UNIVERSIDADE FEDERAL DE UBERLÂNDIA
CURSO DE BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

CONSTRUÇÃO DE COMPILADORES
PROJETO DA DISCIPLINA

Arthur do Prado Labaki
11821BCC017

Uberlândia - MG
2022

Sumário

1 Descrição do Projeto

O projeto da disciplina consiste no desenvolvimento das etapas de um *front-end* de um compilador para a linguagem simplificada demonstrada mais abaixo. Todas as imagens, códigos, diagramas e qualquer outros arquivos utilizados neste trabalho estão inseridos em meu repositório publico no GitHub, o link de acesso está disponível abaixo.
Repositório do projeto

Resumindo, um compilador transforma um programa executável de uma linguagem fonte para um programa executável em uma linguagem destino. Seu *front-end* é formado pelas etapas: análise léxica, análise sintática e análise semântica. A função de todo o *front-end* é de extrair a estrutura do programa, e verificar se não existe erros, produz mensagens apontando onde e quais são os erros, além de produzir uma representação do programa como uma árvore sintática abstrata .

Cada capítulo deste relatório se trata de uma etapa da criação do projeto completo, totalizando quatro etapas. Na primeira etapa, será feito as especificações da linguagem, como a criação da gramática livre de contexto, como também a identificação dos *tokens* e suas respectivas expressões regulares. Já na segunda parte, será feito a análise léxica, especificando o diagrama de transição e a implementação manual do analisador léxico.

Na terceira parte, será realizada a análise sintática, fazendo ajustes na gramática, criando grafos e também implementando o analisador sintático preditivo. Por fim, na quarta e última etapa, será realizado a tradução dirigida por sintaxe, efetuando a análise semântica, gerando o código intermediário, além de realizar algumas etapas programadas, tudo disponível no slide do projeto.

No projeto, o foco será no *front-end* do compilador, não implementando as etapas de otimização e síntese, além de adotar alguns *back-end* disponíveis no ambiente de compilação. Será usado o ambiente de compilação *LLVM*, sendo que deverá ser gerado uma representação intermediaria (IR) compatível com esse ambiente.

Especificando ainda mais, o analisador léxico deve ser uma sub rotina chamada pelo analisador sintático, em que deve retornar o próximo *token* da cadeia de entrada. Ainda, todas as fases do *front-end*, com exceção da tradução dirigida por sintaxe, serão implementadas em um único módulo. Por fim, a semântica será estática, sendo apenas analises em tempo de compilação.

Especificação da linguagem para o projeto:

```
# Estrutura principal:
    programa nome_programa bloco

# Bloco:
    begin declaracao_das_variaveis sequencia_de_comandos end

# Declaração de variáveis:
    tipo: lista_ids ;
    Os tipos serão: int, char e float

# Comando de atribuição:
    id:= expressao ;

# Comentários:
    [ texto_comentario ]

# Comando de seleção:
    if(cond) then bloco else bloco

# Comandos de repetição:
    while(cond) do bloco
    repeat bloco while(cond)

# Condições:
    Igual (=), diferente (~=), menor (<), maior (>),
    menor ou igual (<=), maior ou igual (>=)

# Expressões:
    Soma (+), subtração (-), multiplicação (*),
    divisão (/) e exponenciação (^)

# Observações
    char deve estar entre apostrofo
    int deve estar entre -32768 e +32767
    float pode ser ponto fixo ou notação cientifica
    parenteses para priorizar operações
```

2 Projeto da Linguagem

Nesta etapa, é feito a especificação da linguagem. Para isso, primeiro é necessário definir a gramatica livre de contexto (GLC), com as estruturas da linguagem especificada, além de identificar os *tokens* utilizados nela e definir os padrões para os lexemas aceitos em cada *token*.

2.1 Gramática Livre de Contexto

A Gramática Livre de Contexto é representada por uma quadrupla, e ela especifica regras precisas que descrevem a estrutura sintática e hierárquica das construções de linguagens de programação. A Gramática Livre de Contexto do nosso projeto pode ser representada da forma:

G=({EP, Bloco, Dvar, Scom, Var, IDs, Com, Cond, Expr, Term}, {programa, ID, begin, end, tipo, : , ; , ’,’, (,) , := , if, then, else, while, do, repeat, relop, +, -, *, /, ^, num, ϵ}, P, EP), sendo:

P = {
EP -> **programa** ID Bloco
Bloco -> **begin** DVar SCom **end**
DVar -> Var | Var DVar | ϵ
SCom -> Com | Com SCom | ϵ
Var-> **tipo :** IDs **;**
IDs -> ID | ID , IDs
Com -> **if** (Cond) **then** Bloco **else** Bloco | ID **:=** Expr **;** |
| **while** (Cond) **do** Bloco | **repeat** Bloco **while** (Cond)
Cond -> Expr **relop** Expr
Expr -> Expr Ariop Term | Term
Ariop -> + | - | * | / | ^
Term -> num | ID | (Expr) }

2.2 Identificação dos *Tokens*

Após a criação da Gramática Livre de Contexto, será realizado a identificação dos *tokens* descritos. Para realizar isso, é oportuno utilizar uma tabela para registrar todos os *tokens* identificados, com os seus respectivos lexemas e valores.

Lexemas	Nome do Token	Valor do atributo
Qualquer número	num	Posição na tabela de símbolos
Qualquer ID	ID	Posição na tabela de símbolos
int	tipo	INT
char	tipo	CHAR
float	tipo	FLOAT
programa	programa	-
begin	begin	-
end	end	-
if	if	-
then	then	-
else	else	-
while	while	-
do	do	-
repeat	repeat	-
:	:	-
;	;	-
(	(	-
)	)	-
,	,	-
:=	:=	ATR
=	relop	EQ
~=	relop	NE
<	relop	LT
>	relop	GT
<=	relop	LE
>=	relop	GE
+	+	ADD
-	-	SUB
*	*	MUL
/	/	DIV
^	^	EXP
Qualquer ws	-	-
Qualquer comentário	-	-

Tabela 1 – Tabela dos Tokens

2.3 Expressão Regular para os Lexemas

Para a conclusão dessa etapa do projeto, deve-se definir os padrões, utilizando expressão regular, para os lexemas aceitos em cada *token* encontrado nas seções anteriores.

Com isso, temos:

```
digito     -> [ 0 - 9 ]
digitos    -> digito (digito)*
numero     -> digitos ( . digitos ) ? (E[ + - ] ? digitos) ?
letra    -> [ A - Za - z ]
letra_    -> [ A - Za - z_' ]
ID         -> letra_ ( letra | digito )*
tipo       -> int | char | float
programa   -> programa
begin      -> begin
end        -> end
if         -> if
then       -> then
else       -> else
while      -> while
do         -> do
repeat     -> repeat
:          -> :
;          -> ;
(          -> (
)          -> )
,          -> ,
:=         -> :=
relop      -> =  | ~= | < | > | <= | >=
+          -> +
-          -> -
*          -> *
/          -> /
^          -> ^
ws         -> (" " | \n | \t )
comentario -> "[" [^ "]" ] "]"
```

