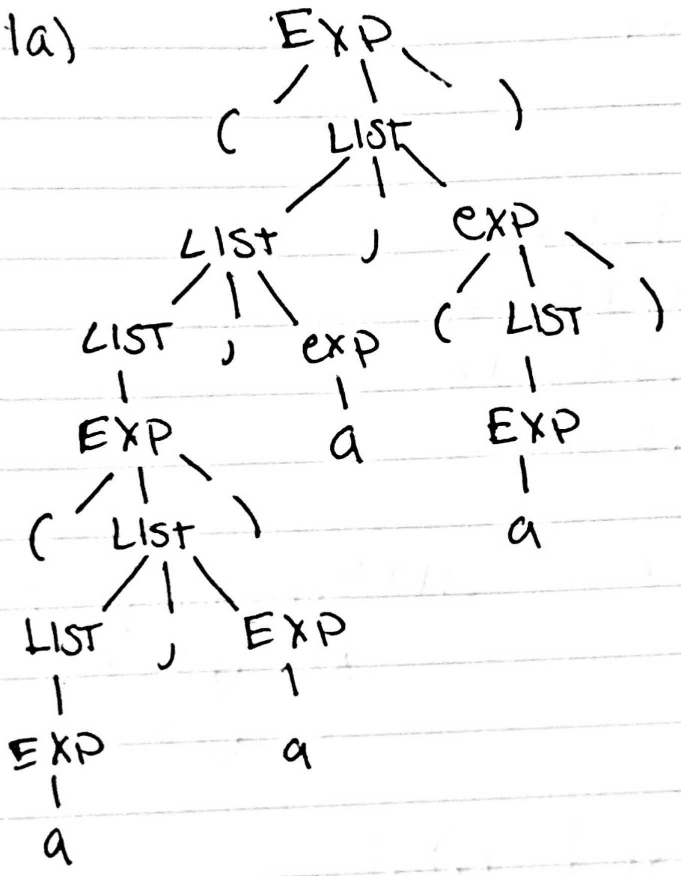
1) EXP ::= (LIST) | a
LIST ::= LIST , EXP | EXP

1a)

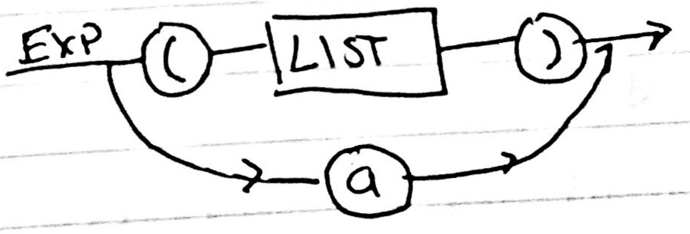


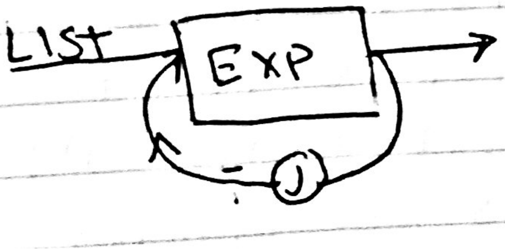
1d) FIRST(EXP) = { a, (}
~~FIRST(LIST) = {) }~~
FIRST(LIST) = FIRST(EXP)
= { a, (}
FOLLOW(LIST) = {) }

FOLLOW(EXP) = FOLLOW(LIST) ∪ { , }
= {), }

1b) EBNF
EXP ::= (LIST) {a}
LIST ::= <exp> { , exp }

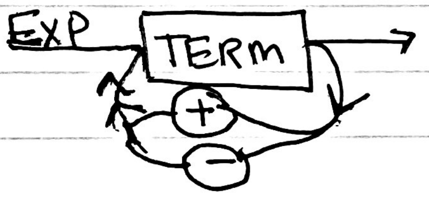
1c)

