ESCOLA POLITÉCNICA DA UNIVERSIDADE DE SÃO PAULO

PCS 2056 – Linguagens e Compiladores

Horae

Projeto do Compilador

Professor:
Prof. Dr. João José Neto

Equipe:
Fernando de Oliveira Gil **4938991**
Fernando Henrique Ginês **3726158**

7/12/2007

Índice

1. Introdução

"As horae (vulgarmente designadas como horas pela corrupção do vocábulo original) constituíam um grupo de deusas gregas que presidiam às estações dos anos. Eram filhas de Zeus e Têmis são: Irene (paz), Dice (justiça) e Eunômia (disciplina); estas são as Horas mais velhas e estão ligadas a legislação e ordem natural, sendo uma extensão dos atributos de sua mãe Têmis. Eumônia está relacionada com a representação da divindade da justiça. Temis e Dice elucidam o lado ético do instinto, a voz miúda e calma no seio do impulso. Dike para a humanidade é a função de base institual muito sintônica com o que chama de instinto para reflexão.

Existem mais nove Horas que são guardiãs da ordem natural, do ciclo anual de crescimento da vegetação e das estações climaticas anuais. (Talo, Carpo, Auxo, Acme, Anatole, Disis, Dicéia, Eupória, Gimnásia)." [1]

Neste trabalho descrevemos a linguagem Horae, em homenagem às deusas citadas no texto acima e o seu compilador para a máquina de Von Neumann.

Procuramos descrever uma linguagem simples, em que fosse possível escrever qualquer programa de computador e que não necessitasse de uma extensa biblioteca de runtime, em face da limitação de memória da máquina utilizada.

A máquina de Von Neumann (Mvn), utilizada aqui através do seu Simulador elaborado para a disciplina PCS2024 - Laboratório de Fundamentos de Engenharia de Computação [2], foi escolhida por ser uma máquina simples de uso geral. Além disso, parte do grupo já havia trabalhado diretamente com este simulador e com esta linguagem de máquina, o que facilitou a implementação da geração de código do compilador.

Sendo assim, o objetivo do trabalho é descrever o projeto de um compilador da linguagem Horae para a Máquina de Von Neumann.

2. Definição da Linguagem

A linguagem Horae elaborada possui as especificações determinadas em aula. Assim, os principais componentes da mesma são:

- estrutura do programa
- declaração de variáveis
- variáveis simples do tipo caracter, booleano e inteiro
- variáveis indexadas – vetor e matriz
- comandos de atribuição
- comandos de entrada
- comandos de saída
- comandos condicionais
- comandos iterativos
- expressões aritiméticas
- expressões booleanas

A seguir faremos uma descrição funcional da linguagem listando e detalhando os seus principais recursos e restrições de acordo com o que foi projetado.

2.1. Recursos da Linguagem

2.1.1. Estrutura do programa

O programa se inicia com a palavra chave START e termina com a palavra chave END. Todas as declarações, de variáveis ou funções, e comandos estarão localizados entre o START e o END.

2.1.2. Declaração de variáveis

Após a palavra chave START inicia-se a declaração de variáveis, no nosso caso a declaração é opcional. A declaração da variável é feita especificando o tipo da variável, seguida de uma cadeia de caracteres que será seu identificador e o caractere ";" apontando o final da declaração. Não é possível a declaração de variáveis distintas com o mesmo identificador.

2.1.3. Variáveis simples

A nossa linguagem aceita três tipos diferentes de variáveis: inteiros (INT), booleano (BOOLEANO) e caracter (CHAR). As variáveis inteiras da linguagem

ocupam áreas de 16 bits, sendo possível armazenar números inteiros situados na faixa de 0 até 65535. Tanto as variáveis do tipo booleana quando as do tipo caracter irão ocupar dois bytes em memória, dos quais só será utilizado o menos significativo. A utilização de 2 bytes ao invés de 1 se deve às restrições da linguagem de montagem.

As operações aritméticas possíveis com as variáveis do tipo inteiro são: subtração, multiplicação e divisão. Vale-se ressaltar que a precedência dos operadores de multiplicação e divisão sobre os demais operadores é respeitada, através de um mecanismo de duas pilhas utilizadas durante a construção do analisador semântico. Mais detalhes desse mecanismo serão apresentados adiante.

2.1.4. Variáveis indexadas – vetor e matriz

As duas estruturas projetadas são vetores e matrizes. O objetivo é que se crie uma forma de acesso de leitura e escrita aos dados contidos nestas estruturas, mas não será implementado nenhuma operação entre elas.

2.1.5. Comandos de atribuição

São comandos utilizados para a alteração do valor de uma variável, seja este vindo de uma outra variável ou através do resultado de expressões. Na linguagem definida o comando de atribuição ocorre através de um identificador da variável que receberá o valor, seguido de um sinal de "=", a expressão que irá definir o novo valor da variável destino, seguido do caractere ";" para indicar o fim do comando.

2.1.6. Comandos de entrada

O comando de entrada implementado aqui é o INPUT que lê um byte e guarda esse valor no endereço fornecido como argumento do comando.

2.1.7. Comandos de saída

O comando de saída aqui projetado é o OUTPUT que imprime na saída um valor ou um resultado armazenado em uma variável (local de memória) acessado pelo programa.

2.1.8. Comandos condicionais

O comando condicional projetado é do tipo IF-THEN-ELSE-ENDIF. Ele testa uma condição, e caso a mesma seja verdadeira, executa o bloco de comandos definidos entre as palavras chaves THEN e ELSE. Caso contrário, executará o bloco de comandos entre as palavras chaves ELSE e ENDIF. A forma IF-THEN-ENDIF também é permitida em nosso projeto.

2.1.9. Comandos iterativos

Este tipo de comando permite ao usuário executar um bloco de comandos repetidamente enquanto uma condição testada for verdadeira. Em nossa linguagem

o comando iterativo é o WHILE e executará o bloco de comandos contidos entre as palavras chaves DO e ENDW enquanto a condição testada for verdadeira.

2.1.10. Expressões aritméticas

A linguagem possui ainda suporte a expressões aritméticas, que podem ser realizadas em atribuições e condições. Pode-se utilizar parênteses para mudar a precedência dos operadores. Os operadores são +, -, * e /.

2.1.11. Expressões booleanas

As expressões booleanas são utilizadas na condição dos comandos de iteração e de condição, para determinar a ação a ser executada. Contém os operadores AND, OR, NOT, ==, !=, <, >, <= e >=. Pode-se utilizar parênteses para mudar a precedência dos operadores.

2.2. Notação BNF

Abaixo temos a linguagem em notação BNF:

```
<Programa> ::= START <seqüência de declarações> <declaração de
funções> <seqüência de comandos> END

<seqüência de declarações> ::= <declarações>

<declarações> :: = <declaração>; | <declarações> <declaração>; | ε

<seqüência de comandos> ::= <comando>; | <seqüência de comandos>

<comando>; | ε

<comando> ::= <atribuição>
                        | <entrada>
                        | <saída>
                        | <iteração>
                        | <condicional>

<declaração> ::= <tipo> <identificador> | <tipo> <vetor> | <tipo>
<matriz>

<tipo> ::= INT | CHAR | BOOLEAN

<identificador> ::= <letra> | <identificador><letra> |
<identificador><número>

<vetor> ::= <identificador> [<número>]

<matriz> ::= <identificador> [<número>][<número>]

<letra> ::= A | B | C | D | E | F | G | H | I | J | K | L | M | N | O
| P | Q | R | S | T | U | V | Y | X | W | Z

<número> ::= <digito> | <número><digito>

<digito> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

```
<atribuição> ::= <identificador> = <exp>
                        | <vetor> = <exp>
                        | <matrix> = <exp>

<exp> ::= <exp> + <termo>
| <exp> - <termo>
| <termo>

<termo> ::= <termo> * <fator>
| <termo> / <fator>
| <fator>

<fator> ::= <identificador>
| <vetor>
| <booleano>
| <função>
| ( <exp> )
| - <exp>
| <matriz>
| <número>

<booleano> ::= TRUE | FALSE

<condição> ::= <exp> <op booleano> <exp> | <exp booleana> | NOT <exp
booleana>

<op booleano> ::= < | > | >= | <= | == | !=

<exp booleana> ::= <condição> <op lógico> <condição>
                        | <booleano>
                        | (<condição>)