3 Análise Léxica

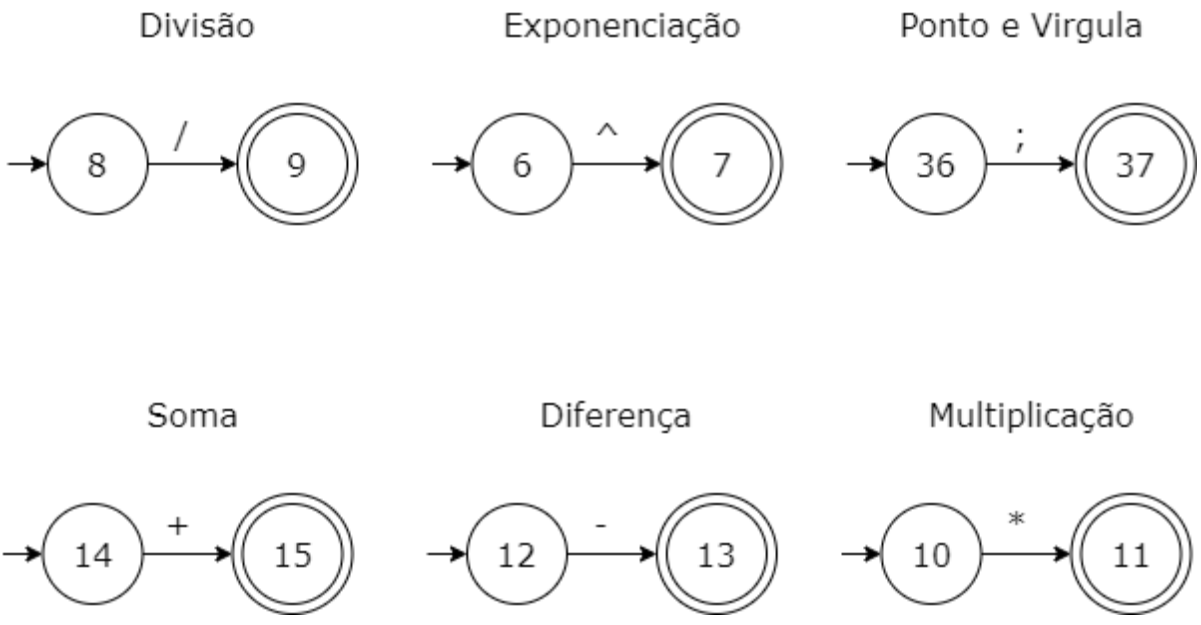
Para cumprir os requisitos e conseguir desenvolver a análise léxica do projeto, além de facilitar o entendimento dele, esse capítulo será dividido em duas seções. A primeira se trata de especificar os diagramas de transição, e a segunda parte a implementação manual do analisador léxico.

3.1 Diagramas de Transição

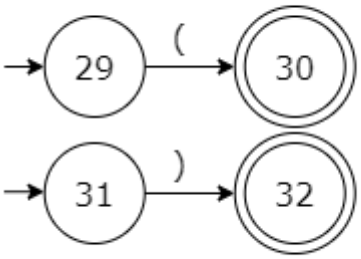
O diagrama de transição de estados é uma representação do estado ou situação em que um objeto pode se encontrar no decorrer da execução de processos de um sistema. A criação dos diagramas de transição será desenvolvida passo a passo, criando diagramas individuais para os *tokens*, realizando a junção de todos em um grande automato finito não determinístico e, por fim, realizando a conversão em um automato determinístico, sendo autômatos no formato de diagramas de transição.

3.1.1 Diagramas de transições individuais

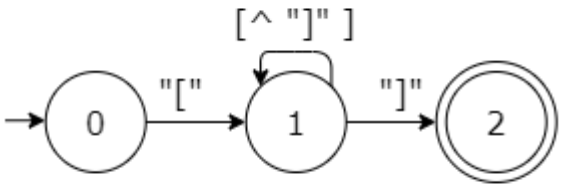
Primeiramente, será criado os diagramas de transição para todos os *tokens* identificados na seção anterior, sendo eles *programa, begin, end, ID, num, +, -, relop,* entre outros. Assim temos:



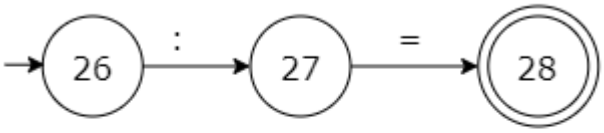
Parênteses



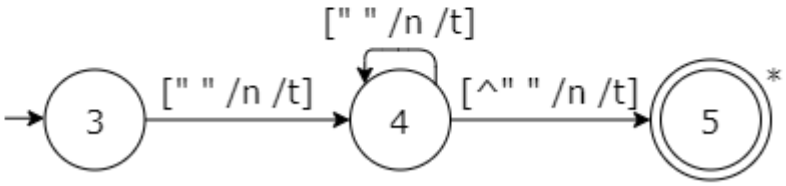
Comentário



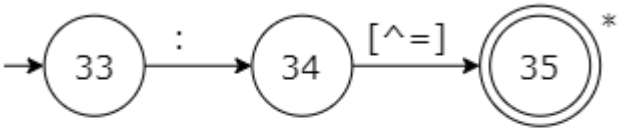
Atribuição



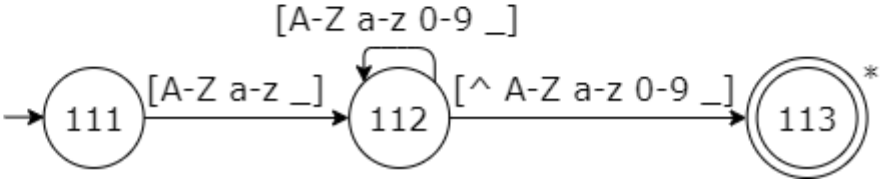
Separadores ws



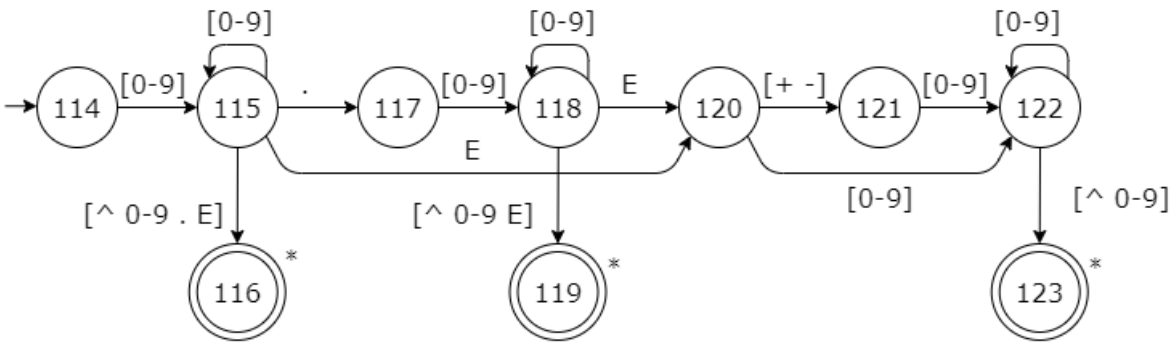
Dois pontos



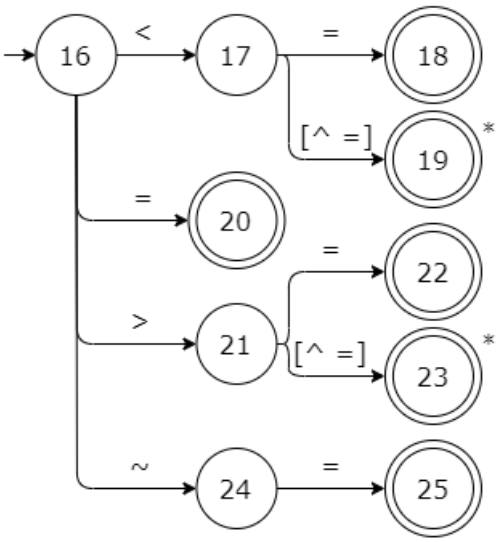
Identificador ID



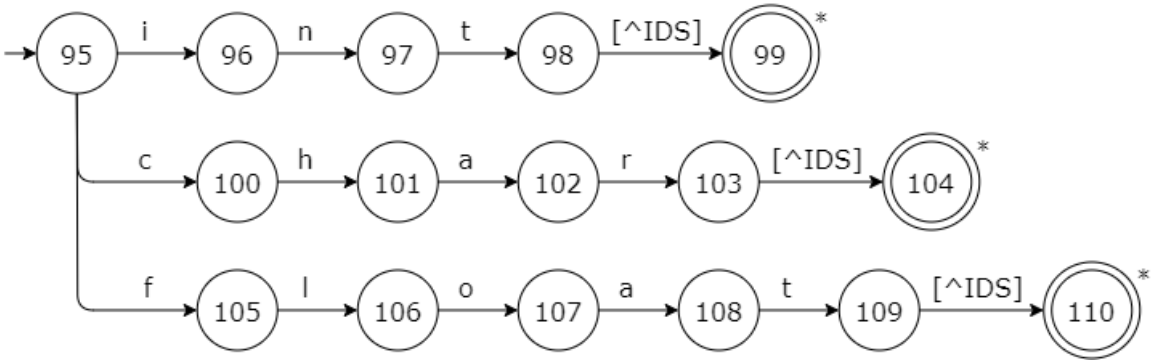
Números Num



Relop

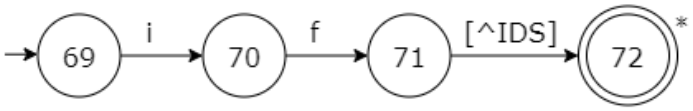


Tipo

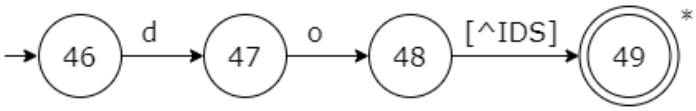


IDS = [A-Z a-z 0-9 _]
(número, letra ou _)

