Compiler Design, Handout No.3
Assignment No.3

OTHER FORMS OF GRAMMARS

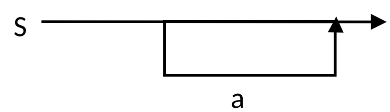
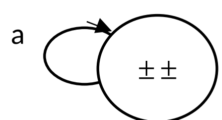
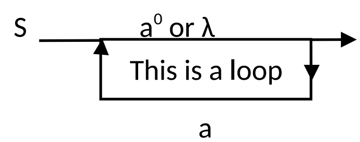
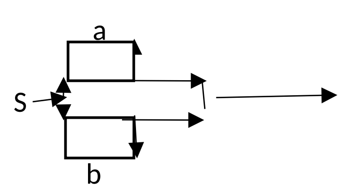
Notations.

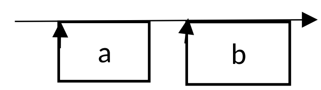
a. [a] means “a” or λ. For example, w=b[a], means either w= bλ =b or w = ba.
b. | means +. For example, a*|b* means a*+ b*.
c. {a} means any power of a including the zero power. {a} means a*

Examples: (a+b)*= { a|b } , a* + b* = {a} | {b}, ab* =a{b}, and a*b* = {a}{b}

To represent a CFG, we can use one of the following methods as well.

i. The ***Syntax diagram***

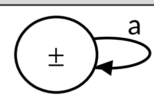
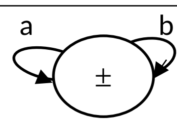
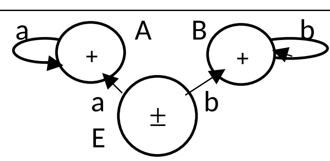
CFG	Syntax diagram
Example: the syntax diagram of **S→[a] is** S→[a] or S→a | λ	 Start at S, the straight line is λ (S=λ). But starting from S if we go down and then up then S=a
Example: Find the syntax diagram of L=a*  CFG: X→aX | λ	Syntax diagram  S→{a} or S→aS | λ Start at S, the straight line is for S=λ=a^0 . Start at S we can go through the loop and go back to the loop before we exit. Hence S=a (go through loop once), S=a^2 (go through loop twice) and so on. Pay attention to the difference between one a or many a’s
Example. Find the syntax diagram of L=a*+b*	The syntax diagram of language L 

Example. Find the syntax diagram of L=(a+b)*	Any powers of a, any powers of b, and any combinations of a's and b's S
Example. Find the Syntax diagram of L=a*b*	S 

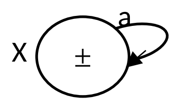
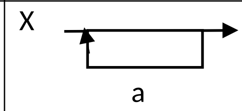
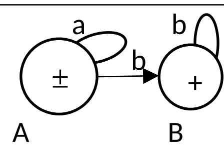
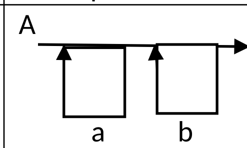
ii. ***The Backus Nour Form (BNF)***, is aregular CFG when we write one rule per line. Example:

CFG	CFG in BNF form
A→aA | bB | λ B→bB | λ	A→aA A→bB B→bB A→λ B→λ

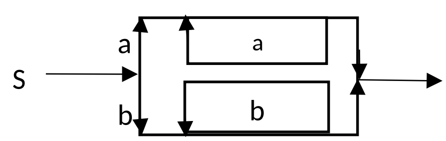
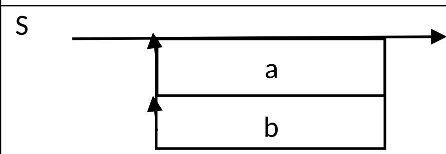
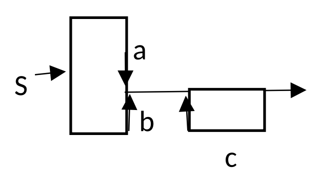
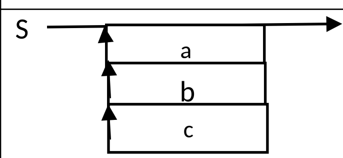
iii. ***Extended BNF (EBNF), i***s a CFG in which we write the language of CFG using the notations [ ], { }, and |

