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

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.

Especificação da linguagem para o projeto:

```
# Estrutura principal:
    programa nome_programa bloco

# Bloco:
    begin declaracao_das_variaveis sequencia_de_comandos end

# Declaracao de variaveis:
    tipo: lista_ids ;
    Sendo tipos: int, char e float

# Comando de atribuicao:
    id:= expressao ;

# Comentarios:
    [ texto_comentario ]

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

```

```
20 # Comandos de repeticao:
21     while(cond) do bloco
22     repeat bloco while(cond)
23
24 # Condicoes:
25     Igual (=), diferente (~=), menor (<), maior (>), menor ou igual
           (<=), maior ou igual (>=)
26
27 # Expressoes:
28     Soma (+), subtracao (-), multiplicacao (*), divisao (/) e
           exponenciacao (^)
29
30 # Observações
31     char deve estar entre apostrofo
32     int deve estar entre -32768 e +32767
33     float pode ser ponto fixo ou notacao cientifica
34     parenteses para priorizar operacoes
```

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.

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, ariop, 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** Com **else** Com | ID **:=** Expr **;** | Expr **;** | ϵ |
| **while** (Cond) **do** Com | **repeat** Com **while** (Cond)
Cond -> Expr **relop** Expr
Expr -> Expr Ariop Term | Term
Ariop -> + | - | * | / | ^
Term -> num | ID | (Expr) }

2.2 Identificação dos *Tokens*

Após a criação e implementação 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.

```
1 digito     -> [ 0 - 9 ]
2 digitos    -> digito (digito)*
3 numero     -> digitos ( . digitos ) ? (E[ + - ] ? digitos) ?
```

```
4  letra_    -> [ A - Za - z_ ]
5  ID         -> letra_ ( letra_ | digito )*
6  tipo       -> int | char | float
7  programa   -> programa
8  begin      -> begin
9  end        -> end
10 if         -> if
11 then       -> then
12 else       -> else
13 while      -> while
14 do         -> do
15 repeat     -> repeat
16 :          -> :
17 ;          -> ;
18 (          -> (
19 )          -> )
20 :=         -> :=
21 relop      -> =  | ~= | < | > | <= | >=
22 +          -> +
23 -          -> -
24 *          -> *
25 /          -> /
26 ^          -> ^
27 ws         -> (" " | \n | \t )
28 comentario -> "[" [^ "]" ] "]"
```

3 Análise Léxica

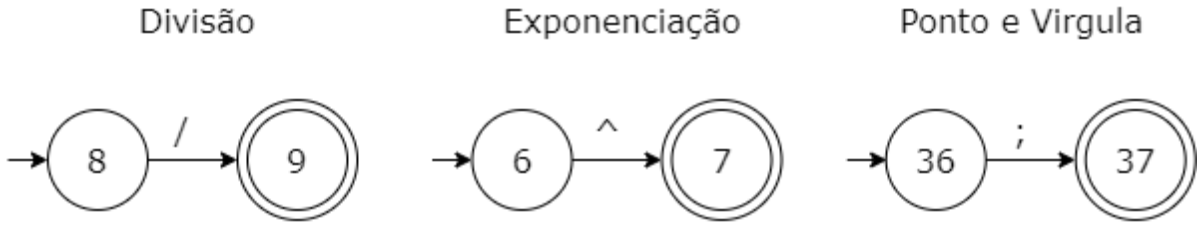
Para cumprir os requisitos e conseguir desenvolver a analise léxica do projeto, podemos dividir esta seção em duas. A primeira se trata de especificar os diagramas de transição, e a segunda parte seria a implementação manual do analisador léxico.

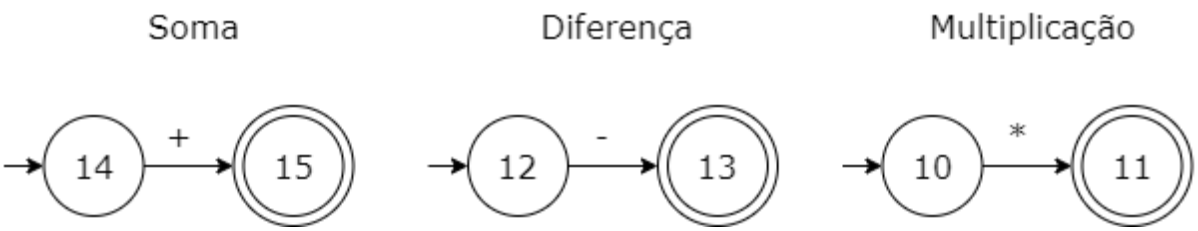
3.1 Diagramas de Transição

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.