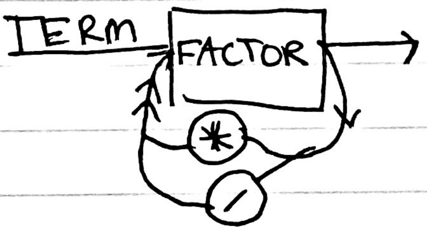


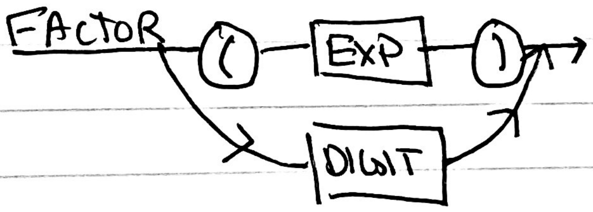


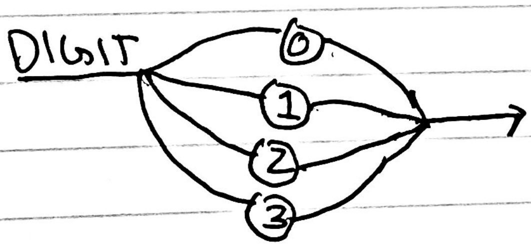
2) EXP::= EXP+TERM | EXP-TERM | TERM
TERM::= TERM *FACTOR | TERM/FACTOR | FACTOR
FACTOR::= (EXP) | DIGIT
DIGIT::= 0|1|2|3

2a) <u>EBNF</u>
EXP::=TERM{(+|-)TERM}
TERM::= FACTOR{(*|/) FACTOR}
FACTOR::= (EXP) | DIGIT
DIGIT::= 0|1|2|3

2b)








2c) First set of any two choices must not have any tokens in common.
ex) FACTOR::=(EXP) | DIGIT
First(EXP) ∩ FIRST(DIGIT) = ∅
• When structures are optional
ex) S → B[A]D
If A is optional, then
FIRST(A) ∩ FOLLOW(A) = ∅

2d) FIRST(DIGIT) = { 0 1 2 3 }

FIRST (FACTOR) = FIRST(EXP) ∪ FIRST(DIGIT)

= { (} ∪ { 0 1 2 3 } = { (0 1 2 3 }

First (TERM) = FIRST(FACTOR) = { (0 1 2 3 }

FIRST(EXP) = FIRST(TERM) = { (0 1 2 3 }

FOLLOW(EXP) = {) }

FOLLOW (TERM) = { + - } ∪ FOLLOW(EXP)

{ + - } ∪ {) } = { + -) }

FOLLOW(FACTOR) = { * / } ∪ FOLLOW(TERM)

= { * / + -) }

FOLLOW(DIGIT) = FOLLOW(FACTOR) = { * / + -) }

2e) FIRST(DIGIT) ∩ FOLLOW(DIGIT) =

{ 0 1 2 3 } ∩ { * / + -) } = ∅

3)
```
inputFunction() {
  inputString = user Input //get expression from use
  token = currentToken //assign pointer to token
}

match(t) {
  If(token = t)
    advanceToken //advance token pointer
  else
    return error
}
exp() { Term() // call non Terminal Term
  If token == "+" or token == "-" {
    match(token) //Check if token is terminal
    Term() //call non terminal Term }
  match($)
}
Term() {
  Factor() // call non terminal Factor function
  If (token == * or token == "/")
    match(token)
    Factor() // call non terminal Factor
  }
  match($) //check for end of string
}
Factor() {
  If (token == "(") {
    match(token)
    exp() // call non terminal exp
  } else If (token == ")")
    match(token)
  } else {
    Digit() // call non terminal Digit.
  }
  match($)
}
Digit() {
  If(token == 0|1|2|3)
    match(token)
  else
    return error
```

This code implements both left and right associative operators. It uses different levels of non-terminals to express operator precedence.

- User Info:
  - Enter string using numbers 0 to 3 and the symbols +, -, *, /, (, and)
  The end string variable will be the dollar sign "$"
  - program checks whether the input string is valid or not.

- Input examples
  - 2/3$ <u>valid.</u>

    1) token = 2
    2) match(2) = true // in Digit function
    3) advance token
    4) token = /
    5) match(/) = true // in term function
    6) advance token
    7) token = 3
    8) match(3) = true // in Digit
    9) next token is $, return true

- Input example

  2/3$ valid

  token = 2

  match(2)=true // @ digit function

  advanceToken; token = /

  match(/) =true // @ term function

  advance Token; token= 3

  match(3) = true // @ digit function

  advance Token; Token = $

  End of input string.

- Input example

  22-3$ //invalid token

  match(22) = False

  Invalid String