CFG	EBNF form of CFG
CFG: A→aA | λ The FA of this grammar is :  The language of this CFG is L=a*	It is much easier to find the EBNF from the Language, For L= a* , then CFG: A→aA |λ EBNF: A→{ a }
CFG: A→aA | bA | λ FA:  Language: L=(a+b)*	L= (a+b)* CFG: A→aA | bA | λ EBNF: A→{ a | b }
E→aA |bB | λ A→aA| λ B→bB | λ FA:  L=a* + b*	L= a* + b* CFG: E→aA|bB|λ, A→aA|λ, B→bB|λ EBNF:E→{a} | {b}

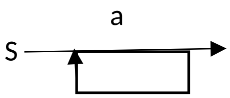
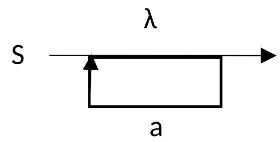
The following are examples to cover all cases at the same time.
Example. For each language, find its (i)FA, (ii)CFG, (iii)CFG in BNF, (iv) CFG in EBNF, (v) Syntax diagram.

Language	FA	CFG	BNF	EBNF	Syntax diagram
L=a*		X→aX | λ	X→aX X→λ	Since L=a*, the EBNF grammar is X→{a}	 the top line is λ
L=a*b*		A→aA|bB|λ B→bB|λ	A→aA A→bB B→bB A→λ B→λ	L=a*b* implies EBNF grammar is A→{a}{b}	

It is clear that to find the **EBNF** and **syntax diagram** there is no need to have the FA or the CFG of the language. The following examples are to find EBNF grammar and the syntax diagram directly from the languages.
Examples: Find the CFG of each language in EBNF and syntax diagram

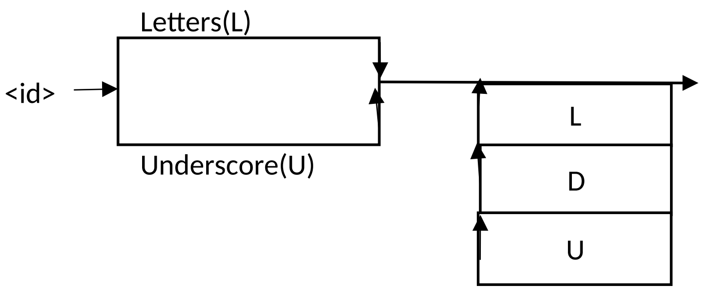
Language (L)	EBNF grammar of L	Syntax diagram to describe grammar of L
L=aa* + bb* Only powers of a or powers of b λ is not in L	L = aa* + bb* S→a{a} | b{b}	
L=(a+b)* ={λ, powers of a, powers of b, any combinations of a's and b's}	The * means { }, and a+b means a|b. Thus S→{ a | b }	 The top straight line is for λ The top loop generates all powers of a, The other loop generates all powers of b, Combination of the loops generate words made up of a's and b's
L=(a+b)c*	S→(a| b){c}=a{c} | b{c}	
L=(a+b+c)*	S→{a|b|c}	

Note: The following syntax diagrams representing L1=aa* and L =a*

L= aa*	L = a*
 At least one “a”, or more a’s	 No “a” (or λ) or more a’s

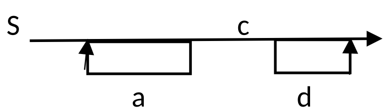
Examples. To write a program for a given grammar, the grammar must be in BNF format. Now, lets practice how to convert a given grammar to another form of grammars.

a. Identifiers in C++. An identifier is a string of letters, digits, and underscores. Identifiers must begin with a letter or underscore. Construct syntax diagram and write its EBNF for the CFG of identifiers in C++



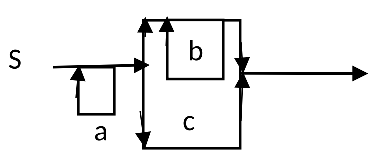
EBNF grammar
<id> → L | U | LX |UX
X →{L | D | U }
L → a|b|...|z|A|B|....|Z
D → 0|1|2|......9
U → _

b. Given **EBNF grammar S→{a} c [d]**, Construct its **syntax diagram**, and write the grammar in form of **BNF.**



The BNF of the grammar look like this
S→ A c D, where A={a}, D=[d], and c is fixed
A→ aA ,
A→ λ
D→d
D→ λ

c. Given L =a*(b* + c), construct the syntax diagram of L. Write the grammar of L in EBNF format.