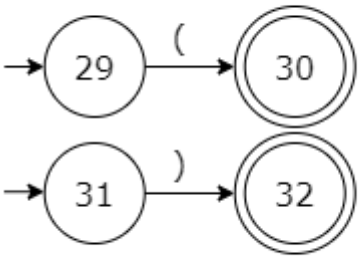
3.1.1 Diagramas de transições individuais

Primeiramente, iremos criar os diagramas de transição para todos os tokens identificados na seção anterior, sendo eles *programa, begin, end, ID, num, +, -, relop,* entre outros. Para essa criação, foi utilizado o aplicativo Diagrams. Assim temos:

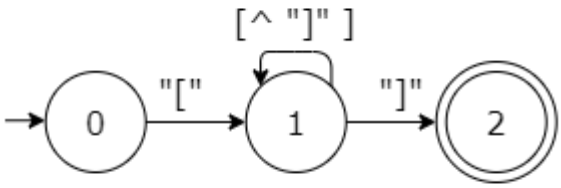




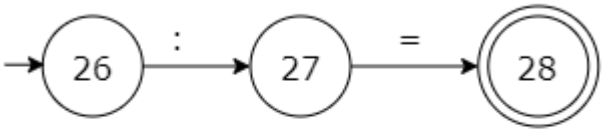
Parênteses



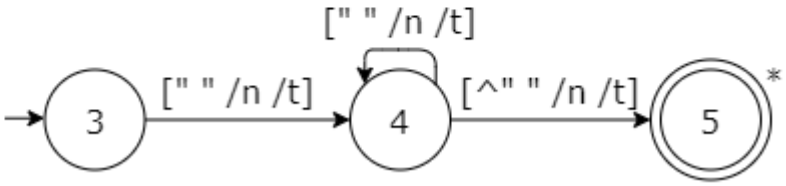