<op lógico> ::= AND | OR

<entrada> ::= INPUT <vetor> | INPUT <identificador>

<saída> ::= OUTPUT <exp> | OUTPUT <cadeia>

<cadeia> ::= '<caracteres>'

<caracteres> :: = <caractere> | <caracteres><caractere>

<caractere> ::= A | B | C | D | E | F | G | H | I | J | K | L | M | N
| O | P | Q | R | S | T | U | V | Y | X | W | Z|   | _ | 0 | 1 | 2 | 3
| 4 | 5 | 6 | 7 | 8 | 9 | + | - | * | / | =

<iteração> :: = WHILE (<condição>) DO < seqüência de comandos> ENDW

<condicional> ::= IF (<condição>) THEN <seqüência de comandos> ENDIF |
IF (<condição>) THEN <seqüência de comandos> ELSE <seqüência de
comandos> ENDIF

<declaração de funções> ::= <declaração de função>
                  | <declaração de funções> <declaração de função> | ε

<declaração de função> ::= FUNCAO <tipo> <identificador>(<declara
parâmetros>) { <seqüência de declarações> <seqüência de comandos>
<retorno_função> }
```

```
<declara parâmetros> ::= <declaração> | <declara parâmetros>,
<declaração>| ε

<função> ::= <identificador>(<parâmetros>)

<parâmetros> ::= <exp> | <parâmetros>, <exp> | ε

<retorno_função> ::= RETURN <exp>;
```

2.3. Notação de Wirth

A partir da notação BNF, criamos a descrição em notação de Wirth abaixo:

```
programa = "START" {declaracao ";"} {declaracao_de_função ";"}
{comando ";"} END.

comando = (atribuicao | entrada | saida | iteracao | condicional).

declaracao = tipo (identificador | vetor | matriz).

tipo = ("INT" | "CHAR" | "BOOLEAN").

identificador = letra {letra | digito}.

vetor = identificador "[" numero "]".

matriz = identificador "[" numero "][" numero "]".

letra= ("A" | ... | "Z") .

numero= digito {digito}.

atribuicao = ( identificador | vetor | matrix ) "=" exp.

exp = (exp ("+"|"-")) termo.

termo = (termo ("*" | "/")) fator.

fator = ( identificador | vetor | matrix | booleano | funcao | "(" exp
")" | "-" exp| numero).

bool = ("TRUE" | "FALSE").

op_bool = "<" | "<" | "==" | "!=" | ">=" | "<=".

condicao = (exp op_bool exp) | {"NOT"} exp_bool .

op_logico = "AND" | "OR" .

exp_bool = (condicao op_logico condicao) | bool | ( "(" condicao ")"
).

entrada = "INPUT" (vetor | identificador | matriz ).

saida = "OUTPUT" (exp | cadeia ).

cadeia = "'" caractere {caractere} "'".
```

```
caractere = ("A" | ... | "Z" | "0" | ... | "9" | " " | "+" | "-" | "_"
| "/" | "*" | "=" ) .

iteracao = "WHILE" "(" condicao ")" "DO" { comando ";" } "ENDW".

condicional = "IF" "(" condicao ")" "THEN" { comando ";" } [ "ELSE" {
comando ";" }] "ENDIF".

declaracao_de_funcao = "FUNCAO" tipo identificador "(" [declaracao {
"," declaracao}] ")" "{" {declaracao ";"} {comando ";"} "RETURN" exp
";""}".

funcao = identificador "(" [exp { "," exp}] ")".
```

3. Manual

Os arquivos executáveis para a execução de testes com o compilador estão disponíveis no CD-ROM anexo. Nós utilizamos o montador fornecido [3] para a tradução do pseudo-código de montagem para linguagem de máquina. A execução do programa será feita através do simulador também fornecido [3].

3.1. Configuração do Ambiente

Para a configuração inicial do ambiente, é necessário copiar os programas executáveis para um diretório no disco. No exemplo utilizado neste manual, o diretório será:

```
D:\horae
```

Também é necessário que esteja disponível uma máquina virtual java versão 1.6 ou superior. Para verificar a versão instalada no sistema execute:

```
D:\horae>java -version

java version "1.6.0_02"
Java(TM) SE Runtime Environment (build 1.6.0_02-b06)
Java HotSpot(TM) Client VM (build 1.6.0_02-b06, mixed mode)
```

3.2. Compilação

O arquivo executável do compilador é o `horae.jar`, ele aceita um arquivo escrito na linguagem Horae como entrada, e gera um arquivo `.asm` com o programa escrito na linguagem de montagem da Mvn. Para compilar um programa execute o comando abaixo, a saída na tela será a lista de tokens e a indicação de estados dos autômatos que foram percorridos.

```
D:\horae>java -jar horae.jar fonte.horae
START - S
...
END - END
Programa - START - Estado Atual: 0
Proximo Estado: 1
Programa - INT - Estado Atual: 1
...
CONSTANTE - INT - EA_3 - main;
CONSTANTE - INT - EA_4 - main;
FIM
D:\horae>
```

3.3. Montagem

Para efetuar a montagem do programa, basta executar o montador passando como argumento o arquivo `.asm`, conforme abaixo:

```
D:\horae>java -jar montador.jar fonte.asm
Montador finalizou corretamente, arquivos gerados.

D:\horae>
```

O compilador também pode executar a montagem em seguida da compilação, para isso basta adicionar o argumento `m` na chamada do compilador:

```
D:\horae>java -jar horae.jar m fonte.horae
START - S
...
END - END
Programa - START - Estado Atual: 0
Proximo Estado: 1
Programa - INT - Estado Atual: 1
...
CONSTANTE - INT - EA_3 - main;
CONSTANTE - INT - EA_4 - main;
FIM
Executando montador...
Montador: Montador finalizou corretamente, arquivos gerados.
D:\horae>
```

3.4. Execução

Os programas podem ser executados através do simulador de mvn, para executá-lo faça:

```
D:\horae>java -jar mvn.jar

Estado INITIALIZE
Estado RUNSTEP
================================================================================
              PCS2302/PCS2024  Simulador da Maquina de Von Neumann
                 Versao 3.0 (c)2005  Todos os direitos reservados
  Initialize(i)  Run(r)  Load(l)  Show(w)  Programa ASCII(p)  Finalizar(f):
```

Digite `p` para carregar um programa na memória, e em seguida o nome do arquivo:

```
================================================================================
              PCS2302/PCS2024  Simulador da Maquina de Von Neumann
                 Versao 3.0 (c)2005  Todos os direitos reservados
  Initialize(i)  Run(r)  Load(l)  Show(w)  Programa ASCII(p)  Finalizar(f):  p
Estado ENTRADA DE PROGRAMA
  Nome do arquivo de programa? fonte.mvn
```

Uma vez com o programa carregado, passe para o modo de execução, através da seqüência abaixo. Responda sim à primeira pergunta caso queira ver o valor dos registradores a cada instrução executada. Não serão utilizadas unidades lógicas.

```
Estado RUNSTEP
================================================================================
              PCS2302/PCS2024  Simulador da Maquina de Von Neumann
                 Versao 3.0 (c)2005  Todos os direitos reservados
  Initialize(i)  Run(r)  Load(l)  Show(w)  Programa ASCII(p)  Finalizar(f):  r
```

```
Estado RUN
  Mostra valor dos registradores a cada passo do ciclo FDE (s/n)? s
  Endereco atual do IC = 0000. Entrar (novo) endereco do IC =

  Entrar unidade logica para o disco (ou ENTER para cancelar):
```

Caso tenha sido solicitada a exibição dos registradores, será possível acompanhar a execução do programa como mostrado abaixo, caso contrário somente os dados do comando OUTPUT serão exibidos.

