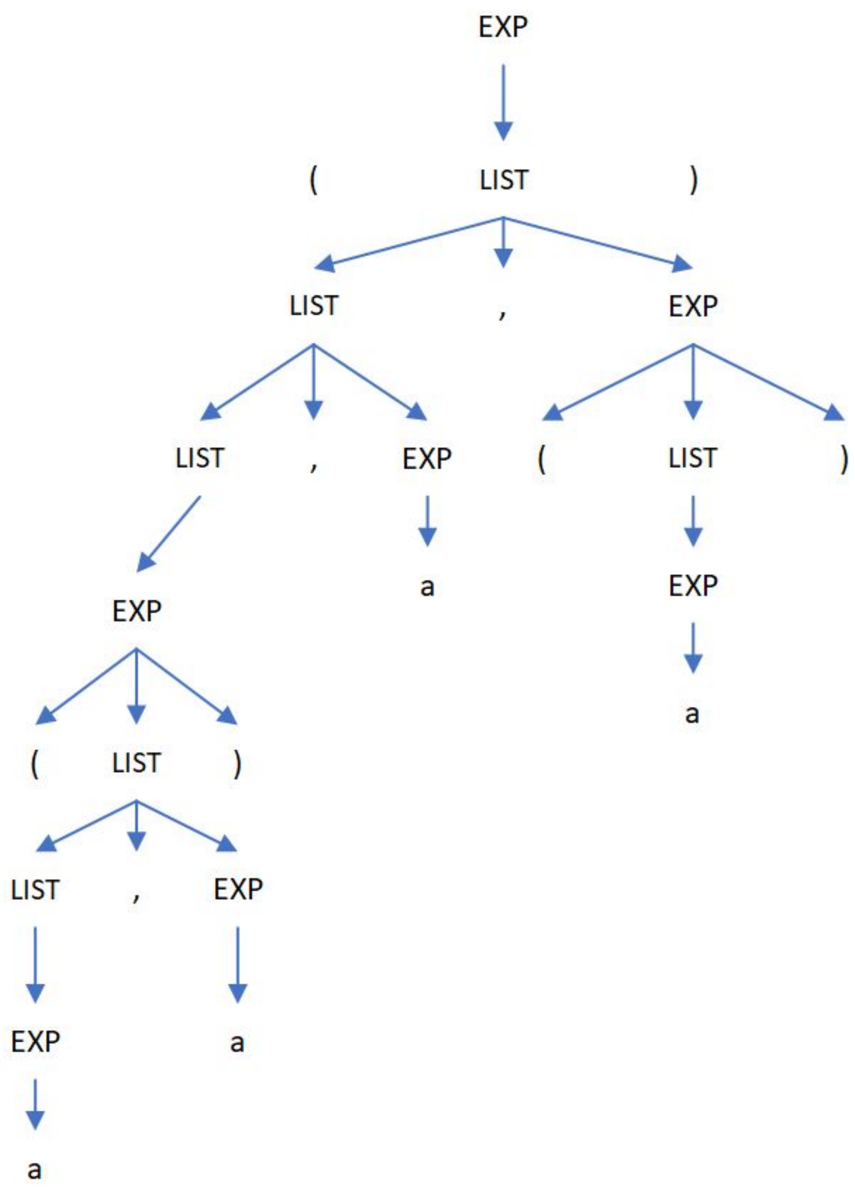
Syntax Description Methods and Recursive Descent Parsing

1a)