Comentário



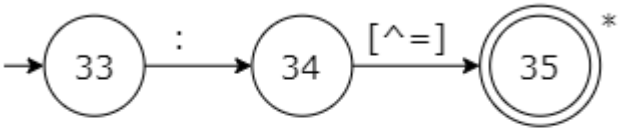
Atribuição



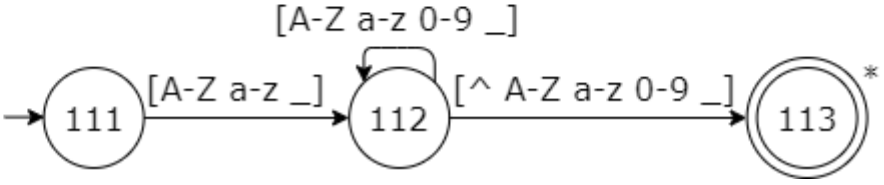
Separadores ws



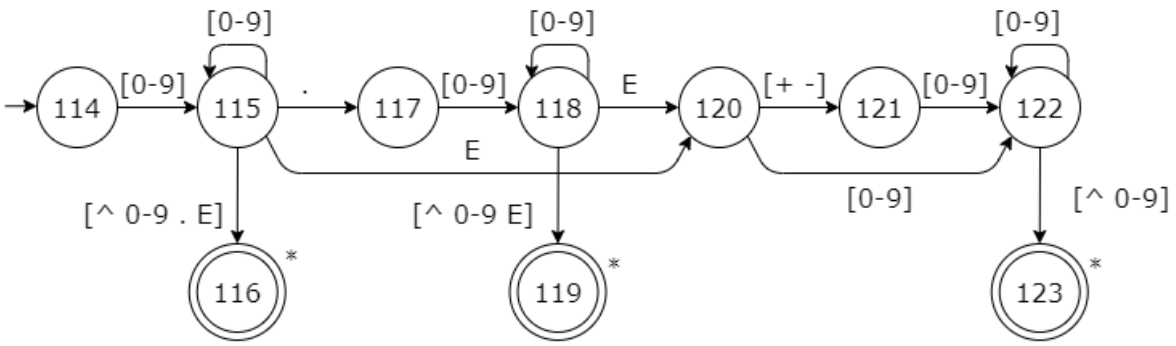
Dois pontos



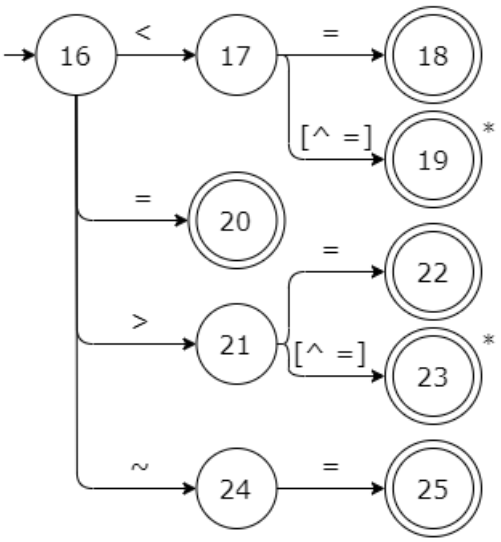
Identificador ID



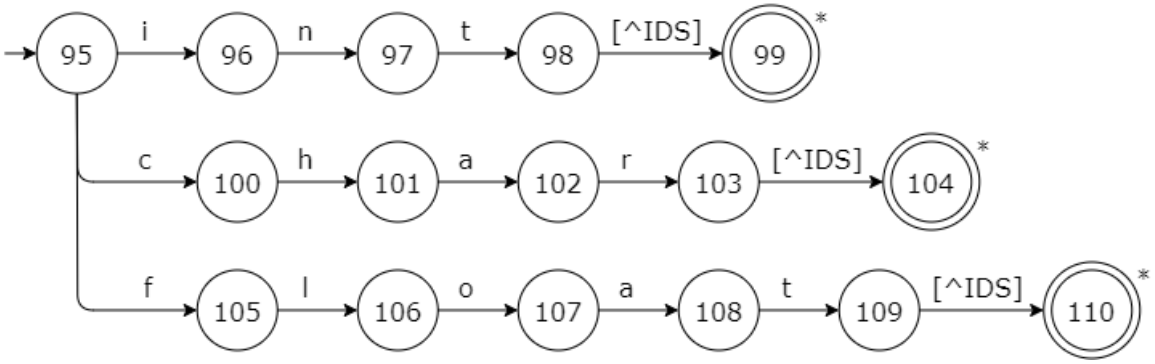
Números Num



Relop

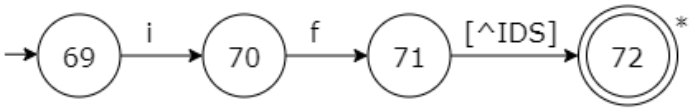


Tipo

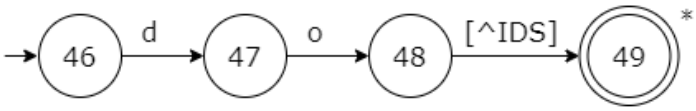


IDS = [A-Z a-z 0-9 _]
(número, letra ou _)