The EBNF grammar of L is:
S→ {a} ({b} | c)

d. Write the **BNF** of the following **EBNF** grammars.

i. **S→{a}{b}**

S→AB,
A→aA
A→ λ,
B→bB
B→λ

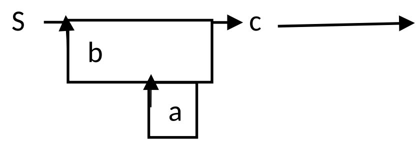
ii. **S→{a|b}**

let T=a|b, then S→{T} or S→TS | λ

Hence:

S→TS
S→ λ
T→a
T→b

e. **Given S→{ [a] b } c** , construct the syntax diagram of this EBNF grammar, and write the grammar in BNF.



In the loop a's are before b

The BNF of this grammar is:

S → Xc , wher X={ [a]b}
X→ YX , where Y=[a]b=ab or Y=λb = b or ab
Y→ab
Y→ b
X→λ

f. Convert **S→[a] { b | c }** from EBNF to a simple CFG

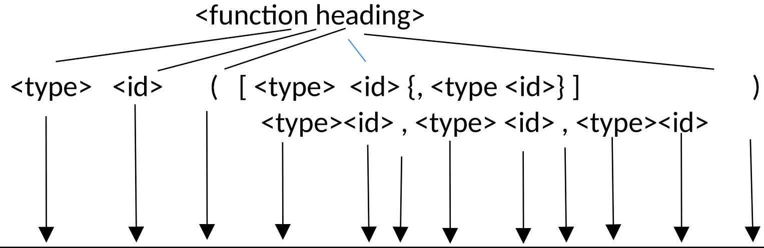
Let A=[a] and X={b | c }, then the grammar becomes S→AX, where A=a or λ and X=(b+c)*
Lets do one more substitution: X=(b+c)* or X=R* where R =b | c. Now put all pieces
Together to have the BNF format of the given grammar

S→AX , where A=[a]=a, λ and X={a|b}=(a+b)* =R* , or X→RX , λ and R=a, b.
A→a | λ
X→RX |λ
R→b | c

g. Construct EBNF grammar for simple f'sunction headings in C++(for example: void f(), int f(int a, int b).

<functions heading> → <type> <id> ([<type><id>{ , <type><id> }])
<type>→void| int |float |string
<id> →(<letter>|<underscore>){<letter>|<digit>|<underscore>}
<letter>→<upper>|<lower>
<upper>→A|B|C|………|Z
<lower>→a|b|c|……….|z
<digit>→0|1|2|………..|9
<underscore>→_

Trace the grammar for function heading: int sum(int a1, int a2, int a3)



int	sum	(	int	a1	,	int	a2	,	int	a3	)

REGULAR AND NON-REGUALR LANGUAGES

Definition. Language L is ***regular*** if we can write L using ( ), * and +. Otherwise the language is called a ***non-regular*** language.
Example.

Regular languages	Non-regular languages
i. L=a*b* ii. L=(a+b)* iii. L=ab* + ba*	L={ $a^n b^n$ |n=1,2,3,...}={ab, aabb, aaabbb,…….} Notice that this language is not L2=a*b*. word aaab is in L2 but is not a member of L

TERMINAL AND NON-TERMINAL SYMBOLS
Consider the following CFG:

A→ aA| bB
B→ bB | λ

The set of Terminals ={a, b, λ } and the set of non-terminals = { A,B }. In the other word, ***terminals*** are symbols that are not expandable and ***non-terminals*** are expandable. Suppose we want to use this grammar to trace the word= aab, then :

```
      A
     / \
    a   A
       / \
      a   A
         / \
        b   B
a   a   b    λ = w
```

, Terminals={a,b,λ}, non-terminals={A, B}

REGULAR AND NON-REGULAR CONTEXT-FREE-GRAMMARS

Definition. CFG is ***regular*** if each rule in the grammar is in one of the following forms:

(a) < non-terminal> → string of terminals with **exactly ONE** non-terminal at the END
Example: A→bD , b is terminal and D is non-terminal at the end
(b) < nonterminal >→ terminals including λ
Example: X→a | λ

Example. For CFG: X→aX | bX | λ, in which terminal={ X } and non-terminals={a,b, λ}. The first two rules satisfy rule (a) and the last one satisfies rule (b). Therefore, this CFG is a regular.

