Escola Politécnica da Universidade de São Paulo

Relatório Final

PCS2056 - Linguagens e Compiladores

Prof. Ricardo Rocha

Data: 01/12/2016

Felipe de Paiva Miranda 7630486
Thiago Ryu Niwa Murakami 7626689

1. Sintaxe da linguagem

A sintaxe da linguagem sofreu diversas modificações durante o desenvolvimento do compilador. Segue a abaixo a versão final da mesma.

1.1. Wirth

OPERADOR = "+" | "-" | "/" | "*" .
OPERADORBOOL= "&" | "|" | "^".
OPERADORCOMP = "<" | "<=" | ">" | ">=" | "==" | "!=" | "&&" | "||" .
DIGITO = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" .
LETRA = "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" | "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" .

EOL = "\n" | "\r" .
SIMBOLO = "!" | "@" | "$" | "_" | " " | "\t" | OPERADOR | OPERADORBOOL | OPERADORCOMP .

BOOL = "true" | "false" .
INTEIRO = DIGITO { DIGITO } .
DECIMAL = ["+" | "-"] INTEIRO .
FLOAT = ["+" | "-"] INTEIRO "." INTEIRO .
STRING = """ {LETRA | SIMBOLO | DIGITO | EOL} """ .
DEF_VARIAVEL = "var" | "persistent" | "const" .

TIPO = "int" | "float" | "bool" | "string" .
IDENTIFICADOR = LETRA { LETRA } .
COMENTARIO = "#" {LETRA | SIMBOLO | DIGITO} EOL .

PROGRAMA = "program" ESCOPO .

DEFINICAO = "deftype" IDENTIFICADOR "{" {TIPO IDENTIFICADOR ","} TIPO IDENTIFICADOR "}" ";" .
FUNCAO = "function" IDENTIFICADOR "(" {TIPO IDENTIFICADOR ","} TIPO IDENTIFICADOR ")" ESCOPO | "function" IDENTIFICADOR "(" {TIPO IDENTIFICADOR ","} TIPO IDENTIFICADOR ")" ":" TIPO ESCOPO .

ESCOPO = "{" { COMANDO } "}" .

VARIAVEL = TIPO IDENTIFICADOR "=" EXPRESSAO ";" | TIPO {IDENTIFICADOR ","} IDENTIFICADOR ";" | TIPO "[" "]" IDENTIFICADOR "=" "{" {EXPRESSAO ","} EXPRESSAO "}" ";" | TIPO "[" INTEIRO "]" {IDENTIFICADOR ","} IDENTIFICADOR ";" .

CONDICIONAL = "(" EXPRESSAO ")" ESCOPO | "if" "(" EXPRESSAO ")" ESCOPO { "elsif" "(" EXPRESSAO ")" ESCOPO } "else" ESCOPO .
REPETICAO = "(" EXPRESSAO ")" ESCOPO .

COMANDO = IDENTIFICADOR "=" EXPRESSAO ";" | IDENTIFICADOR "(" [{EXPRESSAO ","} EXPRESSAO] ")" ";" | "return" EXPRESSAO ";" | DEF_VARIAVEL VARIAVEL | "if" CONDICIONAL | "while" REPETICAO | .

EXPRESSAO = TERMODECIMAL [OPERADOR EXPRESSAO] .
TERMODECIMAL = TERMOCOMP [OPERADORBOOL TERMODECIMAL] .
TERMOCOMP = TERMOPRIMARIO [OPERADORCOMP TERMOCOMP] .
TERMOPRIMARIO = IDENTIFICADOR | IDENTIFICADOR "(" [{EXPRESSAO ","} EXPRESSAO] ")" | DECIMAL | FLOAT | BOOL | STRING | "(" EXPRESSAO ")" .

1.2. Símbolos terminais

1. Operadores:
    a. =
    b. >
    c. <
    d. !
    e. +
    f. -
    g. *
    h. /
    i. ^
    j. &
    k. |
    l. =
    m. >=
    n. <=
    o. !=
    p. &&
    q. ||
2. Palavras reservadas:
    a. int
    b. float
    c. string
    d. bool
    e. function
    f. program
    g. deftype
    h. if
    i. else
    j. elsif

    k. while
    l. return
    m. var
    n. persistent
    o. const
    p. true
    q. false
3. Delimitadores:
    a. {
    b. }
    c. [
    d.]
    e. (
    f.)
    g. ;
    h. :
    i. \n
    j. \r
    k. \t
    l. (espaço em branco)
4. Aspa
    a. “
5. Comentario
    a. #

1.3. Exemplo

```
program {
	const string texto = "Digite um numero:";
	write(texto);
	var int numero = read();

	var int resultado = funcao(numero);
	write("Resultado: ");
	write(resultado);

	write("Digite outro numero:");
	numero = read();

	resultado = funcao(numero);
	write("Resultado: ");
	write(resultado);
}
```