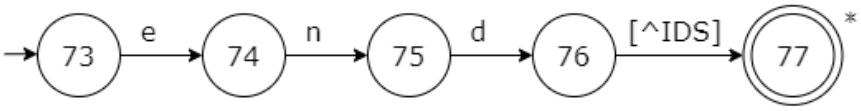
If



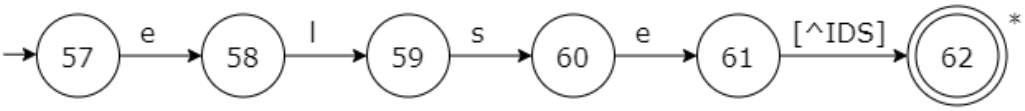
Do



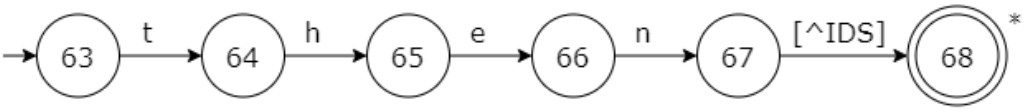
End



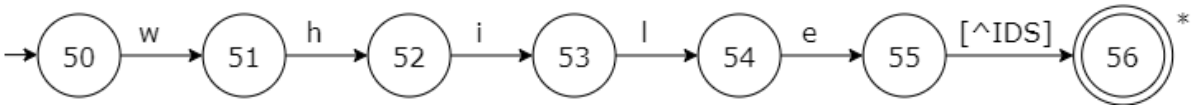
Else

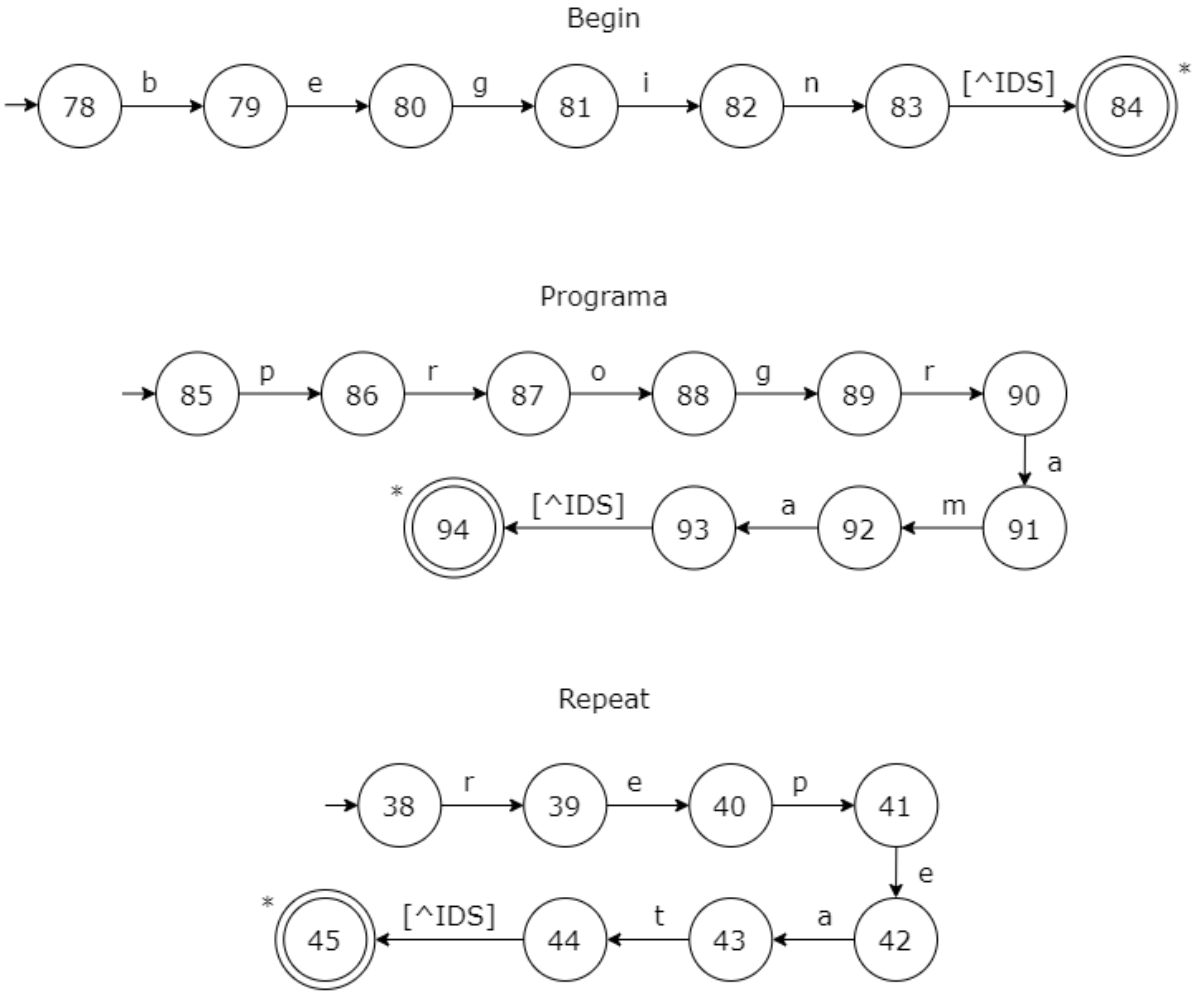


Then



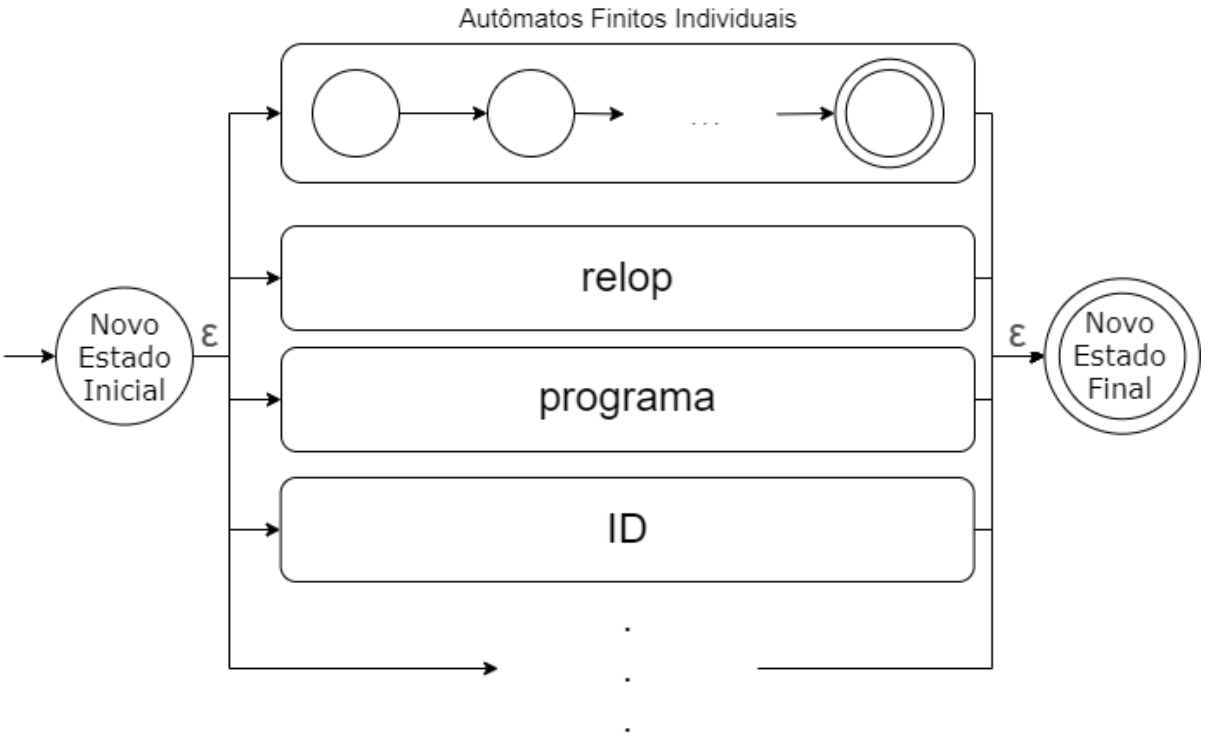
While





3.1.2 Diagrama de transição não determinístico

Após termos criados todos os diagramas de transição de todos os tokens, iremos agrupa-los em um único diagrama de transição, sendo um 'automato' finito não determinístico. Como a imagem dele ficou muito grande, é impossível colocar no relatório sem prejudicar a visualização, então foi criado um esquema demonstrando como que ele foi realizado. O diagrama completo está no repositório disponível na primeira seção.

