Gustavo P. Gouveia (6482819), Victor Lassance (6431325)

Relatório de Compiladores
Relatório Final
Linguagem de programação CZAR

Texto apresentado à Escola Politécnica da Universidade de São Paulo como requisito para a aprovação na disciplina Linguagens e Compiladores no quinto módulo acadêmico do curso de graduação em Engenharia de Computação, junto ao Departamento de Engenharia de Computação e Sistemas Digitais (PCS).

Universidade de São Paulo

Escola Politécnica

Engenharia de Computação - Curso Cooperativo

Professor: Ricardo Luis de Azevedo da Rocha

São Paulo

2013

Sumário

1 Introdução

Este projeto tem como objetivo a construção de um compilador de um só passo, dirigido por sintaxe, com analisador e reconhecedor sintático baseado em autômato de pilha estruturado.

Em um primeiro momento, foi definida uma linguagem de programação e identificados os tipos de átomos. Para cada átomo foi escrito uma gramática linear representativa da sua lei de formação e um reconhecedor para o átomo. Desse modo, as gramáticas assim escritas foram unidas e convertidas em um autômato finito, o qual foi transformado em um transdutor e implementado como sub-rotina, dando origem ao analisador léxico propriamente dito. Também foi criada uma função principal para chamar o analisador léxico e possibilitar o seu teste.

Durante a segunda etapa, a sintaxe da linguagem, denonimada por nós de CZAR, foi definida formalmente a partir de uma definição informal e de exemplos de programas que criamos, misturando palavras-chave e conceitos de diferentes linguagens de programação. As três principais definições foram escritas na notação BNF[1], Wirth[2] e com diagramas de sintaxe.

Na terceira etapa, implementamos o módulo referente à parte sintática para a nossa linguagem. O analisador sintático construído obtém uma cadeia de *tokens* proveniente do analisador léxico, e verifica se a mesma pode ser gerada pela gramática da linguagem e, com isso, constrói a árvore sintática (ALFRED; SETHI; JEFFREY, 1986).

Para a quarta entrega, focamos no ambiente de execução. O compilador por nós criado tem como linguagem de saída um programa que é executado na máquina virtual conhecida como Máquina de von Neumann (MVN).

Já durante as duas últimas entregas, complementamos a especificação do código gerado pelo compilador e das rotinas do ambiente de execução da nossa linguagem de alto nível, a CZAR. Além disso, buscamos integrar as rotinas semânticas no reconhecedor sintático de forma a permitir a geração de código e finalizar o compilador.

Como material de consulta, além de sites sobre o assunto e das aulas ministradas, foi utilizado o livro indicado pelo professor no começo das aulas (NETO, 1987), para pesquisa de conceitos e possíveis implementações.

O documento apresenta a seguir o processo completo de desenvolvimento de um compilador, desde a definição formal da linguagem, passando pelo analisador léxico, re-

[1] Ver http://en.wikipedia.org/wiki/Backus_Naur_Form
[2] Ver http://en.wikipedia.org/wiki/Wirth_syntax_notation

conhecedor sintático, pela definição do ambiente de execução e das rotinas semânticas, terminando com um exemplo de programa traduzido.

2 Definição da linguagem

2.1 Descrição Informal da Linguagem

O programa é composto por quatro partes, explicadas abaixo de forma simplificada, pois a linguagem será definida de forma completa nas seções 2.2 e 2.3 nas notações `BNF` e `Wirth`, respectivamente:

- Definição do programa:

  Um programa em `czar` possui em ordem obrigatória, a importação de bibliotecas, declaração de variáveis globais, definição de funções e procedimentos. O programa deve terminar obrigatoriamente pela declaração da função principal `main`.

  - `PROGRAM = IMPORTS DECLS_GLOBAIS DEF_PROCS_FUNCS DEF_MAIN.`

- Inclusão de bibliotecas:

  - `IMPORTS = { '<' IDENT '>' }.`

- Declaração de tipos, variáveis e constantes de escopo global:

  - `DECLS_GLOBAIS = { DEF_TIPO | DECL }.`
  - `DEF_TIPO = 'struct' IDENT '{' { DECL } '}'.`
  - `DECL = [ 'const' ] TIPO IDENT [ '=' EXPR ]`
    `{ ',' IDENT [ '=' EXPR ] } ';'.`

- Definição dos procedimentos e funções do programa, As funções não devem incluir o procedimento principal (chamado `main`). Estas também possuem retorno final único e obrigatório.

  - `DEF_PROCS_FUNCS = { PROC | FUNC }.`
  - `FUNC = TIPO IDENT LIST_PARAMS`
    `'{' { INSTR_SEM_RET } "return" EXPR [ ";" ] '}'.`
  - `PROC = 'void' IDENT LIST_PARAMS '{' { INSTR_SEM_RET } '}'.`
  - `LIST_PARAMS = '(' [ [ 'ref' ] TIPO IDENT ]`
    `{ ',' [ 'ref' ] TIPO IDENT } ')'.`

- Definição do procedimento principal (chamado `main`):

Não existe passagem explícita de parâmetros para a função `main`, ou seja, a passagem de valores para a mesma deve ocorrer por meio de arquivos ou pela utilização de uma função incluida por alguma biblioteca *built-in* a ser feita, permitindo o acesso dos parâmetros em todas as partes do código.

– `DEF_MAIN = 'main' '(' ')' '{' [ BLOCO ] '}'.`

2.2 Descrição da Linguagem em BNF

Apesar de termos visto em aula que a sintaxe BNF não costuma ter os não-terminais explicitados entre aspas, prefirimos colocar aspas simples para facilitar a leitura, que também é uma forma válida de sintaxe BNF[1].

```
1  <PROGRAM>            ::= <IMPORTS> <DECLS_GLOBAIS> <DEF_PROCS_FUNCS> <DEF_MAIN>
2
3  <IMPORTS>            ::= ϵ
4                          | '<' <IDENT> '>' <IMPORTS>
5
6  <DECLS_GLOBAIS>      ::= ϵ
7                          | <DEF_TIPO> <DECLS_GLOBAIS>
8                          | <DECL_CONST> <DECLS_GLOBAIS>
9                          | <DECL_VAR> <DECLS_GLOBAIS>
10
11 <DEF_TIPO>           ::= 'struct' <IDENT> '{' <DEF_INSTR_TIPO> '}'
12
13 <DEF_INSTR_TIPO>     ::= ϵ
14                         | <DECL_CONST>
15                         | <DECL_VAR>
16
17 <DECL_CONST>         ::= 'const' <DECL_VAR>
18
19 <DECL_VAR>           ::= <TIPO> <IDENT> <DECL_VAR_CONT> ';'
20                         | <TIPO> <IDENT> '=' <EXPR> <DECL_VAR_CONT> ';'
21
22 <DECL_VAR_CONT>      ::= ϵ
23                         | ',' <IDENT> <DECL_VAR_CONT>
24                         | ',' <IDENT> '=' <EXPR> <DECL_VAR_CONT>
25
26 <DEF_PROCS_FUNCS>    ::= ϵ
27                         | <PROC> <DEF_PROCS_FUNCS>
28                         | <FUNC> <DEF_PROCS_FUNCS>
29
30 <FUNC>               ::= <TIPO> <IDENT> <LIST_PARAMS> '{' <INSTRUCOES> 'return'
       <EXPR> '}'
31                         | <TIPO> <IDENT> <LIST_PARAMS> '{' <INSTRUCOES> 'return'
                              <EXPR> ';' '}'
32
33 <PROC>               ::= 'void' <IDENT> <LIST_PARAMS> '{' <INSTRUCOES> '}'
34
35 <INSTRUCOES>         ::= ϵ
36                         | <INSTR_SEM_RET> <INSTRUCOES>
```

