Aula 5 - Implementação incial de um reconhecedor sintático

Para esta entrega, pretende-se descrever um reconhecedor sintático simples e simulá-lo de forma a reconhecer os tokens consumidos e os estados atuais da máquina responsável pelo reconhecimento em sí.

Foi feita uma adaptação da gramática apresentada no exercício, para que se adeque as notações da linguagem Basic utilizada no projeto desta matéria.

Gramática Adaptada

A seguir apresenta-se a gramática implementada:

```
SINTÁTICA:

Program : BStatement BStatement*

BStatement : INTEGER Assign

Assign : LET Var = Exp

Var : ID

Exp: Term (PLUS|MINUS Term)*

Term: Eb ((MUL | DIV) Eb)*

Eb: LPAREN Exp RPAREN | INTEGER | Var

Remark : REM (CHARACTER)*

LÉXICA:
ID: letter(digit|letter)*
INTEGER: digit(digit)*

TERMINAIS:

LET, PLUS, MINUS, MUL, DIV, LPAREN, RPAREN
```

Nota-se aqui que a regra Factor foi substituída por Eb (seguindo a notação da gramática BASIC apresentada), além da presença de BStatement e Remark (comentários).

Adaptação do enuciado para Basic

Neste momento do curso, já tenho implementado e funcional todo o compilador (inclusive geração de código) para a linguagem mostrada anteriormente. Sendo assim, efetuarei algumas mudanças no código fonte apresentado para mostrar o funcionamento já com a linguagem BASIC:

```
ORIGINAL:

LET a = a / 2 + (b - 3) * (3 + b);
LET a = (((b)) * (2 - c));
LET a = b + ((2 * c) / ( d - e) + 1)
```

```
MODIFICADO:

10 LET a = 4
20 LET b = 5
30 LET c = 1
40 LET d = 3
50 LET e = 2

101 REM resultado16
100 LET f = a/2 + (b-3)*(b+2)

109 REM resultado5
110 LET g = (((b)) * (2 - c))

119 REM resultado8
120 LET h = b + ((1+1) + 1)

129 REM resultado29
130 LET i = f + g + h
```

Basicamente as alterações feitas foram para inicializar as variáveis utilizadas com valores que permitisse a execução das contas apenas com inteiros positivos.

Assembly Gerado

O código assembly gerado é o seguinte:

```
    .data

.comm   _a, 4, 4
```

```
.comm   _b, 4, 4
.comm   _c, 4, 4
.comm   _d, 4, 4
.comm   _e, 4, 4
.comm   _f, 4, 4
.comm   _g, 4, 4
.comm   _h, 4, 4
.comm   _i, 4, 4


.text

    .globl  _main
_main:

    movl    $0, %eax
    movl    $4 ,%edx
    movl    %edx, _a

    movl    $0, %eax
    movl    $5 ,%edx
    movl    %edx, _b

    movl    $0, %eax
    movl    $1 ,%edx
    movl    %edx, _c

    movl    $0, %eax
    movl    $3 ,%edx
    movl    %edx, _d

    movl    $0, %eax
    movl    $2 ,%edx
    movl    %edx, _e

    movl    $0, %eax
    movl    _b, %eax
    movl    $3,%edx
    subl    %edx, %eax

    movl    %eax, %ecx
    movl    _b, %eax
    movl    $2,%edx
    addl    %edx, %eax

    movl    %eax, %edx
```

```
imul    %ecx, %edx
movl    %edx, %eax

movl    %eax, %ebx
movl    _a, %eax
movl    $2, %ecx
movl    $0, %edx
idiv    %ecx

addl    %ebx, %eax
movl    %eax, _f

movl    $0, %eax
movl    _b, %ecx
movl    $2, %eax
movl    _c,%edx
subl    %edx, %eax

movl    %eax, %edx
imul    %ecx, %edx
movl    %edx, %eax
movl    %eax, _g

movl    $0, %eax
movl    $1, %eax
movl    $1,%edx
addl    %edx, %eax

addl    $1, %eax

movl    _b, %edx
addl    %edx, %eax
movl    %eax, _h

movl    $0, %eax
movl    _f, %eax
movl    _g,%edx
addl    %edx, %eax

addl    _h, %eax
movl    %eax, _i

ret
```

Este programa assembly, quando montado utilizando o gcc, pode ser executado e retorna o valor 29, como esperado pela soma das três variaveis f, g, h.

Descrição em código do analisador semantico

Descreve-se aqui a representação dos automatos reconhecedores da linguagem BASIC implementados como um parser em linguagem Python.

