PCS2056 – Linguagens e Compiladores

Linguagens e Compiladores

RELATÓRIO FINAL

Alunos:
Renan Martins Zomignani Mendes
Tiago Schelp Lopes
Professor:
Ricardo Luis de Azevedo da Rocha

05.12.2011

Índice

1. INTRODUÇÃO

Este projeto tem como objetivo apresentar o projeto e implementação de um compilador dirigido por sintaxe e baseado em autômato de pilha estruturado.

O primeiro passo consistiu em se definir uma linguagem. Essa etapa foi realizada baseando-se no conhecimento prévio de outras linguagens de programação, principalmente Ruby e C.

Seguiu-se então a realização de um analisador sintático baseado na descrição Wirth da linguagem, gerando as diversas submáquinas e transições de estado de um autômato de pilha estruturado.

A última etapa consistiu em estudar a linguagem objeto, fazer as correspondências entre seus comandos e aqueles definidos pela nossa linguagem, e implementar todas as ações semânticas necessárias à geração do código.

O capítulo 8 apresenta por fim o código em linguagem de alto nível para o cálculo do fatorial de um número inteiro, seguido de sua tradução na linguagem da MVN.

2. DEFINIÇÃO DA LINGUAGEM

A linguagem adotada foi a definida em aula, com algumas modificações ao longo da implementação. Abaixo encontra-se a última versão da linguagem e sua descrição informal e formal através da notação de Wirth.

2.1 Descrição informal da Linguagem

A linguagem que será especificada terá principalmente elementos de C e Ruby.

Primeiramente, os tipos nativos são aqueles encontrados em C - int, float, char, void - acrescidos de boolean para representar resultados de expressões booleanas. Novos tipos podem ser criados fazendo-se uso da palavra reservada newtype. Apesar de fortemente tipada, essa linguagem permite também a realização de castings, dado que os tipos sejam compatíveis.

Além de novos tipos, é possível definir novas estruturas (ou agregados heterogêneos) a partir da palavra-chave struct.

Elementos comuns a C e Ruby foram mantidos, como as formas como as quais são feitas a atribuição e as comparações entre expressões. No entanto, prevaleceu a legibilidade do código ao se adotar os operadores de negação, conjunção e disjunção em sua forma C, usando respectivamente as palavras reservadas !, & e |.

Os identificadores são definidos por uma letra, seguida ou não, de mais letras e números em qualquer ordem.

As chaves foram eliminadas completamente da linguagem, sendo o escopo delimitado pelo início de comandos de iteração e condição, e funções, e pela palavra reservada end.

Também não é utilizado o ponto-e-vírgula. Como em Ruby, os diferentes comandos devem ser separados por uma quebra de linha (aqui simbolizada por \n)

Exemplo de programa:

```
main()
     int fat
     int n
     input fat
     n = fat - 1
     if(fat == 0)
          output 1
     else
          while(n > 0)
```

```
                fat = fat * n
                n = n - 1
            end
            output fat
        end
end
```

2.2 Descrição Wirth da Linguagem

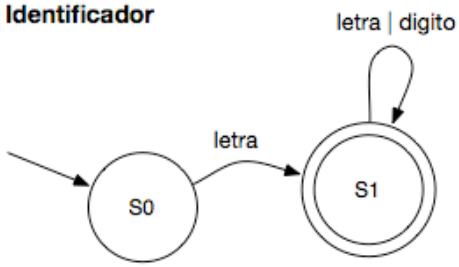
LETRA =	"A"|"B"|"C"|"D"|"E"|"F"|"G"|"H"|"I"|"J"|"K"|"L"|"M"|"N"|"O"|"P"|"Q"|"R"|"S"|"T"|"U"|"V"|"W"|"X"|"Y"|"Z"|"a"|"b"|"c"|"d"|"e"|"f"|"g"|"h"|"i"|"j"|"k"|"l"|"m"|"n"|"o"|"p"|"q"|"r"|"s"|"t"|"u"|"v"|"w"|"x"|"y"|"z".
DIGITO =	"0"|"1"|"2"|"3"|"4"|"5"|"6"|"7"|"8"|"9".
NUM =	NUM_INT | NUM_REAL.
NUM_INT =	DIGITO{DIGITO}.
NUM_REAL =	NUM_INT "." NUM_INT.
BOOLEAN =	"true"| "false".
EXPR =	TERMO {"+" TERMO | "-" TERMO}.
TERMO =	FATOR {"*" FATOR | "/" FATOR | "%" FATOR}.
FATOR =	ID [(REST_ACESSO_VETOR | REST_ACESSO_STRUCT | REST_CHAMADA_FUNCAO)] | NUM | "(" EXPR ")".
REST_ACESSO_VETOR =	"[" NUM_INT "]".
REST_ACESSO_STRUCT =	"." ID [REST_ACESSO_VETOR].
ID =	LETRA{LETRA|DIGITO}.
TIPO =	"int"|"float"|"char"|"boolean"|"void".
DECL_PARAMS =	[TIPO ID {"," TIPO ID}].
COMANDO =	ATR_OU_CHAMADA | COMANDO_COND | COMANDO_ITER | COMANDO_RETORNO | COMANDO_SAIDA | COMANDO_ENTR.
ATR_OU_CHAMADA =	ID (REST_COMANDO_ATR_STRUCT | REST_COMANDO_ATR_ARRAY | REST_COMANDO_ATR | REST_CHAMADA_FUNCAO).
CHAMADA_FUNCAO =	ID REST_CHAMADA_FUNCAO.

REST_COMANDO_ATR_ARRAY =	"[" NUM_INT "]" "=" EXPR.
REST_COMANDO_ATR_STRUCT =	"." ID "=" EXPR.
REST_COMANDO_ATR =	"=" EXPR.
REST_CHAMADA_FUNCAO =	"("PARAM {"," PARAM}")".
CONDICAO =	TERMO_COND {"|" TERM_COND}.
TERMO_COND =	FATOR_COND {"&" FATOR_COND}.
FATOR_COND =	"!" "("CONDICAO")" | ID | BOOLEAN | COMPARACAO.
COMPARACAO =	EXPR (">" | ">=" | "<" | "<=" | "==" | "!=") EXPR.
COMANDO_COND =	"if" "(" CONDICAO ")" "\n" {DECL_OU_COMANDO} {"elsif" "(" CONDICAO ")" "\n" {DECL_OU_COMANDO}} ["else" "\n" {DECL_OU_COMANDO}] "end".
COMANDO_ITER =	"while" "(" CONDICAO ")" "\n" {DECL_OU_COMANDO} "end".
PARAM =	EXPR | CONDICAO.
COMANDO_ENTR =	"input" LISTA_MEM.
LISTA_MEM =	ID {"," ID}.
COMANDO_SAIDA =	"output" LISTA_EXPR.
LISTA_EXPR =	EXPR {"," EXPR}.
COMANDO_RETORNO =	"return" EXPR.
DECL_OU_COMANDO =	[DECL_VAR_SIMP_OU_HOM | DECL_VAR_HET | COMANDO] "\n".
DECL_FUNCAO =	TIPO ID"("[DECL_PARAMS]")" "\n" {DECL_OU_COMANDO} "end".
PROGRAM =	{[(DECL_FUNCAO | DECL_VAR_HET)] "\n"} "main" "(" ")" "\n" {DECL_OU_COMANDO} "end".
DECL_VAR_SIMP_OU_HOM =	TIPO (REST_DECL_VAR_SIMP | REST_DECL_VAR_HOM).
REST_DECL_VAR_SIMP =	ID.
REST_DECL_VAR_HOM =	"[" NUM_INT "]" ID.
DECL_VAR_HET =	"struct" ID "\n" {DECL_VAR_SIMP_OU_HOM "\n"} "end".

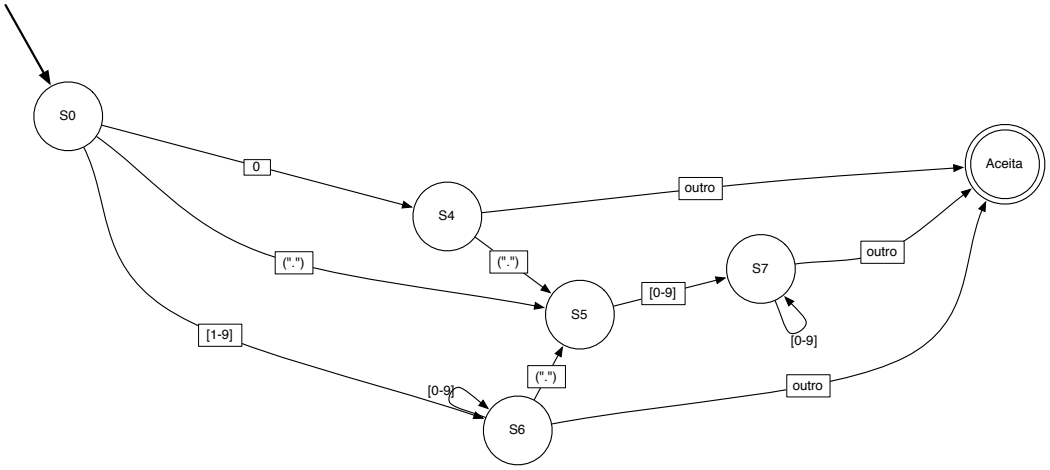
2.3 Autômatos finitos

Autômatos que reconhecem as expressões regulares de cada token:

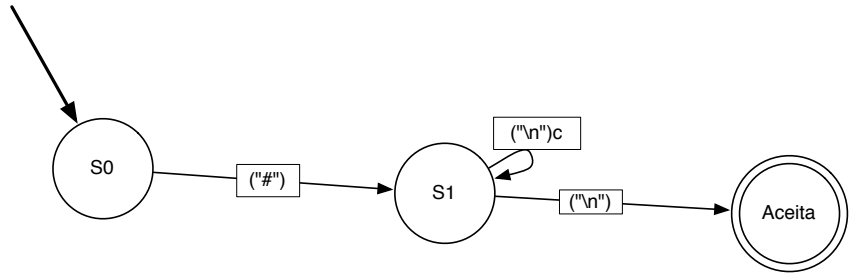
Identificador



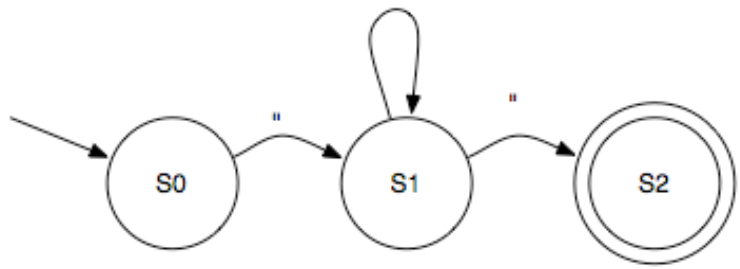
Numero



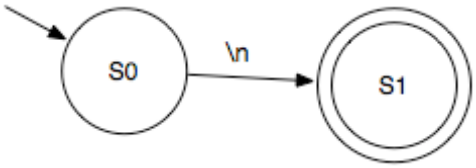
Comentário



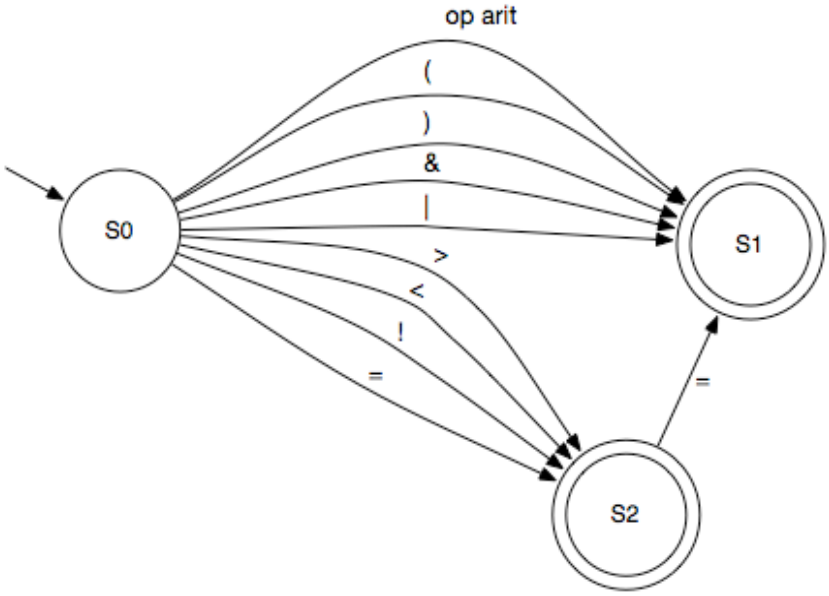
Cadeia de caracter . exceto "



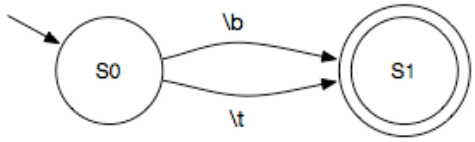
Pontuação



Caracteres Especiais



Espaçador



Autômato finito único

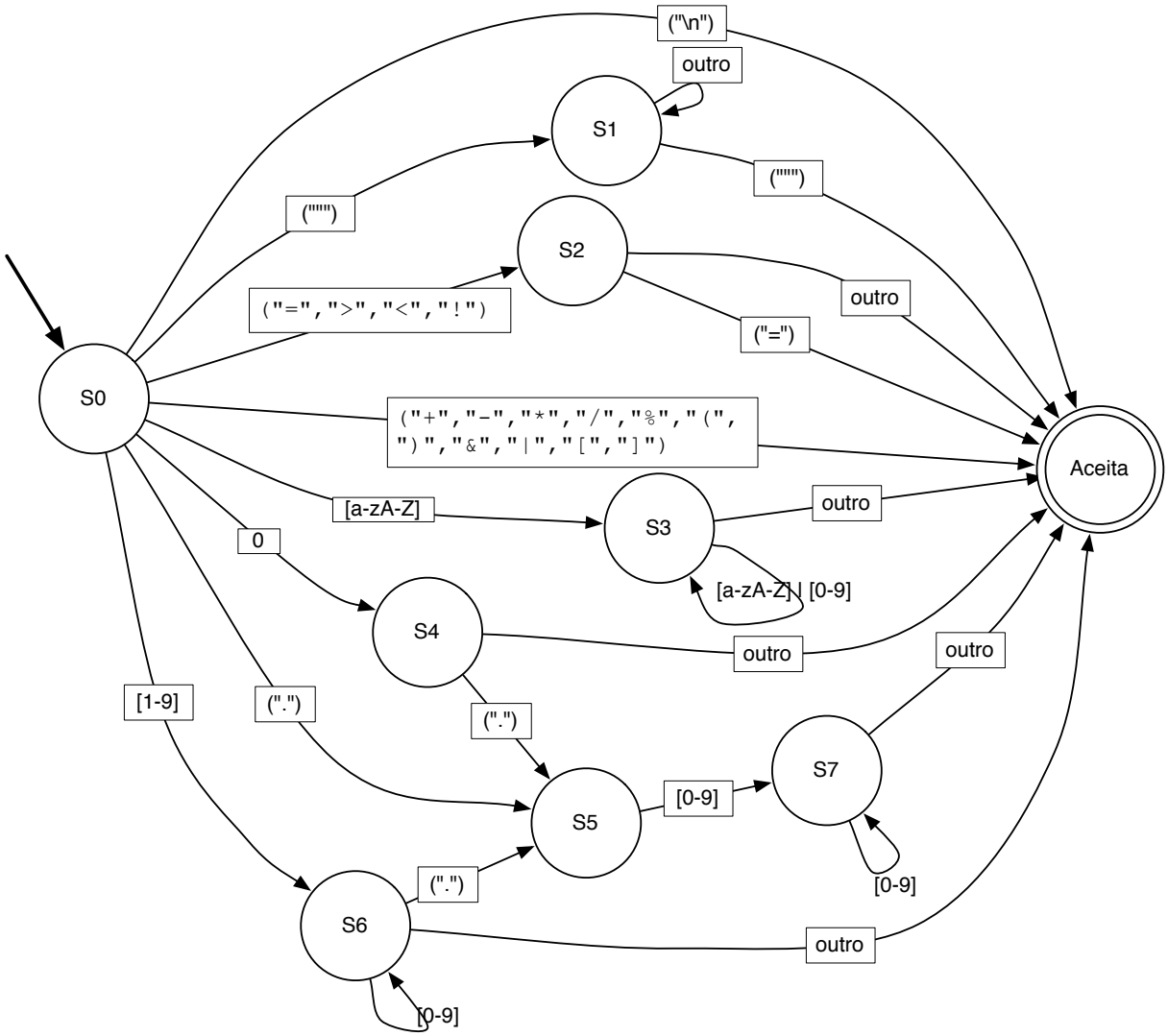


Figura1. Autômato finito equivalente à gramática definida

3. DEFINIÇÃO DO AMBIENTE DE EXECUÇÃO

3.1 Introdução

O ambiente de execução provê suporte para executar o programa gerado pelo compilador. Esse ambiente de execução implementa abstrações incorporadas na definição da linguagem e que não são implementadas pelo código que está sendo

executado. Gerenciam a comunicação com o sistema operacional, dispositivos de entrada e saída e outros programas, alocação de memoria e variáveis declaradas, passagem de parâmetro entre chamadas de função e funções especiais definidas.

No nosso caso, o ambiente de execução do código é a maquina de von Neumann. O código será executado pela máquina virtual utilizada na disciplina PCS2024.

Alguns exemplos de rotinas desse ambiente de execução incluem #INITIALIZE, #LOAD, #STEP #RUN, #EXECUTE e #SHOW.

#INITIALIZE: atribui valores iniciais padrão a todos os elementos importantes do simulador e da arquitetura.

#LOAD: serve para carregar programas e dados para a memória da máquina simulada

#STEP: serve para colocar o simulador no modo de operação passo a passo.

#RUN: serve colocar o simulador no modo de operação contínuo.

#EXECUTE: serve para promover a execução do programa, conforme o modo de operação: execução contínua/uma instrução por vez.

#SHOW: serve para mostrar o conteúdo das memórias da máquina simulada, após a execução de um passo (modo STEP) ou após a execução de um programa (modo RUN).

A máquina de von Neumann disponibiliza ações mais poderosas e ágeis que a máquina de Turing. Por se tratar de código armazenado em memória, existe considerável avanço em flexibilidade e poder computacional se comparado com máquinas de programa fixo, inclusive em tempo de execução, pois na máquina de von Neumann o programa e os dados são armazenados no mesmo espaço de memória. Essa memoria pode ser endereçável aleatoriamente.

Arquitetura básica da MVN :