[1] Ver http://www.cs.man.ac.uk/ pjj/bnf/bnf.html

```
37
38 <LIST_PARAMS>         ::= '(' ')'
39                          | '(' <TIPO> <IDENT> <LIST_PARAMS_CONT> ')'
40                          | '(' 'ref' <TIPO> <IDENT> <LIST_PARAMS_CONT> ')'
41
42 <LIST_PARAMS_CONT>    ::= ϵ
43                          | ',' <TIPO> <IDENT> <LIST_PARAMS_CONT>
44                          | ',' 'ref' <TIPO> <IDENT> <LIST_PARAMS_CONT>
45
46 <DEF_MAIN>            ::= 'main' '(' ')' '{' <INSTRUCOES> '}'
47
48 <INSTR_SEM_RET>       ::= <DECL_VAR>
49                          | <ATRIB>
50                          | <PROC_CALL>
51                          | <FLOW_CONTROL>
52
53 <ATRIB>               ::= <ATRIB_SEM_PV> ';'
54
55 <ATRIB_SEM_PV>        ::= <VARIDENT> <OPER_ATRIB> <EXPR> <ATRIB_SEM_PV_CONT>
56
57 <ATRIB_SEM_PV_CONT>   ::= ϵ
58                          | ',' <VARIDENT> <OPER_ATRIB> <EXPR> <ATRIB_SEM_PV_CONT>
59
60 <PROC_CALL>           ::= <IDENT> '(' ')' ';'
61                          | <IDENT> '(' <EXPR> <PROC_CALL_CONT> ')' ';'
62
63 <PROC_CALL_CONT>      ::= ϵ
64                          | ',' <EXPR> <PROC_CALL_CONT>
65
66 <FLOW_CONTROL>        ::= <FOR_CONTROL>
67                          | <WHILE_CONTROL>
68                          | <IF_CONTROL>
69
70 <FOR_CONTROL>         ::= 'for' '(' <DECL_VAR> <COND> ';' <ATRIB_SEM_PV> ')' '{'
       <INSTRUCOES> '}'
71
72 <WHILE_CONTROL>       ::= 'while' '(' <COND> ')' '{' <INSTRUCOES> '}'
73
74 <IF_CONTROL>          ::= 'if' '(' <COND> ')' '{' <INSTRUCOES> '}'
75                          | 'if' '(' <COND> ')' '{' <INSTRUCOES> '}' 'else' '{' <
                                INSTRUCOES> '}'
76
77 <TIPO>                ::= <IDENT> <TIPO_CONT>
78
79 <TIPO_CONT>           ::= ϵ
80                          | '[' <INT> ']' <TIPO_CONT>
81
82 <IDENT_COLCHETES>     ::= <IDENT> <IDENT_COLCH_CONT>
83
84 <IDENT_COLCH_CONT>    ::= ϵ
85                          | '[' <EXPR> ']' <IDENT_COLCH_CONT>
86
87 <VARIDENT>            ::= <IDENT_COLCHETES> <VARIDENT_CONT>
88
89 <VARIDENT_CONT>       ::= ϵ
90                          | '.' <VARIDENT> <VARIDENT_CONT>
91
```

```
92  <FUNCTION_CALL>       ::= <IDENT> '(' ')'
93                          | <IDENT> '(' <EXPR> <FUNCTION_CALL_CONT> ')'
94
95  <FUNCTION_CALL_CONT>  ::= ϵ
96                          | ',' <EXPR> <FUNCTION_CALL_CONT>
97
98  <COND>                ::= <COND_TERM>
99                          | <COND_TERM> <OPER_BOOL> <COND_TERM>
100
101 <COND_TERM>            ::= '(' <COND> ')'
102                         | <ATOMO_COND>
103                         | <ATOMO_COND> <OPER_COMP> <COND_TERM>
104
105 <ATOMO_COND>           ::= <VARIDENT>
106                         | 'true'
107                         | 'false'
108                         | <INT>
109                         | 'not' <ATOMO_COND>
110
111 <OPER_ATRIB>           ::= '+='
112                         | '-='
113                         | '*='
114                         | '/='
115                         | '%='
116                         | '=='
117
118 <OPER_BOOL>            ::= 'and'
119                         | 'or'
120
121 <OPER_COMP>            ::= '=='
122                         | '!='
123                         | '<='
124                         | '>='
125
126 <OPER_ARIT>            ::= '+'
127                         | '-'
128
129 <OPER_TERM>            ::= '*'
130                         | '/'
131                         | '%'
132
133 <EXPR>                 ::= <TERM>
134                         | <TERM> <OPER_ARIT_TERM_ARR>
135                         | <OPER_ARIT_TERM_ARR>
136
137 <OPER_ARIT_TERM_ARR>  ::= <OPER_ARIT> <TERM>
138                         | <OPER_ARIT> <TERM> <OPER_ARIT_TERM_ARR>
139
140 <TERM>                 ::= '(' <EXPR> ')'
141                         | <ATOMO>
142                         | <ATOMO> <OPER_TERM_ATOMO_ARR>
143
144 <OPER_TERM_ATOMO_ARR>::= <OPER_TERM> <ATOMO>
145                         | <OPER_TERM> <ATOMO> <OPER_TERM_ATOMO_ARR>
146
147 <ATOMO>                ::= <FUNCTION_CALL>
148                         | <OPER_ARIT> <FUNCTION_CALL>
```

```
149                         | <INT>
150                         | <OPER_ARIT> <INT>
151                         | <STRING>
152                         | <CHAR>
153                         | <FLOAT>
154                         | <OPER_ARIT> <FLOAT>
155                         | <BOOL>
156                         | <VARIDENT>
157                         | <OPER_ARIT> <VARIDENT>
```

2.3 Descrição da Linguagem em Wirth

```
1  PROGRAM         = IMPORTS DECLS_GLOBAIS DEF_PROCS_FUNCS DEF_MAIN.
2
3  IMPORTS         = { "<" IDENT ">" }.
4
5  DECLS_GLOBAIS   = { DEF_TIPO | DECL_CONST | DECL_VAR }.
6  DEF_TIPO        = "struct" IDENT "{" { DECL_CONST | DECL_VAR } "}".
7  DECL_CONST      = "const" DECL_VAR.
8  DECL_VAR        = TIPO IDENT [ "=" EXPR ] { "," IDENT [ "=" EXPR ] } ";".
9
10 DEF_PROCS_FUNCS = { PROC | FUNC }.
11 FUNC            = TIPO IDENT LIST_PARAMS "{" { INSTR_SEM_RET } "return" EXPR [
       ";" ] "}".
12 PROC            = "void" IDENT LIST_PARAMS "{" { INSTR_SEM_RET } "}".
13 LIST_PARAMS     = "(" [ [ "ref" ] TIPO IDENT { "," [ "ref" ] TIPO IDENT } ] ")".
14
15 DEF_MAIN        = "main" "(" ")" "{" { INSTR_SEM_RET } "}".
16
17 INSTR_SEM_RET   = DECL_VAR | ATRIB | PROC_CALL | FLOW_CONTROL.
18 ATRIB           = ATRIB_SEM_PV ";".
19 ATRIB_SEM_PV    = VARIDENT OPER_ATRIB EXPR { "," VARIDENT OPER_ATRIB EXPR }.
20 PROC_CALL       = IDENT "(" [ EXPR { "," EXPR } ] ")" ";".
21 FLOW_CONTROL    = FOR_CONTROL | WHILE_CONTROL | IF_CONTROL.
22 FOR_CONTROL     = "for" "(" DECL_VAR COND ";" ATRIB_SEM_PV ")" "{" {
       INSTR_SEM_RET } "}".
23 WHILE_CONTROL   = "while" "(" COND ")" "{" { INSTR_SEM_RET } "}".
24 IF_CONTROL      = "if" "(" COND ")" "{" { INSTR_SEM_RET } "}" ["else" "{" {
       INSTR_SEM_RET } "}"].
25
26 TIPO            = IDENT { "[" INT "]" }.
27 IDENT_COLCHETES = IDENT { "[" EXPR "]" }.
28 VARIDENT        = IDENT_COLCHETES { "." VARIDENT}.
29
30 FUNCTION_CALL   = IDENT "(" [ EXPR { "," EXPR } ] ")".
31
32 COND            = COND_TERM { OPER_BOOL COND_TERM}.
33 COND_TERM       = "(" COND ")" | ATOMO_COND {OPER_COMP ATOMO_COND}.
34 ATOMO_COND      = VARIDENT | BOOL | INT | "not" ATOMO_COND.
35 BOOL            = "true" | "false".
36
37 OPER_ATRIB      = ["+" | "-" | "*" | "/" | "%"] "=".
38 OPER_BOOL       = "and" | "or".
39 OPER_COMP       = ("=" | "!" | "<" | ">") "=".
```

```
40 OPER_ARIT        = "+" | "-".
41 OPER_TERM        = "*" | "/" | "%".
42
43 EXPR             = [ OPER_ARIT ] TERM { OPER_ARIT TERM}.
44 TERM             = "(" EXPR ")" | ATOMO {OPER_TERM ATOMO}.
45 ATOMO            = [ OPER_ARIT ] FUNCTION_CALL | [ OPER_ARIT ] INT | STRING |
       CHAR | [ OPER_ARIT ] FLOAT | BOOL | [ OPER_ARIT ] VARIDENT.
```