Cada regra de formação é transformada em um método. O método "eat" consome um token de tipo específico a partir dos tokens gerados pelo analisador léxico (retorna erro caso não seja um Token do tipo esperado).

```
    def Program(self):
        """
         Program : BStatement BStatement*
        """
        node = self.BStatement()

        nodes = [node]

        while self.current_token.type == INTEGER:
            nodes.append(self.BStatement())

        root = StatementList()
        for node in nodes:
            root.children.append(node)

        return root

    def BStatement(self):
        """
            BStatement : INTEGER Assign | Remark
        """
        node = None
        self.eat(INTEGER)
        if self.current_token.type == LET:
            node = self.Assign()
        elif self.current_token.type == REM:
            node = self.Remark()
        return node

    def Assign(self):
        """
            Assign : LET Var EQUAL Exp
        """
        self.eat(LET)

        left = self.Var()
```

```
        token = self.current_token
        self.eat(EQUAL)

        right = self.Exp()

        node = Assign(left, token, right)
        return node

    def Var(self):
        """
            Var : ID
        """
        node = Var(self.current_token)
        self.eat(ID)
        return node

    def Exp(self):
        """
            Exp: Term (PLUS|MINUS Term)*
        """
        node = self.Term()
        while self.current_token.type in (PLUS, MINUS):
            token = self.current_token
            if token.type == PLUS:
                self.eat(PLUS)
            else:
                if token.type == MINUS:
                    self.eat(MINUS)
            node = BinOp(left=node, op=token, right=(self.Term()))

        return node

    def Term(self):
        """
            Term: Eb ((MUL | DIV) Eb)*
        """
        node = self.Eb()
        while self.current_token.type in (MUL, DIV):
            token = self.current_token
            if token.type == MUL:
                self.eat(MUL)
            else:
                if token.type == DIV:
                    self.eat(DIV)
            node = BinOp(left=node, op=token, right=(self.Eb()))
```

```
        return node

    def Eb(self):
        """
            Eb : PLUS Eb
                  | MINUS Eb
                  | INTEGER
                  | LPAREN Exp RPAREN
                  | Var
        """
        token = self.current_token
        if token.type == PLUS:
            self.eat(PLUS)
            node = UnaryOp(token, self.Eb())
            return node
        if token.type == MINUS:
            self.eat(MINUS)
            node = UnaryOp(token, self.Eb())
            return node
        if token.type == INTEGER:
            self.eat(INTEGER)
            return Num(token)
        if token.type == LPAREN:
            self.eat(LPAREN)
            node = self.Exp()
            self.eat(RPAREN)
            return node
        node = self.Var()
        return node

    def Remark(self):
        self.eat(REM)
        self.eat(ID) # sequencia de caracteres
        return self.empty()
```

Verificação dos estados e transições do reconhecedor

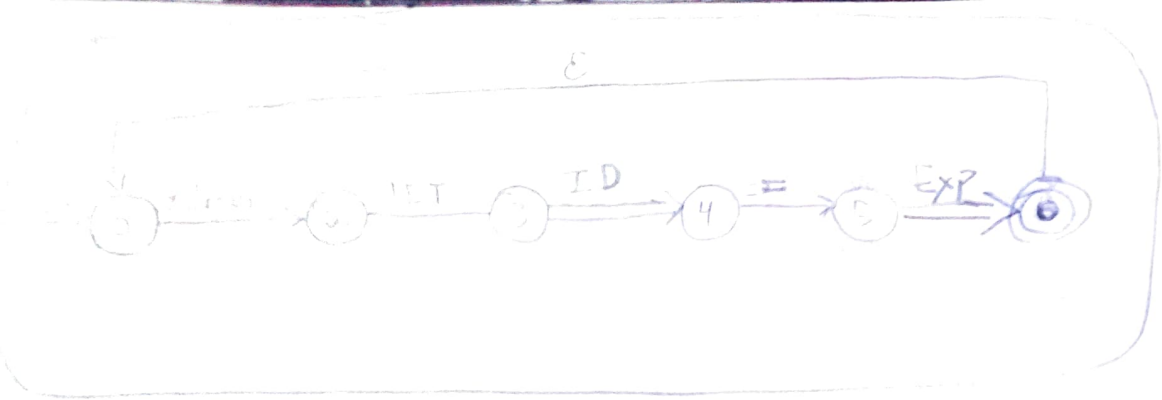
A seguir mostra-se a sequencia de regras de formação (entre parêntesis) e tokens reconhecidos para analisar a formação das 3 linhas de código descritas no enunciado, sendo aqui representadas pelas variáveis 'f', 'g' e 'h'.