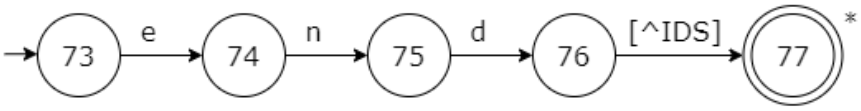
If



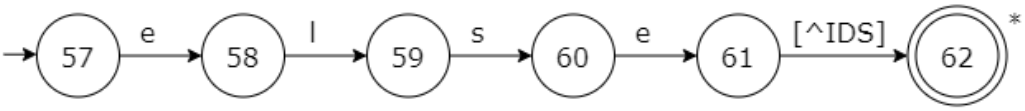
Do



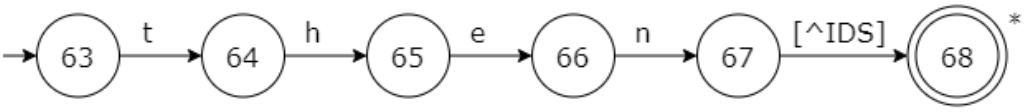
End



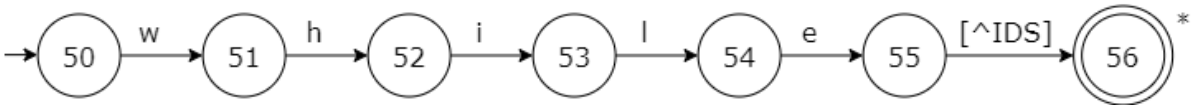
Else

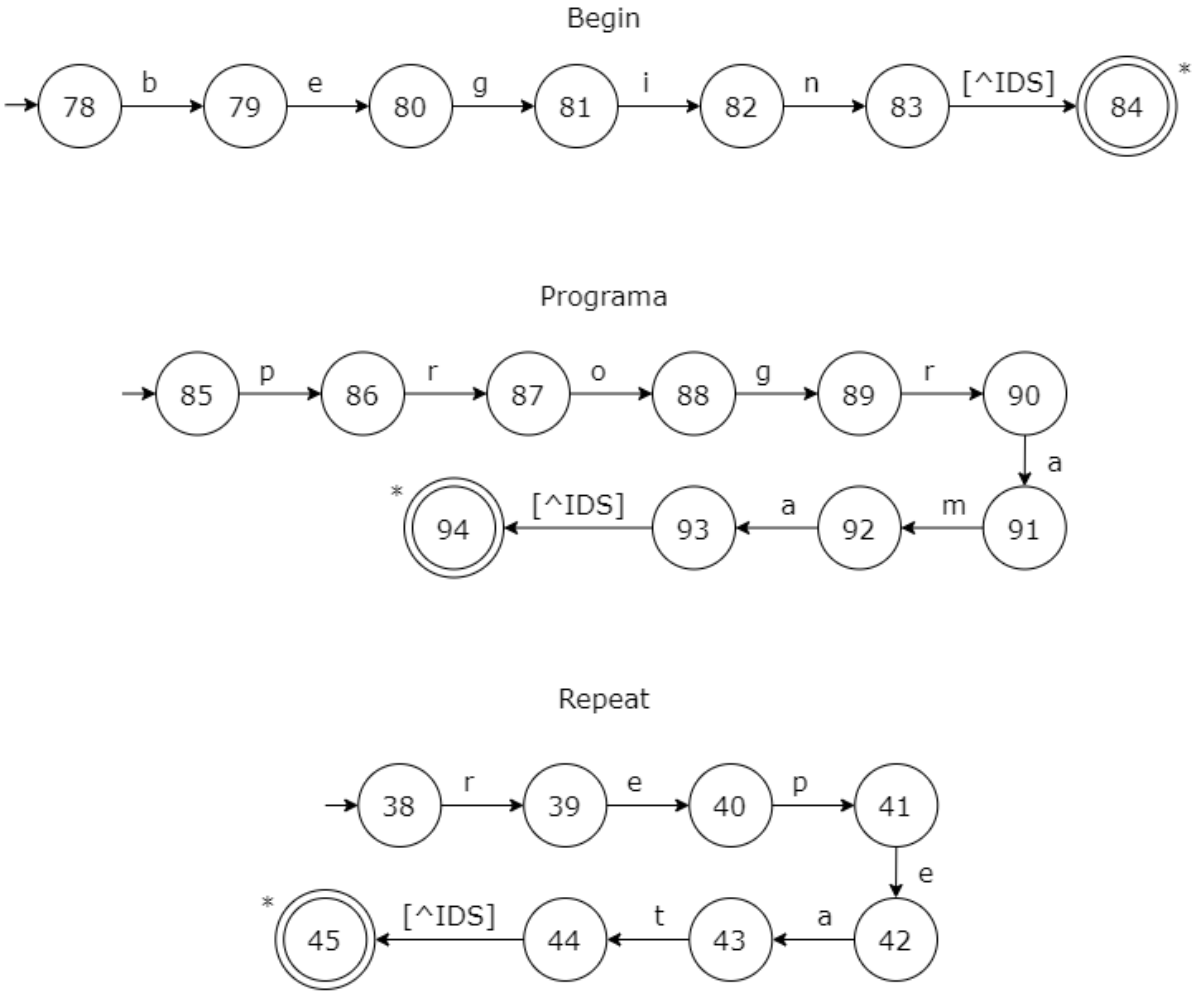


Then



While

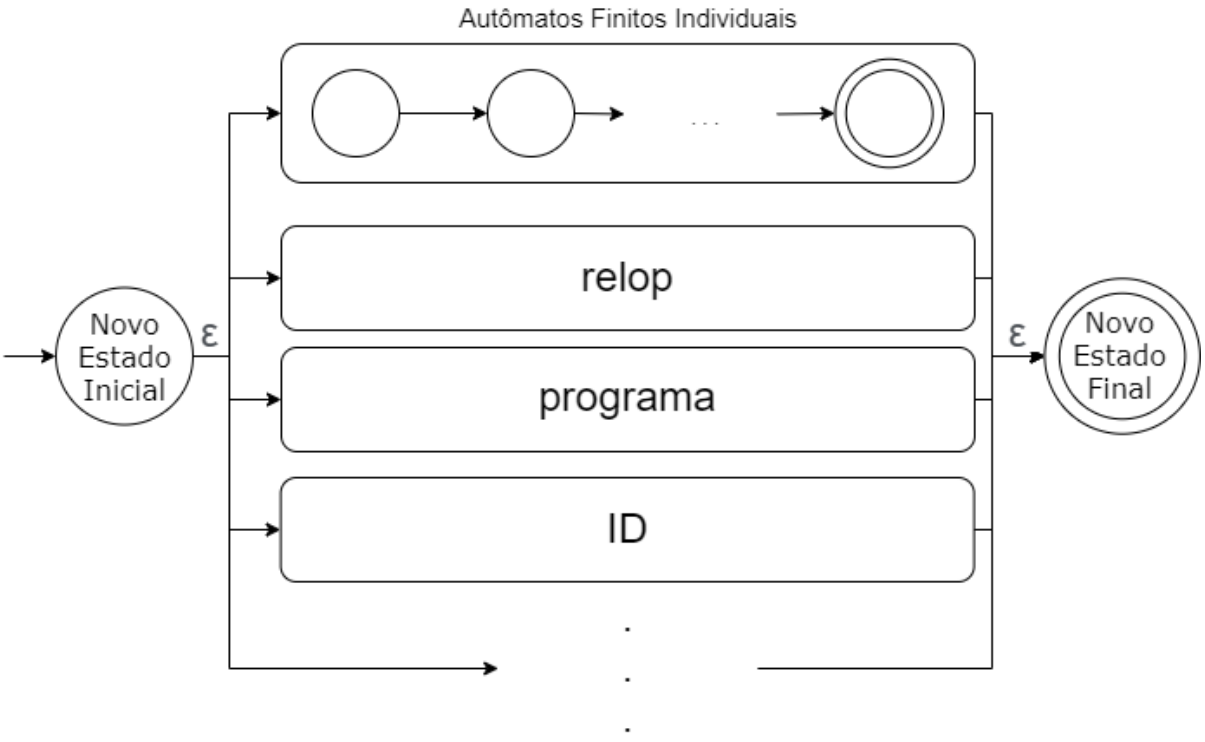




3.1.2 Diagrama de transição não determinístico

Após criados todos os diagramas de transição de todos os *tokens*, eles serão agrupados em um único diagrama de transição, sendo um 'autômato' finito não determinístico. Como a imagem dele ficou muito grande, é impossível colocar no relatório sem prejudicar a sua visualização, então foi criado um esquema demonstrando como que ele foi realizado. O diagrama completo está no repositório cujo acesso está disponível na primeira seção.

Resumindo, esse 'automato' é uma junção de todos os individuais, criando dois novos estados, o inicial e o final. Com isso, é criado transições vazias desse novo estado inicial para todos os estados iniciais de cada 'autômato' individual. O mesmo é feito para o estado final, em que todos os estados finais dos 'autômatos' individuais são levados ao novo estado final por meio de transições vazias.

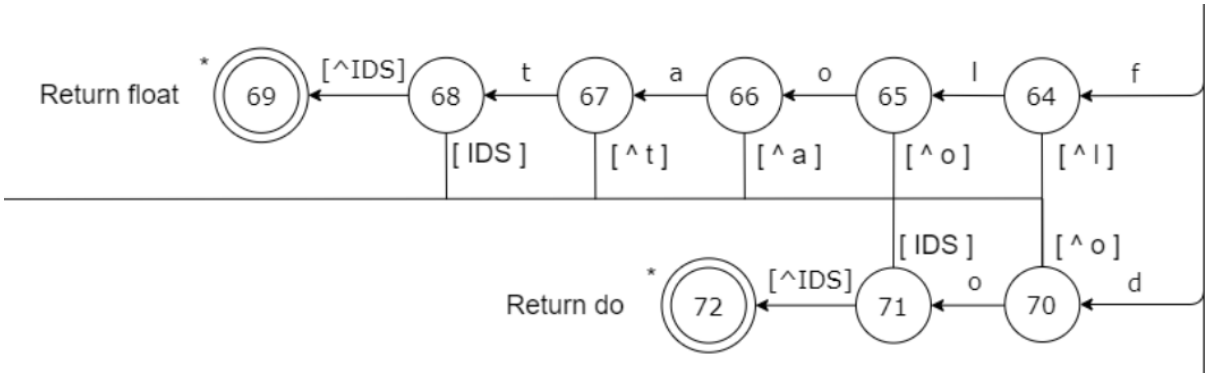


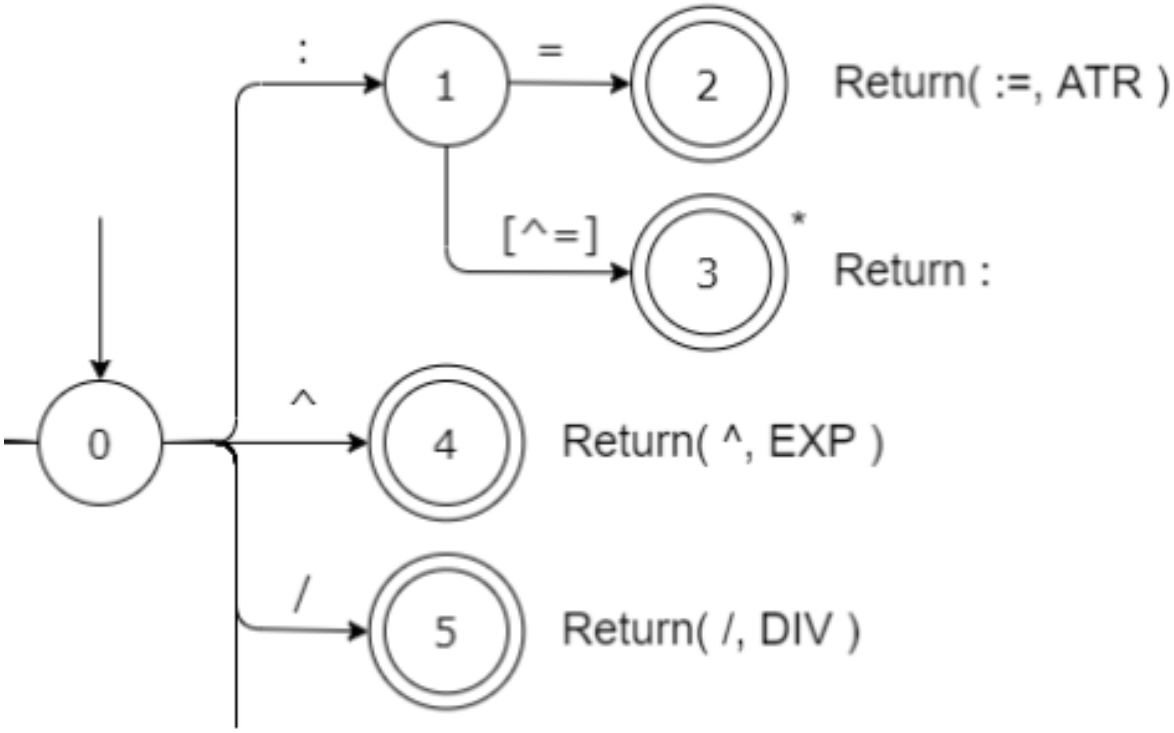
3.1.3 Diagrama de transição determinístico

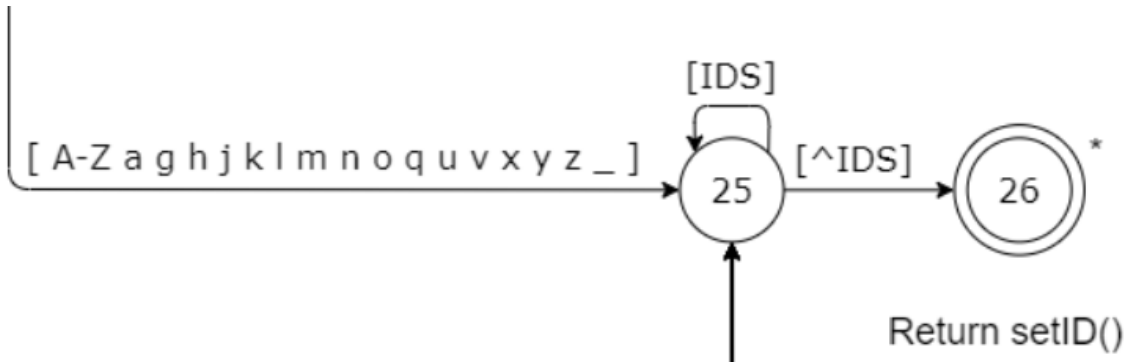
Por fim, será realizado a conversão do 'autômato' finito não determinístico em um 'autômato' finito determinístico. Como o diagrama é muito extenso, com mais de 120 estados, não foi criado tabelas de transições para a conversão, pois cada tabela teria mais de 100 linhas, e mais de 30 colunas, sendo muito trabalhoso.

