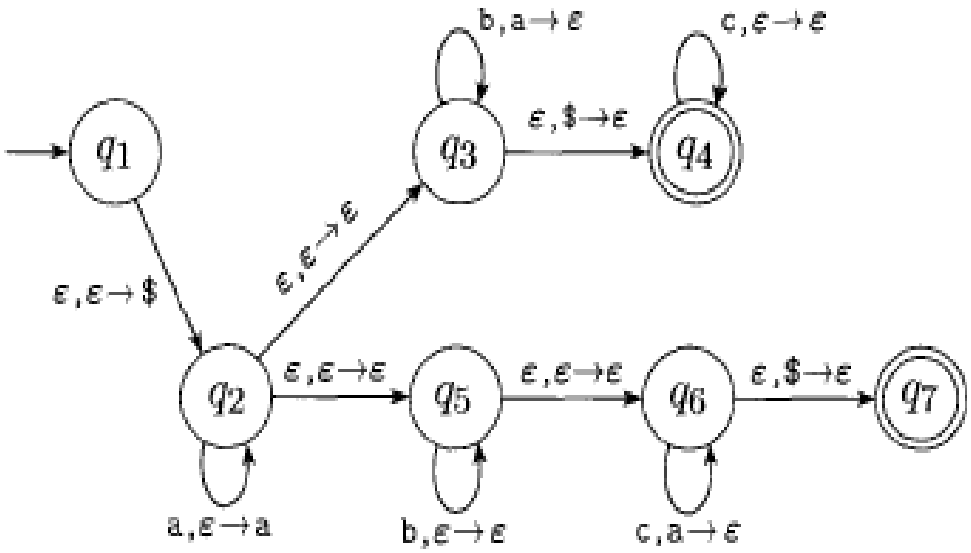


Example 4: Design a PDA that accepts the language
$L = \{a^i b^j c^k : i, j, k \geq O \text{ and } i = j \text{ or } i = k\}$.

Solution :

Let required PDA, $M = (K, \sum, \Gamma, \Delta, S, F)$ where
$K = \{q_1, q_2, q_3, q_4, q_5, q_6, q_7\}$
$\sum = \{a, b, c\}$
$\Gamma = \{a, b\}$
$F = \{q_4, q_7\}$ and
Δ is given by following transition diagram

Informally the PDA for this language works by first reading and pushing the a's. When the a's are done the machine has all of them on the stack so that it can match them with either the b's or the c's. This is a bit tricky because the machine doesn't know in advance whether to match the a's with the b's or the c's. Non-determinism comes in handy here. Using its non-determinism, the PDA can guess whether to match the a's with the b's or with the c's, as shown in the following figure. Think of the machine as having two branches of its non-determinism, one for each possible guess. If either of them match, that branch accepts and the entire machine accepts. In fact we could show, though we do not do so, that non-determinism is essential for recognizing this language with a PDA.



Deterministic PDA (DPDA)

- A PDA is deterministic if there is at most one move for any input symbol, any top stack symbol and at any state
- A no move or move without advancing input string are possible in a deterministic PDA
- A move without advancing input string implies that no other move exists
- The PDA for WCW^R is deterministic
- The set of languages accepted by DPDA's is only a subset of language accepted by non deterministic PDA
- A regular languge is language accepted by a DPDA
- Not all language accepted by DPDA are regular
- All languages accepted by DPDA have an unambiguous VFG
- Not all unambiguous languages are accepted by DPDA.

The PDA is deterministic in the sense that at most one move is possible from any instantaneous description (ID). Formally, we say that a PDA $M = (Q, \sum, \Gamma, \delta, s, F)$, is said to be deterministic if it is an automation as defined in definition of PDA , subject to the restrictions that for each q in Q , b in T and a ϵ $(\sum \cup \{e\})$

1. whenever δ (q, a, b) contains at most one element
2. if δ (q, e, b) is not empty, then δ (q, c, b) must be empty for every c in $\sum$;

The first of these conditions simply requires that for any given input symbol and any stack top ,at most one move can be made. The second condition is that when a e-transitions is possible for some configurations no input consuming alternative is available.

Condition 1 prevents a choice of move for any (q, a, b) or (q, e, b). Note that unlike the finite automaton, a PDA is assumed to be nondeterministic unless we state otherwise.

Condition 2 prevents the possibility of a choice between a move independent of the input symbol (e-move) and a move involving an input symbol.