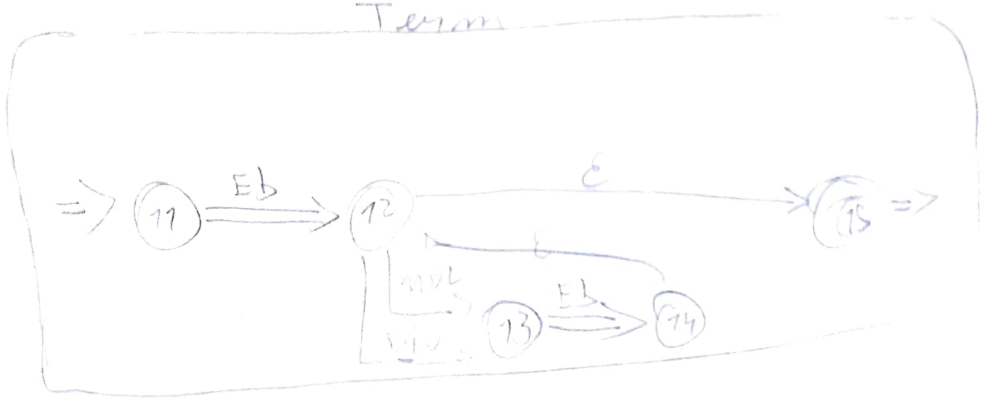
```
100 LET f = a/2 + (b-3)*(b+2)
```

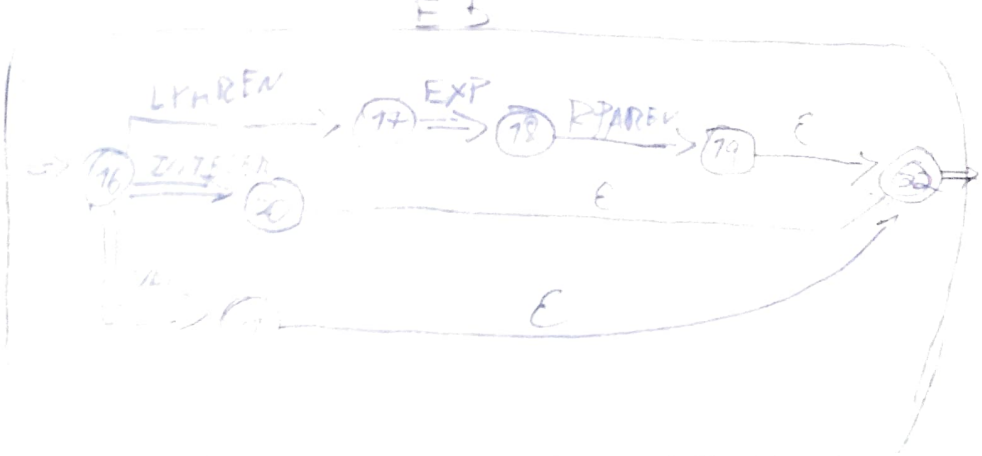
```
(BStatement) -> INTEGER -> (Assign) -> LET -> (Var) -> ID -> -> EQUAL->
```

```
(Exp)-> (Term) -> (Eb) -> (Var) -> ID-> -> DIV -> (Eb) -> INTEGER ->
-> PLUS -> (Term) -> (Eb) -> LPAREN -> (Exp) -> (Term) -> (Eb) -> (Var) ->
ID -> -> MINUS -> (Term) (Eb) -> INTEGER -> -> RPAREN -> -> MUL -> (Eb) ->
LPAREN -> (Exp) -> (Term) -> (Eb) -> (Var) -> ID -> -> PLUS -> (Term) ->
(Eb) -> INTEGER -> -> RPAREN


110 LET g = (((b)) * (2 - c))

(BStatement) ->INTEGER-> (Assign) -> LET -> (Var) -> ID -> -> EQUAL ->
(Exp) -> (Term) -> (Eb) -> LPAREN -> (Exp) -> (Term) -> (Eb) -> LPAREN ->
(Exp) -> (Term) ->(Eb) -> -> LPAREN -> (Exp) -> (Term) -> (Eb) -> (Var) ->
ID -> -> RPAREN -> -> RPAREN -> -> MUL -> (Eb) -> LPAREN -> (Exp) -> (Term) ->
(Eb) -> INTEGER -> -> MINUS -> (Term) -> (Eb) -> (Var) -> ID -> -> RPAREN -> -> RPAREN

120 LET h = b + ((1+1) + 1)

(BStatement) ->INTEGER-> (Assign) -> LET -> (Var) -> ID -> -> EQUAL ->
(Exp) -> (Term) -> (Eb) -> (Var) -> ID -> -> PLUS -> (Term) -> (Eb) ->
LPAREN -> (Exp) -> (Term) -> (Eb) -> LPAREN -> (Exp) -> (Term) -> (Eb) ->
INTEGER -> -> PLUS -> (Term) -> (Eb) -> INTEGER -> -> RPAREN -> -> PLUS ->
(Term) -> (Eb) -> INTEGER -> -> RPAREN
```

Representação escrita do autômato reconhecedor e Log de Estados e Transições







LET a = a/2 + (b-3) * (b-2); (6) → INTEGER → (12) LET → (5) "a" → (4) = → (5) → (6) → (11) →

(16) VAR → (21) → (22) → (12) DIV → (13) INTEGER → (14) → (15) → (16) → (17) → (18) →

VAR → (9) → (11) → (16) LPAREN → (17) → (6) → (11) → (16) VAR → (21) → (22) → (16) →

(7) MINUS → (8) → (11) → (16) INTEGER → (20) → (22) → (11) → (12) → (13) INTEGER → (14) →

(22) → (12) MUL → (13) → (16) LPAREN → (17) → (6) → (11) → (16) INTEGER → (22) →

(22) → (12) → (7) MINUS → (8) → (11) → (16) INTEGER → (22) →

(21) → (9) → (7) → (10) → (0)