Com isso, o diagrama de transição determinístico foi feito manualmente e, dessa forma, provavelmente existe estados idênticos, sendo possível sua minimização. Porém não foi pedido o menor diagrama possível, então a minimização dele não será realizada.

Da mesma forma do diagrama de transição não determinístico, a visualização desse diagrama completo não é possível. Portanto, será inserido alguma partes dele no relatório e sua imagem completa está no repositório mencionado.







Representação do IDS
IDS = [A-Z a-z 0-9 _]
Letras, Números e Underline
Logo [^IDS] = [^ A-Z a-z 0-9 _]

3.2 Analisador Léxico

Concluindo a seção de analise léxica, foi desenvolvido um programa manual do analisador léxico. Esse analisador desenvolvido em *Python* consiste em uma sub rotina do analisador sintático que deve devolver um *token* de cada vez. Ele foi desenvolvido usando a abordagem da codificação direta. O retorno dessa função é representado por:

- **Nome do token:** como 'if', 'relop', 'ID', 'num', ';' entre outros;
- **Valor do atributo:** caso seja necessário, como 'GT', 'NE', 'ADD', 'valor do id na tabela de simbolo' entre outros, ou somente '-' caso não exista;
- **Posição do token:** Definido pela linha e coluna do token.

Por fim, a tabela de símbolos armazena os identificadores e constantes em um arquivo separado, facilitando a futura leitura dos outros analisadores ainda não implementados. Cada ID ou num é representado por uma linha do arquivo criado.

Como o código fonte ficou muito extenso, somente algumas partes dele estão inseridas aqui no relatório. O código completo está disponível no repositório já indicado. Vale ressaltar que o código fonte está mais comentado, facilitando mais o entendimento dele. Vale relembrar que foi criado um video explicando melhor o código, em que ele está disponível no repositório mencionado.

```
1  # Função principal do analisador léxico
2  def buscar_token(arq, tk):  # Buscar token pelo numero tk
3      # Tabela de símbolos
4      Tab_Simb = open("Tab_Simb.txt", "r+")
5
6      # Constante numérica
7      NUM = ['0', '1', '2', '3', '4', '5', '6', '7', '8', '9']
8
9      # Letras maiúsculas
10     LUpp = ['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J', 'K',
11     'L', 'M', 'N', 'O', 'P', 'Q', 'R', 'S', 'T', 'U', 'V', 'W',
12     'X', 'Y', 'Z']
13
14     # Letras minusculas
15     LDnn = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k',
16     'l', 'm', 'n', 'o', 'p', 'q', 'r', 's', 't', 'u', 'v', 'w',
17     'x', 'y', 'z']
18
```

```
19      # Direto para ID
20      ProbID = ['a', 'g', 'h', 'j', 'k', 'l', 'm', 'n', 'o', 'q',
21      'u', 'v', 'x', 'y', 'z', '_', "'"]
22      # Caso seja um ID
23      IDS = ['_', "'"] + LUpp + LDnn + NUM
24
25      nlinha = 1      # numero da linha para possíveis erros
26      ncoluna = 0     # numero da coluna para possíveis erros
27      lah = False     # Look Ahead
28      estado = 0      # Estado atual
29      ntk = 1         # numero do token
30
31      [...]
32
33      # Caso precise utilizar look ahead
34       while 1:
35          if lah == False:  # Look Ahead
36              carac = arq.read(1)
37              ncoluna += 1
38          if not carac:   # eof
39              break
40          lah = False
41
42      [...]
43      # Exemplo de tokens com 2 estados (estado 0 e final)
44          if estado == 0:
45              [...]
46
47              elif carac == '^':  # Caractere ^
48                  if ntk == tk:
49                      return ('^', 'EXP',  (nlinha, ncoluna))
50                  else:
51                      ntk += 1
52                      estado = 0
53              [...]
54
55              elif carac == '/':  # Caractere /
56                  if ntk == tk:
57                      return ('/', 'DIV',  (nlinha, ncoluna))
58                  else:
59                      ntk += 1
60                      estado = 0
```

```
61
62              [...]
63
64              # Caso token for um ID, vai para o estado 25
65              elif carac in ProbID:  # Caractere  do ID
66                  estado = 25
67
68              else:  # Casso de erro
69                  return ('ERRO', 'ERRO', (nlinha, ncoluna))
70
71          [...]
72
73          # Demonstração do estado 1 (começa com :)
74          elif estado == 1:
75              if carac == '=':    # caractere :=
76                  if ntk == tk:
77                      return (':=', 'ATR',  (nlinha, ncoluna))
78                  else:
79                      ntk += 1
80                      estado = 0
81              else:   # caractere =
82                  if ntk == tk:
83                      return (':', NULL,  (nlinha, ncoluna))
84                  else:
85                      ntk += 1
86                      estado = 0
87                      lah = True
88          [...]
89
90          # Demonstração do estado 25 (ID)
91          elif estado == 25:
92              if carac in IDS:
93                  estado = 25
94              else:
95                  if ntk == tk:   # Adicionar na tabela de símbolos
96                      x = addtab(strSimb[:-1], Tab_Simb)
97                      return ('ID', x,  (nlinha, ncoluna))
98                  else:
99                      ntk += 1
100                     strSimb = ''
101                     estado = 0
102                     lah = True
```

4 Análise Sintática

Para uma melhor demonstração e explicação da análise sintática do projeto, ela será realizada em quatro diferentes partes, sendo a primeira fazer ajustes na gramática gerada no capitulo 2, tornando-a do tipo *LL(1)*. A segunda parte consiste em calcular os *FIRST's* e os *FOLLOW's* para os símbolos da gramática. Logo após isso, será feito a construção dos grafos sintáticos referentes a gramática já ajustada. E por fim, a quarta e última parte desse capítulo corresponde ao desenvolvimento do analisador sintático preditivo manual.

4.1 Ajustes na Gramática

Essa primeira etapa do processo de criação da análise sintática pode ser separada em duas partes, sendo a primeira a remoção de recursão a esquerda e a segunda o tratamento das suas ambiguidades. Analisando a gramática criada, é possível perceber que existem esses problemas citados que impedem de ser uma gramática *LL(1)*.

