Segment-1

❖ Context-Free Grammar

A context-free grammar is a set of recursive rules used to generate patterns of strings. Context-free grammars (CFGs) are used to describe context-free languages. A context-free grammar can describe all regular languages and more, but they cannot describe all possible languages.

Context-free grammars are studied in fields of theoretical computer science, compiler design, and linguistics. CFG's are used to describe programming languages and parser programs in compilers can be generated automatically from context-free grammars.

Formal definition of Context-Free Grammar

Context free grammar G can be defined by four tuples as: G= (V, T, P, S) Where,
G describes the grammar
T describes a finite set of terminal symbols.
V describes a finite set of non-terminal symbols
P describes a set of production rules
S is the start symbol

Features of CFG:

- A grammar consists of a collection of substitution rules are called productions.
- Each rule appears as a line in the grammar, comprising a symbol and a string separated by an arrow. The symbol is called a variable.
- The string consists of variables and other symbols called Non terminals. The variable symbols often are represented by capital letters.
- The terminals are the input alphabet and often are represented by lowercase letters, numbers, or special symbols.
- The sequence of substitutions to obtain a string is called a derivation.The process of deriving a string from given grammar is called as derivation.
- The geometrical representation of a derivation is known as a derivation tree or parse tree.

The following is an example of a context-free grammar, which we call G1.

$$A \rightarrow 0A1$$
$$A \rightarrow B$$
$$B \rightarrow \#$$

For example, grammar G1 contains three rules. G1's variables are A and B, where A is the start variable. Its terminals are 0, 1, and #.

Q1. Construct a Context-Free Grammar (CFG) for the regular expression (0 + 1)*0 1*, that is, any combination of 0 and 1 followed by a single 0 and ending with any number of 1.

1. Define the non-terminals:

S: Start symbol
A: Represents the part before the single 0
B: Represents the single 0
C: Represents the part after the single 0

2. Define the terminals:
0: The digit 0
1: The digit 1

3. Define the production rules:
S → A0C
A → ε | 0A | 1A
C → ε | 1C

Explanation:
- S is the start symbol and represents the entire string.
- A represents the part of the string before the single 0. It can be empty (ε), contain one or more 0s, or contain one or more 1s.
- B represents the single 0 in the middle of the string.
- C represents the part of the string after the single 0, which can be empty or contain one or more 1s.

Q2. Construct a CFG for the regular expression 0*1(0+1)*, that is, any number of 0 followed by a single 1 and ending with any combination of 0 and 1

1. **Define the non-terminals:**
S: Start symbol
A: Represents the part before the single 1
B: Represents the single 1
C: Represents the part after the single 1

2. **Define the terminals:**
0: The digit 0
1: The digit 1

3. Define the production rules:
S → A1C
A → ε | 0A
C → ε | 0C | 1C

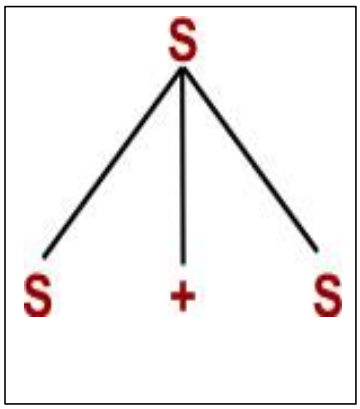
Explanation:
- S is the start symbol and represents the entire string.
- A represents the part of the string before the single 1. It can be empty (ε) or contain zero or more 0s.
- B represents the single 1 in the middle of the string.
- C represents the part of the string after the single 1. It can be empty (ε), contain zero or more 0s, or contain zero or more 1s.

Parse Tree:
The process of deriving a string from given grammar is called as derivation. The geometrical representation of a derivation is known as a derivation tree or parse tree.

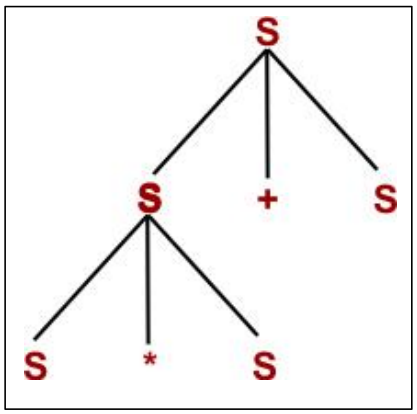
Parse trees hierarchically represent the terminals and non-terminals. They generate input strings from the given grammar.