```
IC   IR   AC  MAR  MBR   OP   RA   OI  [OI]
==== ==== ==== ==== ==== ==== ==== ==== ====
0002 8040 0030 0000 8040 0008 0000 0040 0030
0004 903e 0030 0002 903e 0009 0000 003e 0030
0006 4052 0030 0004 4052 0004 0000 0052 0000
...
0000 c000 0063 003c c000 000c 0000 0000 8040
Estado RUNSTEP
================================================================================
              PCS2302/PCS2024  Simulador da Maquina de Von Neumann
                 Versao 3.0 (c)2005  Todos os direitos reservados
  Initialize(i)  Run(r)  Load(l)  Show(w)  Programa ASCII(p)  Finalizar(f):
```

4. Analisador Léxico

O analisador léxico foi o primeiro componente do compilador a ser desenvolvido por nós. A sua função principal é subdividir o código-fonte a ser compilado em elementos identificáveis, denominados átomos, que serão consumidos pelo analisador sintático mais tarde. Podemos resumir as suas funções às seguintes:

1. Extração e classificação de átomos;
2. Tratamento de identificadores;
3. Identificação de palavras reservadas.

Uma vez que os átomos estiverem identificados e classificados corretamente, eles devem ser analisados sintaticamente para que possa ser feito o tratamento, interpretação desses átomos dispostos como estão.

4.1. Descrição do programa e sua estrutura

Para realizar o desenvolvimento do analisador léxico foi preciso atentar para a linguagem definida. Com isso criamos as tabelas de palavras reservadas e de símbolos:

Palavras Reservadas	
1	START
2	END
3	INT
4	CHAR
5	BOOLEAN
6	INPUT
7	OUTPUT
8	WHILE
9	ENDW
10	FUNCAO
11	IF
12	THEN
13	ELSE
14	ENDIF
15	AND
16	OR
17	RETURN
18	NOT
19	TRUE
20	FALSE
21	DO

Tabela 1 – Tabela de Palavras Reservadas.

Tabela de símbolos:	
1	+
2	-
3	*
4	/
5	;
6	=
7	,
8	(
9	)
10	{
11	}
12	!=
13	>
14	<
15	<=
16	>=
17	==

Tabela 2 - Tabela de Símbolos.

Com essa tabela foi possível desenvolver o autômato a seguir que representa o analisador léxico e no qual nos baseamos durante a implementação.

Nosso analisador recebe cada caracter do código e faz o seu processamento. Quando um token é gerado, o analisador ira armazená-lo em uma fila de tokens. Nós optamos pela utilização de uma fila completa com os tokens gerados e então partir para a análise sintática para facilitar a nossa implementação. A forma com que foi construído o código nos permite facilmente alterar para que o analisador sintático ao invés de armazenar todos os tokens gerados, alimente o analisador sintático passo a passo.

Abaixo está a estrutura de dados utilizada para armazenar as informações referentes a um token:

```
protected String word; Armazena os caracteres que formam o token
protected String type; Armazena a classificação do token, como por
                       exemplo "numero", "identificador" e "=".
```

5. Analisador Sintático

Com a linguagem definida, e descrita na notação de Wirth, prosseguimos com a construção das tabelas de estados a fim de encontrar as máquinas que constituem o analisador sintático, eliminando os elementos não-terminais não essenciais. A simplificação possibilitou a criação de 6 máquinas, cada uma implementada nas seguintes classes (pacote `horae.maquinas`):

- Programa.java
- Declaracao.java
- DeclaracaoFuncao.java
- Comparacao.java
- Expressao.java
- Comando.java

A seguir estão as 6 estruturas das máquinas finais a serem trabalhadas para ajudar na implementação do analisador sintático. Vale ressaltar que escolhemos manter estruturas menores e, com isso, obter menores Autômatos de Pilha Estruturados (APEs) para evitar erros na geração dos mesmos. Assim, diferentes APEs menores compõem grandes APEs principais, mesmo que aqui foram tratados separadamente a fim de facilitar o trabalho com um volume grande de dados.

5.1. Máquina Programa

ESTADO	START	INT	CHAR	BOOLEAN	FUNCAO	IDENTIFICADOR	IF	WHILE	OUTPUT	INPUT	END	E	;
0	1												
1		A-2	A-2	A-2	B-4	C-6	C-6	C-6	C-6	C-6	7	3	
2													1
3					B-4	C-6	C-6	C-6	C-6	C-6	7	5	
4													3
5						C-6	C-6	C-6	C-6	C-6	7	5	
6													5
7													

Sub Máquinas: A - Declaração, B - Função e C - Comando;
Estado Final: 7
Legenda:

7	Estado Final
5	Não Consome (Look-Ahead)
C-6	Chama sub-máquina C e estado de retorno 6

5.2. Máquina Declaração

ESTADO	INT	CHAR	BOOLEAN	IDENTIFICADOR	[	]	;	NÚMERO	E
0	1	1	1						
1				2					
2					4		3		3
3									
4								5	
5						6			
6					7		3		3
7								8	
8						3			

Estado Final: 7
Sub Máquinas: Nenhuma;

5.3. Máquina Declaração Função

ESTADO	FUNCAO	CHAR	BOOLEAN	INT	IDENTIFICADOR	IF	WHILE	OUTPUT	INPUT	(	)	,	{	}	TRUE	FALSE	NÚMERO	RETURN	E	;
0	1																			
1		2	2	2																
2					3															
3										4										
4		A-6	A-6	A-6							5									
5													7							
6											5	15								
7		A-6	A-6	A-6	C-10	C-10	C-10	C-10	C-10									11	10	
8																				7
9					C-10	C-10	C-10	C-10	C-10									11	10	
10																				7
11					E-12										E-12	E-12	E-12			
12																				13
13														14						
14																				
15		A-6	A-6	A-6																

Sub Máquinas: A - Declaração, C – Comando, E – Expressão;
Estado Final: 7

5.4. Máquina Comparação

ESTADO	IDENTIFICADOR	NÚMERO	OR	AND	NOT	(	)	==	!=	>=	>	<=	<	TRUE	FALSE	E	;
0	E-1	E-1				4								10	11		
1			0	0				2	2	2	2	2	2				
2	E-3	E-3				E-3								E-3	E-3		
3			0	0			12									12	
4	E-5	E-5			E-5									E-5	E-5		
5							6										
6			0	0													
10			0	0			12										
11			0	0			12										
12																	

Sub Máquinas: E – Expressão;
Estado Final: 12
Estados Simplificados: 7, 8 e 9.

5.5. Máquina Expressão

ESTADO	IDENT.	NÚMERO	(	)	[	]	=	INPUT	OUTPUT	IF	WHILE	THEN	ELSE	ENDIF	DO	ENDIF	TRUE	FALSE	E	;
0	1							11	12	14	15									
1					3		8													
2							8													
3	E-4	E-4															E-4	E-4		
4						5														
5					6		8													
6	E-7	E-7															E-7	E-7		
7						2														
8	E-9	E-9															E-9	E-9		
9																				10
10																				
11	E-16	E-16															E-16	E-16		
12	E-13	E-13															E-13	E-13		
13																				10
14			22																	
15			32																	
16					17															10
17	E-18	E-18																		
18						19														
19					20															10
20	E-21	E-21																		
21						10														
22	F-23	F-23															F-23	F-23		
23				24																
24												26								
26	C-27							C-27	C-27	C-27	C-27		30	28						
27																				26
28																				10
30	C-31							C-31	C-31	C-31	C-31			28						
31																				30
32	F-33	F-33															F-33	F-33		
33				34																
34															35					
35								C-37	C-37	C-37	C-37									
37																				35
38																				10

Sub Máquinas: C – Comando, E – Expressão, F - Comparação;
Estado Final: 12
Estados Simplificados: 25, 29 e 36.

5.6. Máquina Comando

ESTADO	IDENTIFICADOR	NÚMERO	(	)	,	[	]	-	+	*	/	TRUE	FALSE	E	;	Outra Coisa
0	7	9	4					1				8	8			-
1	E-2	E-2	E-2					-				E-2	E-2			-
2								10	10	10	10			3	3	3
3																-
4	E-5	E-5	E-5									E-5	E-5			-
5				6												-
6				3			3	10	10	10	10				3	3
7			12	3		16	3	10	10	10	10				3	3
8				3											3	3
9								10	10	10	10					-
10	E-11	E-11	E-11									E-11	E-11			-
11				3			3								3	-
12	E-14	E-14	E-14	15								E-14	E-14			-
13	E-15	E-15	E-15									E-15	E-15			-
14				15	13											-
15				3			3	10	10	10	10				3	3
16	E-17	E-17	E-17													-
17							18									-
18						19		10	10	10	10				3	3
19	E-20	E-20	E-20													-
20							21									-
21								10	10	10	10				3	3

Sub Máquinas: E – Expressão;
Estado Final: 3

6. Geração de Código

Rotinas semânticas foram implementadas para cada caso listado a seguir. O disparo das rotinas foi colocado na transição dos estados da máquina de análise sintática.

6.1. Declaração de variáveis

As variáveis declaradas no início do programa são armazenadas em uma tabela de símbolos, cuja estrutura é mostrada abaixo:

Tabela 3 – Tabela de Símbolos

Tipo	Nome	Descrição
String	tipoDeSimbolo	Armazena o tipo de símbolo (CONSTANTES, VAR, etc)
String	Identificador	Nome do identificador declarado
String	tipoDeDado	Armazena o tipo do (INT, BOOL ou CHAR)
String	valorInicial	Armazena o valor inicial, no caso de constantes
String	Escopo	Define o escopo em que foi declarado
int	dimensaoX	Define a primeira dimensão de uma matriz, ou a única de um vetor
int	dimensaoY	Define a segunda dimensão da matriz
boolean	declarado	Indica se o símbolo já foi colocado no código

Esta tabela também armazena as constantes encontradas na analise das expressões aritméticas e booleanas. Ao final da compilação, as variáveis que ainda não foram declaradas são gravadas no final do código.

Abaixo temos a declaração de uma variável e o seu respectivo código:

```
INT CONTADOR;
```

```
CONTADOR    /K    0000
```

6.2. Atribuição

A atribuição é gerada carregando-se o valor de origem no acumulador e em seguida armazenado este na variável de destino. O valor de origem pode ser uma variável, uma função ou um resultado de expressão aritmética. Caso seja uma variável, o seu valor é carregado no acumulador. Caso seja uma expressão, seu resultado estará disponível no símbolo que está no topo da pilhaEA, conforme será dito no item 6.3, assim, o compilador irá carregar o endereço do resultado no acumulador. Abaixo temos uma atribuição de uma variável e após uma atribuição de expressão.

Atribuição de variável:

```
CONTADOR = J;
```

```
LD    J
MM    CONTADOR
```

Atribuição de expressão:

```
CONTADOR = 3 * J;
```

```
LD    EA_2
MM    CONTADOR
```

6.3. Expressões

As expressões são calculadas com auxílio da tabela de símbolos, que armazena todas as constantes encontradas durante a sua análise. Também foi criada uma pilha dupla, chamada pilhaEA, que é utilizada para efetuar as operações respeitando-se a precedência de operadores. Esta classe possui duas pilhas, uma para guardar os operadores e outra para os operandos.

Os símbolos são gerados de forma seqüencial, recebendo o prefixo "EA_" seguido do seu número de série. O resultado calculado fica armazenado como um operando na pilhaEA, e será desempilhado por ocasião do seu uso no próximo comando analisado.

Símbolo Recebido	Topo da Pilha	Ação Requerida
/ ou *	- ou +	Empilha o símbolo recebido
- ou +		Executa a operação pendente Empilha o símbolo recebido
(		Executa a operação pendente Empilha o símbolo recebido
)		Executa operações pendentes até o encontrar o símbolo (Desempilha o símbolo)
/ ou *	/ ou *	Executa a operação pendente Empilha o símbolo recebido
- ou +		Executa a operação pendente Empilha o símbolo recebido
(		Empilha o símbolo recebido
)		Executa operações pendentes até o encontrar o símbolo (Desempilha o símbolo)
/ ou *	(	Empilha o símbolo recebido
- ou +		Empilha o símbolo recebido
(		Empilha o símbolo recebido
)		Desempilha o símbolo)

6.3.1. Codificação

O seguinte código é gerado para executar as operações:

A+B	A*B
LD A + B MM A	LD A * B MM A
A-B	**A/B**
LD A - B MM A	LD A / B MM A

6.4. Condições

As condições, como as expressões, são calculadas com auxílio da tabela de símbolos, que armazena todas as constantes encontradas durante a sua análise. Também foi criada uma pilha dupla, chamada pilhaCO, que é utilizada para efetuar as operações respeitando-se a precedência de operadores quando o programa usa os símbolos "(" e ")", neste caso mais simples do que no caso das expressões. Esta classe possui duas pilhas, uma para guardar os operadores e outra para os operandos.

Os símbolos são gerados de forma seqüencial, recebendo o prefixo "CO_" seguido do seu número de série. O resultado calculado fica armazenado como um operando na pilhaCO, e será desempilhado por ocasião do seu uso no próximo comando analisado.

Para facilitar, foram reservadas duas posições de memória que representam os valores TRUE (1) e FALSE (0). Durante a implementaçao foi convencionado que FALSE seria representado por 0 e TRUE por qualquer valor diferente de 0.

6.4.1. Codificação

A OR B	A AND B
LD A + B MM C	LD A * B MM C
A <= B	**A < B**
LD A - B JN CO_T JZ CO_T LD FALSE JP CO_F CO_T: LD TRUE CO_F: MM C	LD A - B JN CO_T LI 0 JP CO_F CO_T: LD TRUE CO_F: MM C
A >= B	**A > B**
LD B - A JN CO_T JZ CO_T LD FALSE JP CO_F CO_T: LD TRUE CO_F: MM C	LD B - A JN CO_T LD FALSE JP CO_F CO_T: LD TRUE CO_F: MM C
A == B	**A != B**
LD A - B JZ CO_F LD FALSE JP SEQ PROX:LD TRUE SEQ: MM C	LD A - B JZ CO_T LD FALSE JP CO_F CO_T LD TRUE CO_T MM C

6.5. IF

O comando IF foi implementado através a inserção de 3 blocos de código, para efetuar os desvios necessários. É assumido que o símbolo que contém o resultado da condição encontra-se no acumulador. A pilhaIF contém os símbolos que serão utilizados pelo bloco, para permitir a posterior referenciação. Além disso permite o uso de IF aninhados.

A instrução NOP é nada mais do que uma soma com zero, para permitir a colocação de labels apontando logicamente para a mesma instrução, mas em posições diferentes de memória. O montador da mvn não aceita mais de um label por linha.

As tabelas abaixo mostram a compilação do comando IF-THEN-ELSE e IF-THEN, respectivamente.

Trecho de código fonte	Código gerado	Ação na pilha
IF (condição) THEN	JZ labelElse	Empilha labelEndif Empilha labelElse
ELSE	JP labelEndIf labelElse NOP	Desempilha labelElse
ENDIF	labelEndIf NOP	Desempilha labelEndIf

Trecho de código fonte	Código gerado	Ação na pilha
IF (condição) THEN	JZ labelEndif	Empilha labelEndif
ENDIF	labelEndIf NOP	Desempilha labelEndif

6.6. WHILE

O comando WHILE de forma semelhante ao IF. Para ser possível o aninhamento, foi criada a pilhaWH. A diferença principal é que neste caso, é feito um desvio no final do bloco de comandos, para que seja calculada novamente a condição.

As tabelas abaixo mostram a compilação do comando WHILE:

Trecho de código fonte	Código gerado	Ação na pilha
WHILE (condição)	labelCond NOP	Empilha labelFim
DO	JZ labelFim	
ENDW	JP labelCond labelFim NOP	Desempilha labelFim

6.7. Entrada

A rotina que gera o código para o comando INPUT é muito simples, recebendo o valor de uma dada entrada da MVN e então o armazenando em uma posição de memória.