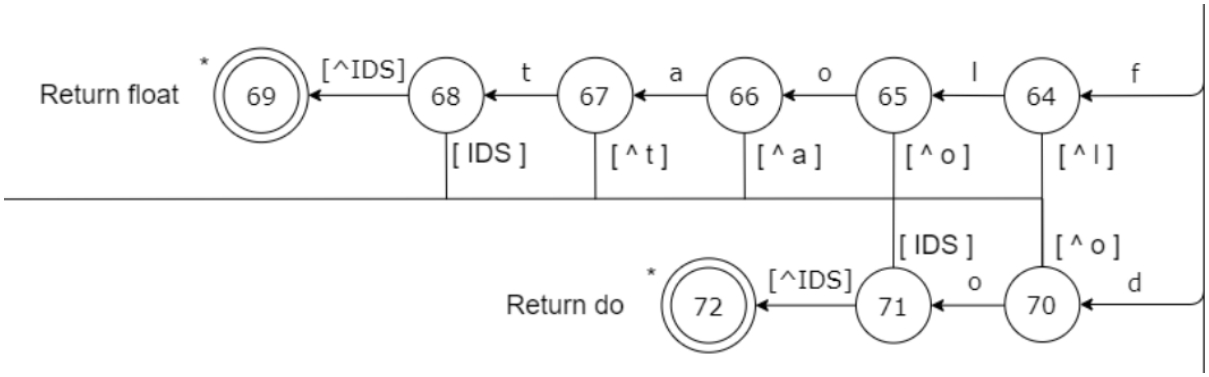


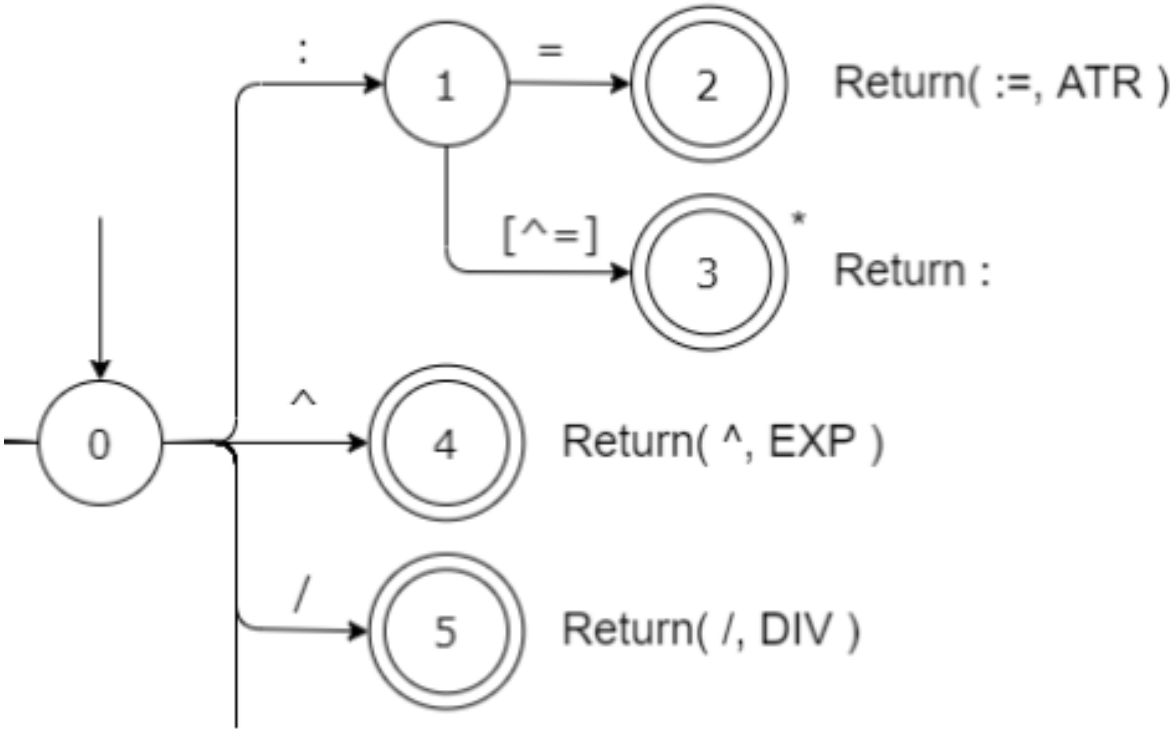
3.1.3 Diagrama de transição determinístico

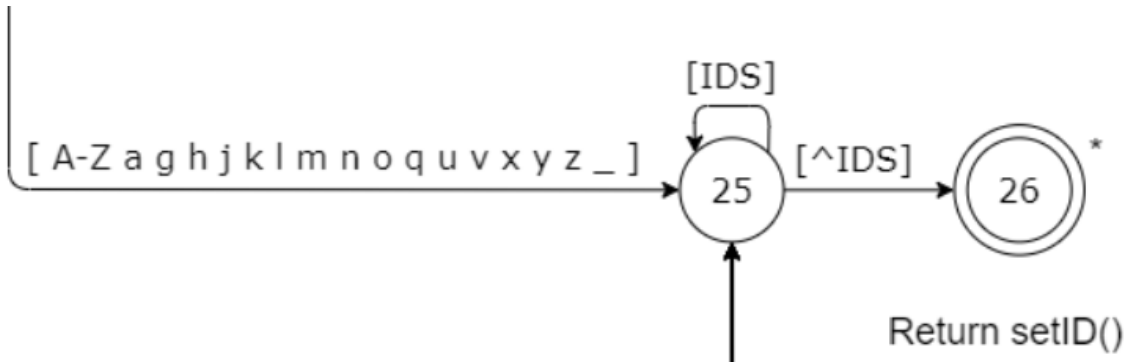
Por fim, iremos converter o 'automato' finito não determinístico em um 'automato' finito determinístico. Como o diagrama é muito extenso, para aumentar a velocidade da conclusão do projeto, não foi criado tabelas de transições para a conversão, pois cada tabela teria mais de 100 linhas, e mais de 30 colunas, sendo muito trabalhoso.