Example
The following Production rules of a Grammar (G)
S → S + S | S * S
S → x|y|z
and Input is (x * y + Z)

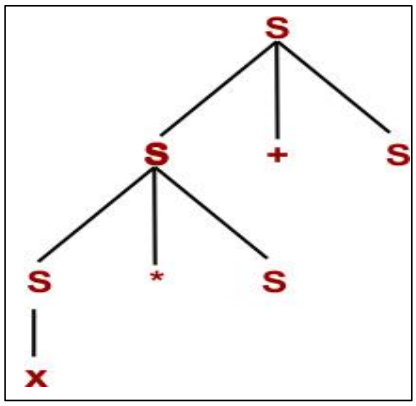
Step 1



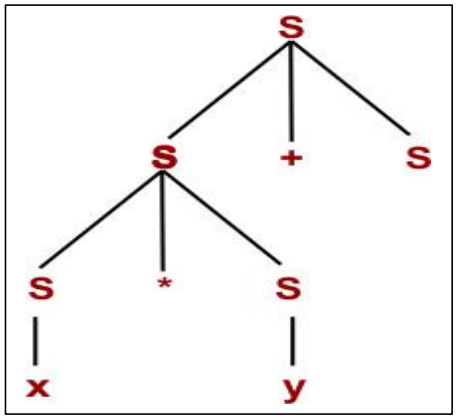
Step 2



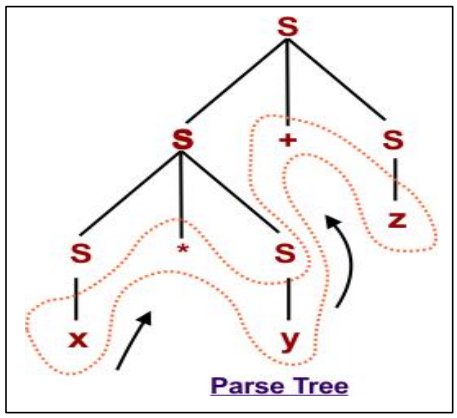
Step 3



Step 4

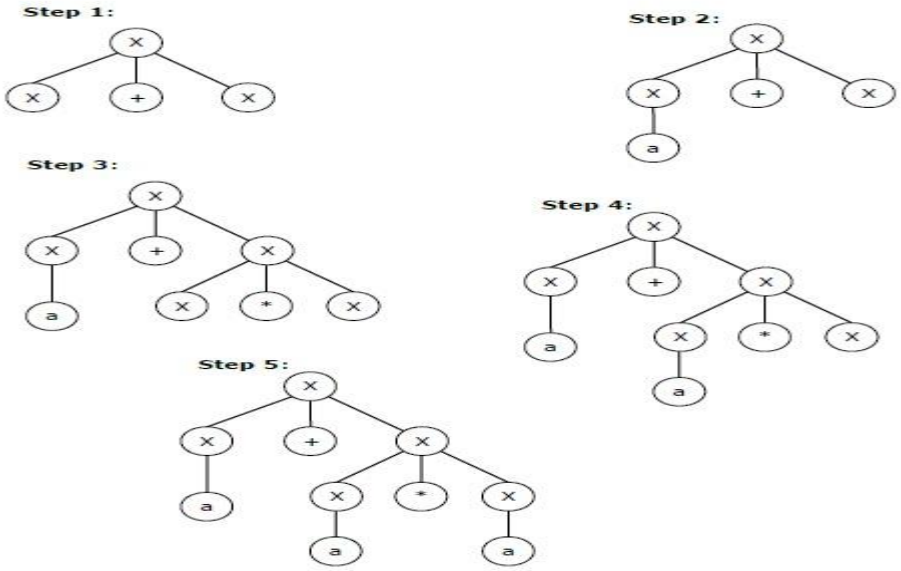


Step 5



Types of Parse Tree or Derivation:
There are two types of derivation. Such as
Leftmost derivation − A leftmost derivation is obtained by applying production to the leftmost variable in each step.
Example:
Let any set of production rules in a CFG be
X → X+X | X*X |X| a over an alphabet {a}.
The leftmost derivation for the string "a+a*a" may be −
X → X+X
X → a+X
X→ a + X*X
X→ a+a*X
X→ a+a*a