```
INPUT DESTINO
GD /0000
/  CONST_SHIFT ; Usado na MVN para colocar o dado de entrada no
               ; byte menos significativo
MM DESTINO
```

6.8. Saída

A rotina que gera o código para o comando OUTPUT é muito simples, enviando para a saída da MVN o resultado da expressão que segue OUTPUT. O resultado da expressão encontra-se no topo da pilhaEA que foi usada para o calcula dessa expressão. Assim, o código gerado será:

```
OUTPUT 5+3
LD EXP        ; Carrega o resultado da operação
PD /0100      ; Envia para a saída do MVN
              ; (0100 é o endereço do vídeo)
```

6.9. Funções

As funções possuem dois blocos de código que precisam ser gerados: a declaração e a chamada. A declaração das funções é feita no início do código gerado, na ordem em que aparecem no código fonte. A chamada de funções é feita através do seu mnemônico específico.

6.9.1. Declaração de Funções

Para que seja possível realizar a passagem de parâmetros, foi implementada uma pilha na mvn, para que faça parte das rotinas de runtime do nosso ambiente. Abaixo temos as rotinas push e pop que são utilizadas para operar a pilha. Deve ser observado que a declaração da pilha é feita ao final do programa, de forma que seu crescimento está limitado apenas pelo tamanho de memória não utilizado pelo código.

Para a implementação da pilha, foi necessário realizar operações de leitura e gravação com endereçamento "indireto". Assim, as instruções GRAVAR e LER são criadas em tempo de execução. O retorno foi implementado através de uma posição fixa de memória para todas as funções, sendo que isso inviabiliza a chamada de sub-funções ou funções recursivas.

Instrução PUSH:

```
PUSH   LD PONTEIRO_TOPO
       +  INST_MM
       MM GRAVAR
       LD TEMP
GRAVAR K  /0000
       LD PONTEIRO_TOPO
       +  DOIS
       MM PONTEIRO_TOPO
       RS PUSH
```

Instrução POP:

```
POP   LD PONTEIRO_TOPO
      - DOIS
      MM PONTEIRO_TOPO
      + INST_LD
      MM LER
LER   K /0000
      MM TEMP
      RS POP
```

Variáveis utilizadas:

```
DOIS          K  /0002
INST_LD       LD /0000
INST_MM       MM /0000
PONTEIRO_TOPO K  TOPO
RETORNO       K  /0000
TEMP          K  /0000
TOPO          K  /0000
```

As funções são declaradas a partir do template abaixo, nele já estão os códigos relativos à obteção dos parâmetros e armazenamento do valor de retorno. Os labels utilizados na geração do código são obtidos a partir do gerador seqüencial de labels seguido do nome da função.

Início da declaração:

```
0_TESTE + FALSE ; NOP
```

Obtenção dos parâmetros da pilha, é repetido para cada parâmetro:

```
SC POP
LD TEMP
MM D
```

Em seguida é colocado o bloco de comandos da função, e para finalizar, o trecho abaixo apresenta o armazenamento do retorno e a chamada da instrução de retorno de sub-rotina.

```
MM RETORNO
RS 0_TESTE
```

6.9.2. Chamada de Funções

A chamada de funções é feita através da instrução "CS", porém, antes da função ser chamada, o compilador coloca na pilha os argumentos para que seja possível a sua posterior recuperação.

O código gerado para a chamada está apresentado abaixo.

```
LD B
MM TEMP
SC PUSH
SC 0_TESTE
LD RETORNO
MM EA_12
```

7. Testes

Foram realizados 3 testes para comprovar o funcionamento do compilador. Eles podem ser encontrados no CD-ROM que contém os arquivos binários.

7.1. Teste 1 – Variáveis, Atribuição, Expressão, Entrada e Saída

Este teste verifica a correta compilação de declaração de variáveis , atribuição (com expressão e constante), entrada e saída.

```
START
INT K;
INT J;
INPUT J;
K = 3;
J = J + K * 2;
OUTPUT K;
END
```

O código foi gerado de forma correta e está apresentado abaixo. Nota-se a presença da declaração das variáveis no final do código, bem como os símbolos utilizados para o cálculo da expressão e as constantes de runtime.

```
@ /0
GD /0000
/ CONST_SHIFT
MM J
LD EA_1
MM K
LD K
*  EA_2
MM EA_3
LD J
+  EA_3
MM EA_4
LD EA_4
MM J
LD K
PD /0100
HM /00
K K /0000
J K /0000
EA_1 K /0003
EA_2 K /0002
EA_3 K /0000
EA_4 K /0000
TRUE K /0001
FALSE K /0000
CONST_SHIFT K /0100
```

7.2. Teste 2 – Condição, IF e IF aninhado

Este código efetua o teste da execução de condições e IFs aninhados.

```
START
INT K;
INT J;
K = TRUE;
INPUT J;
IF ( K OR FALSE ) THEN
J = 3;
ELSE
IF ( J == 4 ) THEN
J = 5;
ENDIF;
ENDIF;
OUTPUT J;
END
```

O código gerado se encontra abaixo. Pode-se notar os símbolos utilizados na condição "CO_", as instruções de NOP. Também é possível observar que os IFs aninhados não causam problemas na geração do código.

```
@ /0
LD EA_1
MM K
GD /0000
/ CONST_SHIFT
MM J
LD K
+  CO_1
MM CO_2
JZ IF_1
LD EA_2
MM J
JP IF_2
IF_1 + FALSE ; NOP
LD J
-  EA_3
JZ CO_4
LD FALSE
JP CO_5
CO_4 LD TRUE
CO_5 MM CO_3
JZ IF_3
LD EA_4
MM J
IF_3 + FALSE ; NOP
IF_2 + FALSE ; NOP
LD J
PD /0100
HM /00
K K /0000
J K /0000
EA_1 K /0001
CO_1 K /0000
CO_2 K /0000
EA_2 K /0003
EA_3 K /0004
```

```
CO_3 K /0000
EA_4 K /0005
TRUE K /0001
FALSE K /0000
CONST_SHIFT K /0100
```

7.3. Teste 3 – WHILE e WHILE aninhado

Neste programa é possível observar tanto o funcionamento de outra condição, com o operador “>”, como o WHILE aninhado.

```
START
INT K;
INT J;
K = 48;
WHILE ( K != 58 ) DO
OUTPUT K;
K = K + 1;
J = 3;
WHILE ( J > 0 ) DO
OUTPUT 46;
J = J - 1;
ENDW;
ENDW;
END
```

O código gerador encontra-se abaixo:

```
@ /0
LD EA_1
MM K
W_2 + FALSE ; NOP
LD K
-  EA_2
JZ CO_2
LD TRUE
CO_2 MM CO_1
JZ W_1
LD K
PD /0100
LD K
+  EA_3
MM EA_4
LD EA_4
MM K
LD EA_5
MM J
W_4 + FALSE ; NOP
LD EA_6
-  J
JN CO_4
LD FALSE
JP CO_5
CO_4 LD TRUE
CO_5 MM CO_3
JZ W_3
LD EA_7
PD /0100
```

```
LD J
-  EA_8
MM EA_9
LD EA_9
MM J
JP W_4
W_3 + FALSE ; NOP
JP W_2
W_1 + FALSE ; NOP
HM /00
K K /0000
J K /0000
EA_1 K /0030
EA_2 K /003a
CO_1 K /0000
EA_3 K /0001
EA_4 K /0000
EA_5 K /0003
EA_6 K /0000
CO_3 K /0000
EA_7 K /002e
EA_8 K /0001
EA_9 K /0000
TRUE K /0001
FALSE K /0000
CONST_SHIFT K /0100
```

7.4. Teste 4 – Funções

Neste programa é possível observar a declaração de funções e a sua posterior chamada, com passagem de parâmetros e obtenção do valor de retorno.

```
START
int b;
int e;
funcao int teste(int d) {
output 46;
while (d < 10) do
output 58;
d=d+1;
endw;
return d;
};
b=1;
e=6;
while(b < 10) DO
output 46;
b=b+1;
e=teste(b);
ENDW;
END
```

O código gerado encontra-se abaixo. Pode-se notar na chamada que a variável B é colocada na pilha antes da chamada da função, e o valor de retorno é obtido após o seu retorno, sendo armazenado como resultado da expressão. Na declaração é importante notar o consumo dos parâmetros da pilha e o seu armazenamento em variáveis da função. O código foi comentado para facilitar o entendimento.