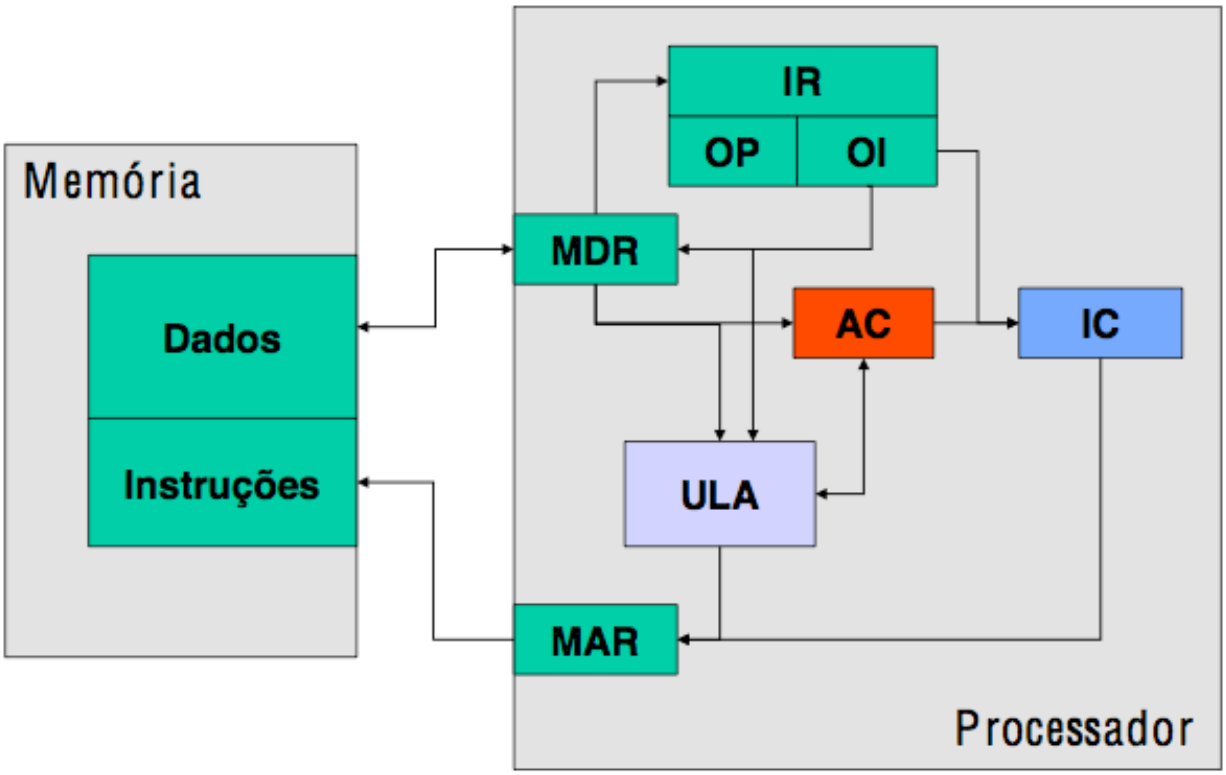


Figura 5 – Arquitetura da máquina de Von Neuman

3.2 Instruções da linguagem de saída

Operação	Nome	Mnemônico
0	Jump	JP
1	Jump if Zero	JZ
2	Jump if Negative	JN
3	Load Value	LV
4	Add	+
5	Subtract	-
6	Multiply	*
7	Divide	/
8	Load	LD
9	Move to Memory	MM
A	Subroutine Call	SC
B	Return from Subroutine	RS
C	Halt Machine	HM
D	Get Data	GD
E	Put Data	PD
F	Operating System	OS

Detalhamento das instruções da linguagem de saída

JP : desvio incondicional
Registrador de instrução = 0

- modifica o conteúdo do registrador de Endereço de Instrução (IC) armazenando nele o conteúdo do registrador de operando (OI) IC := OI

JZ : desvio se acumulador é zero

Registrador de instrução = 1

- se o conteúdo do acumulador for zero, então modifica o conteúdo do registrador de Endereço de Instrução (IC), armazenando nele o conteúdo do registrador de operando (OI)

JN : desvio se negativo
Registrador de instrução = 2

- se o conteúdo do acumulador (AC) for negativo, isto é, se o bit mais significativo for 1, então modifica o conteúdo do registrador de Endereço de Instrução (IC) armazenando nele o conteúdo do registrador de operando (OI)

Se AC < 0 então IC := OI senão IC := IC + 1

LV : constante para acumulador
Registrador de instrução = 3

- Armazena no acumulador (AC) o número relativo de 12 bits contido no registrador de operando (OI), estendendo seu bit mais significativo (bit de sinal) para completar os 16 bits do acumulador

AC := OI IC := IC +1

+ : soma
Registrador de instrução = 4

- Soma ao conteúdo do acumulador (AC) o conteúdo da posição de memória indicada pelo registrador de operando MEM[OI] • Guarda o resultado no acumulador

AC := AC + MEM[OI] IC := IC + 1

- : subtração
Registrador de instrução = 5

- Subtrai do conteúdo do acumulador (AC) o conteúdo da posição de memória indicada pelo registrador de operando MEM[OI] • Guarda o resultado no acumulador

AC := AC - MEM[OI] IC := IC + 1

*** :** multiplicação
Registrador de instrução = 6

- Multiplica o conteúdo do acumulador (AC) pelo conteúdo da posição

de memória indicada pelo registrador de operando MEM[OI] • Guarda o resultado no acumulador

AC := AC * MEM[OI] IC := IC + 1

/ : divisão inteira
Registrador de instrução = 7)

- Dividir o conteúdo do acumulador (AC) pelo conteúdo da posição de

memória indicada pelo registrador de operando MEM[OI] • Guarda a parte inteira do resultado no acumulador

AC := int (AC / MEM[OI]) IC := IC + 1

LD : memória para acumulador

Registrador de instrução = 8

- Armazena no acumulador (AC) o conteúdo da posição de memória cujo endereço é o conteúdo do registrador de operando MEM[OI] AC := MEM[OI]

IC := IC + 1

MM : acumulador para memória

Registrador de instrução = 9

- Guarda o conteúdo do acumulador (AC) na posição de memória indicada pelo registrador de operando MEM[OI] MEM[OI] := AC

IC := IC + 1

SC : desvio para subprograma

Registrador de instrução = A

- Armazena o conteúdo do registrador de Endereço de Instrução (IC), incrementado de uma unidade, na posição de memória apontada pelo registrador de operando MEM[OI]
- Armazena no registrador de Endereço de Instrução (IC) o conteúdo do registrador de operando incrementado de uma unidade (OI)

MEM[OI] := IC + 1

IC := OI + 1

RS : retorno de subprograma

Registrador de instrução = B

- Armazena no registrador de Endereço de Instrução (IC) o conteúdo que está na posição de memória apontada pelo registrador de operando MEM[OI]

IC := MEM[OI]

HM : Parar Máquina

Registrador de instrução = C

- Modifica o conteúdo do registrador de Endereço de Instrução (IC) armazenando nele o conteúdo do registrador de operando (OI) IC := OI

GD : input

Registrador de instrução = D

- Aciona o dispositivo indicado, fazendo a leitura de dados do mesmo

Transfere dado para o acumulador

(solicita dado do dispositivo) AC := dado de entrada IC := IC + 1

PD : output
Registrador de instrução = E

- Aciona o dispositivo indicado

Transfere o conteúdo do acumulador (AC) para o dispositivo, esperando que este termine de executar a operação de gravação
dado de saída := AC (aciona dispositivo) IC := IC + 1

OS : supervisor call
Registrador de instrução = F
(não implementada: por enquanto esta instrução não faz nada)
IC := IC + 1

3.3 Pseudo-instruções da linguagem de saída

Símbolo	Pseudo-instrução	Exemplo
@	Origem absoluta do programa	@ /50; indica /050 como origem do código
#	Fim do programa	# X;
K	Área preenchida por uma constante de 2 bytes	XYZ K /10; gera /10 na posição XYZ
$	Bloco de memória com número especificado de bytes	$ =30; reserva 30 bytes e o primeiro chama-se XYZ
&	Origem relocável	& /50; próximo código se localizará no endereço /050 relativo à origem do código

3.4 Características gerais

O ambiente de execução do código objeto possui 4096 posições em memória e 7 registradores específicos.

Registrador	Função
MAR	Registrador de endereço de memória
MDR	Registrador de dados da memória

IC Registrador de endereço de instrução
IR Registrador de instrução
OP Registrador de código de operação
OI Registrador de operando de instrução
RA Registrador de endereço de retorno
AC Acumulador

Os parâmetros tanto de chamada de função quanto de retorno ficam armazenados espaços endereços de memória específicos para cada caso.

A o espaço de memória é dividido em 3 partes:

- **Área de programa:** Armazena o código do programa.
- **Área de dados:** Área onde ficam armazenados os valores das variáveis declaradas no programa.
- **Área de pilha:** Área onde serão armazenados valores temporários auxiliares. Essa área é implementada a partir do endereço de memória 0x0F00.

Não existem áreas exclusivas. O espaço de programa inicia na posição 0x0000 da memória e logo que termia o programa inicia a área de memória, o que inclui tanto variáveis quanto constantes declaradas e utilizadas ao longo do programa.

4. Tradução – Linguagem de montagem

A linguagem de montagem que será utilizada como código objeto é a mesma vista em PCS2024 - Laboratório de Fundamentos de Engenharia de Comutação. Ela foi apresentada em detalhes no item anterior. A seguir o resumo de algumas características dessa linguagem:

- Cada instrução é composta de um mnemônico e o seu operando.
- Entre os elementos de uma linha deve haver ao menos um espaço.
- As instruções podem ser escritas com um operando numérico ou com um operando simbólico(label).
- Se o primeiro caractere de uma linha for um espaço, esta linha não caracteriza um label.
- À direita de um ponto-e-vírgula, todo texto é ignorado (comentário).
- Mnemônicos e significado das pseudo-instruções:

  @ Operando numérico: define endereço da instrução seguinte, usado para marcar espaço de memória reservados como área de programa, de pilha ou variáveis.

  # Final físico do texto-fonte. Operando=endereço de execução

  K Constante. Operando numérico = valor da constante, em hexadecimal

  / Hexadecimal

  = Decimal

4.1 Tradução de estruturas de controle de fluxo

Desvio:

Nossa linguagem

Desvio -> não implementado

Linguagem de saída

```
JP endereço_destino
```

If-then:

Nossa linguagem

```
If(condicao)
   comandos
end
```

Linguagem de saída

If_<id>	OS /0
	;(expressão)
	JZ fim_if_<id>
	;(comandos)
fim_if_<id>	

If-then-else:

Nossa linguagem:

```
If(condicao)
   comandos
elsif(condicao)
     comandos
end
```

Linguagem de saída:

If_<id>	OS /0
	;(expressão)
	JZ esle_<id>
	;(comandos)
	JP fim_if _<id>
else_<id>	OS /0
	;(comandos)
fim_if_<id>	OS /0

While:

Nossa linguagem

```
while(condicao)
     comandos
end
```

Linguagem de saída

while_<id>	OS /0
	;(expressão)
	JZ fim_while_<id>
	;(comandos)
	JP while_<id>
fim_while_<id>	OS /0

4.2 Tradução de comandos imperativos

Atribuição de valor:

Nossa linguagem:

```
a = 4
```

Linguagem de saída

	LD K4
	MM a

K4 é constante em tempo de compilação

Leitura (entrada):

Nossa Linguagem

```
input a
```

Linguagem de saída

	SC input ; converte de ASCII para decimal e guarda no acumulador
	MM a ;

Impressão (saída)

Nossa Linguagem

```
output a
```

Linguagem de saída

	output ; converte para ASC

Chamada de subrotina

Nossa linguagem:

```
Y = 2
a = subrotina(1, Y)
```

Linguagem de saída:

Área de programa

	LD K2
	MM Y
	LD K1
	MM subrotina_p1
	LD Y
	MM subrotina_p2
	SC SUBROTINA
	LD retorno
	MM a

Subrotina(Param1,Param2)

SUBROTINA	OS /0
	; (comandos)
	MM retorno ; armazena resultado
	RS SUBROTINA

Área de variáveis

subrotina_p1	K /0000
subrotina_p2	K /0000
Y	K /0000
retorno	K /0000
a	K /0000
K2	K /0002
K1	K /0001

“retorno” é uma variável usada para retorno de funções

Essa tradução é uma simplificação do modelo teórico que implementa registros de ativação.