2.4 Diagrama de Sintaxe da Linguagem

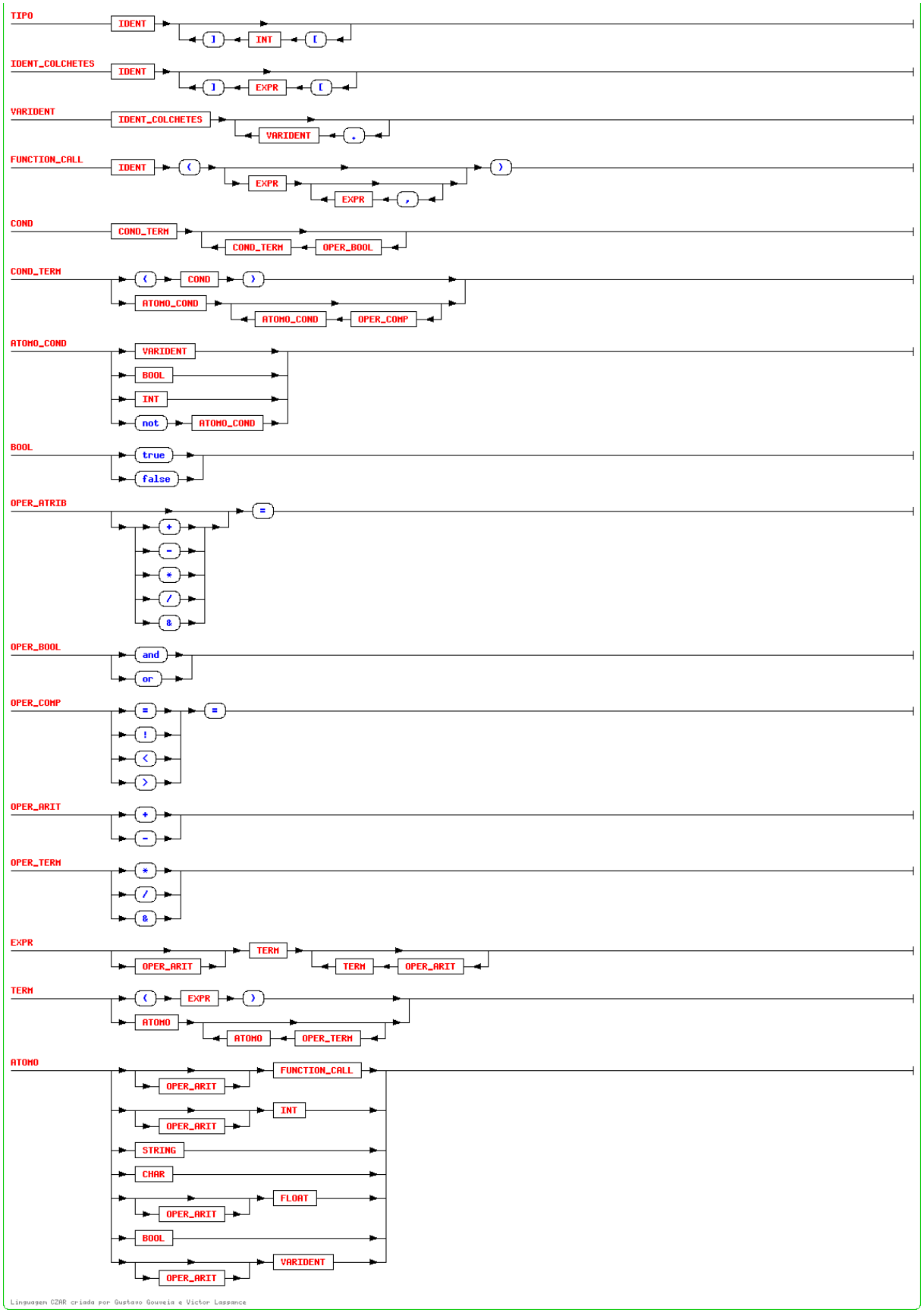


Figura 1 – Diagrama de Sintaxe da Linguagem CZAR

3 Descrição do analisador léxico

O analisador léxico atua como uma interface entre o reconhecedor sintático, que forma, normalmente, o núcleo do compilador, e o texto de entrada, convertendo a sequência de caracteres de que este se constitui em uma sequência de átomos.

Para a consecução de seus objetivos, o analisador léxico executa usualmente uma série de funções, todas de grande importância como infraestrutura para a operação das partes do compilador mais ligadas à tradução propriamente dita do texto-fonte. As principais funções são listadas abaixo:

- Extração e Classificação de Átomos;
  - Principal funcionalidade do analisador;
  - As classes de átomos mais usuais: identificadores, palavras reservadas, números inteiros sem sinal, números reais, strings, sinais de pontuação e de operação, caracteres especiais, símbolos compostos de dois ou mais caracteres especiais e comentários.
- Eliminação de Delimitadores e Comentários;
- Conversão numérica;
  - Conversão numérica de notações diversas em uma forma interna de representação para manipulação de pelos demais módulos do compilador.
- Tratamento de Identificadores;
  - Tratamento com auxílio de uma tabela de símbolos.
- Identificação de Palavras Reservadas;
  - Verificar se cada identificador reconhecido pertence a um conjunto de identificadores especiais.
- Recuperação de Erros;
- Listagens;
  - Geração de listagens do texto-fonte.
- Geração de Tabelas de Referências Cruzadas;
  - Geração de listagem indicativa dos símbolos encontrados, com menção à localização de todas as suas ocorrências no texto do programa-fonte.

- Definição e Expansão de Macros;
  - Pode ser realizado em um pré-processamento ou no analisador léxico. No caso do analisador, deve-se haver uma comunicação entres os analisadores léxico e sintático.
- Interação com o sistema de arquivos;
- Compilação Condicional;
- Controles de Listagens.
  - São os comandos que permitem ao programador que ligue e desligue opções de listagem, de coleta de símbolos em tabelas de referência cruzadas, de geração, e impressão de tais tabelas, de impressão de tabelas de símbolos do programa compilador, de tabulação e formatação das saídas impressas do programa-fonte.

Nas próximas seções definiremos com detalhes cada um dos passos para a criação e teste do analisador léxico.

3.1 Descrição do funcionamento

O analisador léxico lê um arquivo de configuração da máquina de estados (transdutor). O mesmo pode ser comparado à seguinte regex:

```
(.)([^\1]*)\1\s*([A-Za-z]+(:?_[0-9]+)?)\s*([A-Za-z]+(:?_[0-9]+)?)
```

Cada linha possui uma lista de caracteres delimitados por um caractere especial (por exemplo '+' ou '`#`') e dois identificadores que designam os estados inicial e final da transição. O caractere `@` designa todas as transições, este é usado principalmente para encaminhar qualquer aceitação final de um sub-autômato ao estado `Q0`, para então ser tratado normalmente.