```
@ /0
JP INICIO_0
0_TESTE + FALSE ; NOP e declaração da função
SC POP          ; consume da pilha
LD TEMP
MM D
LD EA_1
PD /0100
W_2 + FALSE ; NOP
LD D
-  EA_2
JN CO_2
LD FALSE
JP CO_3
CO_2 LD TRUE
CO_3 MM CO_1
JZ W_1
LD EA_3
PD /0100
LD D
+  EA_4
MM EA_5
LD EA_5
MM D
JP W_2
W_1 + FALSE ; NOP
LD D
MM RETORNO  ; armazenando o valor de retorno
RS 0_TESTE  ; retorno da função
INICIO_0 + FALSE ; NOP
LD EA_6
MM B
LD EA_7
MM E
W_4 + FALSE ; NOP
LD B
-  EA_8
JN CO_5
LD FALSE
JP CO_6
CO_5 LD TRUE
CO_6 MM CO_4
JZ W_3
LD EA_9
PD /0100
LD B
+  EA_10
MM EA_11
LD EA_11
MM B
LD B
MM TEMP
SC PUSH      ; colocação dos argumentos na pilha
SC 0_TESTE  ; chamada da função
LD RETORNO  ; obtenção do retorno
MM EA_12
LD EA_12
MM E
JP W_4
W_3 + FALSE ; NOP
HM /00
```