Da maneira como foi traduzido, é possível que uma função chame outras funções, sem perder a referência do endereço de retorno nem misturar as variáveis em cada escopo.

Modelo implementado

Da maneira como foi apresentado uma função pode chamar outra função, evitando que o conteúdo das variáveis sejam sobrepostos ou que os endereços de retorno de cada função sejam perdidos.

Os endereços de memória das variáveis de cada subrotina são únicos, mesmo que o nome das variáveis na linguagem que definimos sejam as mesmas, em tempo de compilação o mecanismo pensado cria um label para criar essa diferenciação, nesse

momento o compilador adiciona um prefixo no nome da variável com o nome da subrotina de modo a obter: NOMESUBROTINA_NOMEVARIAVEL.

Uma limitação desse modo de implementação é o fato de não ser possível o uso de chamdas recursivas.

4.3 Exemplo teórico de programa traduzido

Código utilizando nossa linguagem

```
#Programa
int main()
     int resultado, a
         input a
resultado = fatorial(a)
output resultado
end

#Sub Programa
int fatorial(int n)
     int fat
     fat = 1
          while(n > 0)
               fat = fat * n
               n=n-1
          end
     return fatorial
end
```

Código gerado pelo compilador

```
@ /0000
JP inicio
;*********************************
; programa
;*********************************
inicio         OS /0
          GD /000   ;le do teclado
          /  K100
          -  K30         ; converte numeros lido
          MM a      ; input a
          LD a
          MM fatorial_n
          SC fatorial    ;fatorial a
          LD retorno
          MM resultado
          LD resultado   ; le retorno da funcao
          + K30          ; converte para ASC
```

```
          PD /100   ; imprime no monitor
          HM /0


;**********************************
;Subrotina fatorial
;**********************************

fatorial  OS /0
          LD K1
          MM fatorial_fat     ; fat = 1
WHILE          OS /0
          LD fatorial_n
          JN FIMWHILE    ;  sai do whole se fatorial_n <= 0
          JZ FIMWHILE
          LD fatorial_fat     ; fat = fat * n
          *  fatorial_n
          MM fatorial_fat
          LD fatorial_n       ; n = n-1
          -  K1
          MM fatorial_n
          JP WHILE
FIMWHILE  OS /0
          LD fatorial_fat
          MM retorno          ; return fat
          RS fatorial

;**********************************
;area de variaveis e constantes
;**********************************
resultado K /000
a         K /000
retorno   K /000
fatorial_n     K /000
fatorial_fat   K /000
K30       K /030
K100      K /100
K1        K /001
K2        K /002


# inicio
```

5. ANÁLISE LÉXICA

5.1 Extração e classificação de átomos

Associa o texto-fonte à outro texto formado pelos átomos que os símbolos componentes do texto-fonte representam. O analisador léxico também classifica esses átomos em:

- *Identificadores*
- *palavras reservadas*
- *números inteiros sem sinal*
- *números reais*
- *cadeias de caracteres*
- *sinais de pontuação e de operação*
- *caracteres especiais*
- *símbolos compostos de dois ou mais caracteres*
- *comentários*

5.2 Implementação do Analisador Léxico: Vantagens e Desvantagens

Como uma fase separada do processamento da linguagem de programação o analisador léxico apresenta vantagens e desvantagens em relação à sua implementação como sub-rotina que extrai um átomo a cada chamada. Na tabela a seguir são apresentadas as principais vantagens e desvantagens.

Implementação	Vantagens	Desvantagens
Fase separada do processamento	Simplicidade do compilador final: funções nitidamente separadas entre as diversas etapas	A comunicação entre o analisador léxico deverá ser feita por meio de arquivos intermediários ou ainda com todos os tokens carregados em memória
Sub-rotina	Carrega em memória somente a quantidade necessária de tokens à análise sintática.	
	Lê cada token somente uma vez	

Tabela 1 . Vantagens e desvantagens da implementação do analisador léxico em fase separada

5.3 Sintaxe dos Tokens

A tabela a seguir apresenta a definição formal, através de expressões regulares sobre o conjunto de caracteres ASCII, da sintaxe de cada um dos tipos de átomos a serem extraídos do texto-fonte pelo analisador léxico, bem como de cada um dos espaçadores e comentários.

Token	Expressão Regular
ID	[a-zA-Z]([a-zA-Z]|[0-9])*
RESERVED_KEYWORD	(if|else|elsif|end|while|int|float| boolean|char|true|false|return|input| output|main|struct)
NUM	[0-9]+(\.[0-9]+)*
STRING	“({\”} ^{c})*”
PONTUACAO	\n
CARACTER_ESPECIAL	(+|-*|/|%|=|(|)|>|<|!|>=|<=|!=|==|&||[|])
EoF	\377
SPACE	(|\t)
COMENTARIO	#{\n}^{c}\n

Tabela 2. Sintaxe dos Tokens

5.3 Transdutor

A partir do autômato definido, foi criado um transdutor, que emite como saída o átomo encontrado ao abandonar cada um dos estados finais para iniciar o reconhecimento de mais um átomo do texto.

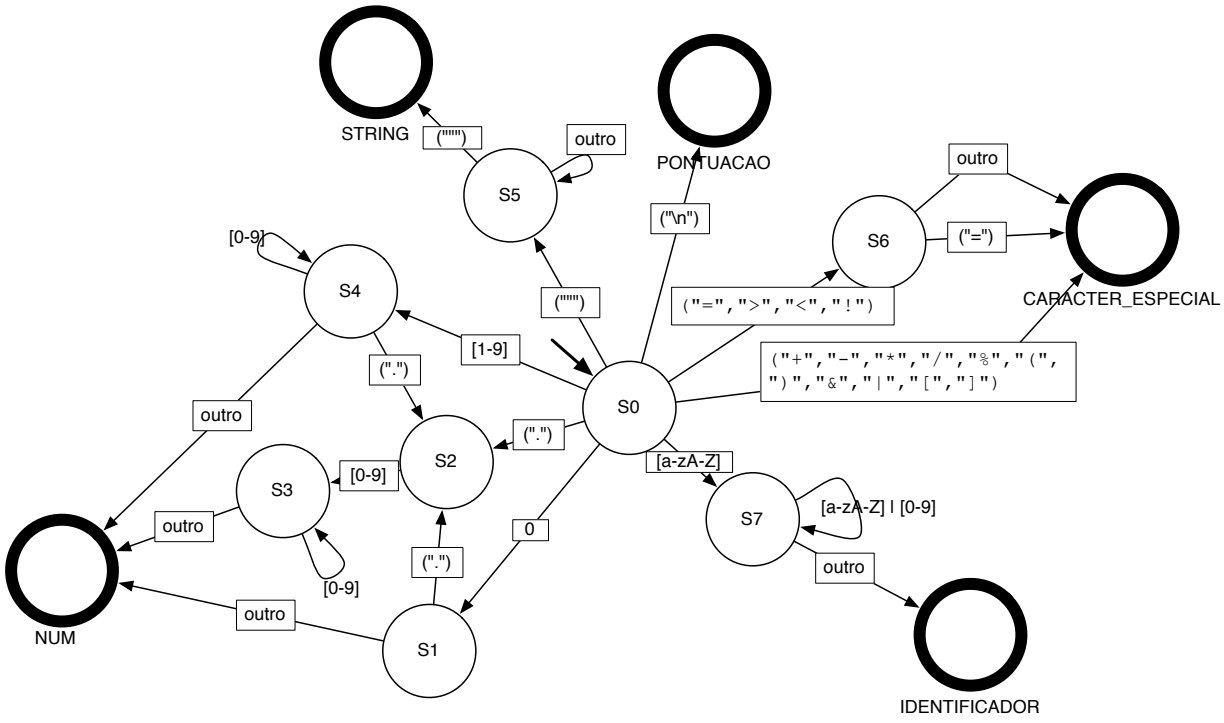


Figura2. Transdutor

5.4 Descrição do Analisador Léxico

5.4.1 Descrição da Estrutura

As estruturas que compões o analisador léxico são:

- **Token**

O token possui um valor que é o seu lexema, possui um tipo, de acordo com o seu conteúdo, definido no transdutor, uma linha e uma coluna correspondente à sua posição no arquivo de entrada.

```
typedef struct token {
 tipoToken tipo;
 char valor[256];
 int linha;
 int coluna;
} Token;
```

- **TiposToken**

Os tipo que podem ser atribuídos aos tokens estão definidos no tipo enum tiposToken, detalhado abaixo.

```
typedef enum tiposToken {
 PONTUACAO,
 NUM,
 STRING,
 ID,
 RESERVED_KEYWORD,
 SPECIAL_CHARACTER,
 ERRO,
 EoF,
    NDEF
} tipoToken;
```

- **Palavras_reservadas**

```
 const char* Tabela_Palavras_Reservadas[] = {
      "if",
      "else",
      "elsif",
      "end",
      "while",
      "int",
      "float",
      "boolean",
      "char",
       "void",
      "true",
      "false",
      "return",
      "input",
      "output",
      "main",
      "struct"
};
```

- **TabelaDeEstados**

A máquina finita de estados que implementa o transdutor do item 3.3 pode ser representado por uma tabela com 8 estados e 256 transições a partir de cada estado, totalizando 2048 transições de estado.

Estado atual / Condição	Estado A	Estado B	Estado C
Condição X	...	...	...
Condição Y	...	Estado C	...
Condição Z	...	...	...

Figura 3. Modelo de tabela que implementa máquina de estados

Tal tabela pode ser construída manualmente, porém o trabalho é exaustivo e a chance de erro são grandes, para contornar tal problema foi implementada uma função para a construção dessa tabela.

5.4.2 Descrição do Funcionamento

IDENTIFICANDO OS TOKENS:
Token *getNextToken(FILE *, char *ch, int *linha,int *coluna);

O analisador léxico foi implementado para gerar tokens a partir do arquivo de entrada.