Ex:

```
+abcdefghijklmnopqrstuvxywz+    Q0      IDENT
+ABCDEFGHIJKLMNOPQRSTUVXYWZ+    Q0      IDENT
+_+                             Q0      IDENT
```

Este exemplo vai reconhecer todos os caracteres a-z e A-Z mais o underscore (_) como transições do estado `Q0` e `IDENT`.

Após a leitura do arquivo de configuração e de um arquivo com *keywords*, faz-se a leitura do arquivo fonte, por meio do transdutor, percorrendo-se o arquivo fonte token a token. Uma função `next_useful_token` oferece o não retorno dos tokens de espaço, tal qual a quebra de linha e espaços normais, além de ignorar comentários. O motor do

lex também substitui classes `IDENT` para `RESERVED` se a palavra se encontra na lista de identificadores reservados.

3.2 Autômatos finitos por *Token*

- DELIM: `/[{}()\[\];]/`

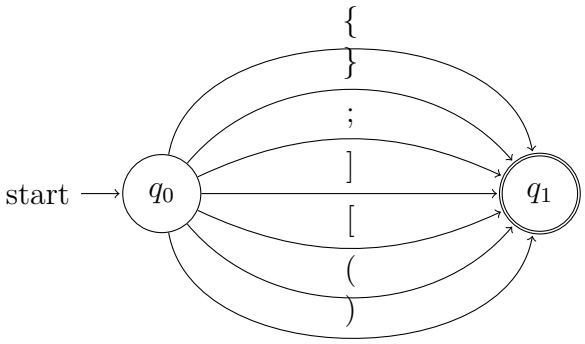


Figura 2 – Autômato finito DELIM

- SPACE: `/[ \t\r\n\v\f]+/`

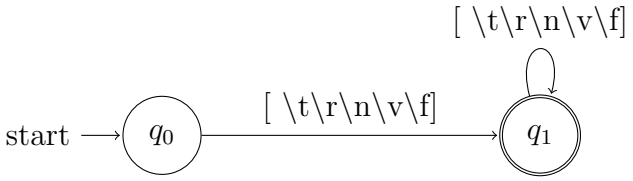


Figura 3 – Autômato finito SPACE

- COMMENT: `/#[^\n]*/`

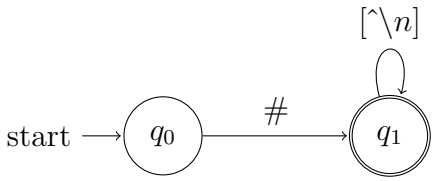


Figura 4 – Autômato finito COMMENT

- IDENT: `/[a-zA-Z_][a-zA-Z0-9_]*/`

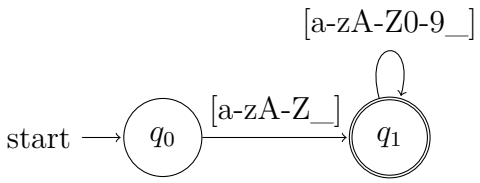


Figura 5 – Autômato finito IDENT

- INTEGER: `/[0-9]+/`

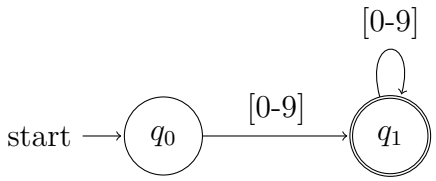


Figura 6 – Autômato finito INTEGER

- FLOAT: `/[0-9]*\.[0-9]+/`

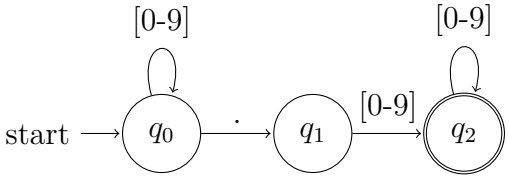


Figura 7 – Autômato finito FLOAT