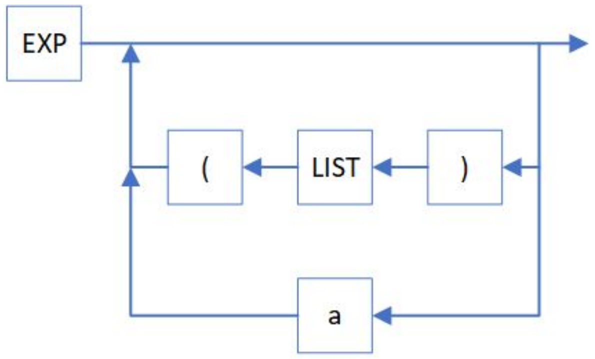


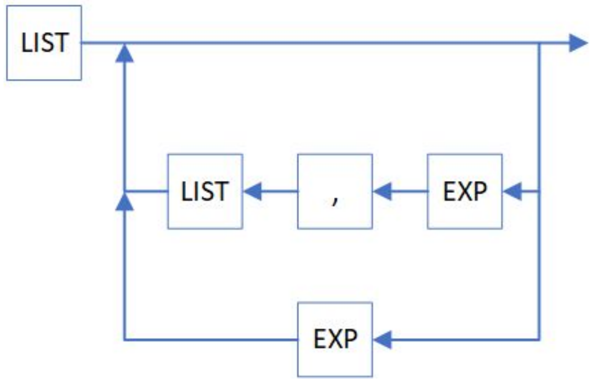
1b) An EBNF that represents the BNF is

EXP ::= (LIST) | a
LIST ::= LIST {, EXP}

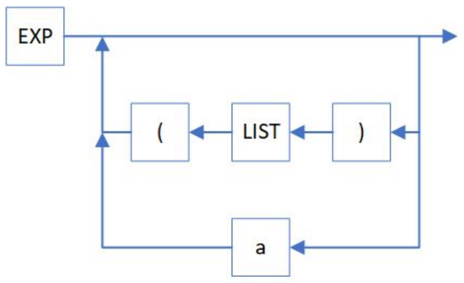
1c)

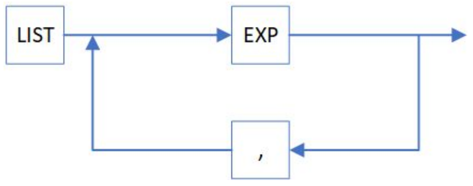
BNF





EBNF





1d)