Com o auxílio da uma tabela de estados, implementada a partir do transdutor do item 3.3, a função principal do analisador léxico é percorrer essa tabela e assim que chegar em um estado final, chama a função Token *criarToken(char *Lexema, tipoToken tipo, int linha, int coluna), que por sua vez identifica o tipo do token, consulta a tabela de palavras reservadas para os casos de tipo identificador, e atribui os valores passados como parâmetros.

O diagrama a seguir apresenta graficamente o funcionamento do analisador descrito.

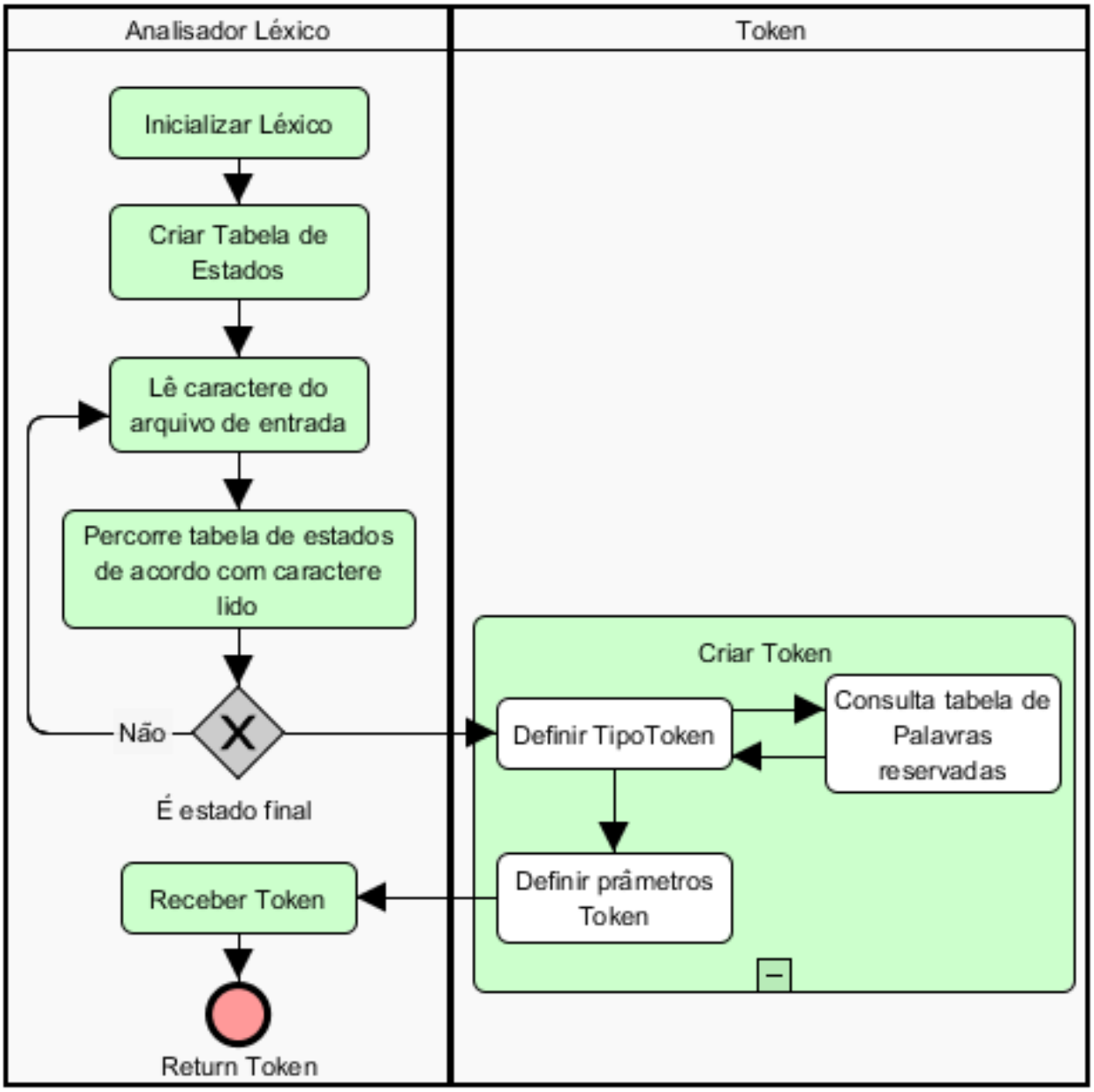


Figura 4. Funcionamento do Analisador Léxico

Nos casos em que os lexemas são definidos pelo primeiro caractere do próximo token, e não pelo último do token em questão, foi preciso atenção especial com a leitura dos próximos caracteres. Por exemplo, sempre que um token do tipo número chega ao fim, só percebemos isso depois de ler o próximo caractere do arquivo, foi preciso atenção nesse ponto para não iniciar nova leitura e descartar esse caractere já lido. Esse tratamento é feito em determinados estados dependendo da necessidade. O tratamento é apresentado a seguir:

```
// Descarta o último caractere do BufferLexema
if ( estado_Atual == Terminal_IDENT ||
estado_Atual == Terminal_NUM ||
estado_Atual == Terminal_CARACTER_ESPECIAL_1_Digitos){
     BufferLexema[strlen(BufferLexema)-1] = '\0';
}

//Lê o próximo caractere do arquivo de entrada
else if(estado_Atual == Terminal_STRING ||
estado_Atual == Terminal_CARACTER_ESPECIAL_2_Digitos ||
estado_Atual == Terminal_CARACTER_ESPECIAL ||
estado_Atual == Erro_Lexico){
     ch = getc(inputFile);
     coluna++;
}

//atualiza o contador de linha para o caso de fim de //linha
else if(estado_Atual == Terminal_PONTUACAO) {
     ch = getc(inputFile);
     coluna=0;
     linha++;
}
```

DECORATIVOS:

Os decorativos são descartados no início da função getNextToken. São tratados como decorativos os comentários(#), espaços em branco (‘ ’) e Tabs(\t).

ERROS:

Os caracteres não identificados no estado S0 são armazenados em um Token com o tipoToken ERRO, este erro pode ser localizado através da indicação de linha e coluna.

6. ANÁLISE SINTÁTICA

O analisador sintático é o componente central do compilador. A partir dele aciona-se tanto as funcionalidades do analisador léxico (busca do próximo token) quanto do analisador semântico (verificações referentes aos escopos, tradução para código de máquina, etc).

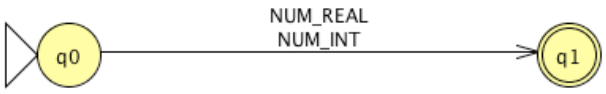
A função principal do analisador sintático é, no entanto, a validação da cadeia de tokens como pertencente à linguagem, além de apontar a localização dos erros de sintaxe.

6.1 Lista de submáquinas do APE

As submáquinas geradas na página da internet indicada no enunciado da questão 4 são as seguintes (algumas abreviações para as mais simples):

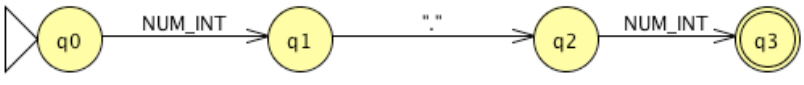
LETRA
initial: 0
final: 1
(0, "A") -> 1
(0, "B") -> 1
...
(0, "Z") -> 1
(0, "a") -> 1
(0, "b") -> 1
...
(0, "z") -> 1

DIGITO
initial: 0
final: 1
(0, "0") -> 1
(0, "1") -> 1
...
(0, "9") -> 1



NUM
initial: 0
final: 1
(0, NUM_INT) -> 1
(0, NUM_REAL) -> 1

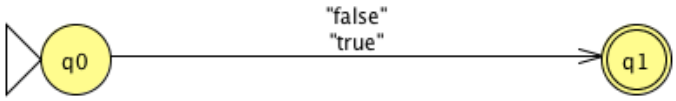
NUM_INT
initial: 0
final: 1
(0, DIGITO) -> 1
(1, DIGITO) -> 1



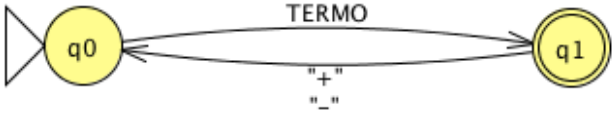
NUM_REAL
initial: 0
final: 3
(0, NUM_INT) -> 1
(1, ".") -> 2

(2, NUM_INT) -> 3

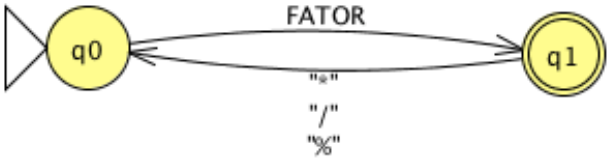
BOOLEAN
initial: 0
final: 1
(0, "true") -> 1
(0, "false") -> 1



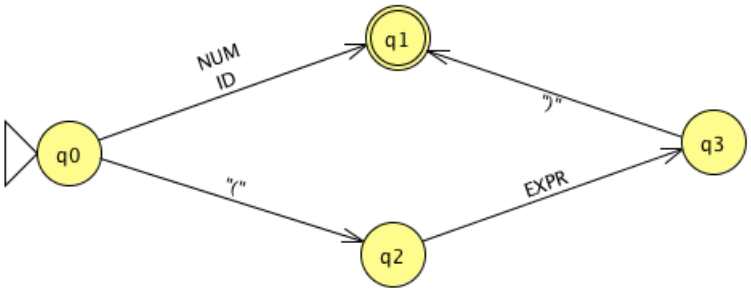
EXPR
initial: 0
final: 1
(0, TERMO) -> 1
(1, "+") -> 0
(1, "-") -> 0



TERMO
initial: 0
final: 1
(0, FATOR) -> 1
(1, "*") -> 0
(1, "/") -> 0
(1, "%") -> 0