For finite automata, the deterministic and nondeterministic models were equivalent with respect to the languages accepted. The same is not true for PDA. In fact ww^R is accepted by a nondeterministic PDA, but not by any deterministic PDA.

The Languages of a PDA

We have assumed that a PDA accepts its input by consuming it and entering an accepting state. We call this approach acceptance by final state. We may also define for any PDA the language accepted by empty stack, that is, the set of strings that cause the PDA to empty it stack, starting from the initial ID.

These two methods are equivalent, in the sense that a language L has a PDA that accepts it by final state if and only if L has a PDA that accepts it by empty stack.

However for a given PDA P, the languages that P accepts by final state and by empty stack are usually different. We will show conversion of a PDA accepting L by final state into another PDA that accepts L by empty stack, and vice-versa.

Acceptance by Final State

Let $M = (K,\Sigma, \delta, s, \Gamma, F)$ be a PDA. Then $L_f(M)$, the language accepted by M by final state, is

$L_f(P) = \{ w \in \Sigma^* : (s, w, e) \overset{*}{\vdash} (f, e, \alpha))\}$ for some state $f \in F$ and any stack string α

That is, starting in the initial ID with w waiting on the input, M consumes w from the input and enters an accepting state. The content of the stack at that time is irrelevant.

Acceptance by Empty Stack

Let $M = (K,\Sigma, \delta, s, \Gamma, F)$ be a PDA. Then $L_e(M)$, the language accepted by M by empty stack , is

$L_e(M) = \{w \in \Sigma^* : (s, w, e) \overset{*}{\vdash} (q, e, e))\}$ for some state $q \in K$

That is, the set of input w that M can consumes and at the same time empty its stack. w from the input and enters an accepting state. The content of the stack at that time is irrelevant.

Two methods are equivalent in the sense that a language L has a PDA that accepts it by final state if and only if L has a PDA that accepts it by empty stack.

- **From Empty Stack PDA to Final State PDA**
- **From Final state PDA to Empty stack PDA**

Proof Read yourself

Equivalence of PDA and CFG

Theorem: The class of languages accepted by PDA is exactly the class of context free languages.

Lemma1: Each CFL is accepted by some PDA.
Lemma2: If a language is accepted by a PDA , it is a CFL.

Proof of Lemma 1:

Construction of PDA equivalent to a CFG.

Let G= (V, $\sum$, R, S) be a CFG, we must construct a PDA M such that L(M) = L(G) . The machine we construct has only two states p and q and remains permanently in state q after its first move. Also M uses V : set of terminals and $\sum$ set of non terminals as its stack alphabet.

Let M= (K, $\sum$, Γ ,Δ , S , F) where
K = {p, q}
$\sum = \sum$
$\Gamma = V \cup \sum$
S=p and
Δ (transition relation) is defined as

1. ((p, e ,e), (q, S)) as S is starting non terminals of CFG.
2. ((q, e, A), (q, x)) for each rule $A \rightarrow x$
3. ((q, a, a) ,(q, e)) for each a $\in \sum$

Example
Consider the grammar G = (V, $\sum$, R, S) with V = {S, a, b ,c}, $\sum$ = {a, b, c}, and R = {S $\rightarrow$ aSa, S $\rightarrow$ bSb, S $\rightarrow$ e), which generates the language $\{WcW^R : w \in \{a, b\}^*\}$.
Design a pushdown automaton.
Solution

The corresponding pushdown automaton, according to the construction above, is
M = (Q, $\sum$, Γ, Δ, S, F) where
Q= {p, q}
$\sum = \sum$ = {a, b, c }
Γ ={S, a, b ,c}
S= p
F={ q} and
Δ= {((p, e, e), (q, S)), T1
((q, e, S), (q, aSa)), T2
((q, e, S), (q, bSb)), T3
((q, e, S), (q, e)), T4
((q, a, a), (q, e)), T5
((q, b, b), (q, e)), T6
((q, c, c), (q, e))} T7

The string **abbcbba** is accepted by M through the following sequence of moves.

State	Unread input	stack content	Transition used
p	abbcbba	e	-
q	abbcbba	S	T1
q	abbcbba	aSa	T2
q	bbcbba	Sa	T5
q	bbcbba	bSba	T3
q	bcbba	Sba	T6
q	bcbba	bSbba	T3
q	cbba	Sbba	T6
q	cbba	cbba	T4
q	bba	bba	T7
q	ba	ba	T6
q	a	a	T6
q	e	e	T5

Closure Properties of CFL

Theorem 1 : The context-free languages are closed under union, concatenation, and Kleene star.