```
      B K /0000
      E K /0000
      D K /0000
      EA_1 K /002e
      EA_2 K /000a
      CO_1 K /0000
      EA_3 K /003a
      EA_4 K /0001
      EA_5 K /0000
      EA_6 K /0001
      EA_7 K /0006
      EA_8 K /000a
      CO_4 K /0000
      EA_9 K /002e
      EA_10 K /0001
      EA_11 K /0000
      EA_12 K /0000
      TRUE K /0001
      FALSE K /0000
      CONST_SHIFT K /0100
      PUSH LD PONTEIRO_TOPO
      + INST_MM
      MM GRAVAR
      LD TEMP
      GRAVAR      K /0000
      LD PONTEIRO_TOPO
      + DOIS
      MM PONTEIRO_TOPO
      RS PUSH
      POP   LD PONTEIRO_TOPO
      - DOIS
      MM PONTEIRO_TOPO
      + INST_LD
      MM LER
      LER   K /0000
      MM TEMP
      RS POP
      DOIS K /0002
      INST_LD LD /0000
      INST_MM MM /0000
      PONTEIRO_TOPO K TOPO
      RETORNO K /0000
      TEMP K /0000
      TOPO K /0000
```

8. Referências

[Id] Autor, Titulo, <url>. Acessado em 02 de novembro de 2007.

[1] Horas – Wikipédia, < http://pt.wikipedia.org/wiki/Horae>. Acessado em 02 de novembro de 2007.

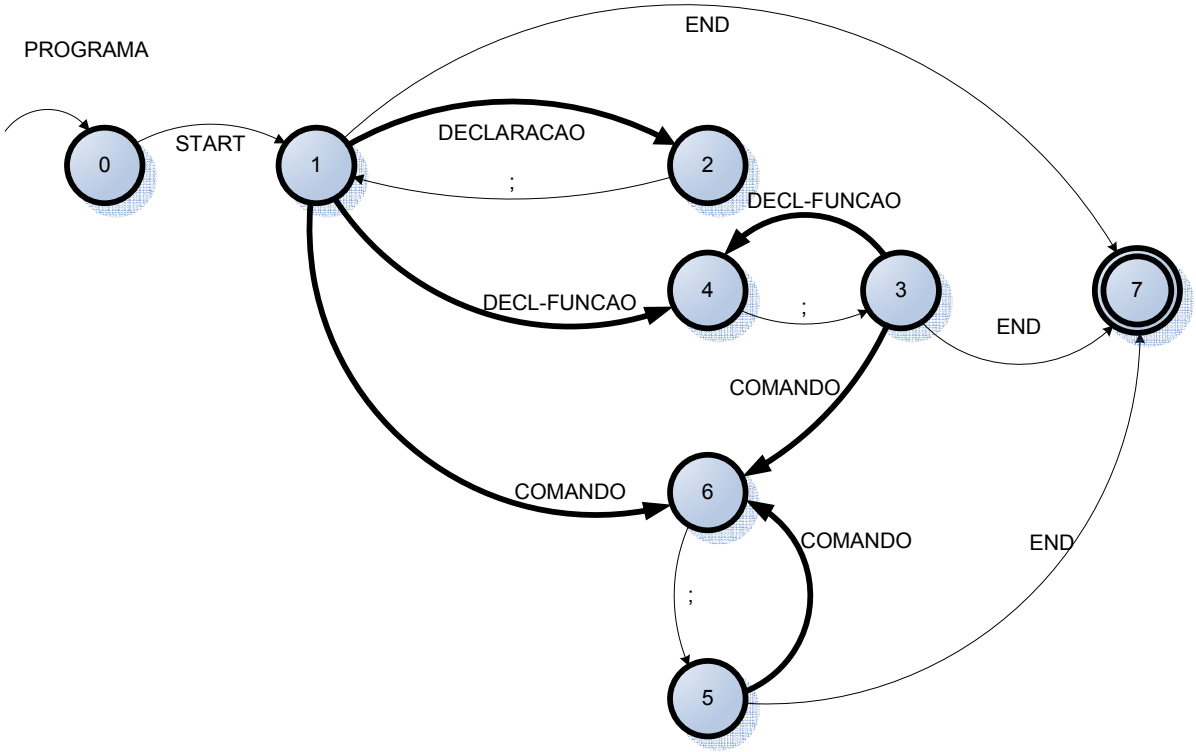
[2] SICHMAN, J. S. & JOSÉ NETO, J. & SILVA, P. S. M. & ROCHA, R. L. A., Notas de Aula de PCS2024, 2º semestre de 2005.

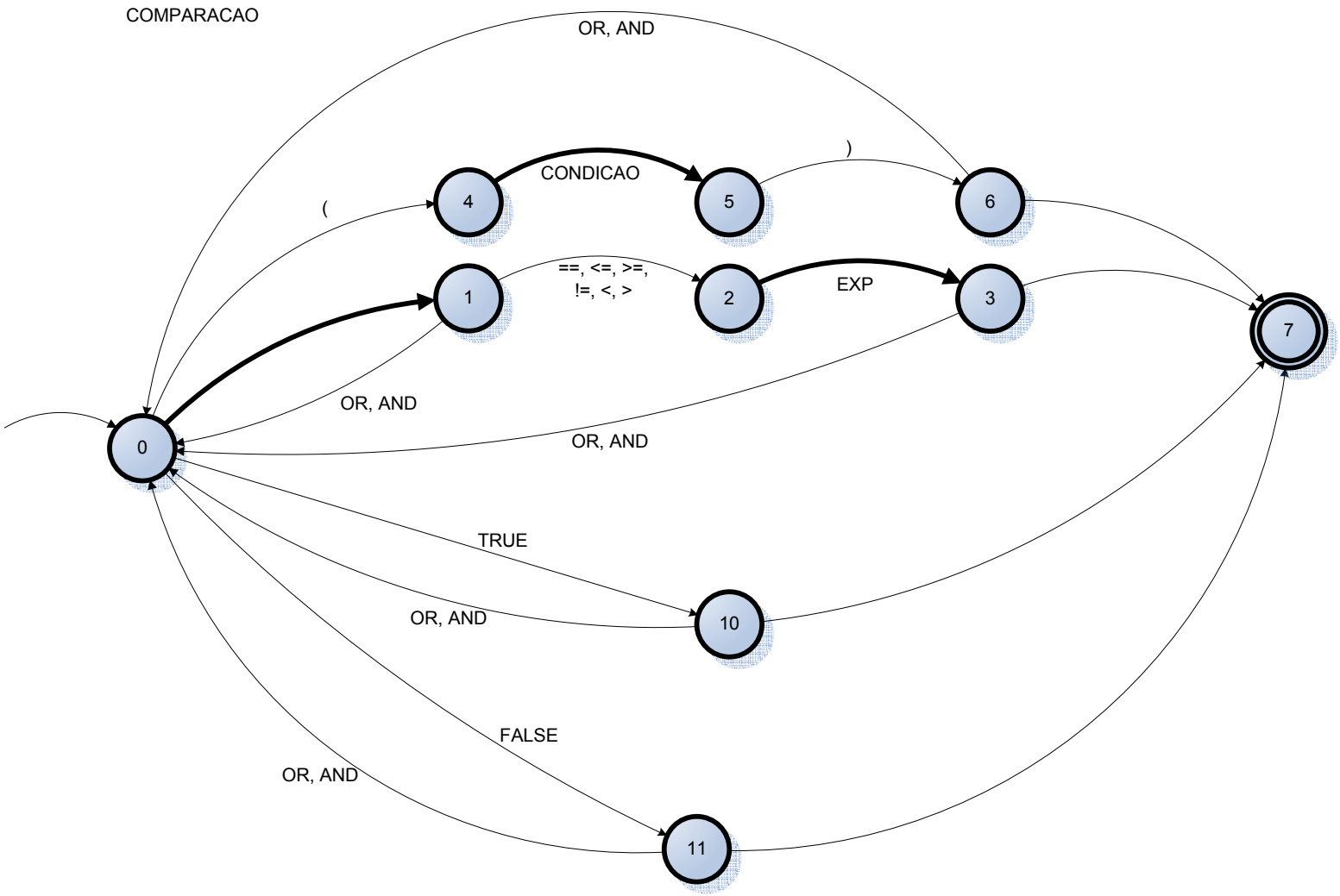
[3] PCS2302/PCS2024 Simulador da Maquina de Von Neumann, Versão 3.0, 2005.