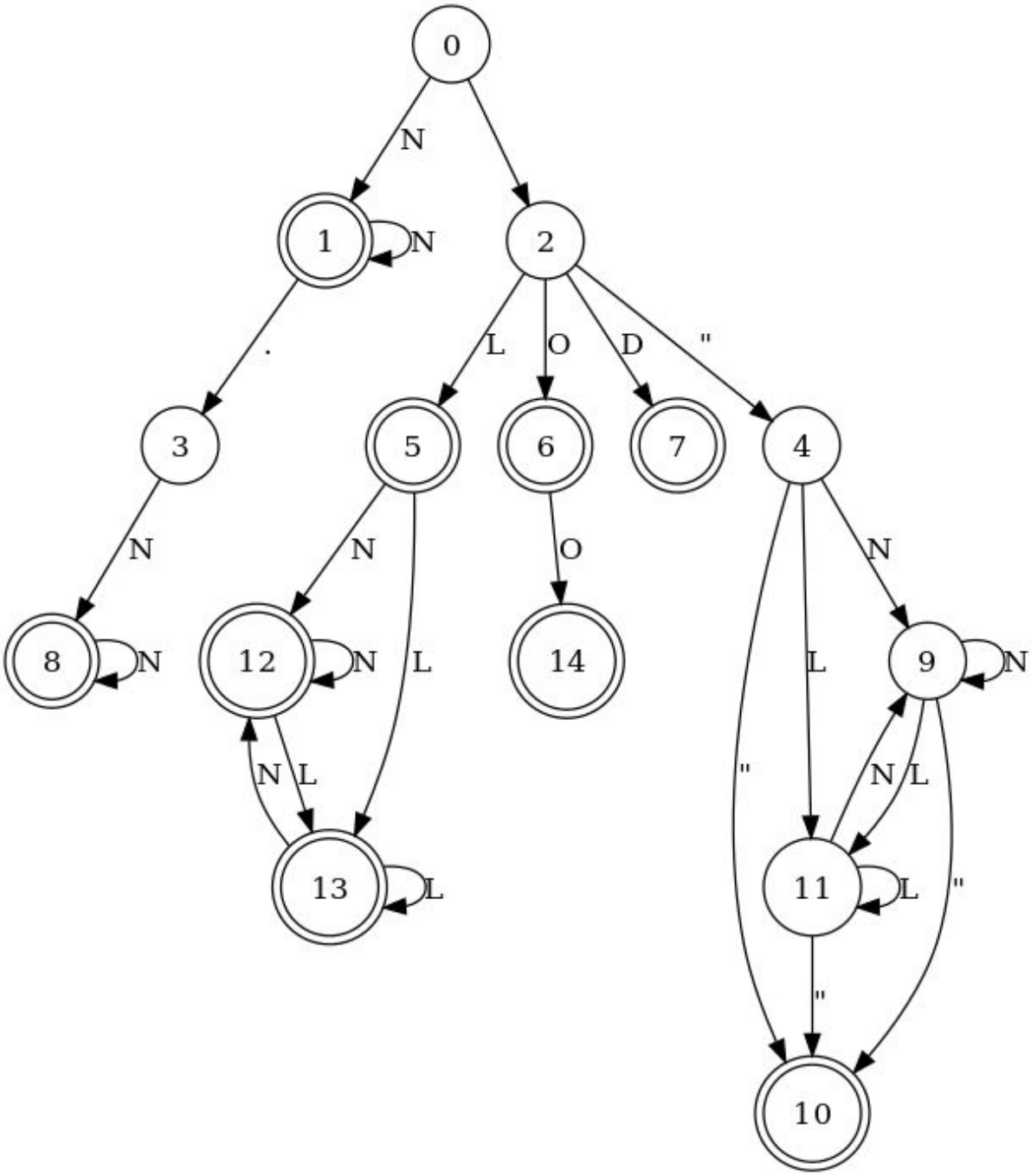
2. Analisador léxico

O analisador léxico tem como entrada o código-fonte e deve traduzir essa cadeia de caracteres em símbolos léxicos, ou tokens , que são trechos elementares completos e com identidade própria. É função do analisador léxico também, classificar esses tokens segundo o tipo, como identificadores, palavras reservadas, números, cadeias de caracteres, sinais de pontuação, etc e extrair corretamente os valores associados a eles, como por exemplo, realizar conversões numéricas. Além de extrair e classificar, o analisador deve eliminar delimitadores, como espaços em branco, e comentários, pois estes não são utilizados na geração do código.

2.1. Tokens

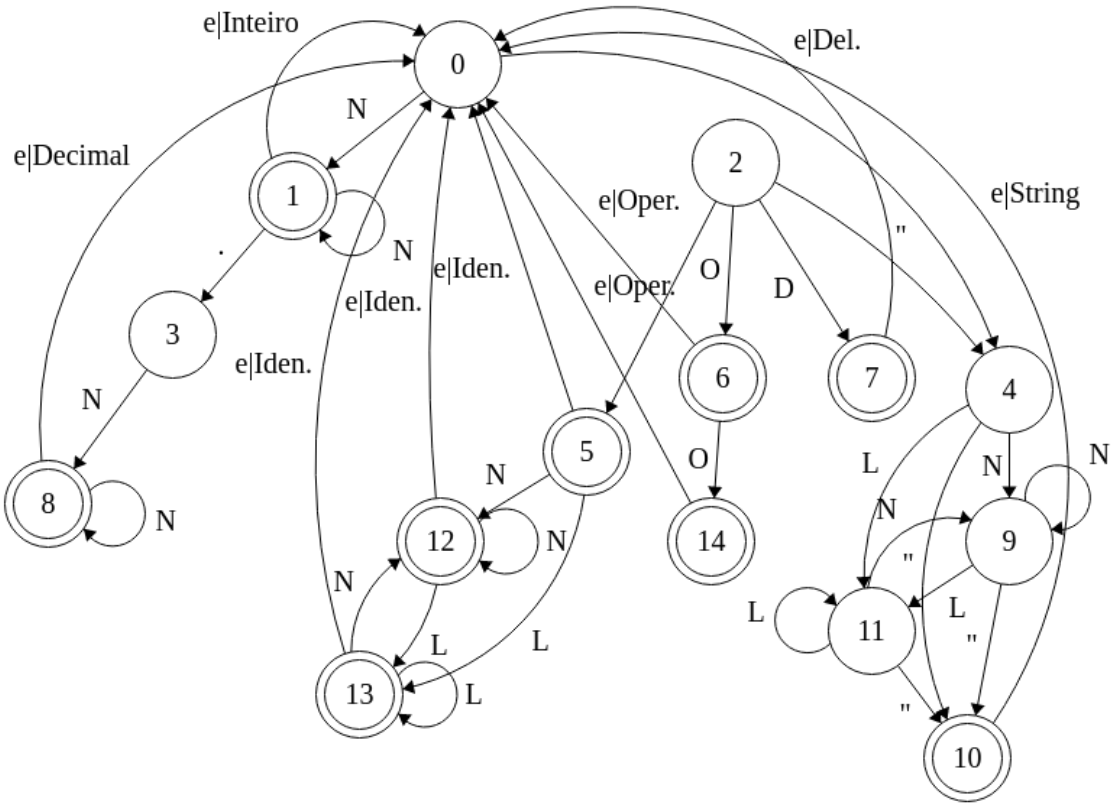
<números>: 0-9
<alfabeto>: a-z|A-Z
<operador>: ‘=’ | ’>’ | ‘!’ | ‘+’ | ‘-’ | ‘*’ | ‘/’ | ‘^’ | ‘&’ | ‘|’
<delimitador>: ‘{’ | ‘}’ | ‘[’ | ‘]’ | ‘(’ | ‘)’ | ‘,’ | ‘;’
<inteiros>: <números><números>*
<decimais>: <inteiros>’.’<inteiros>
<string literal>: ‘“‘(<alfabeto>|<números>)*’”’
<identificadores>: <alfabeto>(<alfabeto>|<números>)*
<operadores>: <operador>(<operador>|ε)

2.2. Autômato de estados finitos



onde N = 0 a 9.
onde L = a-z|A-Z
onde O = ‘=’ | ’>’ | ‘!’ | ‘+’ | ‘-’ | ‘*’ | ‘/’ | ‘^’ | ‘&’ | ‘|’
onde D = ‘{’ | ‘}’ | ‘[’ | ‘]’ | ‘(’ | ‘)’ | ‘,’ | ‘;’

2.3. Transdutor



onde N = 0 a 9.
onde L = a-z|A-Z
onde O = ‘=’ | ’>’ | ‘!’ | ‘+’ | ‘-’ | ‘*’ | ‘/’ | ‘^’ | ‘&’ | ‘|’
onde D = ‘{’ | ‘}’ | ‘[’ | ‘]’ | ‘(’ | ‘)’ | ‘,’ | ‘;’

3. Reconhecedor sintático

Após a etapa de análise léxica, o analisador sintático verifica a validade da sequência, isto é, se os átomos estão agrupados de forma sintaticamente corretas, de acordo com as regras especificadas. Essa atividade denomina-se reconhecimento sintático, e além dessa função de reconhecer erros de sintaxe, o analisador deve ter mecanismos de recuperação de erros sintáticos para que possa prosseguir a análise do programa mesmo na detecção de erros.

A função mais importante do analisador sintático, no entanto, é, a partir dos átomos, levantar a estrutura sintática do texto-fonte. Ou seja, a partir da sequência de *tokens* gerados pela análise léxica, construir a árvore sintática do texto-fonte, relacionando as construções às gramáticas que definem a linguagem. A árvore gerada pode ser parte da árvore completa do código-fonte e pode apresentar modificações, como eliminação de redundâncias e elementos supérfluos, tal árvore é denominada árvore sintática abstrata.

Logo, um analisador sintático deve verificar se a sintaxe está correta, recuperar-se de erros sintáticos e continuar o processo, e gerar uma árvore sintática do texto-fonte.

Para a construção desse analisador, foi usado o modelo baseado no autômato de pilha estruturado. Para tal, a partir da sintaxe formal, geraram-se as sub-máquinas do autômato, listadas a seguir.