Gramatica original:

G=({EP, Bloco, Dvar, Scom, Var, IDs, Com, Cond, Expr, Term}, {programa, ID, begin, end, tipo, : , ; , ',' , (,) , := , if, then, else, while, do, repeat, relop, +, -, *, /, ^, num, ϵ}, P, EP), sendo:

P = {
EP -> **programa** ID Bloco
Bloco -> **begin** DVar SCom **end**
DVar -> Var | Var DVar | ϵ
SCom -> Com | Com SCom | ϵ
Var-> **tipo :** IDs **;**
IDs -> ID | ID , IDs
Com -> **if** (Cond) **then** Bloco **else** Bloco | ID **:=** Expr **;** |
| **while** (Cond) **do** Bloco | **repeat** Bloco **while** (Cond)
Cond -> Expr **relop** Expr
Expr -> Expr Ariop Term | Term
Ariop -> + | - | * | / | ^
Term -> num | ID | (Expr) }

4.1.1 Remoção de recurção à esquerda

Analisando melhor a gramática, é possível perceber que existe recursão à esquerda na variável *Expr*, então é solucionado esse problema criando uma nova variável *Expr'*, sendo:

. . .
Expr -> Term Expr' | Term
Expr' -> Ariop Term Expr' | Ariop Term

4.1.2 Tratamento de ambiguidades

Analisando novamente a gramática, pode-se verificar que existe ambiguidade de precedência nos operadores aritméticos (*Ariop*), então resolveremos esse ponto definindo suas precedências, assim temos:

. . .
Expr -> Expr + Pre1 | Expr - Pre1 | Pre1
Pre1 -> Pre1 * Pre2 | Pre1 / Pre2 | Pre2
Pre2 -> Pre2 ^ Term | Term
Term -> num | ID | (Expr)

Resolvendo essa ambiguidade de procedência, foi criado uma recurção à esquerda, no qual também deve ser removida:

. . .
Expr -> Pre1 Expr' | Pre1
Expr' -> + Pre1 | - Pre1 | + Pre1 Expr' | - Pre1 Expr'
Pre1 -> Pre2 Pre1' | Pre2
Pre1' -> * Pre2 | / Pre2 | * Pre2 Pre1' | / Pre2 Pre1'
Pre2 -> Term Pre2' | Term
Pre2' -> ^ Term | ^ Term Pre2'
Term -> num | ID | (Expr)

Por fim, para tornar a linguagem *LL(1)*, deve-se fatorar as variáveis que começam se repetindo, para não ter dúvidas sobre qual produção de um não-terminal deve ser usada no analisador sintático que será criado, fazendo a fatoração à esquerda. Após uma breve analise da gramática, foram encontrados algumas variáveis que contém esse erro, que são: *DVar, SCom, IDs* e precedentes de *Expr* criadas. Então, suas formas fatoradas serão:

DVar -> Var DVar | ϵ
Var -> tipo : ID IDs;
IDs -> , ID IDs | ϵ
SCom -> Com SCom | ϵ
Expr -> Pre1 Expr'
Expr' -> + Pre1 Expr' | - Pre1 Expr' | ϵ
Pre1 -> Pre2 Pre1'
Pre1' -> * Pre2 Pre1' | / Pre2 Pre1' | ϵ
Pre2 -> Term Pre2'
Pre2' -> ^ Term Pre2' | ϵ

Com todas essas melhorias criadas, foi gerado a nova gramática que é *LL(1)*:

G=(
{EP, Bloco, Dvar, Scom, Var, IDs, Com, Cond, Expr, Expr', Pre1, Pre1', Pre2, Pre2', Term},
{programa, ID, begin, end, tipo, : , ; , ',' , (,) , := , if, then, else, while, do, repeat, relop,
+, -, *, /, ^, num, ϵ}, P, EP)
sendo:

P = {
EP -> **programa ID** *Bloco*
Bloco -> **begin** *DVar SCom* **end**
DVar -> *Var DVar* | ϵ
SCom -> *Com SCom* | ϵ
Var -> **tipo : ID** *IDs* **;**
IDs -> **,** **ID** *IDs* | ϵ
Com -> **if** (*Cond*) **then** *Bloco* **else** *Bloco* | **ID :=** *Expr* **;** |
| **while** (*Cond*) **do** *Bloco* | **repeat** *Bloco* **while** (*Cond*)
Cond -> *Expr* **relop** *Expr*
Expr -> *Pre1 Expr'*
Expr' -> **+** *Pre1 Expr'* | **-** *Pre1 Expr'* | ϵ
Pre1 -> *Pre2 Pre1'*
Pre1' -> ***** *Pre2 Pre1'* | **/** *Pre2 Pre1'* | ϵ
Pre2 -> *Term Pre2'*
Pre2' -> ^ *Term Pre2'* | ϵ
Term -> **num** | **ID** | (*Expr*)

4.2 Cálculo dos FIRST's e FOLLOW's

O *FIRST* e o *FOLLOW* são funções que auxiliam na construção de analisadores sintáticos, ajudando a escolher qual produção aplicar com base no próximo *token*. Para desenvolver o analisador sintático preditivo manual, será necessário saber os valores dessas funções, então elas serão calculadas nessa etapa.

4.2.1 FIRST

Essa função deve retornar o conjunto de símbolos terminais que iniciam as cadeias derivadas. Para calcular essas funções de cada cadeia, é necessário seguir as regras explicadas em aula, além de começar de baixo para cima. É importante ressaltar que o *FIRST* de símbolos terminais serão sempre os próprios terminais, logo eles não serão escritos aqui (*Ex: FIRST(+) = { + }*).

Com isso, temos os valores dos *FIRST's* :

FIRST(Term) = { num, ID, (}
FIRST(Pre2') = { ^, ϵ }
FIRST(Pre2) = FIRST(Term) = { num, ID, (}
FIRST(Pre1') = { *, /, ϵ }
FIRST(Pre1) = FIRST(Pre2) = FIRST(Term) = { num, ID, (}
FIRST(Expr') = { +, -, ϵ }
FIRST(Expr) = FIRST(Pre1) = FIRST(Pre2) = FIRST(Term) = { num, ID, (}
FIRST(Cond) = FIRST(Expr) = FIRST(Pre1) = FIRST(Pre2) = FIRST(Term) = { num, ID, (}
FIRST(Com) = { if, ID, while, repeat } $\cup$ FIRST(Expr) = { if, ID, while, repeat }
FIRST(IDs) = { ',' , ϵ }
FIRST(Var) = { tipo }
FIRST(SCom) = { ϵ } $\cup$ FIRST(Com) = { if, ID, while, repeat, ϵ }
FIRST(DVar) = { ϵ } $\cup$ FIRST(Var) = { tipo, ϵ }
FIRST(Bloco) = { begin }
FIRST(EP) = { programa }

4.2.2 FOLLOW

Já essa função deve retornar o conjunto de símbolos terminais que podem aparecer imediatamente à direita do simbolo não terminal em alguma forma sentencial. Para calcular essa função para as cadeias, também deve-se seguir as regras explicadas. Diferente do *FIRST*, essa função deve-se começar do topo da gramática.

Assim, temos os valores dos *FOLLOW's*:

FOLLOW(EP) = { $ }
FOLLOW(Bloco) = FOLLOW(EP) ∪ FIRST(else) ∪ FIRST(while) ->
∪ FOLLOW(Com) = FIRST (SCom) ∪ FOLLOW (SCom) = FIRST(end) ->
FOLLOW(Bloco) ={ $, else, while, if, ID, repeat, end }
FOLLOW(DVar) = FIRST(SCom) ∪ FOLLOW(SCom) = { if, ID, while, repeat, end }
FOLLOW(SCom) = FIRST(end) = { end}
FOLLOW(Var) = FIRST(DVar) ∪ FOLLOW(DVar) = { tipo, if, ID, while, repeat, end }
FOLLOW(IDs) = FIRST(;) = { ; }
FOLLOW(Com) = FIRST(SCom) ∪ FOLLOW(SCom) = { if, ID, while, repeat, end }
FOLLOW(Cond) = FIRST ( )) = {) }
FOLLOW(Expr) = FIRST(;) ∪ FIRST(relop) ∪ FIRST()) ∪ FOLLOW(Cond) = { ; , relop,) }
FOLLOW(Expr') = FOLLOW(Expr) = { ; , relop,) }
FOLLOW(Pre1) = FIRST(Expr') ∪ FOLLOW(Expr') ∪ FOLLOW(Expr) = { +, -, ; , relop,) }
FOLLOW(Pre1') = FOLLOW(Pre1) = { +, -, ; , relop,) }
FOLLOW(Pre2) = FIRST(Pre1') ∪ FOLLOW (Pre1') = { *, /, +, -, ; , relop,) }
FOLLOW(Pre2') = FOLLOW(Pre2) = { *, /, +, -, ; , relop,) }
FOLLOW(Term) = FIRST(Pre2') ∪ FOLLOW(Pre2') = { ^, *, /, +, -, ; , relop,) }