Let $G_1=(V_1, \sum_1, R_1,S_1)$ and $G_2=(V_2, \sum_2, R_2,S_2)$ be two context-free grammars, and without loss of generality assume that they have disjoint sets of non-terminals , that is, V_1, - $\sum_1$ and V_2 - $\sum_2$ are disjoint.

Union:
Let S be a new symbol and let G= =(V, ∑, R, S) where
$V= V_1 \cup V_2 \cup S$
$\sum= \sum_1 \cup \sum_2$

$R = R_I \cup R_2 \cup \{S \rightarrow S_1 , \ S \rightarrow S_2\}$.

Then we claim that $L(G) = L(G_1) \cup L(G_2)$ For the only rules involving S are $S \rightarrow S_1$ and $S \rightarrow S_2$,
so $S \Rightarrow^*_G w$ if and only if either $S_1 \Rightarrow^*_G w$ or $S_2 \Rightarrow^*_G w$ and since G_1 and G_2 have disjoint sets of non-terminals, the last disjunction is equivalent to saying that $w \in L(G1) \cup L(G_2)$.

For example, to get a grammar for the language $\{0^n1^n | n \geq 0\} \cup \{1^n0^n | n \geq 0\}$, first construct the grammar

$$S_1 \rightarrow 0S_11 \mid \varepsilon$$

for the language $\{0^n1^n | \ n \geq 0\}$ and the grammar

$$S_2 \rightarrow 1S_20 \mid \varepsilon$$

for the language $\{1^n0^n | \ n \geq 0\}$ and then add the rule $S \rightarrow S_1 \mid S_2$ to give the grammar

$$\begin{aligned} S &\rightarrow S_1 \mid S_2 \\ S_1 &\rightarrow 0S_11 \mid \varepsilon \\ S_2 &\rightarrow 1S_20 \mid \varepsilon . \end{aligned}$$

\

Concatenation

The construction is similar: $L(G_1) .L(G_2)$ is generated by the grammar $G = (V, \sum, R, S)$ where
$V = V_1 \cup V_2 \cup S$
$\sum = \sum_1 \cup \sum_2$
$R = R_I \cup R_2 \cup \{S \rightarrow S_1 S_2\}$.

For a string $x = x_1x_2 \in L_1L_2$, where $x_1 \in L_1$ and $x_2 \in L_2$, a derivation of x in G_c is

$$Sc \Rightarrow S_1S_2 \Rightarrow^* x_1S_2 \Rightarrow^* x_1x_2$$

Kleene Star.

$L(G_1)^*$ is generated by $G = (V, \sum, R, S)$ where
$V = V_1 \cup S$
$\sum = \sum_1$

$R = R_I \cup \{S \rightarrow e , \ S \rightarrow SS_2\}$.

Theorem 2: The class of context-free languages is not closed under intersection or complementation.

This is not very surprising: Recall that our proof that regular languages are closed under intersection depended on closure under complementation; and that construction required that the automaton be deterministic. And not all context-free languages are accepted by deterministic pushdown automata (e.g WW^R is context free language but not accepted by Deterministic PDA but can be accepted by non deterministic PDA)

Theorem: The intersection of a context-free language with a regular language is a context-free language.

Proof: If L is a context-free language and R is a regular language, then L = L(M1) for some pushdown automaton $M_1 = (K_1, \Sigma, \Gamma, \Delta, S_1, F_1)$, and R = L($M_2$) for some deterministic finite automaton $M_2 = (K_2, \Sigma, \delta, S_2, F_2)$ The idea is to combine these machines into a single pushdown automaton M that carries out computations by M_1 and M_2 in parallel and accepts only if both would have accepted. Specifically, let M = (K, Σ, Γ, Δ, s, F), where

K = K_1 x K_2, the Cartesian product of the state sets of M_1 and M_2;
f = f_1 ;
s = (S_1, S_2);
F = F_1 X F_2, and
Δ, the transition relation, is defined as follows.
For each transition of the pushdown automaton $((q_1, \alpha,, \beta), (p_1, \gamma)) \in \Delta$, and for each state $q_2 \in K_2$, we add to Δ. the transition $(((q_1, q_2), \alpha, \beta), ((p_1, \delta(q2, a)), \gamma))$; and for each transition of the form $((q_1, e, \beta, (p_1 , \gamma)) \in \Delta$ and each state $q_2 \in K_2$, we add to Δ the transition $(((q_1 , q_2), e, \beta), ((p1 , q2), \gamma))$ That is, M passes from state (ql, q_2) to state (p_1, p_2) in the same way that M_1 passes from state q_1 to p_1, except that in addition M keeps track of the change in the state of
M_2 caused by reading the same input.