	First	Follow
EXP	{(a}	{,) $}
LIST	{(a}	{,)}

2a) EBNF

EXP ::= TERM { (+ | -) TERM }

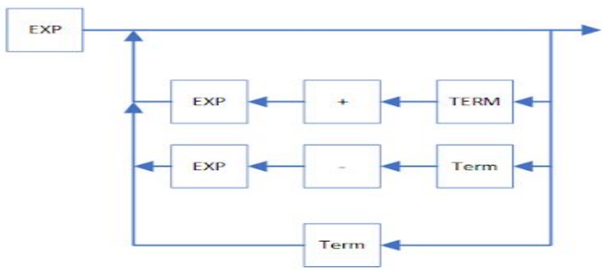
TERM ::= FACTOR { (* | /) FACTOR }

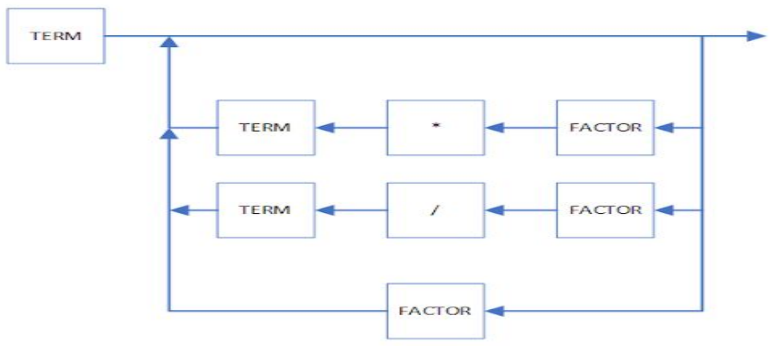
FACTOR ::= (EXP) | DIGIT

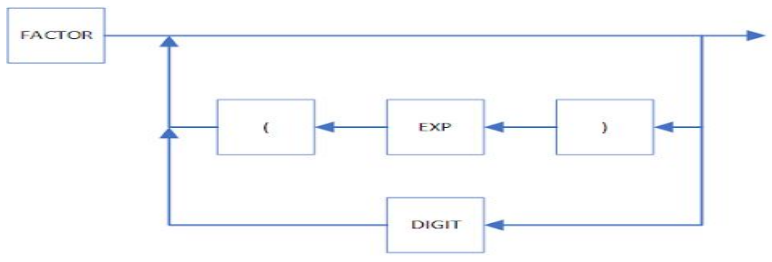
DIGIT ::= 0 | 1 | 2 | 3

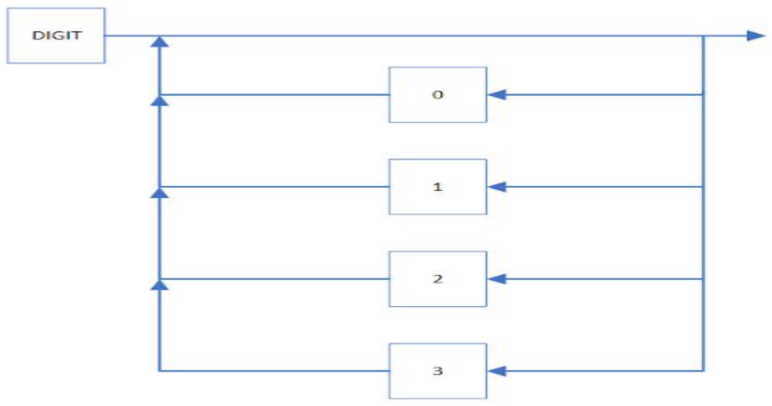
2b)

BNF

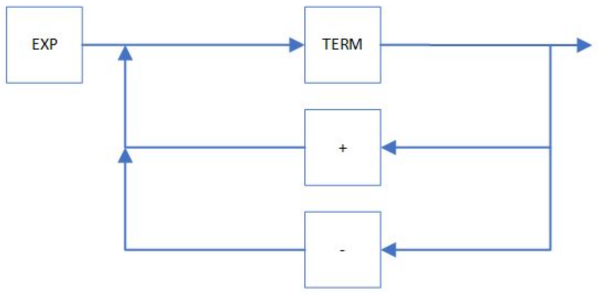

EXP
EXP
+
TERM
EXP
-
Term
Term

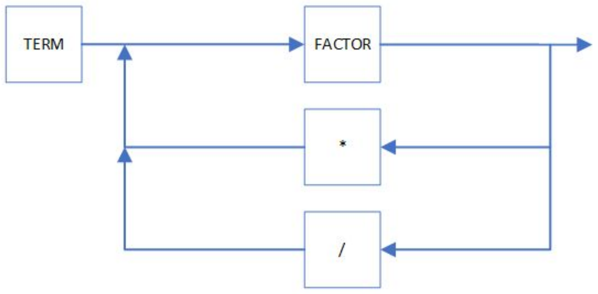

TERM
TERM
*
FACTOR
TERM
/
FACTOR
FACTOR

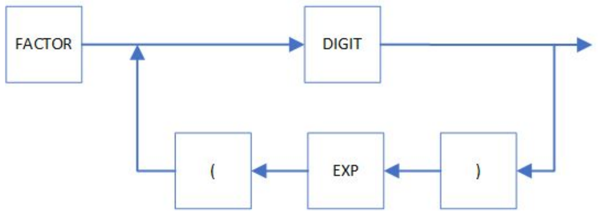

FACTOR
(
EXP
)
DIGIT

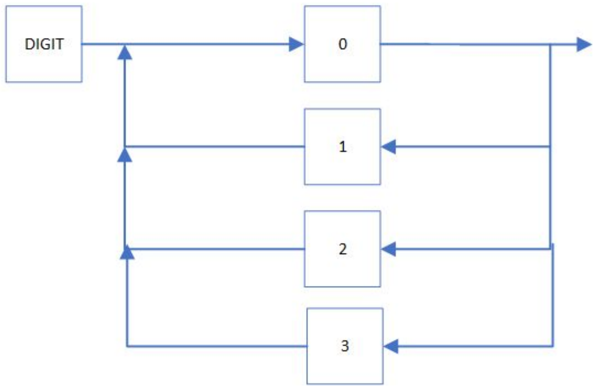

DIGIT
0
1
2
3

EBNF









2c) The two requirements are that it should not be left recursive and it should not be non-deterministic.

2d)

	First	Follow
EXP	{ (0 1 2 3 }	{ + -) $ }
TERM	{ (0 1 2 3 }	{ + - * /) $ }
FACTOR	{ (0 1 2 3 }	{ + - * /) $ }
DIGIT	{ 0 1 2 3 }	{ + - * /) $ }

3) **Recursive Descent Recognizer Pseudocode**

```
define recogniser()    //
{
  if token == "$"     // Assume the input stream terminates with a $.
      Legal           // Report "legal" or "errors found" (not both!).
  else
  {
    r=match(token)
    if r==error
      {
       Illegal string
       break
      }
    else
     check_grammar(token)
  }
}

define match (token t)
{
  if token == t
    advanceToken
  else
return error
}
```