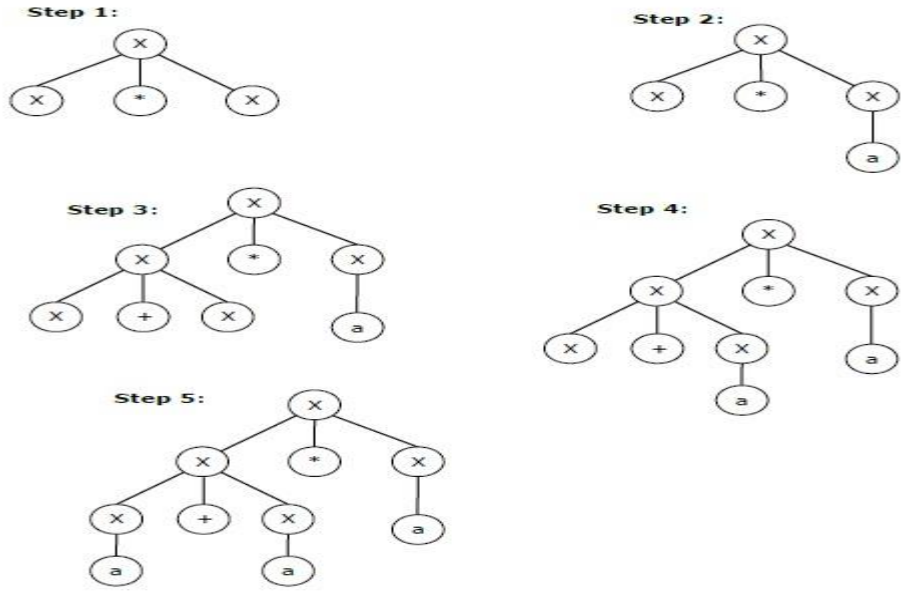
The leftmost derivation Parse tree for the given string is shown as below −



Rightmost derivation − A rightmost derivation is obtained by applying production to the rightmost variable in each step.

Example
Let any set of production rules in a CFG be
X → X+X | X*X |X| a over an alphabet {a}.

The rightmost derivation for the above string "a+a*a" may be −
X → X*X
X → X*a
X → X+X*a
X→ X+a*a
X→ a+a*a
The rightmost derivation Parse tree for the given string is shown as below −



AMBIGUITY

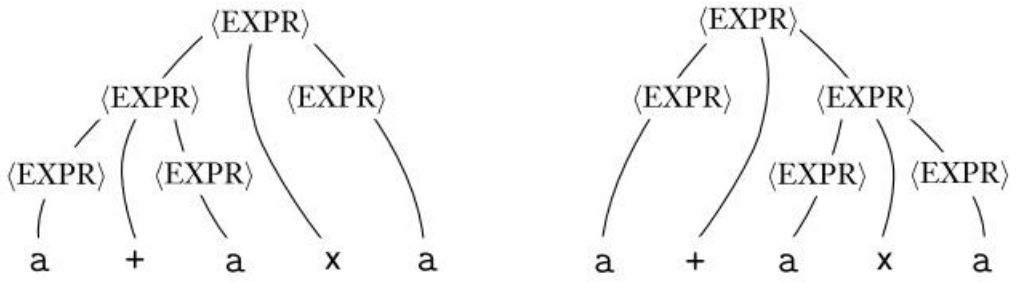
If a grammar generates the same string in several different ways, we say that the string is derived *ambiguously* in that grammar. If a grammar generates some string ambiguously, we say that the grammar is *ambiguous*.

Problem 1:

For example, consider grammar G:

$$\langle \text{EXPR} \rangle \rightarrow \langle \text{EXPR} \rangle + \langle \text{EXPR} \rangle \mid \langle \text{EXPR} \rangle \times \langle \text{EXPR} \rangle \mid (\langle \text{EXPR} \rangle) \mid \text{a}$$

This grammar generates the string a+axa ambiguously. The following figure shows the two different parse trees.



Problem 2:

Check whether the following grammar is ambiguous or not for string w = ab

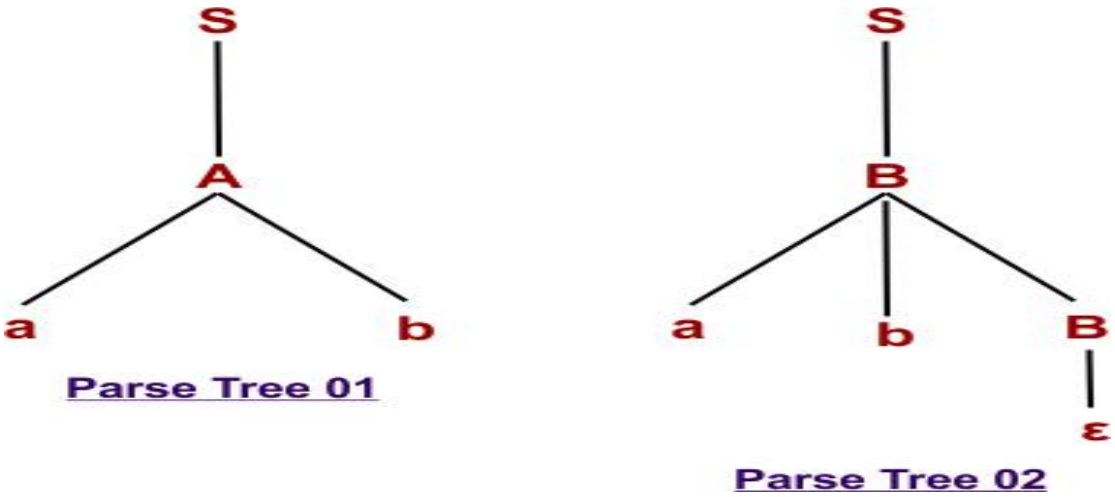
S → A / B

A → aAb / ab

B → abB / ∈

Solution

Now we draw more than one parse trees to get string w = ab.



As original string (w =ab) can derived through two different parse trees. So, the given grammar is ambiguous.

❖ **Chomsky normal form:**

A context-free grammar is in ***Chomsky normal form*** if every rule is of the form

$A \rightarrow BC$

$A \rightarrow a$

where a is any terminal and A, B, and C are any variables

❖ Conversion of CFG into CNF

Convert the following CFG into CNF

$S \rightarrow ASA \mid aB$

$A \rightarrow B \mid S$

$B \rightarrow b \mid \varepsilon$

(1) Since **S** appears in R.H.S, we add a new state **S0** and **S0→S** is added to the production set and it becomes −

S0→S

S→ ASA | aB

$A \rightarrow B \mid S$

$B \rightarrow b \mid \varepsilon$

(2) Now we will remove the null productions $B \rightarrow \varepsilon$ and $A \rightarrow \varepsilon$

After removing $B \rightarrow \varepsilon$, the production set becomes −

S0→S

S→ ASA | aB | a

$A \rightarrow B \mid S \mid \varepsilon$

$B \rightarrow b$

After removing $A \rightarrow \varepsilon$, the production set becomes −

S0→S

S→ ASA | aB | a | AS | SA | S

$A \rightarrow B \mid S$

$B \rightarrow b$

(3) Now we will remove the unit productions.

After removing $S \rightarrow S$, the production set becomes −

S0→S

S→ ASA | aB | a | AS | SA

$A \rightarrow B \mid S$

$B \rightarrow b$

After removing S0→ S, the production set becomes −

S0→ ASA | aB | a | AS | SA

S→ ASA | aB | a | AS | SA

A → B | S

B → b

After removing A→B, the production set becomes −

S0 → ASA | aB | a | AS | SA

S→ ASA | aB | a | AS | SA

A → S | b

B → b

After removing A→ S, the production set becomes −

S0 → ASA | aB | a | AS | SA

S→ ASA | aB | a | AS | SA

A → b |ASA | aB | a | AS | SA

B → b

(4) Now we will find out more than two variables in the R.H.S

Here, S0→ ASA, S → ASA, A→ ASA violates two Non-terminals in R.H.S.

Hence we will apply step 4 and step 5 to get the following final production set which is in CNF −

S0→ AX | aB | a | AS | SA

S→ AX | aB | a | AS | SA

A → b |AX | aB | a | AS | SA

B → b

X → SA

(5) We have to change the productions S0→ aB, S→ aB, A→ aB

And the final production set becomes −

S0→ AX | YB | a | AS | SA

S→ AX | YB | a | AS | SA

A → b A → b |AX | YB | a | AS | SA

B → b

X → SA

Y → a

For More Practice see this tutorial:
https://www.youtube.com/watch?v=Xw5iXARgTXk

❖ Pushdown Automata

Pushdown Automata is a finite automata with extra memory called stack which helps Pushdown automata to recognize Context Free Languages.