- CHAR: `/'(?:\\[0abtnvfre\\'"]|[\x20-\x5B\x5D-\x7E])'/`

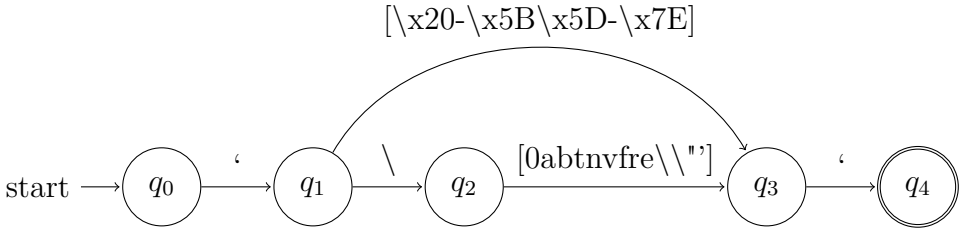


Figura 8 – Autômato finito CHAR

- STRING: `/"(?:\\"|[^"])*"/`

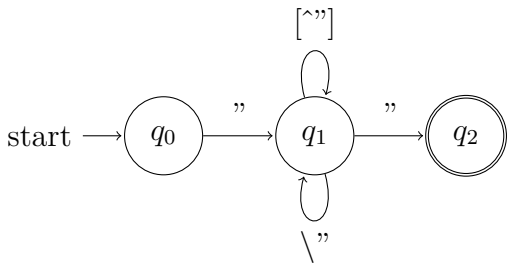


Figura 9 – Autômato finito STRING

- OPER: `/[\+\-\*\/%=!<>][=]?/`

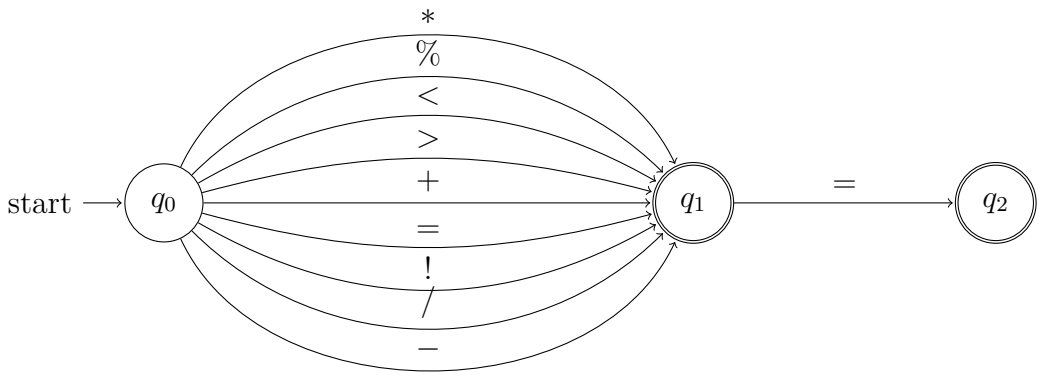
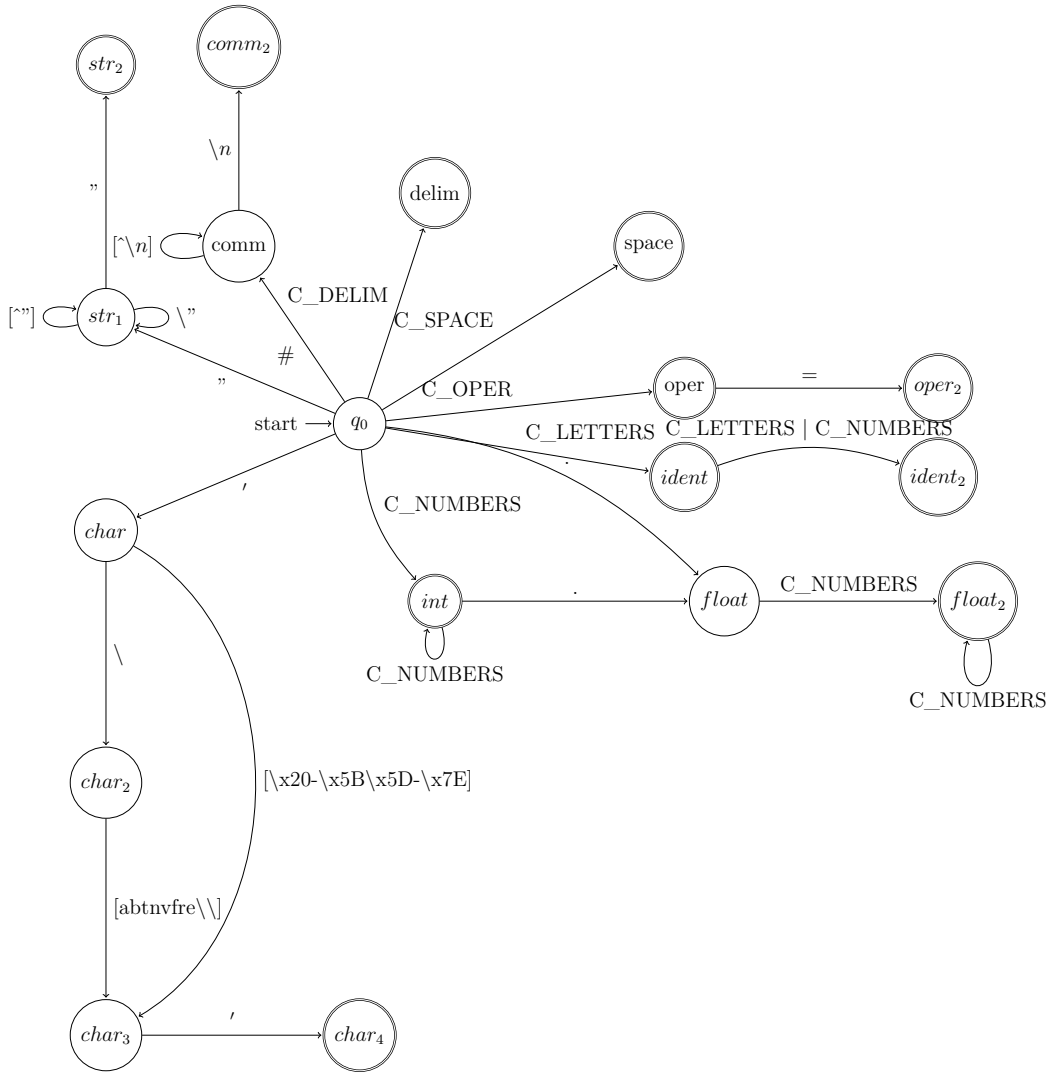


Figura 10 – Autômato finito OPER

3.3 Autômato finito único

- C_DELIM = $[91, 93, 123, 125, 40, 41, 59]$
- C_SPACE = $[32, 9, 10, 11, 12, 13]$
- C_OPER = $[42, 37, 60, 62, 43, 61, 33, 47, 45]$
- C_LETTERS = $[65, \ldots, 90, 97, \ldots, 122, 95]$
- C_NUMBERS = $[48, 57]$



3.4 Trandutor léxico

O transdutor obtido a partir do autômato finito únco pode ser visto na Figura 11.

3.5 Testes

Um código de exemplo que foi utilizado para teste está listado abaixo:

ENTRADA.txt

```
1  int main() {
2      zhis = 2;
3      print(zhis);
4      ozer = "a\"nother";
5      abc = '\r';     # this is a comment
6      while ( a >= 0 ) {
7          if ( b == 0 ) {
8              b = a;
```

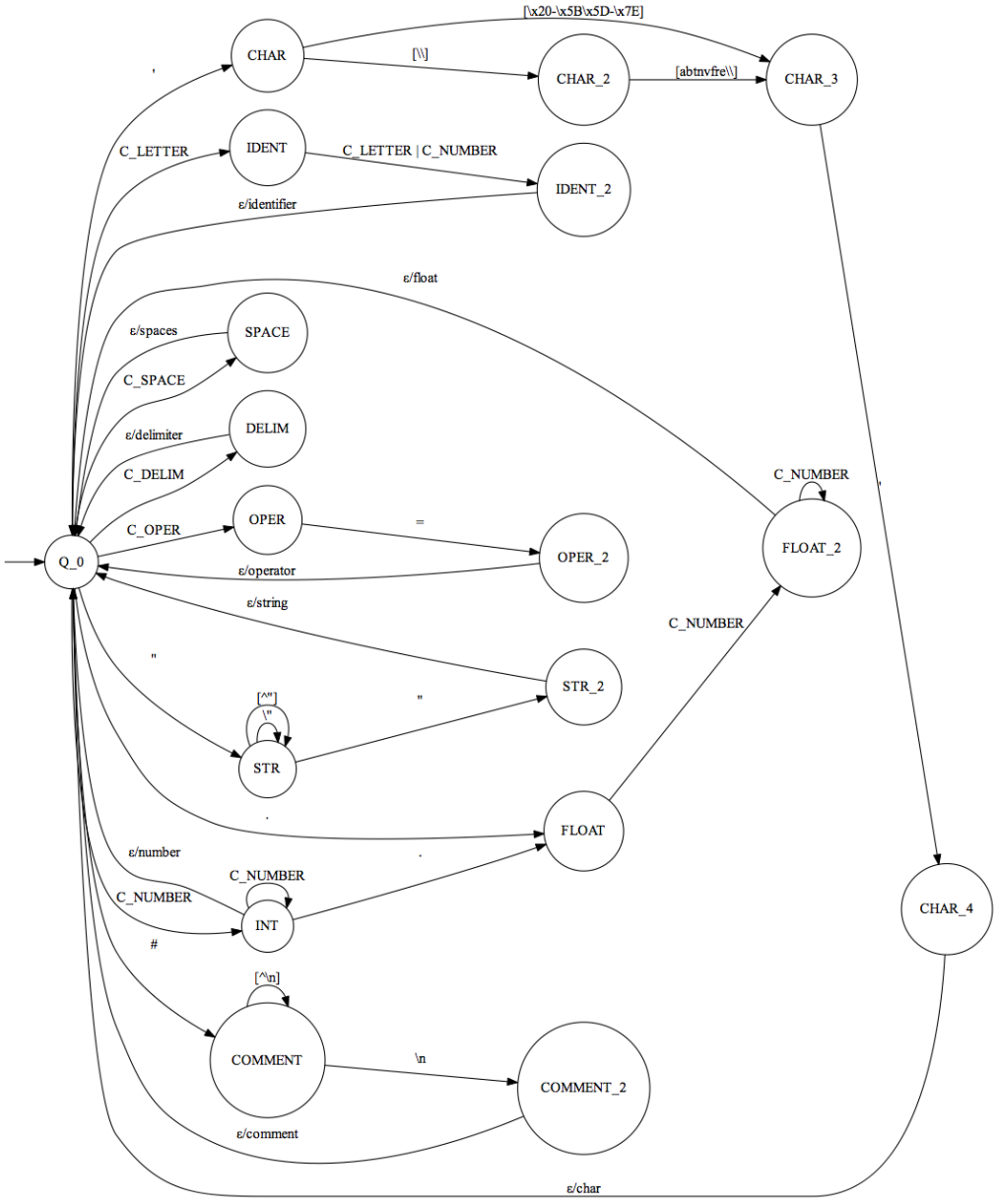


Figura 11 – Transdutor léxico da Linguagem *CZAR*

```
9           }
10          a = a - 1;
11      }
12      print(ozer);
13      eis = 'I';
14      e = '\n';
15      cerr = 'i';
16      return 0;
17  }
```

Ao utilizar o código acima como *input*, obtivemos o seguinte resultado, que foi de acordo com o esperado:

Resultado sem erros

```
1  > [RESERVED] >>int<< at (1, 1), with size 3
2  > [RESERVED] >>main<< at (1, 5), with size 4
3  > [DELIM] >>(<< at (1, 9), with size 1
4  > [DELIM] >>)<< at (1, 10), with size 1
5  > [DELIM] >>{<< at (1, 12), with size 1
6  > [IDENT] >>zhis<< at (2, 5), with size 4
7  > [OPER] >>=<< at (2, 10), with size 1
8  > [INT] >>2<< at (2, 12), with size 1
9  > [DELIM] >>;<< at (2, 13), with size 1
10 > [IDENT] >>print<< at (3, 5), with size 5
11 > [DELIM] >>(<< at (3, 10), with size 1
12 > [IDENT] >>zhis<< at (3, 11), with size 4
13 > [DELIM] >>)<< at (3, 15), with size 1
14 > [DELIM] >>;<< at (3, 16), with size 1
15 > [IDENT] >>ozer<< at (4, 5), with size 4
16 > [OPER] >>=<< at (4, 10), with size 1
17 > [STR] >>"a\"nother"<< at (4, 12), with size 11
18 > [DELIM] >>;<< at (4, 23), with size 1
19 > [IDENT] >>abc<< at (5, 5), with size 3
20 > [OPER] >>=<< at (5, 9), with size 1
21 > [CHAR] >>'\r'<< at (5, 11), with size 4
22 > [DELIM] >>;<< at (5, 15), with size 1
23 > [RESERVED] >>while<< at (6, 5), with size 5
24 > [DELIM] >>(<< at (6, 11), with size 1
25 > [IDENT] >>a<< at (6, 13), with size 1
26 > [OPER] >>>=<< at (6, 15), with size 2
27 > [INT] >>0<< at (6, 18), with size 1
28 > [DELIM] >>)<< at (6, 20), with size 1
29 > [DELIM] >>{<< at (6, 22), with size 1
30 > [RESERVED] >>if<< at (7, 9), with size 2
31 > [DELIM] >>(<< at (7, 12), with size 1
32 > [IDENT] >>b<< at (7, 14), with size 1
33 > [OPER] >>==<< at (7, 16), with size 2
```

```
> [INT] >>0<< at (7, 19), with size 1
> [DELIM] >>)<< at (7, 21), with size 1
> [DELIM] >>{<< at (7, 23), with size 1
> [IDENT] >>b<< at (8, 13), with size 1
> [OPER] >>=<< at (8, 15), with size 1
> [IDENT] >>a<< at (8, 17), with size 1
> [DELIM] >>;<< at (8, 18), with size 1
> [DELIM] >>}<< at (9, 9), with size 1
> [IDENT] >>a<< at (10, 9), with size 1
> [OPER] >>=<< at (10, 11), with size 1
> [IDENT] >>a<< at (10, 13), with size 1
> [OPER] >>-<< at (10, 15), with size 1
> [INT] >>1<< at (10, 17), with size 1
> [DELIM] >>;<< at (10, 18), with size 1
> [DELIM] >>}<< at (11, 5), with size 1
> [IDENT] >>print<< at (12, 5), with size 5
> [DELIM] >>(<< at (12, 10), with size 1
> [IDENT] >>ozer<< at (12, 11), with size 4
> [DELIM] >>)<< at (12, 15), with size 1
> [DELIM] >>;<< at (12, 16), with size 1
> [IDENT] >>eis<< at (13, 5), with size 3
> [OPER] >>=<< at (13, 9), with size 1
> [CHAR] >>'I'<< at (13, 11), with size 3
> [DELIM] >>;<< at (13, 14), with size 1
> [IDENT] >>e<< at (14, 5), with size 1
> [OPER] >>=<< at (14, 7), with size 1
> [CHAR] >>'\n'<< at (14, 9), with size 4
> [DELIM] >>;<< at (14, 13), with size 1
> [IDENT] >>cerr<< at (15, 5), with size 4
> [OPER] >>=<< at (15, 10), with size 1
> [CHAR] >>'i'<< at (15, 12), with size 3
> [DELIM] >>;<< at (15, 15), with size 1
> [RESERVED] >>return<< at (16, 5), with size 6
> [INT] >>0<< at (16, 12), with size 1
> [DELIM] >>;<< at (16, 13), with size 1
> [DELIM] >>}<< at (17, 1), with size 1

Lista de identificadores:

>> zhis
>> print
>> ozer
>> abc
>> a
>> b
>> eis
>> e
```

```
>> cerr
```

Ao introduzir um erro colocando mais de uma letra como caracter, obtivemos, como esperado, o seguinte resultado:

Resultado com erro

```
> [RESERVED] >>int<< at (1, 1), with size 3
> [RESERVED] >>main<< at (1, 5), with size 4
> [DELIM] >>(<< at (1, 9), with size 1
> [DELIM] >>)<< at (1, 10), with size 1
> [DELIM] >>{<< at (1, 12), with size 1
> [IDENT] >>zhis<< at (2, 5), with size 4
> [OPER] >>=<< at (2, 10), with size 1
> [INT] >>2<< at (2, 12), with size 1
> [DELIM] >>;<< at (2, 13), with size 1
> [IDENT] >>print<< at (3, 5), with size 5
> [DELIM] >>(<< at (3, 10), with size 1
> [IDENT] >>zhis<< at (3, 11), with size 4
> [DELIM] >>)<< at (3, 15), with size 1
> [DELIM] >>;<< at (3, 16), with size 1
> [IDENT] >>ozer<< at (4, 5), with size 4
> [OPER] >>=<< at (4, 10), with size 1
> [STR] >>"a\"nother"<< at (4, 12), with size 11
> [DELIM] >>;<< at (4, 23), with size 1
> [IDENT] >>abc<< at (5, 5), with size 3
> [OPER] >>=<< at (5, 9), with size 1
buff_token (2): <'ab>, error at line 5 column 13

Lista de identificadores:

>> zhis
>> print
>> ozer
>> abc
```

4 Descrição do reconhecedor sintático

4.1 Descrição Atualizada da Linguagem em Wirth

```
1  PROGRAM          = IMPORTS DECLS_GLOBAIS DEF_PROCS_FUNCS "main" DEF_MAIN.
2  IMPORTS          = { "<" IDENT ">" } .
3
4  DECLS_GLOBAIS    = { "struct" IDENT "{" { [ "const" ] IDENT { "[" INT "]" } IDENT
       [ "=" EXPR ] { "," IDENT [ "=" EXPR ] } ";" } "}" | [ "const" ] IDENT { "["
       INT "]" } IDENT [ "=" EXPR ] { "," IDENT [ "=" EXPR ] } ";" } "meth".
5
6  DEF_PROCS_FUNCS = { "void" IDENT LIST_PARAMS "{" { INSTR_SEM_RET } "}" | IDENT {
       "[" INT "]" } IDENT LIST_PARAMS "{" { INSTR_SEM_RET } "return" EXPR [ ";" ]
       "}" }.
7  LIST_PARAMS      = "(" [ [ "ref" ] IDENT { "[" INT "]" } IDENT { "," [ "ref" ]
       IDENT { "[" INT "]" } IDENT } ] ")".
8
9  DEF_MAIN         = "(" ")" "{" { INSTR_SEM_RET } "}".
10
11 INSTR_SEM_RET    = IDENT ( "[" ( [ "+" | "-" ] ( "(" EXPR ")" |( INT | FLOAT |
       IDENT ( "(" [ EXPR { "," EXPR } ] ")" | { "[" EXPR "]" } { "." VARIDENT } ))
       ) | STR | CHAR | BOOL { "*" | "/" | "%" ATOMO } { ( "+" | "-" ) TERM} "]" {
       "[" EXPR "]" } { "." VARIDENT } ["+" | "-" | "*" | "/" | "%" ] "=" EXPR {
       "," VARIDENT OPER_ATRIB EXPR } ";" | INT "]" { "[" INT "]" } IDENT [ "="
       EXPR ] { "," IDENT [ "=" EXPR ] } ";" ) | IDENT [ "=" EXPR ] { "," IDENT [
       "=" EXPR ] } ";" | { "." VARIDENT } [ "+" | "-" | "*" | "/" | "%"] "=" EXPR
       { "," VARIDENT OPER_ATRIB EXPR } ";" | "(" [ EXPR { "," EXPR } ] ")" ";" ) |
       "for" "(" IDENT { "[" INT "]" } IDENT [ "=" EXPR ] { "," IDENT [ "=" EXPR ]
       } ";" COND ";" VARIDENT OPER_ATRIB EXPR { "," VARIDENT OPER_ATRIB EXPR } ")
       " "{" { INSTR_SEM_RET } "}" | "while" "(" COND ")" "{" { INSTR_SEM_RET } "}"
       | "if" "(" COND ")" "{" { INSTR_SEM_RET } "}" ["else" "{" { INSTR_SEM_RET }
       "}"].
12
13 VARIDENT         = IDENT { "[" EXPR "]" } { "." IDENT { "[" EXPR "]" } } .
14
15 FUNCTION_CALL    = IDENT "(" [ EXPR { "," EXPR } ] ")".
16
17 BOTH = IDENT ( { "[" EXPR "]" } { "." IDENT { "[" EXPR "]" } } | "(" [ EXPR {
       "," EXPR } ] ")").
18
19 COND             = COND_TERM { ("and" | "or") COND_TERM}.
20 COND_TERM        = "(" COND ")" | ATOMO_COND { ("==" | "!=" | "<=" | ">=")
       ATOMO_COND}.
21 ATOMO_COND       = VARIDENT | BOOL | INT | "not" ATOMO_COND.
22
23 OPER_ATRIB       =  "+=" | "-=" | "*=" | "/=" | "%=" | "=".
24
25 EXPR             = [ "+" | "-" ] TERM { ( "+" | "-" ) TERM } .
26 TERM             = "(" EXPR ")" | ATOMO { ( "*" | "/" | "%" ) ATOMO } .
27 ATOMO            = ([ "+" | "-" ] ( BOTH | INT | FLOAT )) | STR | CHAR | BOOL .
```