3.1. Lista de sub-máquinas do Autômatos de Pilha Estruturados

3.1.1. Lista de transições

Foram encontradas 12 submáquinas em nossa linguagem que são:

1. **PROGRAMA**
   initial: 0
   final: 2
   (0, "program") -> 1
   (1, **ESCOPO**) -> 2

2. **DEFINICAO**
   initial: 0
   final: 7
   (0, "deftype") -> 1
   (1, IDENTIFICADOR) -> 2
   (2, "{") -> 3
   (3, TIPO) -> 4
   (4, IDENTIFICADOR) -> 5
   (5, ",") -> 3
   (5, "}") -> 6
   (6, ";") -> 7

3. **FUNCAO**
   initial: 0
   final: 7
   (0, "function") -> 1
   (1, IDENTIFICADOR) -> 2
   (2, "(") -> 3
   (3, TIPO) -> 4
   (4, IDENTIFICADOR) -> 5
   (5, ",") -> 3
   (5, ")") -> 6
   (6, **ESCOPO**) -> 7
   (6, ":") -> 8
   (8, TIPO) -> 9
   (9, **ESCOPO**) -> 7

4. **ESCOPO**
   initial: 0
   final: 2
   (0, "{") -> 1
   (1, **COMANDO**) -> 1
   (1, "}") -> 2

5. **VARIAVEL**
   initial: 0
   final: 5
   (0, TIPO) -> 1
   (1, IDENTIFICADOR) -> 2
   (1, "[") -> 3
   (2, "=") -> 4
   (2, ";") -> 5
   (2, ",") -> 6
   (3, "]") -> 7
   (3, INTEIRO) -> 8
   (4, **EXPRESSAO**) -> 10
   (6, IDENTIFICADOR) -> 9
   (7, IDENTIFICADOR) -> 11
   (8, "]") -> 6
   (9, ";") -> 5
   (9, ",") -> 6
   (10, ";") -> 5
   (11, "=") -> 12
   (12, "{") -> 13
   (13, **EXPRESSAO**) -> 14
   (14, ",") -> 13
   (14, "}") -> 10

6. **CONDICIONAL**
   initial: 0
   final: 7
   (0, "(") -> 1
   (0, "if") -> 2
   (1, **EXPRESSAO**) -> 3
   (2, "(") -> 4
   (3, ")") -> 5
   (4, **EXPRESSAO**) -> 6
   (5, **ESCOPO**) -> 7
   (6, ")") -> 8
   (8, **ESCOPO**) -> 9
   (9, "elsif") -> 2
   (9, "else") -> 5

7. **REPETICAO**
   initial: 0
   final: 4
   (0, "(") -> 1
   (1, **EXPRESSAO**) -> 2
   (2, ")") -> 3
   (3, **ESCOPO**) -> 4

8. **COMANDO**
   initial: 0
   final: 7
   (0, IDENTIFICADOR) -> 1
   (0, "return") -> 2
   (0, DEF_VARIAVEL) -> 3
   (0, "if") -> 4
   (0, "while") -> 5
   (1, "=") -> 2
   (1, "(") -> 6
   (2, **EXPRESSAO**) -> 9
   (3, **VARIAVEL**) -> 7
   (4, **CONDICIONAL**) -> 7
   (5, **REPETICAO**) -> 7
   (6, **EXPRESSAO**) -> 8
   (6, ")") -> 9
   (8, ",") -> 10
   (8, ")") -> 9
   (9, ";") -> 7
   (10, **EXPRESSAO**) -> 8

9. **EXPRESSAO**
   initial: 0
   final: 1, 3
   (0, **TERMODECIMAL**) -> 1
   (1, OPERADOR) -> 2
   (2, **EXPRESSAO**) -> 3

10. **TERMODECIMAL**
    initial: 0
    final: 1, 3
    (0, **TERMOCOMP**) -> 1
    (1, OPERADORBOOL) -> 2
    (2, **TERMODECIMAL**) -> 3

11. **TERMOCOMP**

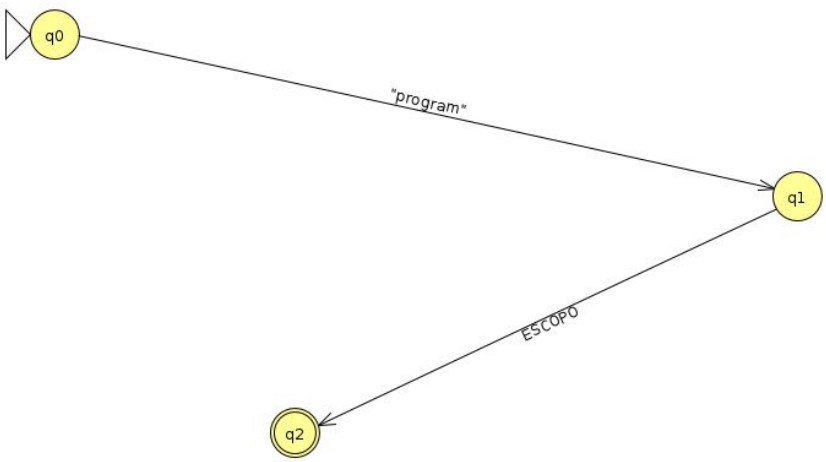
initial: 0
final: 1, 3
(0, **TERMOPRIMARIO**) -> 1
(1, OPERADORCOMP) -> 2
(2, **TERMOCOMP**) -> 3

12. TERMOPRIMARIO

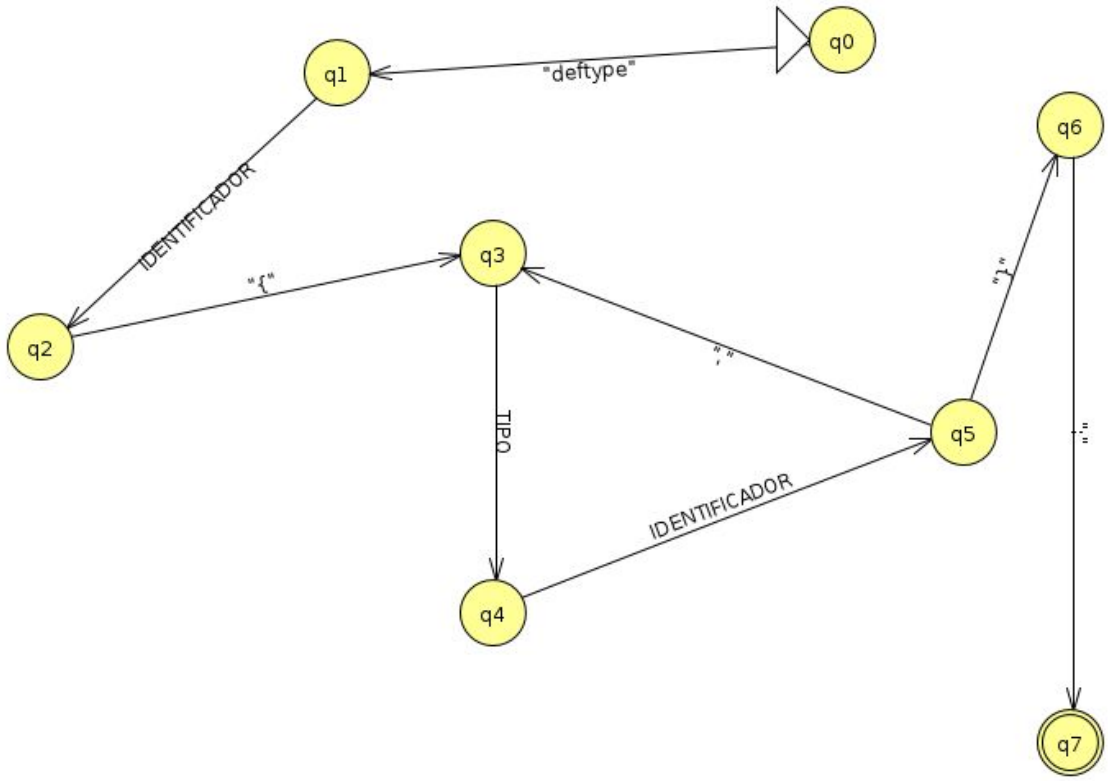
initial: 0
final: 1, 3
(0, IDENTIFICADOR) -> 1
(0, "(") -> 2
(0, DECIMAL) -> 3
(0, FLOAT) -> 3
(0, BOOL) -> 3
(0, STRING) -> 3
(1, "(") -> 4
(2, **EXPRESSAO**) -> 7
(4, **EXPRESSAO**) -> 5
(4, ")") -> 3
(5, ",") -> 6
(5, ")") -> 3
(6, **EXPRESSAO**) -> 5
(7, ")") -> 3