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

Da mesma forma do diagrama de transição não determinístico, a visualização desse diagrama 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;
- **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.

```
1  def addtab(elem):   # Função para inserir na tabela de simbolos
2      nelem = 1
3      Tab_Simb.seek(0, 0)
4      for linha in Tab_Simb:
5          nelem += 1
6          if elem in linha:
7              Tab_Simb.close()
8              return nelem
9
10     Tab_Simb.writelines(elem+"\n")
11     return nelem
12
13 def buscar_token(arq, tk):  # Busacar token numero tk
14     .
15   [...]
16     .
17     NUM = ['0', '1', '2', '3', '4', '5', '6', '7', '8', '9']  #
          Constante numerica
18     LUpp = ['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J', 'K', 'L
          ', 'M', 'N', 'O', 'P', 'Q', 'R', 'S', 'T', 'U', 'V', 'W', 'X',
           'Y', 'Z']
19     ProbID = ['a', 'g', 'h', 'j', 'k', 'l', 'm', 'n', 'o', 'q', 'u',
          'v', 'x', 'y', 'z', '_']
```

```
20      strSimb = ''
21      ProbID.extend(LUpp)
22      nlinha = 1  # numero da linha    .tell() mostra a posição do
           ponteiro
23      ncoluna = 0  # numero da coluna  arq.seek(-1, 1) volta 1 casa
24      lah = False   # Look Ahead - como o maximo é de 1 caractere, não
           é necessario voltar mais do que 1
25      estado = 0  # Estado atual
26      ntk = 1  # numero do token
27      .
28    [...]
29      .
30       while 1:
31          if lah == False:  # Look Ahead
32              carac = arq.read(1)
33              ncoluna += 1
34          if not carac:   # eof
35              break
36          lah = False
37  ################### Estado 0 ####################
38          if estado == 0:
39              if carac == ':':
40                  estado = 1
41
42              elif carac == '^':  # Caractere ^
43                  if ntk == tk:
44                      return ('^', 'EXP',  (nlinha, ncoluna))
45                  else:
46                      ntk += 1
47                      estado = 0
48
49              elif carac == '/':  # Caractere /
50                  if ntk == tk:
51                      return ('/', 'DIV',  (nlinha, ncoluna))
52                  else:
53                      ntk += 1
54                      estado = 0
55
56      .
57    [...]
58      .
59
```

```
60              elif carac in ProbID:  # Caractere  do ID
61                  estado = 25
62
63              else:  # Erro
64                  return ('ERRO', 'ERRO', (nlinha, ncoluna))
65
66  ################### Estado 1 ####################
67          elif estado == 1:
68              if carac == '=':    # caractere :=
69                  if ntk == tk:
70                      return (':=', 'ATR',  (nlinha, ncoluna))
71                  else:
72                      ntk += 1
73                      estado = 0
74              else:   # caractere =
75                  if ntk == tk:
76                      return (':', NULL,  (nlinha, ncoluna))
77                  else:
78                      ntk += 1
79                      estado = 0
80                      lah = True
81      .
82    [...]
83      .
84      ################### Estado 33 ####################
85          elif estado == 33:
86              if carac in NUM:  # Num
87                  strSimb = strSimb + str(carac)
88                  estado = 34
89              else:
90                  return ('ERRO', 'ERRO', (nlinha, ncoluna))
91      .
92    [...]
93      .
94        return ('RELOP', 'ERRO',  (nlinha, ncoluna))
```

4 Análise Sintática

5 Tradução Dirigida por Sintaxe