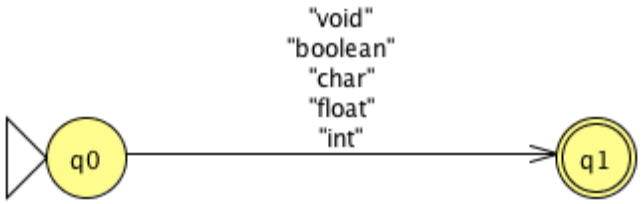


FATOR
initial: 0
final: 1
(0, ID) -> 1
(0, NUM) -> 1
(0, "(") -> 2
(2, EXPR) -> 3
(3, ")") -> 1

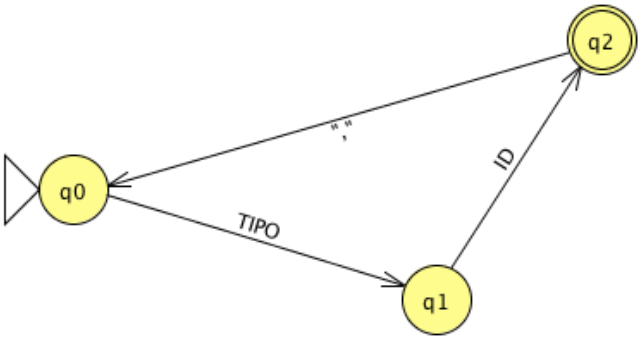


ID
initial: 0
final: 1
(0, LETRA) -> 1
(1, LETRA) -> 1
(1, DIGITO) -> 1

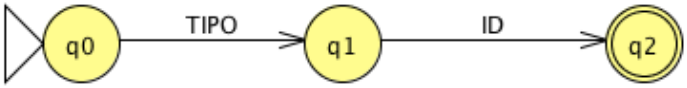
TIPO
initial: 0
final: 1
(0, "int") -> 1
(0, "float") -> 1
(0, "char") -> 1
(0, "boolean") -> 1
(0, "void") -> 1



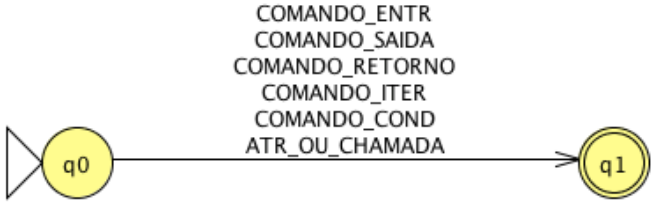
DECL_SIMP_VAR
initial: 0
final: 2
(0, TIPO) -> 1
(1, ID) -> 2



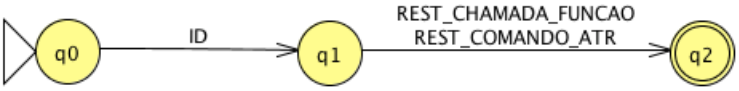
DECL_PARAMS
initial: 0
final: 2
(0, TIPO) -> 1
(1, ID) -> 2
(2, ",") -> 0



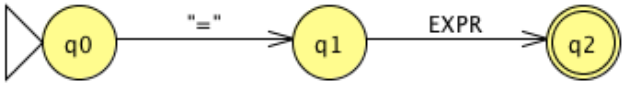
COMANDO
initial: 0
final: 1
(0, ATR_OU_CHAMADA) -> 1
(0, COMANDO_COND) -> 1
(0, COMANDO_ITER) -> 1
(0, COMANDO_RETORNO) -> 1
(0, COMANDO_SAIDA) -> 1
(0, COMANDO_ENTR) -> 1



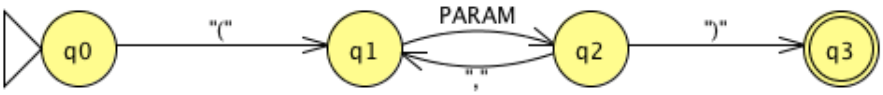
ATR_OU_CHAMADA
initial: 0
final: 2
(0, ID) -> 1
(1, REST_COMANDO_ATR) -> 2
(1, REST_CHAMADA_FUNCAO) -> 2



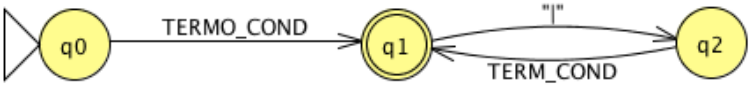
REST_COMANDO_ATR
initial: 0
final: 2
(0, "=") -> 1
(1, EXPR) -> 2



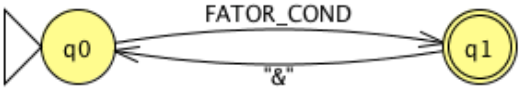
REST_CHAMADA_FUNCAO
initial: 0
final: 3
(0, "(") -> 1
(1, PARAM) -> 2
(2, ",") -> 1
(2, ")") -> 3



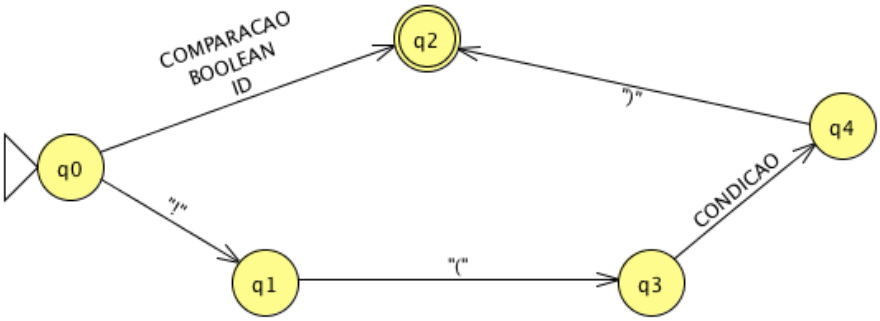
CONDICAO
initial: 0
final: 1
(0, TERMO_COND) -> 1
(1, "|") -> 2
(2, TERM_COND) -> 1



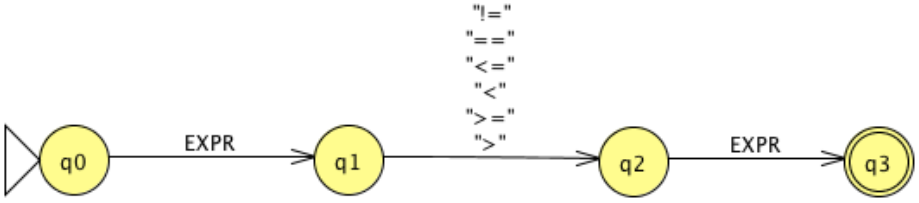
TERMO_COND
initial: 0
final: 1
(0, FATOR_COND) -> 1
(1, "&") -> 0



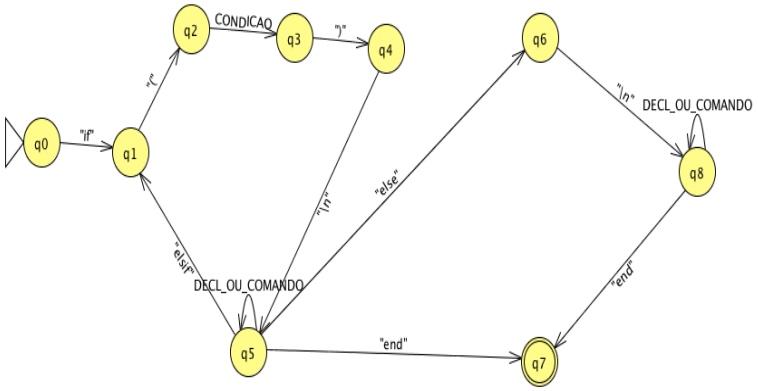
FATOR_COND
initial: 0
final: 2
(0, "!") -> 1
(0, ID) -> 2
(0, BOOLEAN) -> 2
(0, COMPARACAO) -> 2
(1, "(") -> 3
(3, CONDICAO) -> 4
(4, ")") -> 2



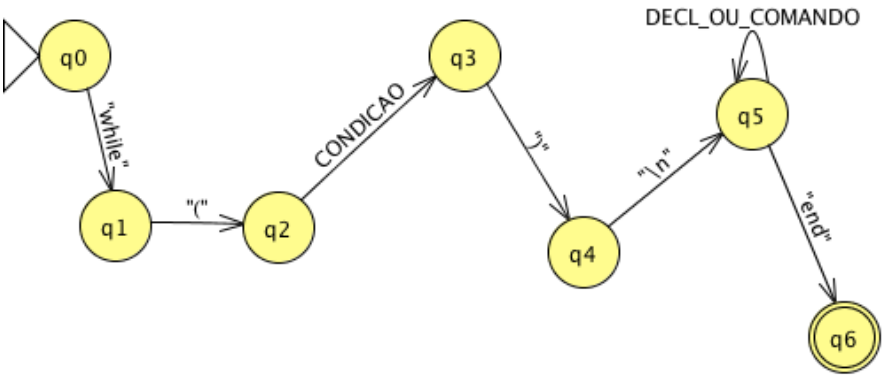
COMPARACAO
initial: 0
final: 3
(0, EXPR) -> 1
(1, ">") -> 2
(1, ">=") -> 2
(1, "<") -> 2
(1, "<=") -> 2
(1, "==") -> 2
(1, "!=") -> 2
(2, EXPR) -> 3



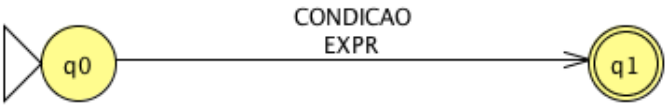
COMANDO_COND
initial: 0
final: 7
(0, "if") -> 1
(1, "(") -> 2
(2, CONDICAO) -> 3
(3, ")") -> 4
(4, "\n") -> 5
(5, DECL_OU_COMANDO) -> 5
(5, "elsif") -> 1
(5, "else") -> 6
(5, "end") -> 7
(6, "\n") -> 8
(8, DECL_OU_COMANDO) -> 8
(8, "end") -> 7



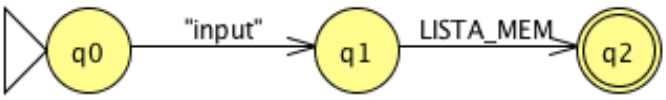
COMANDO_ITER
initial: 0
final: 6
(0, "while") -> 1
(1, "(") -> 2
(2, CONDICAO) -> 3
(3, ")") -> 4
(4, "\n") -> 5
(5, DECL_OU_COMANDO) -> 5
(5, "end") -> 6