3.1.2. Lista de autômatos

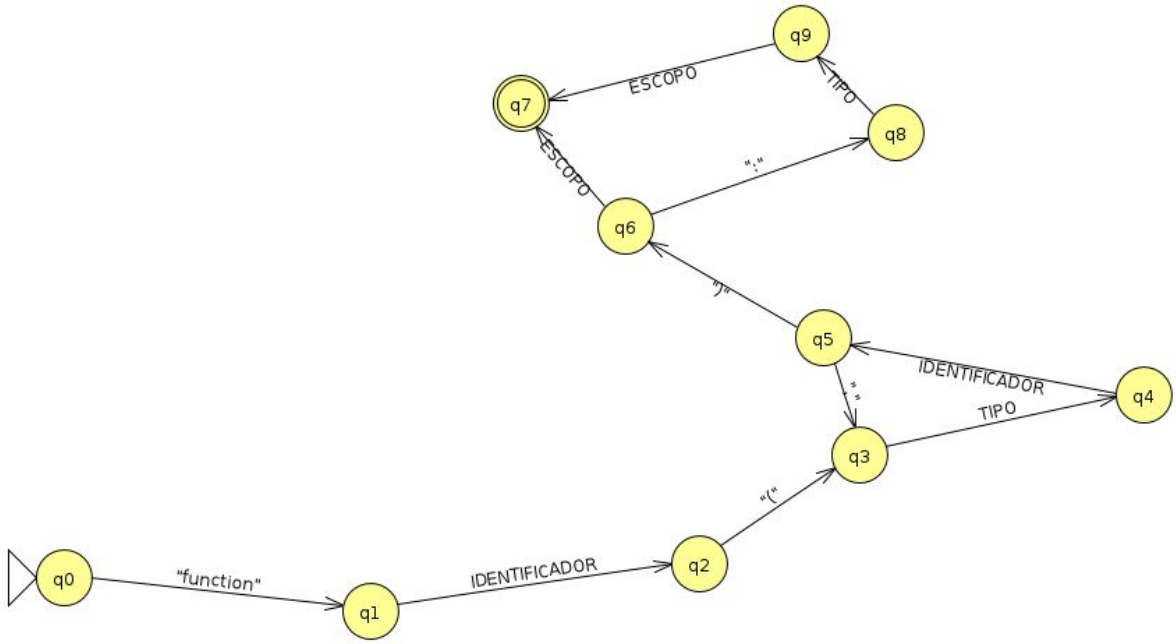
1. PROGRAMA



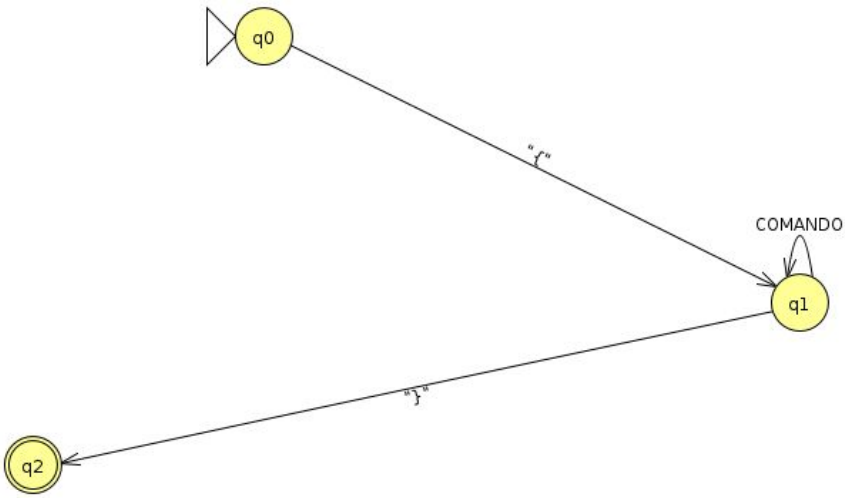
2. DEFINICAO


q0
q1
"deftype"
IDENTIFICADOR
q2
"{"
q3
TIPO
q4
IDENTIFICADOR
","
q5
"}"
q6
";"
q7