Para melhorar a visualização dos valores das funções *FIRST* e *FOLLOW*, foi criado uma tabela contendo os resultados de cada variável analisada, com seu respectivo *FIRST* e *FOLLOW*:

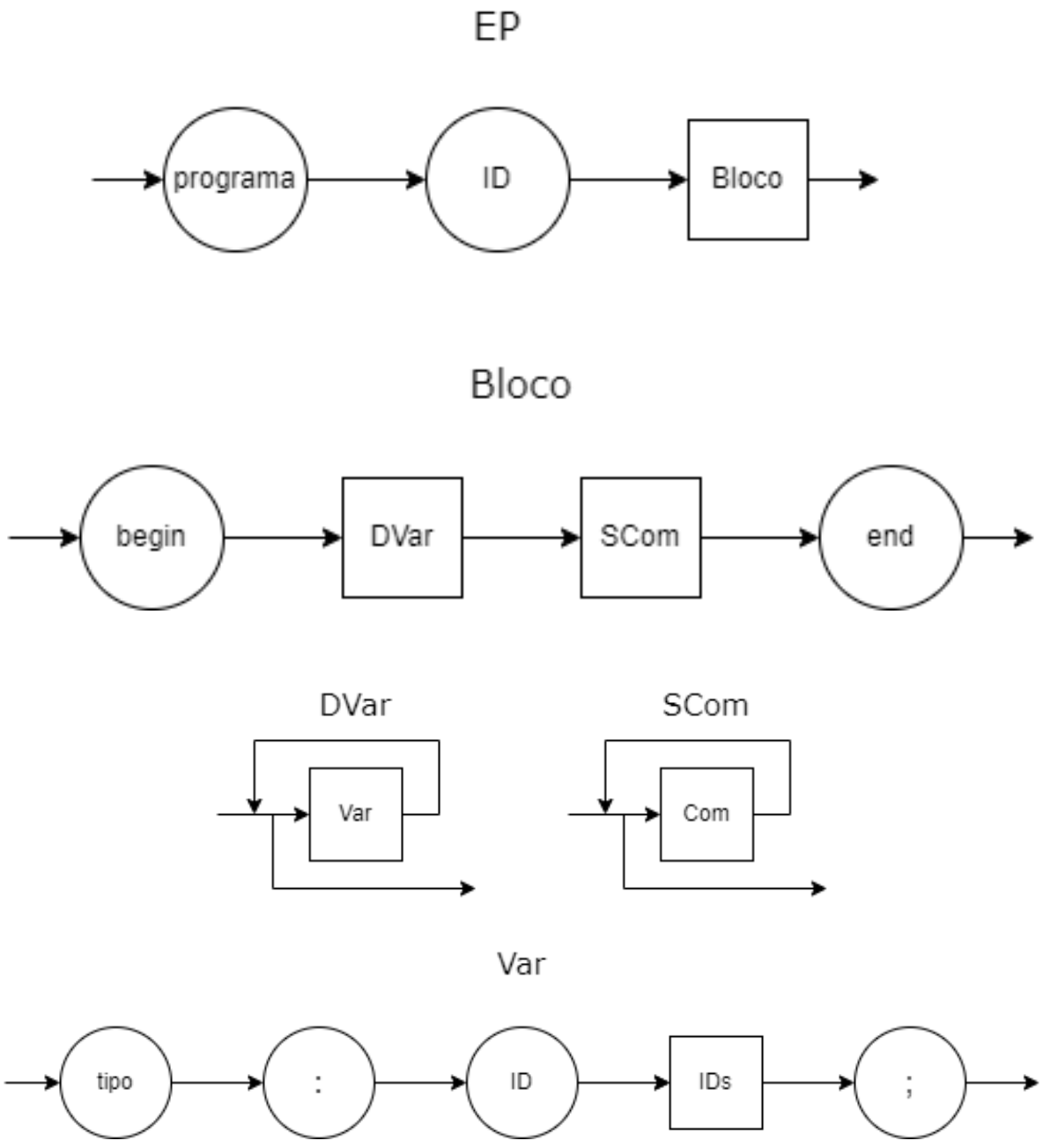
Variáveis	FIRST	FOLLOW
EP	{ programa }	{ $ }
Bloco	{ begin }	{ $, else, while, if, ID, repeat, end }
DVar	{ tipo, ϵ }	{ if, ID, while, repeat, end }
SCom	{ if, ID, while, repeat, ϵ }	{ end}
Var	{ tipo }	{ tipo, if, ID, while, repeat, end }
IDs	{ ',' , ϵ }	{ ; }
Com	{ if, ID, while, repeat }	{ if, ID, while, repeat, end }
Cond	{ num, ID, (}	{) }
Expr	{ num, ID, (}	{ ; , relop,) }
Expr'	{ +, -, ϵ }	{ ; , relop,) }
Pre1	{ num, ID, (}	{ +, -, ; , relop,) }
Pre1'	{ *, /, ϵ }	{ +, -, ; , relop,) }
Pre2	{ num, ID, (}	{ *, /, +, -, ; , relop,) }
Pre2'	{ ^, ϵ }	{ *, /, +, -, ; , relop,) }
Term	{ num, ID, (}	{ ^, *, /, +, -, ; , relop,) }

Tabela 2 – Tabela das Funções

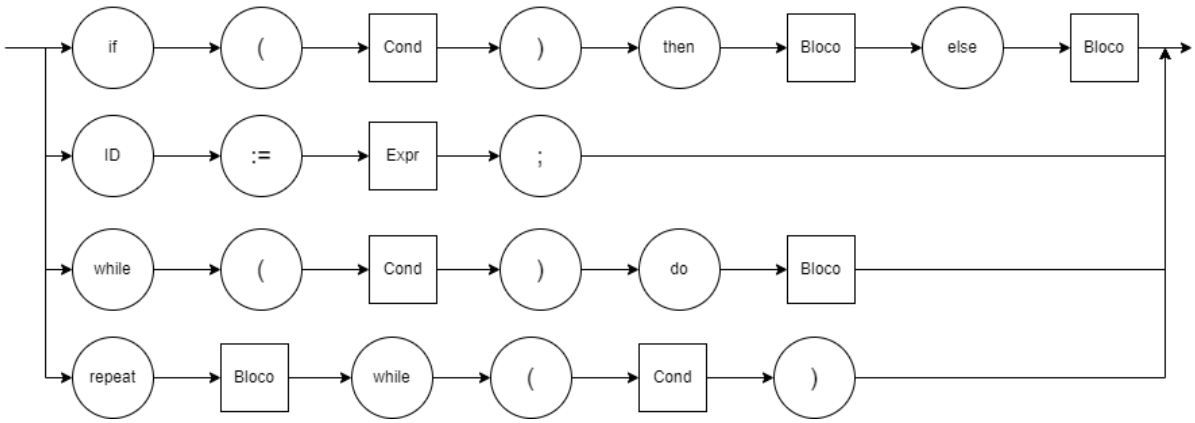
4.3 Grafos Sintáticos

Grafos sintáticos são uma maneira diferente de representar uma linguagem. Essa representação facilita na visualização do tipo de cadeias ou formas sentenciais que cada não-terminal pode gerar, além de auxiliar na construção do analisador sintático manual que será desenvolvido na próxima seção.