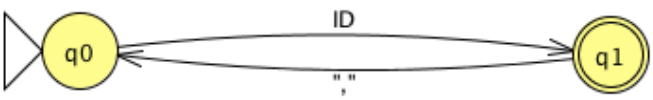
PARAM
initial: 0
final: 1
(0, EXPR) -> 1
(0, CONDICAO) -> 1



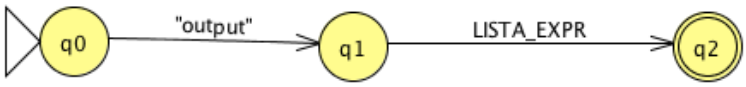
COMANDO_ENTR
initial: 0
final: 2
(0, "input") -> 1
(1, LISTA_MEM) -> 2



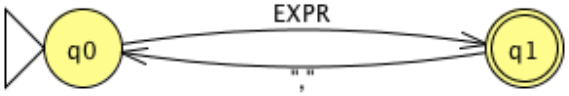
LISTA_MEM
initial: 0
final: 1
(0, ID) -> 1
(1, ",") -> 0



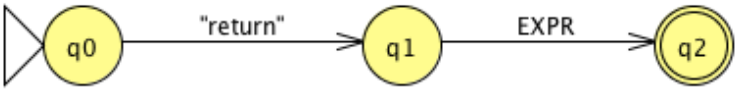
COMANDO_SAIDA
initial: 0
final: 2
(0, "output") -> 1
(1, LISTA_EXPR) -> 2



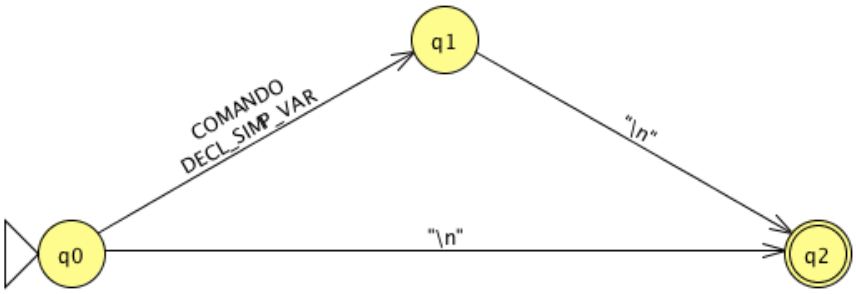
LISTA_EXPR
initial: 0
final: 1
(0, EXPR) -> 1
(1, ",") -> 0



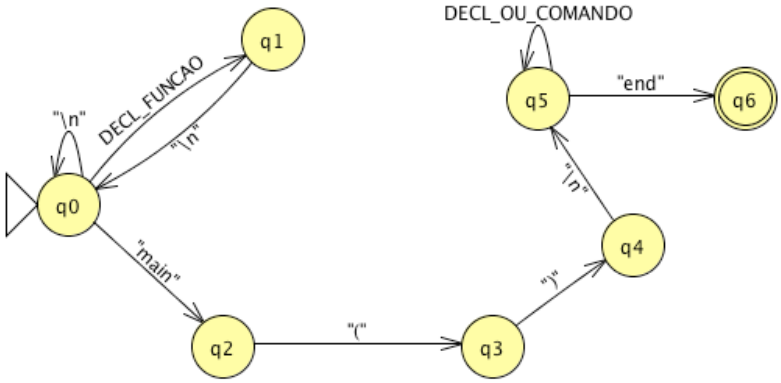
COMANDO_RETORNO
initial: 0
final: 2
(0, "return") -> 1
(1, EXPR) -> 2



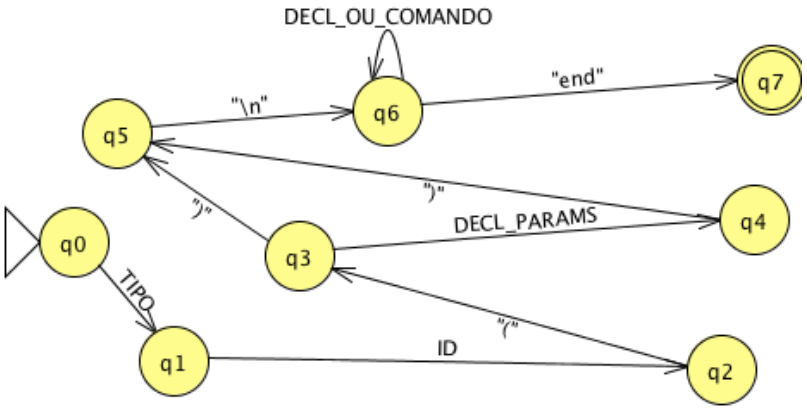
DECL_OU_COMANDO
initial: 0
final: 2
(0, DECL_SIMP_VAR) -> 1
(0, COMANDO) -> 1
(0, "\n") -> 2
(1, "\n") -> 2



DECL_FUNCAO
initial: 0
final: 7
(0, TIPO) -> 1
(1, ID) -> 2
(2, "(") -> 3
(3, DECL_PARAMS) -> 4
(3, ")") -> 5
(4, ")") -> 5
(5, "\n") -> 6
(6, DECL_OU_COMANDO) -> 6
(6, "end") -> 7



PROGRAM
initial: 0
final: 6
(0, DECL_FUNCAO) -> 1
(0, "\n") -> 0
(0, "main") -> 2
(1, "\n") -> 0
(2, "(") -> 3
(3, ")") -> 4
(4, "\n") -> 5
(5, DECL_OU_COMANDO) -> 5
(5, "end") -> 6



6.2 implementação

A implementação do analisador sintático foi realizada segundo o método do Autômato de Pilha Estruturado, em que os diferentes componentes da gramática são reconhecidos por meio de submáquinas.

Para tal, foi necessário primeiramente identificar todas as transições possíveis, tanto as transições internas quanto as chamadas de submáquina. As primeiras foram guardadas na tabela transicoes e as últimas na tabela chamadas.

Dado um token e um estado, o analisador sintático verifica se existe uma transição interna adequada. Se existir, consome-se o token, ou seja, analisa-se o próximo estado com o próximo token. Caso não exista transição interna, procura-se uma chamada de submáquina apropriada. Se encontrada, empilha-se o estado de retorno e realiza a transição. Nesse caso o token não se consome.

Existem, no entanto, situações em que não haverá nem transição interna nem chamada de submáquina adequadas. Nesse caso, verifica-se se o estado corrente corresponde a um estado final de alguma submáquina utilizando-se a função estadoFinal(). Em caso afirmativo, retorna-se ao estado desempilhado do topo da pilha.

Em último caso, indica-se um erro de sintaxe na linha do token atual.

Algumas modificações tiveram que ser feitas na gramática exibida na entrega parcial precedente a esta para que fosse possível a implementação do analisador sintático utilizando o método do APE. Essa gramática encontra-se explicitada no arquivo WIRTH.txt.

7. Análise Semântica

A funcionalidade do compilador, de modo simplificado é, ler um arquivo de entrada, verificar o código de entrada de acordo com a gramática definida para a linguagem de

programação criada e gerar um código de saída na linguagem especificada de acordo com os itens 3 e 4.

O diagrama a seguir apresenta o funcionamento do programa com simplificações para não atrapalhar o entendimento do processo, as estruturas de dados e ações semânticas serão explicadas em mais detalhes a seguir, o processo que verifica se a partir de um estado e dado um token existe transição ou chamada de submáquina acontece antes de cata ponto de decisão consultando a respectiva tabela de transições e de submáquinas.

Importante ressaltar que um novo token é lido apenas em caso de transição, já que quando uma submáquina é chamada, esta inicia com o mesmo token.

As ações semânticas são chamadas sempre que ocorre transição, chamada de submáquina ou quando está em estado de aceitação da submáquina, porém nem sempre executam alguma função. Isso por que em determinadas transações é preciso realizar ações de controle de tokens, atualizar alguma tabela ou pilha da estrutura de dados ou escopo, para o caso de if por exemplo, é preciso atualizar a pilhaIfs, criar um label para esse if, criar um escopo novo, entre outras ações.