Definition. If the CFG is not regular, then it is called a ***non-regular CFG***

Example. Given CFG: E→AB, A→aA | λ, B→Bb. The first rule E→AB does not satisfy rule (a) above (there are more than one non-terminal (A,B)on the right-hand-side) therefore this grammar is non-regular

Recall:

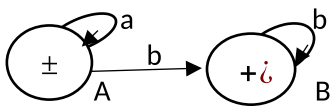
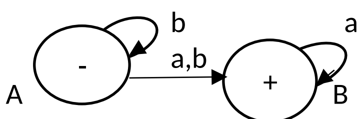
FA

Regular language ↔ **Regular CFG**

This indicates that if you have the FA, you can find the regular CFG and also a regular language for the machine. We have covered all these conversion cases in handout 1 and 2.

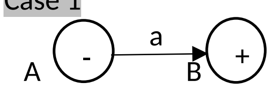
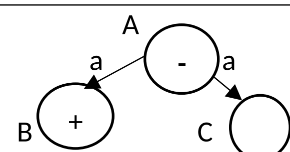
CONSTRUCTING REGUALR AND NON-REGULAR CFG FOR A GIVEN REGULAR LANGUAGE

There are two methods to find a CFG for a given regular language
Example.

Regular language	**Regular CFG**: construct FA and then use it to write regular CFG	**Non-regular CFG**: Write the non-regular CFG directly using the language
L=a*b*	Construct an FA to accept L  Use the FA to write a regular CFG A→aA| bB|λ B→bB|λ	L = a* b* Let A=a* and B=b*, then Write non-regular CFG S→AB , two nonterminals; make it Non-regular CFG A→aA |λ B→bB|λ
L=b*(a+b)a* λ is not in L, thus initial state is not a final state	FA:  Regular CFG: A→bA |aB | bB B→aB|λ	L = b*(a+b)a* Let B=b*, X=a+b, and A=a*. Then Non-regular CFG look like this S→ BXA B→bB|λ X→a | b A→ aA|λ

DETERMINISTIC AND NON-DETERMINSITC FINITE AUTOMATOA

Consider the following two cases:

Case 1  a is accepted by this machine	Case 2  Start at state A, for input a we have two choices, either enter state B (a is accepted) or enter state C (a is not accepted). But in handout 1, we said word is accepted by FA if "there is a way" to start at initial state and enter a final state. Hence a is accepted by this machine. This is an example of a ***non-deterministic FA***, from state A the input "a" gives us **two choices**, either enter state B or state C

In case 1, input **a** takes us from state A to only one next state B, this is an example of ***deterministic FA***. That is for each input there is only ONE next state to enter.

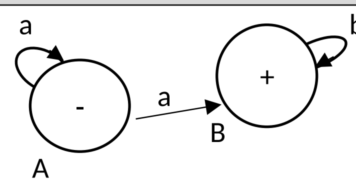
In case 2, at state A for input **a**, we have a choice to enter state B or state C (means there are more than one next states for input a), this is an example of ***non-deterministic FA*** (we are not determine whether to enter state B or state C)

When you design an FA to represent a grammar or a language, it is acceptable for the FA to be non-deterministic. But, when you want to write a program for that FA you have to make sure the FA is a deterministic. Following is a technique to convert non-deterministic FA to a deterministic FA.

CONVERTING NON-DETERMINITIS FA TO A DETEMINITISTIC FA

Example: Convert the given non-deterministic FA (NDFA) to a deterministic FA (DFA).

Non-deterministic FA: NDFA



This FA is non-deterministic. At state A the input a issues two next options, back to state A or enter state B. Hard to make the right decision when you write a program for this FA.

Following steps on the right column to remove this problem. Means convert this non-deterministic FA to a new deterministic FA so that the new machine accepts all words that were accepted by the original machine.

Deterministic FA: DFA

Step 1. Construct the following table

states	Input a	Input b
{A}	{A , B} At state A with input a you have a choice to enter state A or state B. A→aA means from state A input a takes you back to A. A→aB, means from state A input a takes you to B. We write it as set {A, B}	{ } At state A, input b is not declared, we use an empty set for no destination.
{B}	{ } At state B, input a is not declared	{B} B→bB, means at state B input b takes you back to state B
A new state {A,B} is created, lets find the next state using inputs a and b at this state. Since {A,B}={A}U{B}, so the next state using input a is the same as the next state using a as input at {A} union with the next state using input a at {B}		
{A,B}	{A,B}={A}U{B}, input a ={A,B}U{ } ={A,B}	{A,B}={A}U{B}, input b ={ }U{B} ={B}
There are no new states, the table is complete New FA states are the first column of the table: {A},{B}, {A,B}		