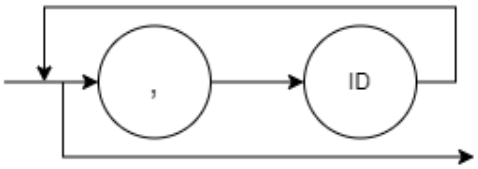
De acordo com a nossa gramática, os grafos sintáticos gerados são:



Com

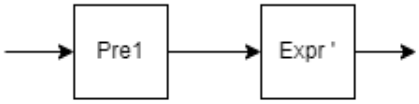


IDs

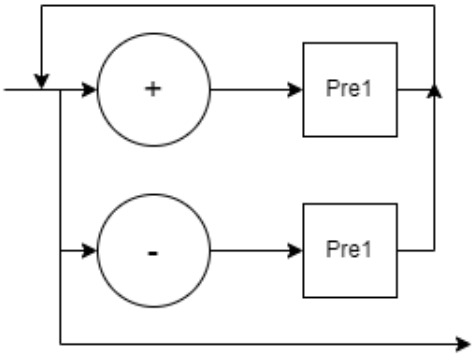


Cond

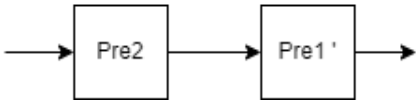
Expr

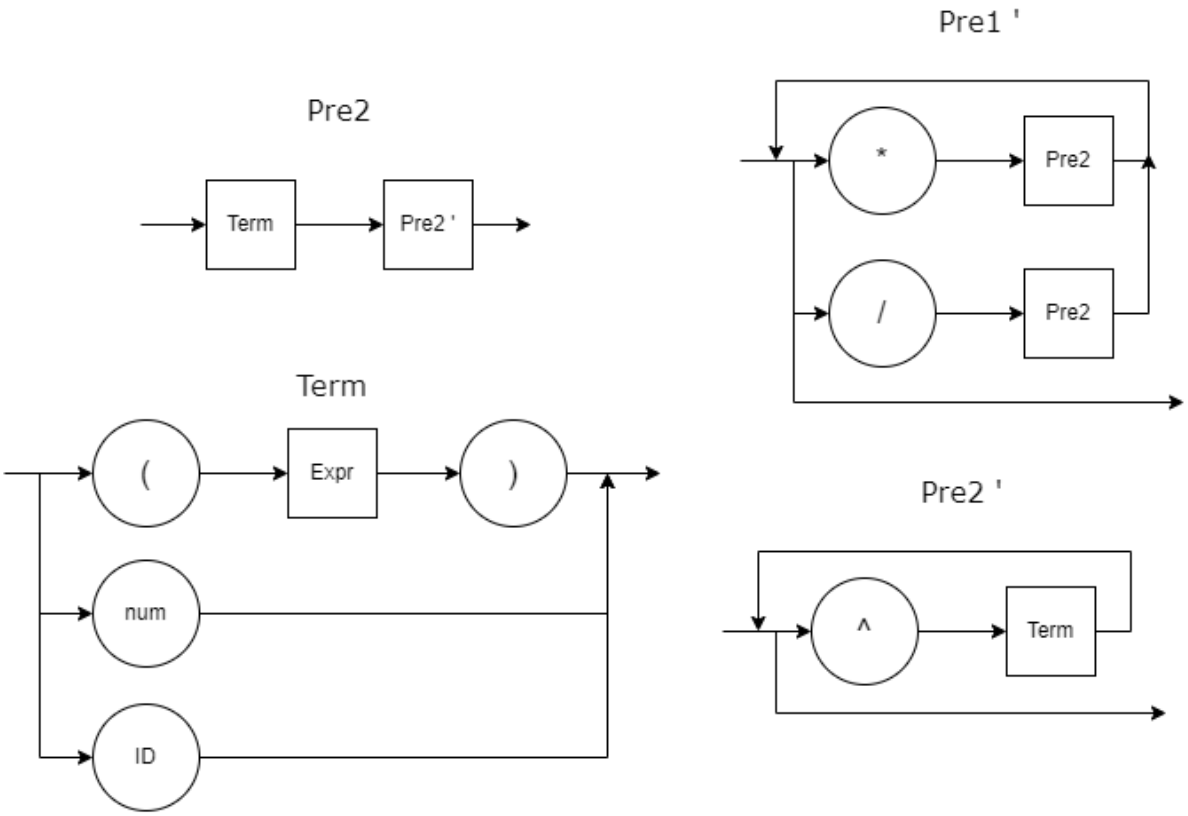


Expr '



Pre1





4.4 Analisador Sintático Preditivo

Para concluir esse etapa do projeto, será construído um analisador sintático preditivo manual. Analisador sintático preditivo significa que ele é baseado em estratégias descendentes ou *top-down*, utilizando uma especie de árvore de derivação construída da raiz para as folhas. Dessa maneira, será utilizado a estrategia da descida recursiva, que consiste em utilizar chamadas recursivas implementadas utilizando pilha implícita, a fim de determinar se o fluxo de *tokens* possui as sentenças válidas para a linguagem escolhida.

O código feito em *Python* utiliza estrategias descendentes e recursão da esquerda para a direita e ele também pode ser separado em três partes principais. A primeira é o *main*, que basicamente abre os arquivos e envia para a análise da gramática (procedimentos), fazendo a recursão. Os procedimentos são os grafos escritos em códigos, muito semelhantes aos apresentados em aula, neles são alocados as produções em uma pilha até resultar nos *tokens*, gerando a árvore sintática correspondente com a gramática. Por fim existe a função *lex*, que busca o novo *token*, chamando o analisador léxico feito no capítulo anterior.

Da mesma forma que o analisador léxico, o código do analisador sintático ficou muito extenso, então somente algumas partes dele estão inseridas aqui no relatório, sendo que o código completo está disponível no repositório indicado.

```
# Main
arq = open("teste1.txt", "r")  # abre arquivo
Tab_Simb = open("Tab_Simb.txt", "w")  # limpar tabela de símbolos
Tab_Simb.close()
proxToken = ()
pilha = ['EP']
i = 1
proxToken = lex(i)
Procedimento_EP()  # Primeiro grafo
printarpilha(pilha)
if proxToken[1] != 'EOF':
    erro(proxToken, "EOF")
print("\nCodigo condiz com gramatica analisada!")
arq.close()

[...]

# Exemplo de Procedimentos
def Procedimento_EP():  # Procedimento EP (primeiro)
    global proxToken, pilha
    printarpilha(pilha)
    pilha.pop()
    if proxToken[0] == 'programa':
        proxToken = lex(i)
        pilha.append("programa")
        if proxToken[0] == 'ID':
            proxToken = lex(i)
            pilha.append("ID")
            pilha.append("Bloco")
            Procedimento_Bloco()
        else:
            erro(proxToken, "ID")
    else:
        erro(proxToken, "programa")
    return 0

[...]
```

```
38 def Procedimento_Term():    # Procedimento para Termos
39     global proxToken, pilha
40     printarpilha(pilha)
41     pilha.pop()
42     if proxToken[0] == 'num':
43         proxToken = lex(i)
44         pilha.append("num")
45     elif proxToken[0] == 'ID':
46         proxToken = lex(i)
47         pilha.append("ID")
48     elif proxToken[0] == '(':
49         proxToken = lex(i)
50         pilha.append("(")
51         pilha.append("Expr")
52         Procedimento_Expr
53         if proxToken[0] == ')':
54             proxToken = lex(i)
55             pilha.append(")")
56         else:
57             erro(proxToken, ")")
58     else:
59         erro(proxToken, "ID, num, (")
60     return 0
61
62  [...]
63
64  # Função para buscar os tokens
65  def lex(token):
66     (nome, atr, (linha, coluna)) = buscar_token(arq, token)
67     global i
68     i += 1
69     arq.seek(0, 0)
70     if atr == NULL:
71         atr = '-'
72     if nome == 'ERRO':
73         print(
74             f"Erro no caractere {atr} \nEncontrado
75             na linha {linha} e coluna {coluna}")
76         exit()
77     if nome == 'EOF':
78         return (nome, 'EOF', (linha, coluna))
79     return (nome, atr, (linha, coluna))
```

5 Tradução Dirigida por Sintaxe