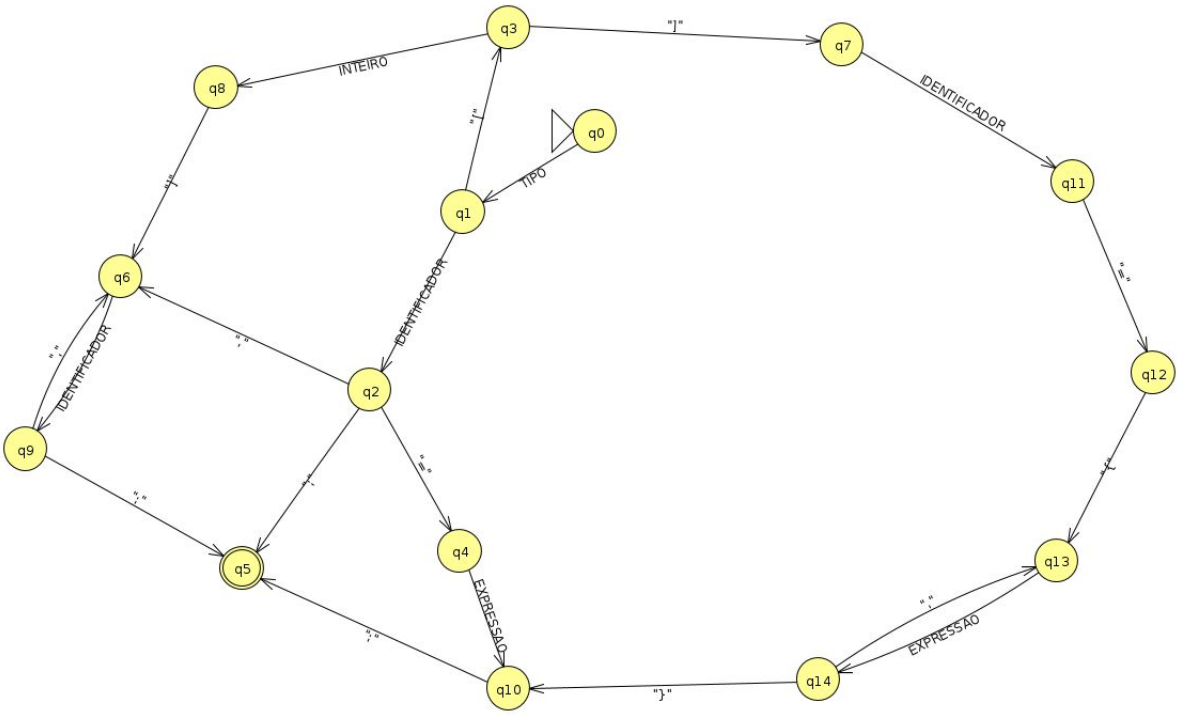
3. FUNCAO



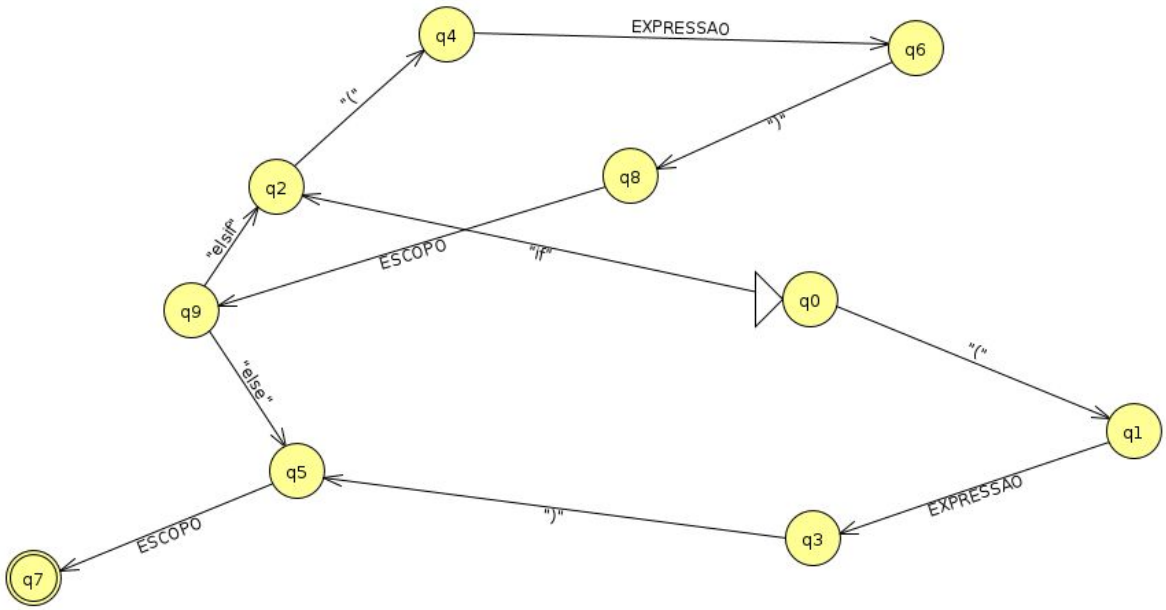
4. ESCOPO



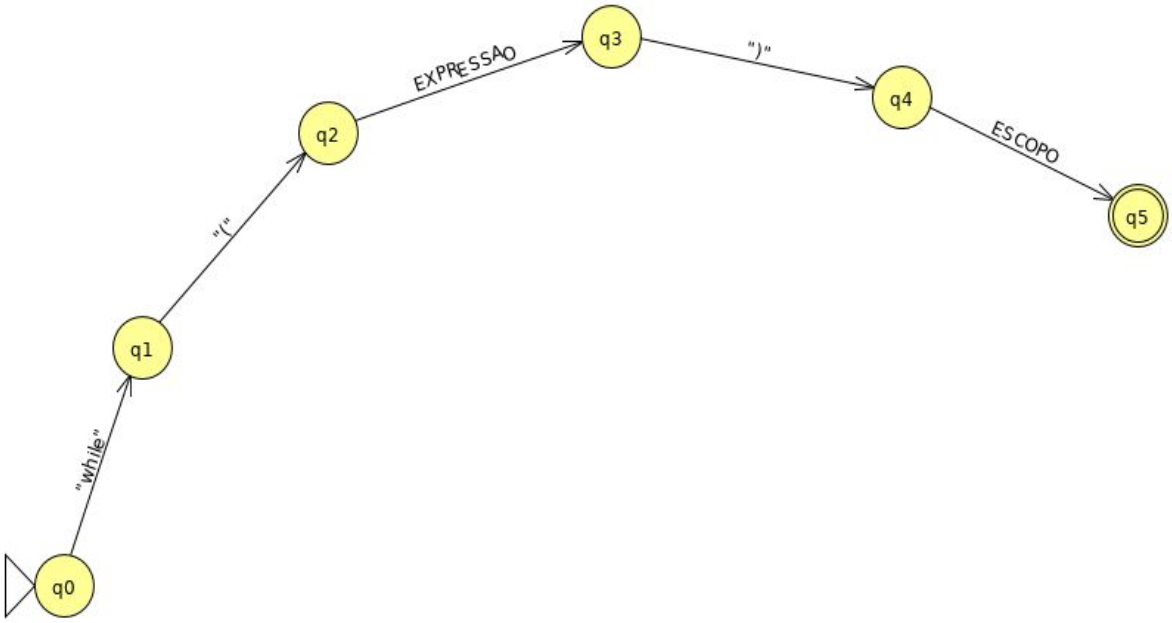
5. VARIAVEL



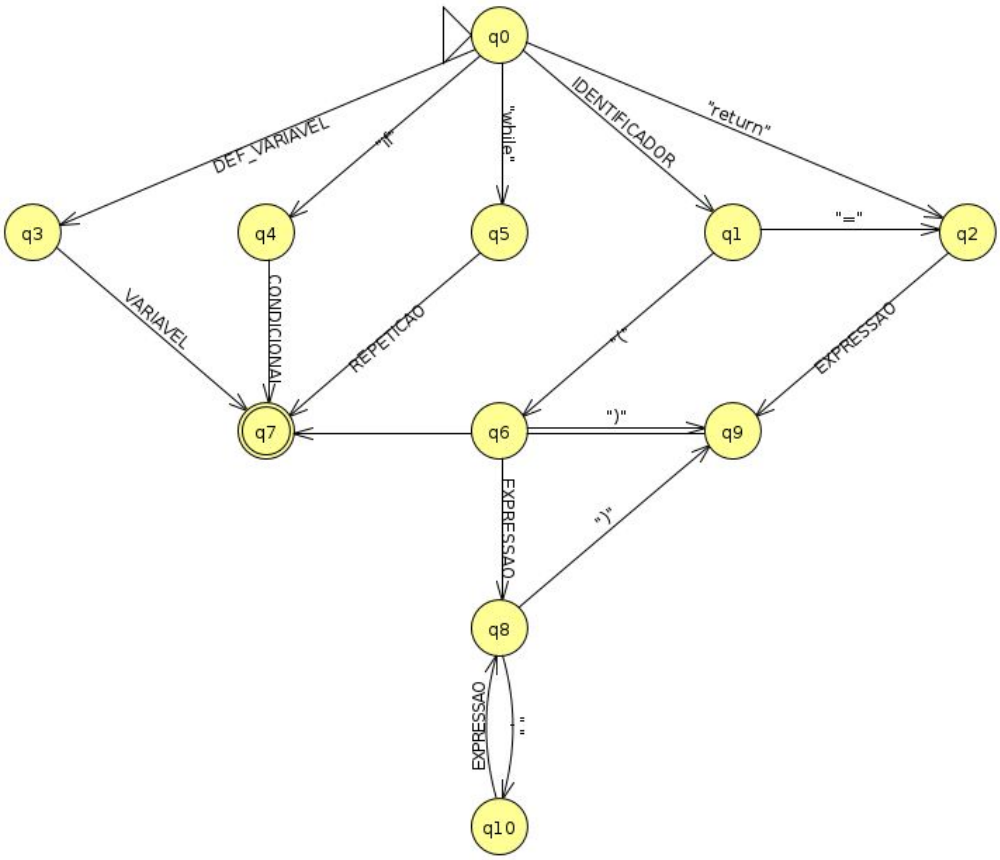
6. CONDICIONAL



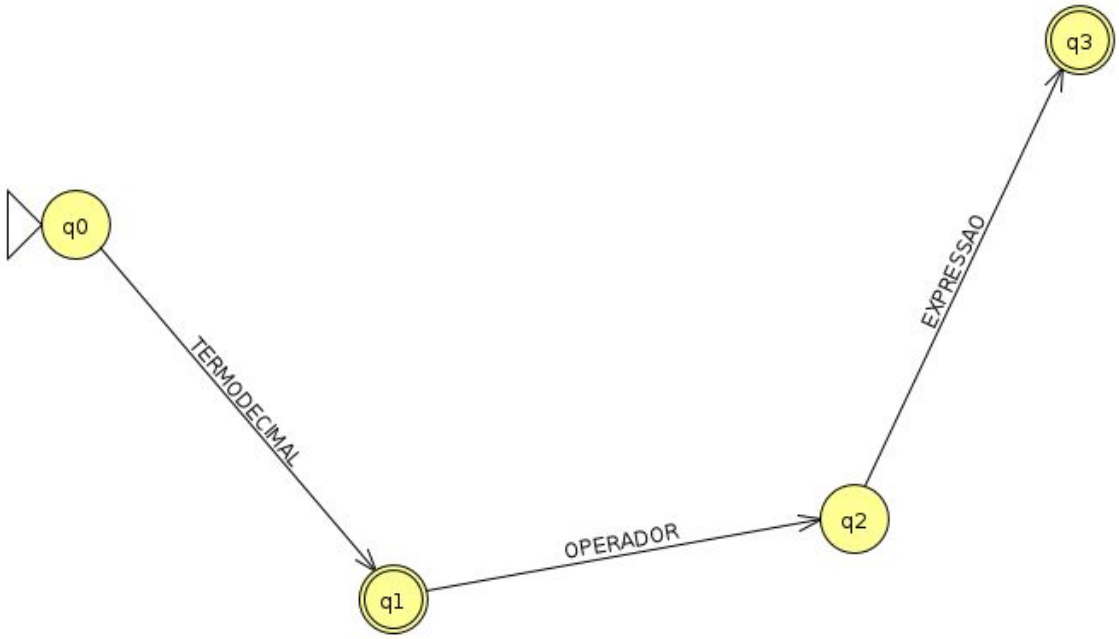
7. REPETICAO



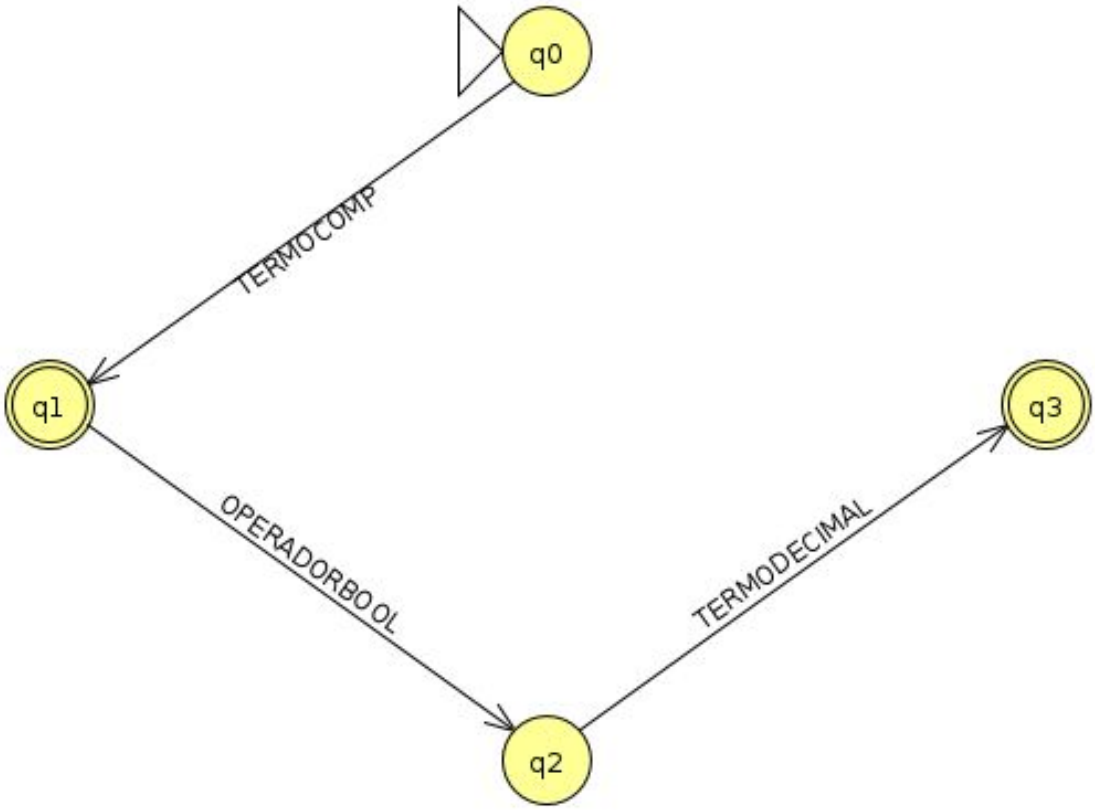
8. COMANDO



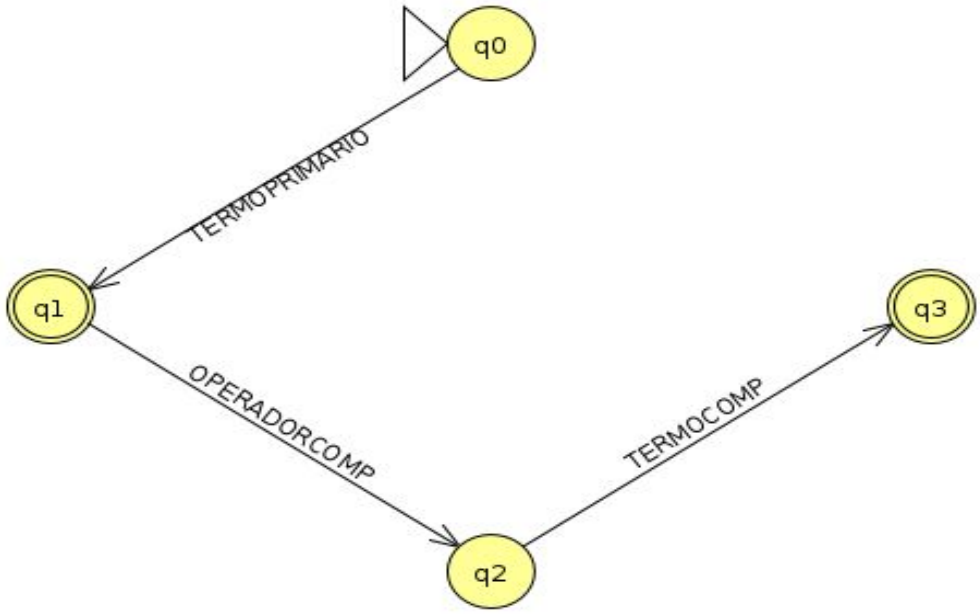
9. EXPRESSAO



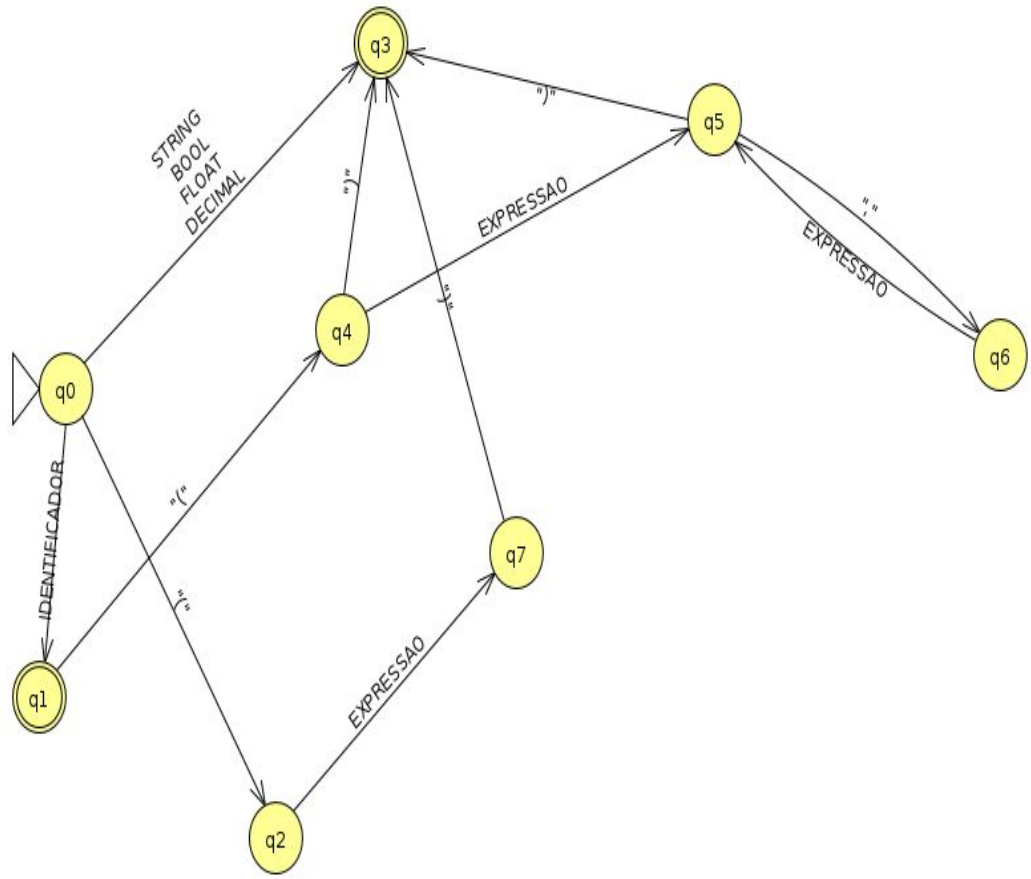
10. TERMODECIMAL



11. TERMOCOMP



12. TERMOPRIMARIO



3.2. Implementação

A implementação do analisador léxico possui o arquivo *syntatic.c* e seu header *syntatic.h*. Ele possui uma função principal que está exposta chamada *compile* que possui como parâmetro o texto-fonte de entrada. Essa função:

1. Inicializa o estado do analisador sintático
2. Verifica se deve pegar outro token
3. Se sim, chama o analisador léxico e pega o nextToken. Caso ele seja nulo, interrompe-se a execução da análise e retorna.
4. Em seguida, atualiza-se o estado do autômato atual em função do estado do autômato atual e do token.
5. Caso seja necessário chamar uma sub-máquina, empilha-se a máquina atual e define-se o estado de retorno, altera-se o estado interno da analisador para utilizar a nova sub-máquina a partir do estado inicial.
6. Ao chegar em um estado final, desempilha-se a pilha de sub-máquinas e atualiza-se o estado interno do analisador.
7. Verifica-se se não houve erro na análise, e repete-se a partir de 1 caso não exista erros.

Para implementar essa função, foi necessário definir o estado interno do analisador sintático, com o estado atual do autômato, token atual e autômato atual e também uma pilha de sub-máquinas bem como funções de empilhar e desempilhar que além de alterar a pilha, atualizam o estado do analisador. Foi necessário também implementar funções que representassem os autômatos listados em 3.2.2.

Para integrar ao compilador, substituiu-se o loop do analisador léxico por uma chamada à função principal do analisador sintático.

4. Ambiente de execução