Regular Grammars

A Context free grammar G = (V, ∑, R, S) is said to be regular if each production has a right hand side that consists of a string of terminals followed by at most one non terminal.

- A grammar is regular if it is either left- or right-linear.
- A linear grammar may have a mix of left and right productions. Its only restriction is that there is at most one symbol on the right.
- All regular grammars are linear, but not all linear grammars are regular!

Example 1:
Let G = (V, ∑, R, S) where
V= { A,B,S, a,b}
∑= { a, b}

R= {
S → abA/B/baB/e
A → bS

B →sS/b

}

Right Linear Grammars

Right-linear: at most one variable on the right side of the production must be the right-most symbol.

$A \rightarrow xB$

$A \rightarrow x$

x is any string of terminals.

Right Regular Grammars:

Rules of the forms

$A \rightarrow \varepsilon$

$A \rightarrow a$

$A \rightarrow aB$

A, B: variables (Non terminals) and

a: terminal

Left Linear Grammars

Left-linear: at most one variable on the right side of the production, must be the left-most symbol.

$A \rightarrow Bx$

$A \rightarrow x$

x is any string of terminals.

Left Regular Grammars:

Rules of the forms

$A \rightarrow \varepsilon$

$A \rightarrow a$

$A \rightarrow Ba$

A, B: variables (Non terminals) and

a: terminal

Prove: All right-linear grammars describe regular languages

- Idea: Find a way to convert any right-linear grammar into a FSA.
  - Terminals before a non-terminal represent arcs
  - Non-terminals represent non-final states.
  - Terminals without non-terminals following represent final states.

Regular Grammar to NFA

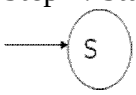
Example: For the production rules of Regular Grammar given below find the NFA.

$S \rightarrow aD$

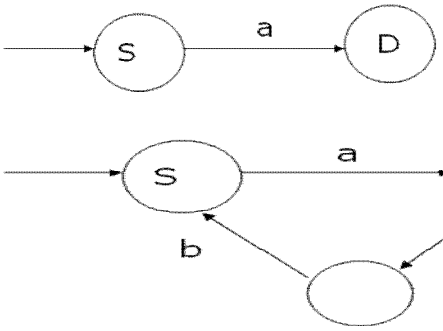
$D \rightarrow abS$

$D \rightarrow b$

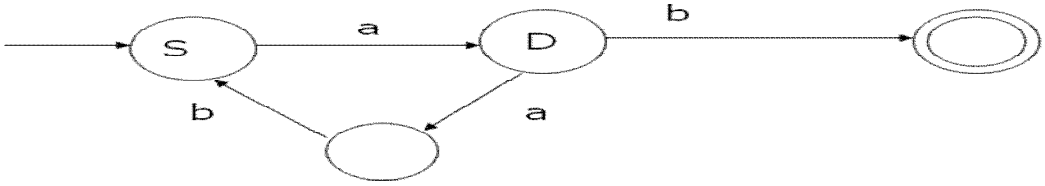
Step 1: Start variable is the initial node.



Step 2: Each rule ending in a non-terminal is a chain of arcs followed by a non-final node.



Step 3: Each rule ending in a terminal leads to a final state.

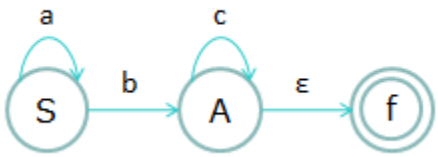


Example : Transform the following Right Regular grammar in an equivalent NFAε.

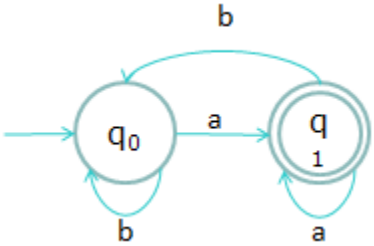
$S \rightarrow aS \mid bA$

$A \rightarrow cA \mid \varepsilon$

Solution:



Transform the following DFA to a right regular grammar



Solutions

$$Q_0 \rightarrow aQ_1 \mid bQ_0$$
$$Q_1 \rightarrow aQ_1 \mid bQ_0 \mid \varepsilon$$

Simplification of CFG

1. Elimination of useless symbols

2. Elimination of empty production (e-productions)
3. Elimination of unit Productions