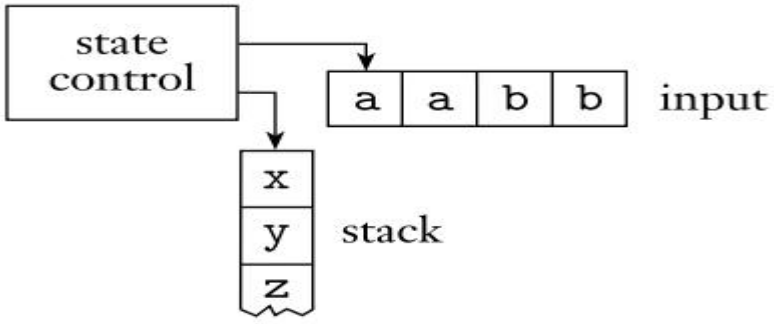
Pushdown automata is a way to implement a CFG in the same way we design DFA for a regular grammar. A DFA can remember a finite amount of information, but a PDA can remember an infinite amount of information.

Basically a pushdown automaton is −

"Finite state machine" + "a stack"

A pushdown automaton has three components −

- an input tape,
- a control unit, and
- a stack with infinite size.



Format of Transition:

- A,B->c where A is the input symbol,B is the stack symbol read or pop,C is the insert symbol on stack

Formal Definition of Pushdown Automata:

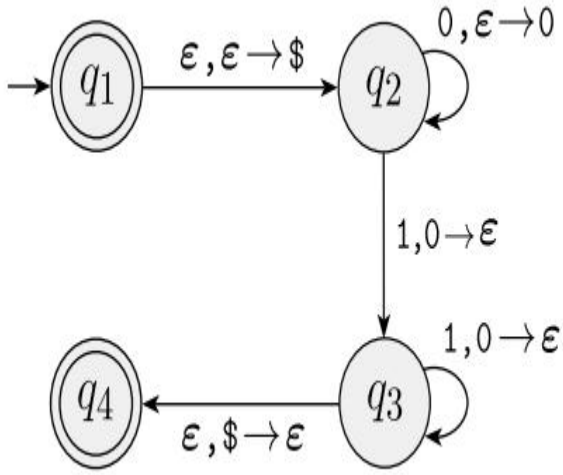
A Pushdown Automata (PDA) can be defined as 7 tuples – M = (Q, Σ, Γ, δ, q0, Z, F)

where

- Q is a finite set of states
- Σ is a finite set which is called the input alphabet
- Γ is a finite set which is called the stack alphabet
- δ is a finite subset of the transition .
- q0 ∈ Q is the start state
- Z ∈ Γ is the initial stack symbol
- F ⊆ Q is the set of accepting states

Example:
Construct PDA for the language that recognizes $\{0^n1^n | n \geq 0\}$

The Input is: 00001111



Stack

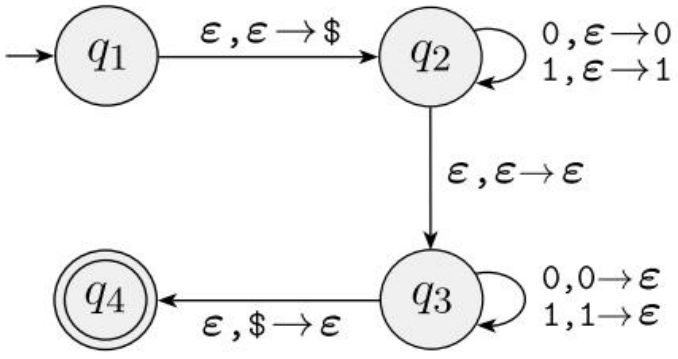
0
0
0
0
$

Transition Table:

Input:	0			1			ε		
Stack:	0	$	ε	0	$	ε	0	$	ε
q_1									$\{(q_2, \$)\}$
q_2			$\{(q_2, 0)\}$	$\{(q_3, \varepsilon)\}$					
q_3				$\{(q_3, \varepsilon)\}$				$\{(q_4, \varepsilon)\}$	
q_4									