O ambiente de execução é composto pela simulação de um processador muito simples. Este simulador apresenta um conjunto de elementos de armazenamento e dados, são eles: memória principal, acumulador e registradores auxiliares.

Este processador contém somente 9 instruções que são extremamente simples. Por exemplo, este processador não contém uma uma unidade de ponto flutuante. Entretanto, este contém as funções matemáticas básicas, sendo possível construir programas como o cálculo de fibonacci e fatorial.

4.1. Instruções da linguagem de saída

Operação	Mnemônico	Operando	Descrição
Jump	JP	Endereço/Rótulo de	Desvio incondicional

		desvio	
Jump if Zero	JZ	Endereço/Rótulo de desvio	Desvio se valor no acumulador é zero
Jump if Negative	JN	Endereço/Rótulo de desvio	Desvio se valor no acumulador é negativo
Load Value	LV	Constante de 12 bits	Deposita uma constante no acumulador
Add	+	Endereço/Rótulo do operando	Soma o conteúdo do acumulador com o operando
Subtract	-	Endereço/Rótulo do subtraendo	Subtração do conteúdo do acumulador com o subtraendo
Multiply	*	Endereço/Rótulo do multiplicador	Multiplicação do conteúdo do acumulador com o multiplicador
Divide	/	Endereço/Rótulo do divisor	Divisão do conteúdo do acumulador com o divisor
Load	LD	Endereço/Rótulo do dado	Copia valor contido no endereço de memória para acumulador
Move to Memory	MM	Endereço/Rótulo de destino do dado	Copia valor do acumulador para a memória
Subroutine Call	SC	Endereço//Rótulo do subprograma	Desvio para subprograma
Return from Subroutine	RS	Endereço/Rótulo de retorno	Retorno de subprograma
Halt Machine	HM	Endereço/Rótulo do desvio	Parada
Get Data	GD	Dispositivo de E/S	Entrada
Put Data	PD	Dispositivo de E/S	Saída
Operating System	OS	Constante	Chamada de Supervisor

4.2. Pseudo-instruções da linguagem de saída

Pseudoinstrução	Descrição
@	Recebe um operando numérico, define o endereço da instrução seguinte, uma origem absoluta para o código a ser gerad