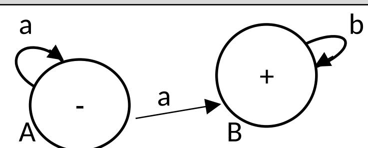
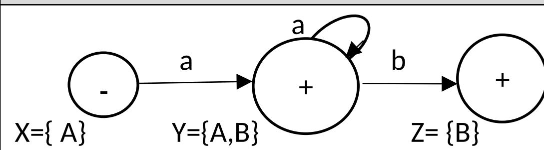
Redraw the above table without explanations to use it to construct the deterministic FA

states	Input a	Input b
{A}	{A,B}	{ }
{B}	{ }	{B}
{A,B}	{A,B}U{ }={A,B}	{ }U{B}= {B}

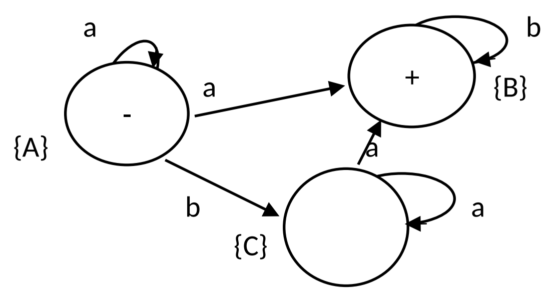
Step 2. Now we use the table to construct a new FA

	which is the deterministic form of the original FA i. The initial state remains the same: {A} a a b { A} {A,B} {B} ii.To decide which states are final states, go back to the original FA. Since the B was a final state in the original FA, the final states of this new FA are any state whose name contains label B. Hence, {A,B} and {B} are final states of this new FA. This machine has the same language as the original FA, and because it is deterministic, its easy to write a program to accepted or rejected any word.

Note that while we convert NDFA to DFA, their grammar also changed from non-deterministic CFG (NDCFG) to a deterministic CFG (DCFG)

NDFA and its NDCFG	DFA and its DCFG
 NDCFG: A→aA |aB, B→bB | λ	 For simplicity, rename the states DCFG: X→aY, Y→aY |bZ |λ, B→λ

Example. Convert the following NDFA to a DFA. At the same time show how the Grammar will change to a DCFG



i. initial state:{A}
ii. Final state: {B}

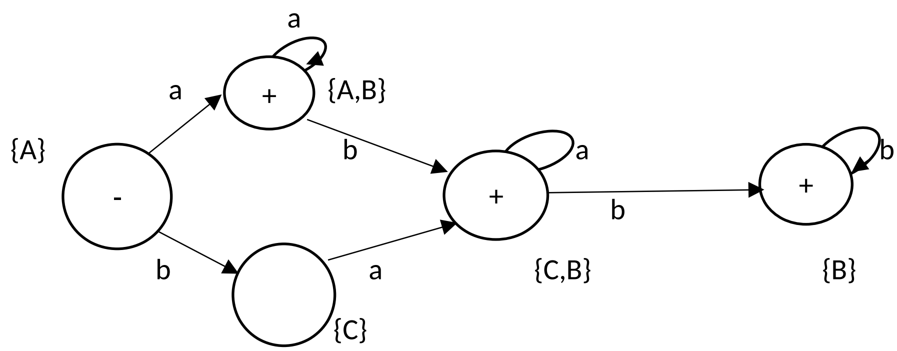
Step 1. Construct the transition table

States	Input a	Input b
{A}	{A,B}	{ C }
{B}	{ }	{B}
{C}	{ C, B }	{ }
New states: {A,B} and {C,B}, add them to the table		
{A,B}	{A,B}={A}U{B} = {A,B}U{ } = {A,B}	{A,B}={A}U{B}={c}U{B}={C,B}

{C,B}	{C,B}={C}U{B}={C,B}U{ } = {C,B}	{C,B}={C}U{B}={ }U{B}={B}
No newer state. States of the new machine are {A}, {B}, {C}, {A,B}, {C,B}		

Step 2: Use the table to construct DFA

In the DFA, Initial state={A} and Final states are states whose B is a member of their name: {A, B}, {B}, and {C,B}, mark all three states as final state. Use the summary table to construct DFA.