Construct PDA for the language that recognizes $L=\{ww^R | w \in \{0,1\}^* \}$

The Input is: 1110 0001



Stack

0
1
1
1
$

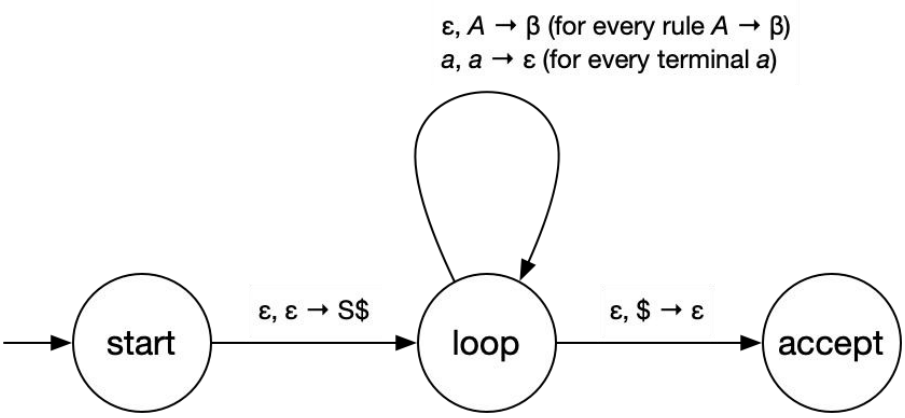
https://www.youtube.com/watch?v=kRzZ2WjvINQ
https://www.youtube.com/watch?v=kRzZ2WjvINQ&list=PLE4rcHuYzdpJnurHeU01Rol9WzCTP33B7&index=10

Conversion of CFG to PDA

Conversion of CFG to PDA consists of five steps. The steps are as follows:

- Convert the CFG productions into GNF.
- There will only be one state, "q," on the PDA.
- The CFG's first symbol will also be the PDA's initial symbol.
- Include the following rule for non-terminal symbols:
  $\delta(q, \varepsilon, A) = (q, \alpha)$, Where the production rule is $A \rightarrow \alpha$
- Include the following rule for terminal symbol:
  $\delta(q, a, a) = (q, \varepsilon)$ for every terminal symbol

General Format of Converted PDA



The following grammar should be converted to a PDA that supports the same language.

$S \rightarrow 0S1 \mid A$
$A \rightarrow 1A0 \mid S \mid \varepsilon$

Solution

Unit productions can be removed in order to first simplify the CFG:

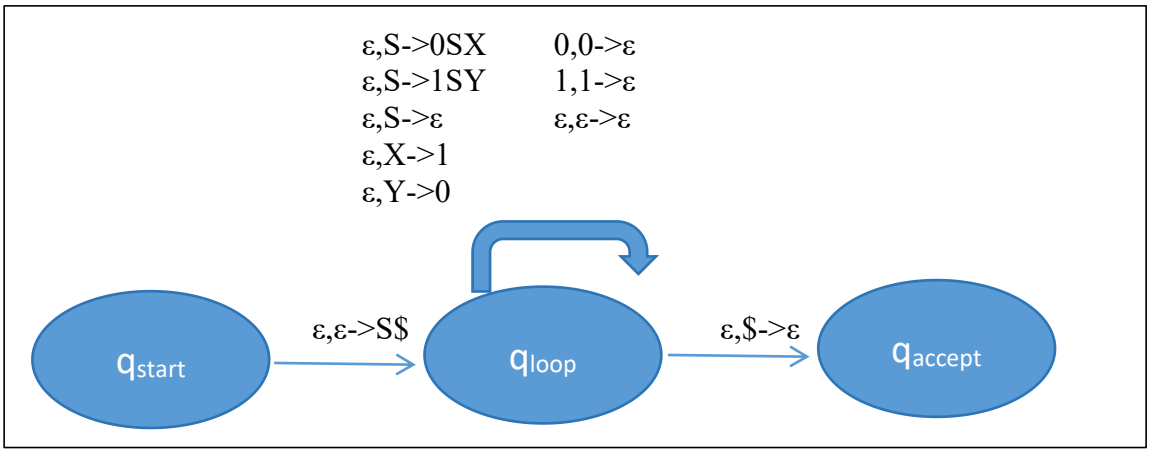
$S \rightarrow 0S1 \mid 1S0 \mid \varepsilon$

We shall now translate this CFG into GNF:

$S \rightarrow 0SX \mid 1SY \mid \varepsilon$
$X \rightarrow 1$
$Y \rightarrow 0$

A PDA could be:

R1: $\delta(q, \varepsilon, S) = \{(q, 0SX) \mid (q, 1SY) \mid (q, \varepsilon)\}$
R2: $\delta(q, \varepsilon, X) = \{(q, 1)\}$
R3: $\delta(q, \varepsilon, Y) = \{(q, 0)\}$
R4: $\delta(q, 0, 0) = \{(q, \varepsilon)\}$
R5: $\delta(q, 1, 1) = \{(q, \varepsilon)\}$



For More Practice

Convert any one of the following context-free grammar (CFG) to an equivalent pushdown automaton

$S \rightarrow aSb \mid bY \mid Ya$

$Y \rightarrow bY \mid aY \mid c \mid \varepsilon$

OR

$S \rightarrow aXbX$

$X \rightarrow aY \mid bY \mid \varepsilon$

$Y \rightarrow X \mid c$