K	Define área preenchida por uma constante, o operando numérico tem o valor da constante de 2 bytes (em hexadecimal)
$	Define um bloco de memória com número especificado de bytes, o operando numérico define o tamanho da área a ser reservada (em bytes)
#	Define o fim do texto fonte
&	Define uma origem relocável para o código a ser gerado, o operando é o endereço em que o próximo código se localizará (relativo à origem do código corrente)
>	Define endereço simbólico de entrada (Entry Point)
<	Define um endereço simbólico que referencia um entry-point externo

4.3. Características gerais

O ambiente de execução é a MVN disponibilizada, que simula o Modelo de Von Neumann como um processador simples composto por: Memória de 4096 posições e endereços de 12 bits, Acumulador e Registradores Auxiliares.

Na memória principal, são armazenadas as instruções dos programas e os seus dados. O acumulador é um registrador especial utilizado para operações aritméticas e lógicas, é utilizado também nas operações de desvio condicional.

Os registradores auxiliares são utilizados em operações intermediárias e estão descritos na tabela a seguir:

Registrador Auxiliar	Descrição
Registrador de Dados da Memória (MDR)	Utilizado para tráfego de dados entre a memória e outros elementos da MVN
Registrador de Endereço de Memória (MAR)	Contém a origem ou destino dos dados que se encontram no MDR
Registrador de Endereço de Instrução (IC)	Armazena a próxima instrução a ser executada pela máquina
Registrador de Instrução (IR)	Representa a instrução em execução, é composto de duas parcelas: o código de operação (OP) e o operando da instrução (OI).

As variáveis são acessadas diretamente pela memória. Utiliza-se complemento de 2 para determinação de sinal do dado.

Nas chamadas de subrotina, deve-se guardar o endereço de IC (que será a instrução de retorno da subrotina). Após armazenar o endereço do IC, coloca-se nesse mesmo registrador o endereço de execução da subrotina. Assim que a subrotina é executada, para retornar, basta retornar o endereço armazenado ao IC.