[4] JOSÉ NETO, J., Introdução à compilação – Livros técnicos e científicos, Editora S.A. Rio de Janeiro, 1987.

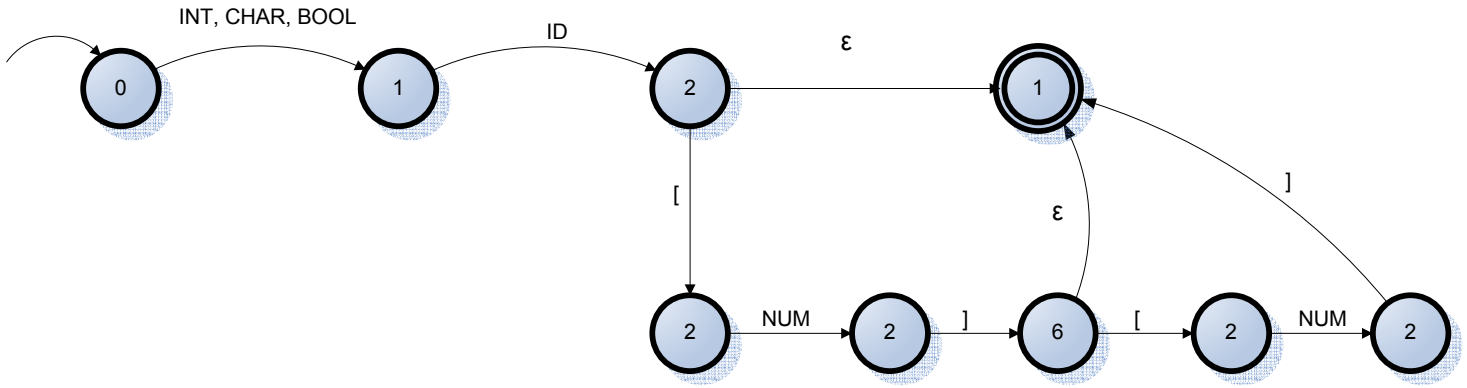
[2] AHO, A. V. & SETHI, R. & ULLMAN, J. D. & LAM, MONICA S., Compilers. Principles, Techniques and tools, Editora Pearson Education, 2ª Edição, 2006.

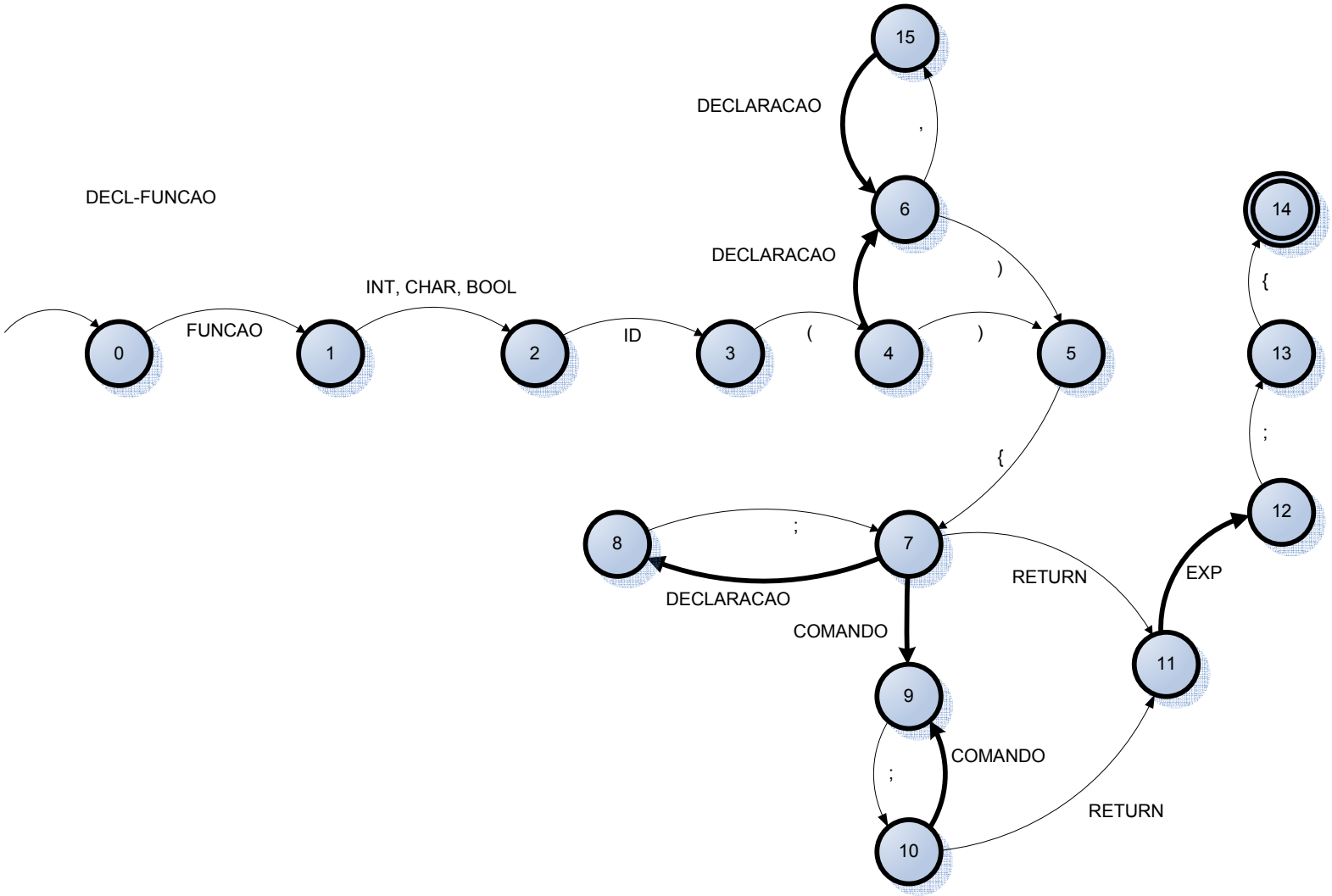
Anexo I – Autômatos do analisador sintático

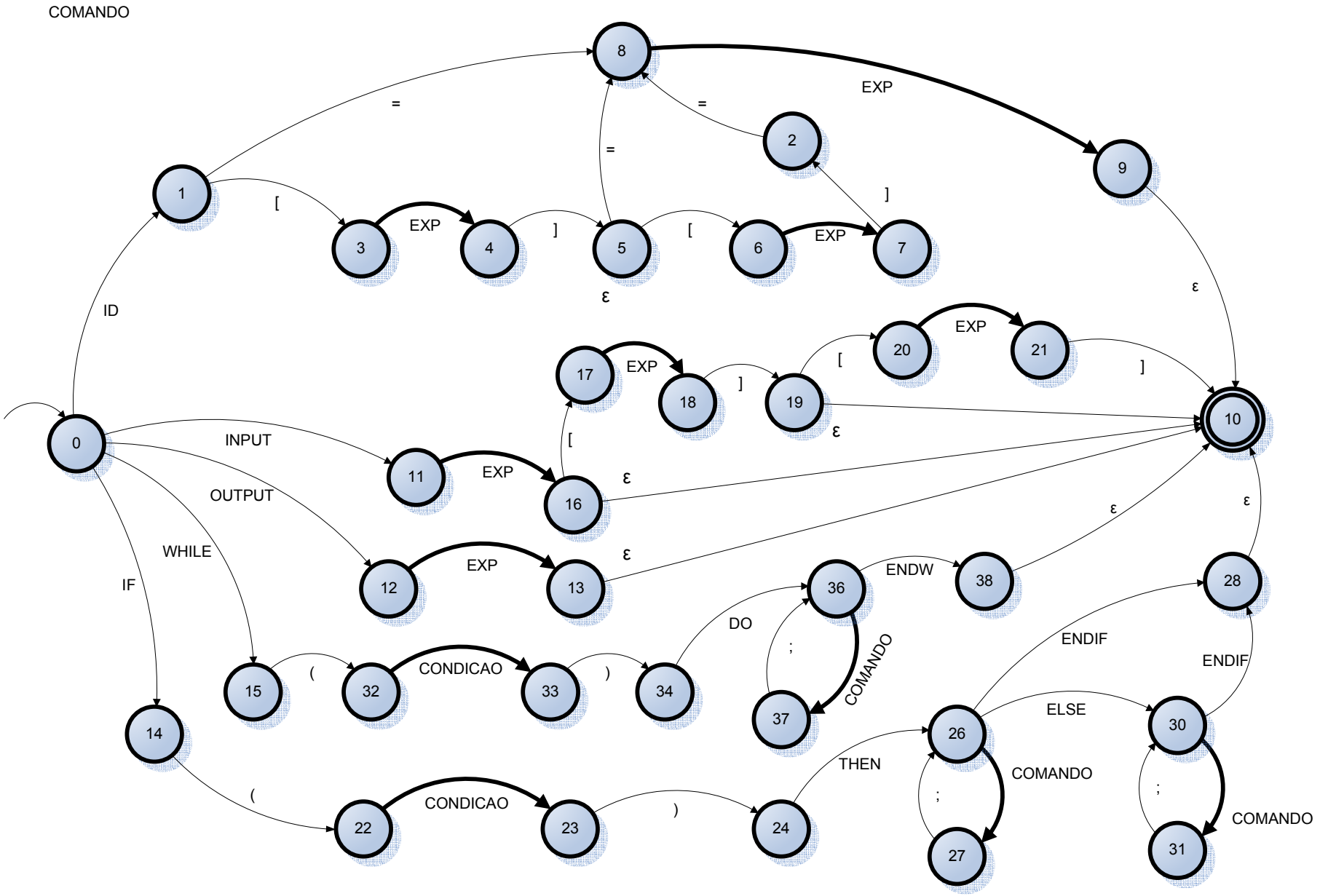

PROGRAMA
END
0
START
1
DECLARACAO
;
2
DECL-FUNCAO
4
;
3
7
DECL-FUNCAO
END
COMANDO
COMANDO
6
COMANDO
END
;
5


COMPARACAO
OR, AND
(
4
CONDICAO
5
)
6
1
==, <=, >=,
!=, <, >
2
EXP
3
7
OR, AND
0
OR, AND
TRUE
OR, AND
10
FALSE
OR, AND
11

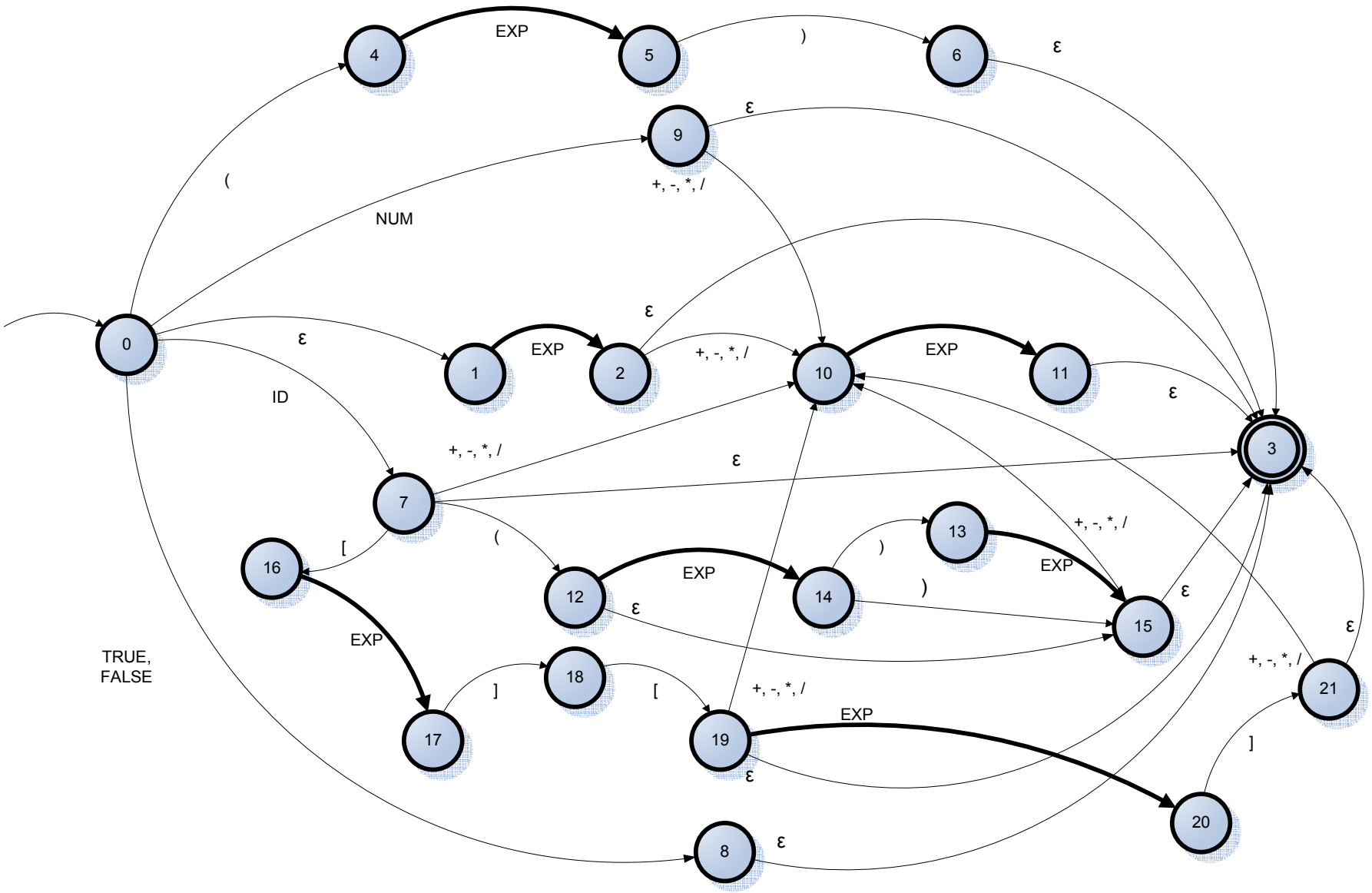
DECLARACAO




DECL-FUNCAO
0
FUNCAO
1
INT, CHAR, BOOL
2
ID
3
(
4
DECLARACAO
15
,
DECLARACAO
6
)
)
5
{
7
;
8
DECLARACAO
RETURN
EXP
12
;
13
{
14
COMANDO
9
11
;
COMANDO
RETURN
10


COMANDO
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
26
27
28
30
31
32
33
34
36
37
38
ID
=
=
=
EXP
[
EXP
]
[
EXP
]
ε
ε
INPUT
OUTPUT
WHILE
IF
EXP
[
EXP
]
[
EXP
]
ε
ε
ε
ε
(
CONDICAO
)
DO
;
COMANDO
ENDW
(
CONDICAO
)
THEN
;
COMANDO
ELSE
ENDIF
;
COMANDO
ENDIF

EXPRESSAO


0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
EXP
(
)
ε
NUM
+, -, *, /
ID
[
]
TRUE,
FALSE