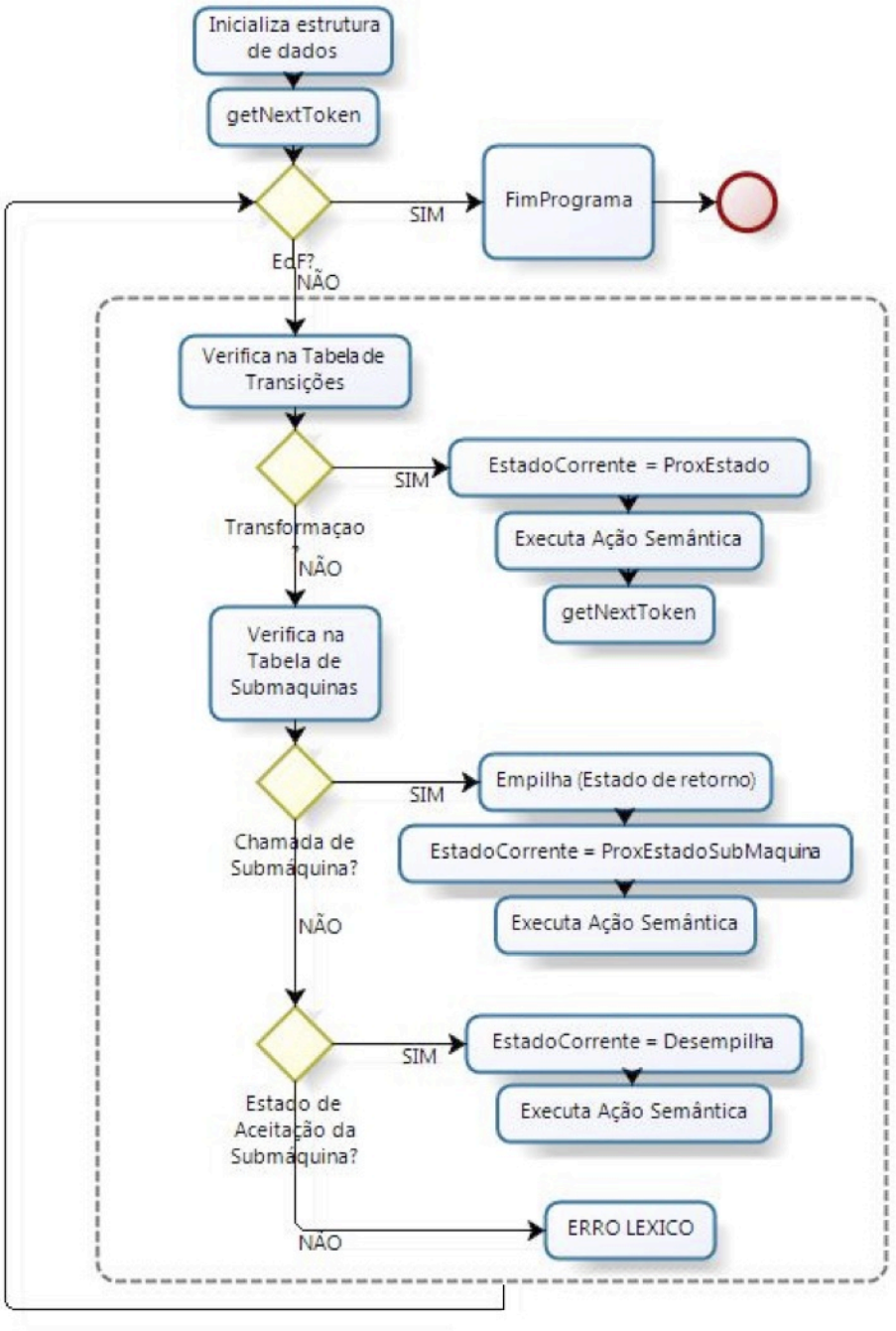


Figura 4. Funcionamento do Compilador

7.1 Estrutura de dados

7.1.1 Escopo

Essa estrutura de dados foi implementada para garantir o controle sobre elementos de diferentes escopos. Cada escopo possui uma tabela de símbolos, referência a um escopo anterior e a escopos internos e um nome.

```
typedef struct _Escopo {
   Simbolo simbolos[MAX_SIMBOLOS];
   struct _Escopo* anterior;
   struct _Escopo* internos[MAX_ESCOPOS];
   int numSimbolos;
   int numInternos;
     char* nome;
} Escopo;
```

7.1.2 Tabelas

Tabela de Simbolos

Essa estrutura de dados foi implementada para garantir o controle das variáveis criadas separando-as por escopo. Cada símbolo é composto por um tipo, o nome (identificador) e o label. Na imlementação do presente compilador foi usado somente tipo int.

Tabela de Constantes

Essa tabela foi implementada para permitir ao compilador armazenar todas as constantes utilizadas no código do programador e para poder declarar essas constantes na área prevista em tempo de compilação. Essa tabela também armazena o label criado pelo compilador para cada constante, da forma `K<valor_da_constante>`.

Tabela de Variáveis Temporárias

Essa tabela foi implementada para guardar os labels das variáveis temporárias criadas em tempo de compilação, essas responsáveis pelo armazenamento de valores intermediários nos cálculos de expressões.

7.1.3 Pilhas

As pilhas a seguir servem para empilhar dois possíveis elementos: ids de comandos de controle de fluxo, isto é, ifs, elses e whiles, ou tokens, no caso de operando e operadores.

Os ids desses comandos de controle são armazenados a fim de se compor labels únicos no código compilado (por exemplo, fim_if_1). Já os tokens são armazenados para se realizar as operações correspondes com os operandos e operadores corretos.

pilhaIfs

Sempre quando houver chamada da submáquina COMANDO_COND (para if ou if-else), empilha-se o id desse comando, que é gerado por um contador.

pilhaElses

Analogamente a pilhaIfs, a pilhaElses também armazena os ids dessas estruturas, sendo que tais ids são sempre iguais aos ids dos ifs correspondentes.

pilhaWhiles

Análoga às anteriores: serve para armazenar os ids dos whiles

pilhaOperadores

Armazena os operadores para a construção de expressões.

pilhaOperandos

Armazena os operandos, tokens relativos aos identificadores, isto é, nomes de variáveis, e constantes. Em conjunto com a pilhaOperadores, são peças-chave para a construção de expressões.

7.2 Tabela de ações semânticas

CABECALHO: escreve o início do programa - @/0000 e JP main

PROGRAM_MAIN: escreve o comando “main OS /0”

PROGRAM_END_MAIN: escreve o comando “HM /0“

VARIAVEL_NA_TABELA: é chamada quando se encontra a declaração de um identificador. Adiciona-se então o símbolo encontrado na tabela de símbolos. Caso essa variável já tenha sido declarada no mesmo escopo ou em escopo mais externo, lança-se o erro de redeclaração de variável. Nessa ação semântica é gerado o label dessa variável. Esse label depende do nome dado ao escopo e do nome da variável.

EMPILHA_IF: empilha o id do if encontrado na pilhaIfs, explicado anteriormente e cria um novo escopo.

APOS_CONDICAO_IF: acrescenta o comando “JZ fim_if_<id>”

TERMINA_IF: acrescenta o label “fim_if_<id>“ para indicar o fim do if e retorna ao escopo anterior.

TERMINA_IF_EMPILHA_ELSE: determina que ao término da execução do código dentro do if (no caso de condição verdadeira) deve-se saltar para o final do else que segue. Nessa mesma ação semântica empilha-se o id do else.

TERMINA_ELSE: escreve o label “fim_else_<id>”

EMPILHA_WHILE: empilha o id do while encontrado na pilhaWhiles, explicado anteriormente e cria um novo escopo.

APOS_CONDICAO_WHILE: acrescenta o comando “JZ while_if_<id>”

TERMINA_WHILE: acrescenta o comando “JP while_<id>“ para retornar ao início do while e em seguida o label “fim_while_<id>“ para indicar o fim do while e retorna ao escopo anterior.

EMPILHA_OPERANDO: somente empilha o token do operando conforme descrito na seção anterior.

EMPILHA_OPERADOR: além de empilhar o token do operador em pilhaOperadores, verifica algumas condições para realizar o cálculo de expressões.

EMPILHA_OPERADOR_PRIORIDADE: análogo ao EMPILHA_OPERADOR, mas com as condições de cálculo de expressões adaptadas para o caso de operadores “*” e ”/”.

SAIDA_EXPRESSAO: termina de calcular a expressão utilizando os operadores e operandos que restaram nas suas respectivas pilhas.

GUARDA_LVALUE: armazena o token relativo à variável a qual será atribuído o valor calculado na expressão.

REALIZA_ATRIBUICAO: move o valor do acumulador para a posição de memória indicado pelo label da variável anterior.

GUARDA_TIPO_COMP: armazena o token de comparação “>”, “<“, “>=“, “<=“, “!=“ e “==”.

REALIZA_COMPARACAO: gera o código relativo a comparação dependendo do seu tipo.

OUTPUT: faz chamada de subrotina para output.

INPUT: faz chamada de subrotina para input e armazena o valor de input na posição de memória da variável indicada.

8. Exemplo de compilação

Funcionamento do Compilador:

- O arquivo a ser compilado deve estar na mesma pasta do compilador e deve ter o nome "ENTRADA.txt";
- Se durante a compilação algum problema for encontrado, o compilador exibirá a linha e a coluna do arquivo de entrada que gerou o erro e o tipo de erro, se léxico ou sintático.
- Se não houver nenhum erro de compilação o compilador irá gerar, também no mesmo diretório, um arquivo chamado "SAIDA.txt" com o resultado da compilação.
- Para verificar se a compilação está correta utilizamos o montador para gerar o bytecode e para finalmente executar o MVN.

8.1 Código Compilado

```
main()
     int fat
     int n
     input fat
     n = fat - 1
     if(fat == 0)
          output 1
     else
          while(n > 0)
               fat = fat * n
               n = n - 1
          end
          output fat
     end
end
```

8.2 Código Gerado

```
@ /000
 JP main
;*******************************
; Programa
;*******************************
```

```
main OS /0
 SC input
 MM main_fat
 LD main_fat
- K1
 MM TEMP0
 MM main_n
if_0 OS /0
 LD main_fat
 MM TEMP1
 LD K0
 - TEMP1
 JZ comp_true_0
comp_false_0 LV /0
 JP fim_comp_0
comp_true_0 LV /1
fim_comp_0 OS /0
 JZ fim_if_0
 LD K1
 SC output
 JP fim_else_0
fim_if_0 OS /0
while_0 OS /0
 LD main_n
 MM TEMP2
 LD K0
 - TEMP2
 JN comp_true_1
comp_false_1 LV /0
 JP fim_comp_1
comp_true_1 LV /1
fim_comp_1 OS /0
 JZ fim_while_0
 LD main_fat
* main_n
 MM TEMP3
 MM main_fat
 LD main_n
- K1
 MM TEMP4
 MM main_n
 JP while_0
fim_while_0 OS /0
 LD main_fat
 SC output
fim_else_0 OS /0
```

```
 HM /0
;*******************************
; Subrotinas da MVN
;*******************************
output OS /0
 MM NUM
 LD NUM
w OS /0
 JZ fim_w
 / Ka
 MM NUM_PROX
 * Ka
 MM NUM_TEMP
 LD K9K
 + SP
 + SP_INICIO
 MM EMPILHA
 LD SP
 + K2
 MM SP
 LD NUM
 - NUM_TEMP
EMPILHA OS /0
 LD NUM_PROX
 MM NUM
 JP w
fim_w OS /0
w1 OS /0
 LD SP
 JZ fw0
 LD SP
 - K2
 MM SP
 LD K8K
 + SP
 + SP_INICIO
 MM Des
Des OS /0
 + K48
 PD /100
 JP w1
fw0 OS /0
 RS output
input OS /0
 LV /30
 PD /100
 LV /2d
 PD /100
 LV /38
 PD /100
```

```
 LV /3e
 PD /100
 LV /20
 PD /100
 GD /0
 / K256
 - K48
 RS input
;*******************************
; Area de variaveis e constantes
;*******************************
TEMP1 K /0
TEMP2 K /0
TEMP3 K /0
TEMP4 K /0
main_fat K /0
main_n K /0
TEMP0 K /0
K1 K /1
K0 K /0
K256 K /100
K48 K /30
K2 K /2
NUM         K /0000
NUM_PROX    K /0000
NUM_TEMP    K /0000
SP          K /0000
SP_INICIO   K /0F00
Ka          K /000A
K9K         K /9000
K8K         K /8000
# main
```