5. Tradução de comandos semânticos

5.1. Tradução de estruturas de controle de fluxo

5.1.1. If-then

Linguagem de Entrada	**Linguagem de Saída**

if (condicao) {
  # comandos
}
	LD condicao
	JZ END_IF
	<comandos>
	END_IF

5.1.2. If-then-else

Linguagem de Entrada	**Linguagem de Saída**

if (condicao) {
  #comandos_if
} else {
  #comandos_else
}
	LD condicao
	JZ ELSE
	<comandos_if>
	JP END_IF
	ELSE <comandos_else>
	END_IF

5.1.3. While

Linguagem de Entrada	**Linguagem de Saída**

while (condicao) {
  #comandos
}
	LOOP LD condicao
	JZ END_WHILE
	<comandos>
	JP LOOP
	END_WHILE

5.2. Tradução de comandos imperativos

5.2.1. Atribuição de valor

Linguagem de Entrada	**Linguagem de Saída**
identificador = valor;	identificador K=0 LD valor MM identificador

5.2.2. Leitura (entrada)

Linguagem de Entrada	**Linguagem de Saída**
int numero = read();	numero K =0 SC read LD read_number MM numero

Subrotina

```
read_number   K =0 ; Variável de retorno
r_negative    K =0 ; Número digitado é negativo
zero          K =0
one           K =1
num_256       K =256
minus_sign    K =45
ascii_cr      K /D
ascii_lf      K /A
ascii_offset  K /30
r_temp        K =0
r_temp2       K =0

read          JP /0000
              LD zero ; Inicialização
              MM read_number
              MM r_negative
              MM r_temp
              MM r_temp2
              GD /0000 ; Leitura de número negativo
              MM r_temp ; Guarda caracteres lidos
              / num_256
              MM r_temp2
              - minus_sign
              JZ is_negative ; Verifica se número digitado é negativo
              JP char1
```

```
is_negative     LD one ; Carrega o i_negative com FFFF
                MM r_negative
                JP char2
r_loop          GD /0000 ; Loop de leitura
                MM r_temp ; Guarda caracteres lidos
                / num_256
                MM r_temp2
char1           - ascii_cr ; Primeiro caractere
                JZ r_end ; Verifica se é o fim
                LD r_temp2
                - ascii_lf
                JZ r_end ; Verifica se é o fim
                LD read_number ; Não é o último caracter
                * ten ; Aumenta uma dezena no resultado
                MM read_number
                LD r_temp2 ; Converte caracter lido em número
                - ascii_offset
                + read_number
                MM read_number ; Atualiza o resultado de retorno
char2           LD r_temp2 ; Segundo caracter
                * num_256
                MM r_temp2
                LD r_temp
                - r_temp2
                MM r_temp2
                - ascii_cr ; Verifica se é o fim
                JZ r_end
                LD r_temp2
                - ascii_lf
                JZ r_end ; Verifica se é o fim
                LD read_number
                * ten
                MM read_number
                LD r_temp2 ; Converte caracter lido em número
                - ascii_offset
                + read_number
                MM input_number ; Atualiza o resultado de retorno
                JP i_loop ; Proximo caracter
r_end           LD r_negative ; Transforma em negativo se negativo
                JZ r_return
                LD zero
                - read_number
                MM read_number
i_return        LD read_number
                RS input ;
```

5.2.3. Impressão (saída)

Linguagem de Entrada	**Linguagem de Saída**
write(numero);	LD numero MM write_number SC write

Subrotina

```
write_number    K =0      ; Número a ser impresso
minus_one       K /FFFF ; valor -1
one             K =1      ; valor 1
ten             K =10     ; valor 10
minus_sign      K =45     ; Sinal de menos em ASCII
ascii_offset    K =48     ; Offset para o código de um número na tabela ASCII
w_temp1         K =0      ; Guarda o valor da última dezena
w_temp2         K =10     ; Indicador da dezena

write           JP /0000
                LD write_number
                JN w_negative       ; Número negativo
                JP w_start          ; Número posítivo
w_negative      LD minus_sign       ; imprime "-"
                PD /0100
                LD minus_one
                - write_number
                + one               ; inverte o número
                MM write_number
w_start         MM w_temp1
w_loop          LD write_number
                / w_temp2
                JZ w_print          ; É o número mais a esquerda?
                MM w_temp1          ; Guarda a última casa decimal visitada
                LD w_temp2
                * ten
                MM w_temp2          ; Próxima casa decimal
                JP w_loop
w_print         LD w_temp1          ; Número a ser impresso
                + ascii_offset
                PD /0100
                LD w_temp2
                / ten
                MM w_temp2
                - one
                JZ w_end            ; Verifica se é o último número
                LD w_temp1
                * w_temp2
                MM w_temp1
                LD write_number
                - w_temp1           ; Atualiza o número para impressão
                MM write_number
                MM w_temp1
```

```
                LD ten
                MM w_temp2      ; Próxima dezena
                JP w_loop       ; Próximo caractere
w_end           RS output
```

5.2.4. Chamada de subrotina

Linguagem de Entrada	**Linguagem de Saída**
funcao(a, b, c, ...)	LD func_size ; Carrega o tamanho do R.A. MM call_stack_size LD return_adr ; Carrega o endreço de retorno do R.A. MM call_stack_adr SC create_call_stack ; Cria R.A. LD 1 ; Carrega parâmetro da função MM arg_pos SC store_cs_pos SC função ; Executa a função

Subrotina do ambiente de execução

```
two                 K =2
STOP                K /0FF0
load_instruction    LD /0000   ; Instrução para o acesso indireto
store_instruction   MM /0000 ; Instrução para store indireto
arg_pos             K =0       ; Posição do argumento da função

load_cs_pos         JP /0000   ; Ponto de entrada da subrotina
                    LD STOP  ; Carrega topo da pilha do R.A.
                    - two ; Diminui um endereço na pilha do R.A.
                    - arg_pos ; Accumulador com o endereço correto
                    + load_instruction ; Cria nova instrução
                    MM instruct ; Armazena como proxima instrução
instruct            K /0 ; Reservado para guardar a instrução recém-montada
                    RS load_cs_pos
store_cs_pos        JP /0000
                    LD STOP ; Carrega topo da pilha do R.A.
                    - two ; Diminui um endereço na pilha do R.A.
                    - arg_pos ; Accumulador com o endereço correto
                    + store_instruction ; Here's the magic: Cria instrução nova!
                    MM instruct2 ; Armazena como proxima instrução
instruct2           K /0 ; Reservado para guardar a instrução recém-montada
                    RS store_cs_pos
call_stack_size     K =0
call_stack_adr      K =0400
create_call_stack   JP /0000
```

```
LD STOP
+ two
MM STOP
LD zero
MM arg_pos
LD call_stack_adr
SC store_cs_pos
LD STOP
+ call_stack_size
MM STOP
RS create_call_stack
```

5.3. Exemplo de programa traduzido

Na tabela abaixo, segue um programa exemplo que calcula o fatorial de um número digitado.

Cálculo de Fatorial

```
program {
        int fat;
        int num = read();

        if(num < 0){
                fat = 0;
        }else{
                fat = 1;
                while(num > 0){
                        fat = fat * num;
                        num = num -1;
                }
        }
        write(fat);
}
```

Na tabela abaixo, encontra-se o código gerado (programa traduzido) pelo compilador.

Cálculo de Fatorial (Programa traduzido)

```
one     K=1

        @ /0200
temp    K =0
fat     K =0 ; int fat
num     K =0 ; num = read()
```

```
                SC write
                LD read_number
                MM num
                LD num ; (num < 0)
                - zero
                JN TRUE1
FALSE1          LV =0
                JP END1
TRUE1           LV =1
END1            MM temp
                LD temp ; if () {} else{}
                JZ ELSE1
                LV =0 ; fat = 0
                MM fat
                JP END_IF1
ELSE1           LV =1 ; fat = 1
                MM fat
LOOP1           LV =0 ; while(num > 0) {}
                - num
                JN TRUE2
FALSE2          LV =0
                JP END2
TRUE2           LV =1
END2            MM temp
                LD temp
                JZ END_WHILE1
                LD fat ; fat = fat*num
                * num
                MM fat
                LD num ; num= num - 1
                - one
                MM num
                JP LOOP1
END_WHILE1 JP END_IF1
END_IF1     LD fat
                MM write_number
                SC write
                HM /00
```