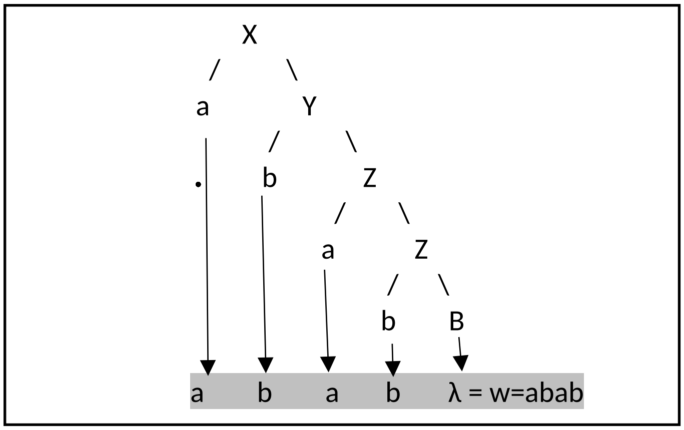


	a	b
{A}	{A,B}	{C}
{B}	{ }	{B}
{C}	{C,B}	{ }
{A,B }	{A,B}	{C,B}
{C,B}	{C,B}	{B}

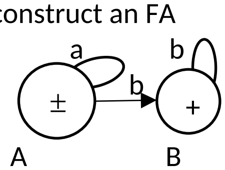
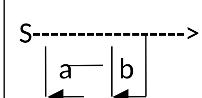
This is the summary table equivalent to the above table

To write its deterministic CFG, let X= {A}, Y={A, B}, Z={C,B}, D ={C}, and E={B}, then

$X \rightarrow aY \mid bD$
$Y \rightarrow aY \mid bZ \mid \lambda$
$D \rightarrow aZ$
$Z \rightarrow aZ \mid bB \mid \lambda$
$B \rightarrow bB \mid \lambda$
Use parsing tree to trace the word w=abab



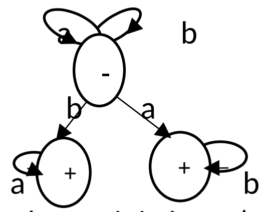
Handout summary

Languag e	Reg.CFG	NonReg-CFG	EBNF	Syntax Diagram	BNF
L=a*b*	To find L's regular CFG, we must construct an FA  Regular CFG $A \rightarrow aA \mid bB \mid \lambda$ $B \rightarrow bB \mid \lambda$	Directly from L to grammar L=a*b* $S \rightarrow AB$ $A \rightarrow aA \mid \lambda$ $B \rightarrow bB \mid \lambda$	Directly from L to grammar L=a*b* $S \rightarrow \{a\}\{b\}$	Directly from L to grammar L=a*b* 	From FA to grammar (one rule per line) $A \rightarrow aA$ $A \rightarrow bB$ $B \rightarrow bB$ $A \rightarrow \lambda$ $B \rightarrow \lambda$

Compiler Design **Names ……………………………………………row…………….**

Assignment No. 3 (70 points.) Different forms of CFG, Regular and non-regular languages, regular and non-regular CFG, Deterministic and non-deterministic FAs, converting NDFA to DFA)

1.(10 points) Given the language L= (a + b)*(ba* + ab*)



a. Construct an FA for L (FA is already given)
b. Convert the non-deterministic FA (NDFA) to a deterministic FA (DFA)
c. Use the DFA to write a deterministic CFG for L

2.(10 points) Given the following non-deterministic CFG :

S → aA | aB | bB | λ
B→b B | λ
A→aA |aB

Convert the grammar to a deterministic CFG (Hint use the CFG to construct a non-deterministic FA, convert NDFA to DFA, and then write a new DCFG)

3.(10 points) Write a regular and non-regular CFG for languages: (i) L= a*b (a+b)* (ii) L=a*b*

4.(20 points) Given the following EBNF grammars. (i) draw their syntax diagram (ii)write each grammar in form of BNF

a. S→ [a] { b } d
b. S→ {a|b} {c}
c. S→ {a} {b} [c] {d}

K-mart
23andMe
456
Tax 2018
While
switch
do_it
_Fall_20
_Jan 19

<u>Programming (10 points each)</u>

1. Write a program to read one token at a time from the given text file and determine whether the token is
   i. A number
   ii. An identifier (must start with underscore or a letter, followed by more letters, more digits, or more underscores
   iii. A reserved word. List of reserved words: string reserved[5]={“while”, “for”, “switch”, “do”, “return” };

<u>Sample output for #1</u>

Token	number	identifier	reserved word
K-mart	no	no	no
23andMe	no	no	no
456	yes	no	no
……….			

2. Given CFG: **S→aS |bB|cC**
   B→bB | aC |λ
   C→aS |λ

   Write **<u>a program</u>** to determine whether an input string is accepted or rejected by the grammar. Hint, first you have to complete the FA

   Try input strings: w1=abbbcaaa$, w2=ccccbbb$, w3=aabbaac$