4.2 Lista de Autômatos do APE

- ATOMO-COND:

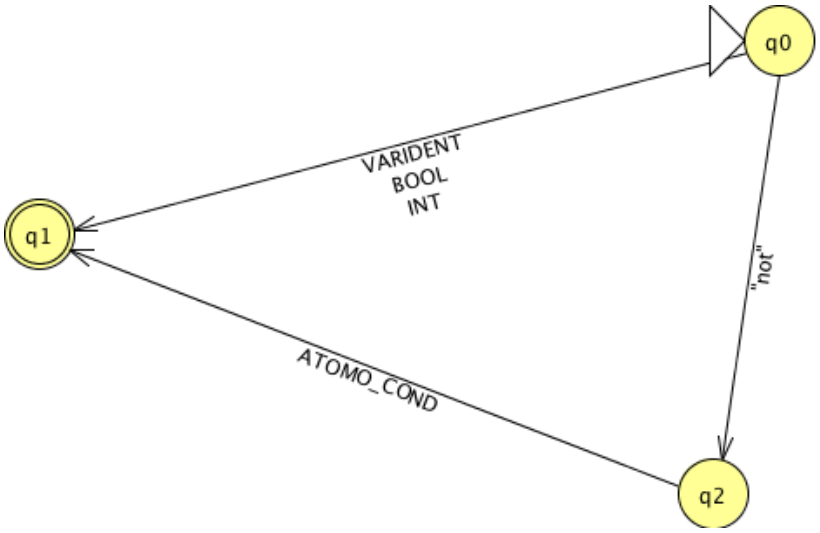


Figura 12 – Autômato ATOMO-COND

- ATOMO:

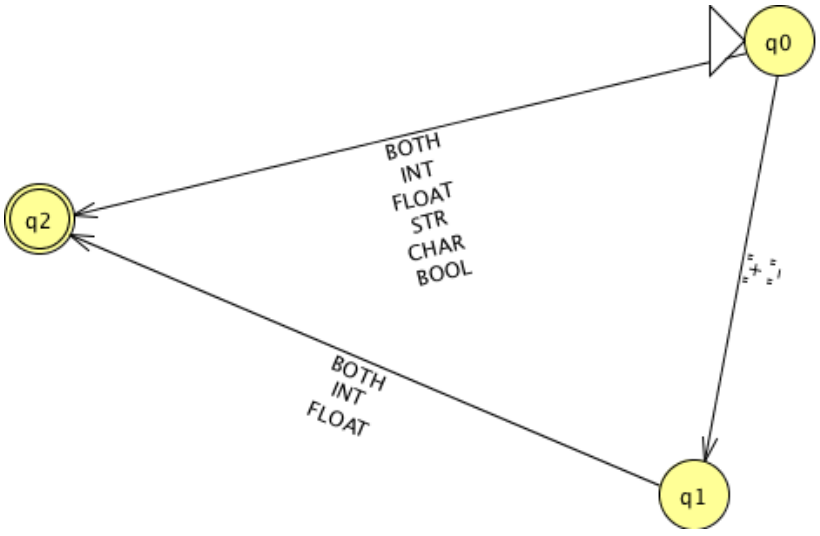


Figura 13 – Autômato ATOMO

- BOTH:

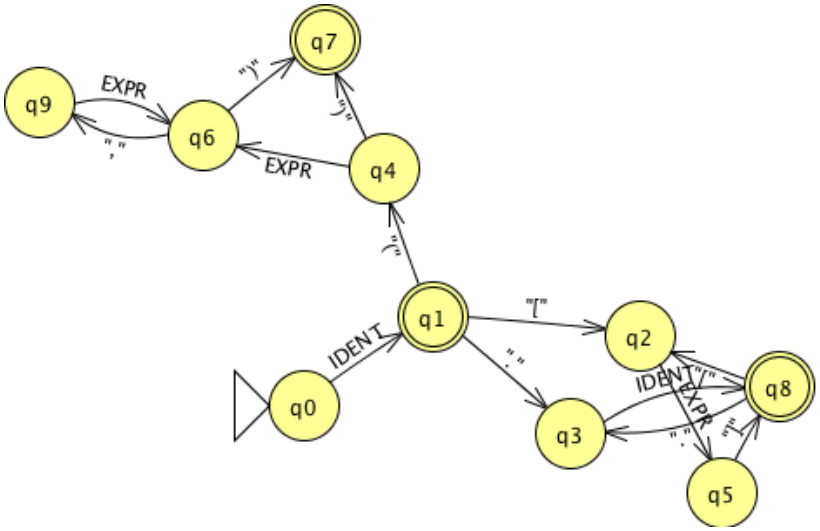


Figura 14 – Autômato BOTH

- COND-TERM:

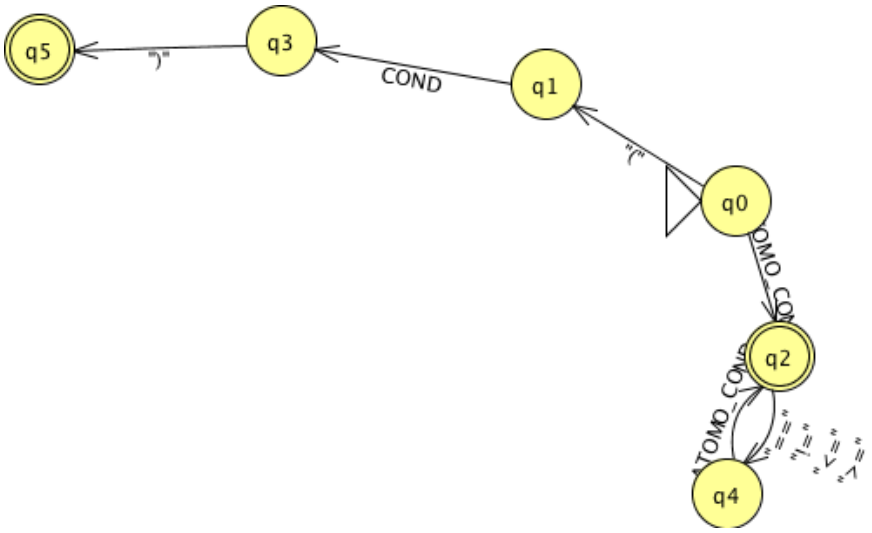


Figura 15 – Autômato COND-TERM

- COND:

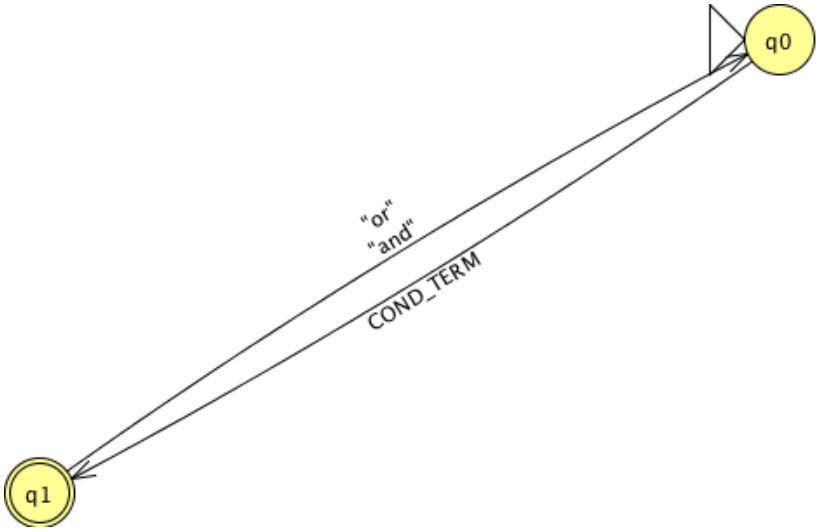


Figura 16 – Autômato COND

- DECLS-GLOBAIS:

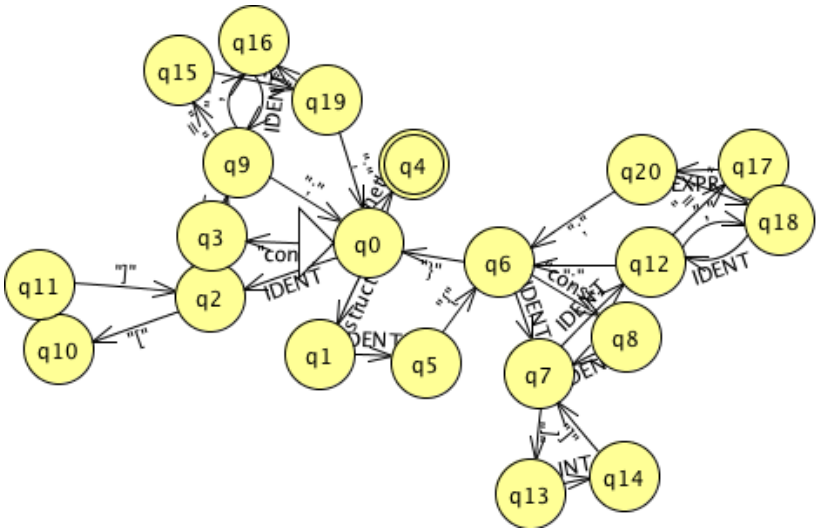


Figura 17 – Autômato DECLS-GLOBAIS

- DEF-MAIN:

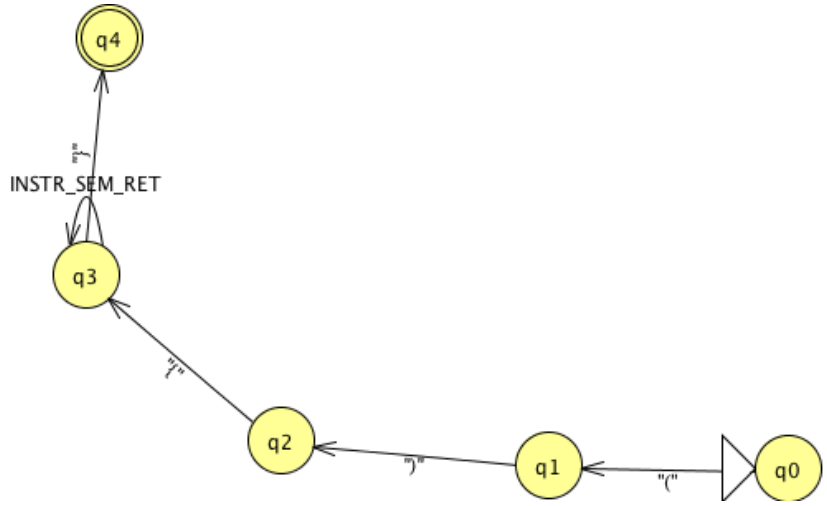


Figura 18 – Autômato DEF-MAIN

- DEF-PROCS-FUNCS:

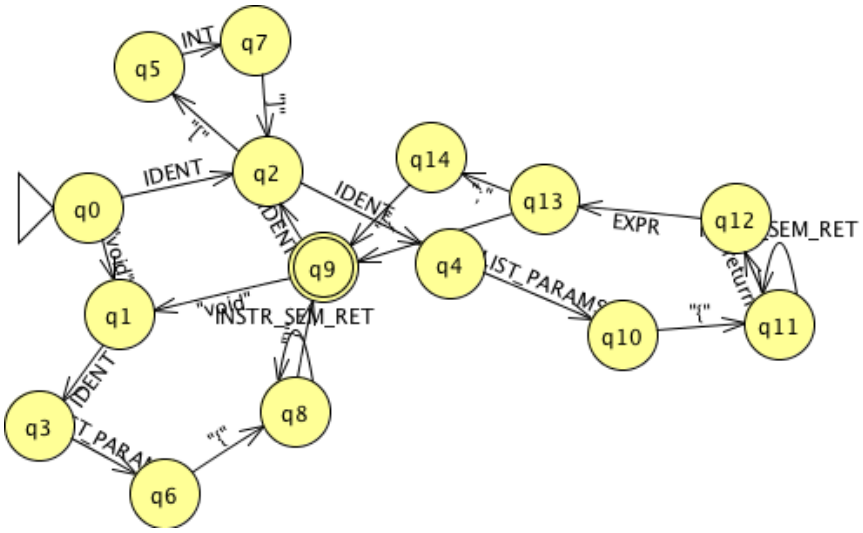


Figura 19 – Autômato DEF-PROCS-FUNCS

- EXPR:

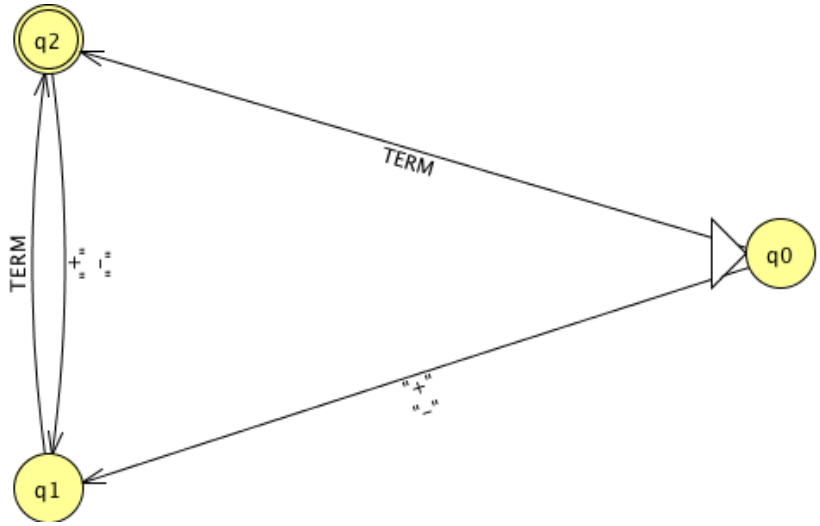


Figura 20 – Autômato EXPR

- FUNCTION-CALL:

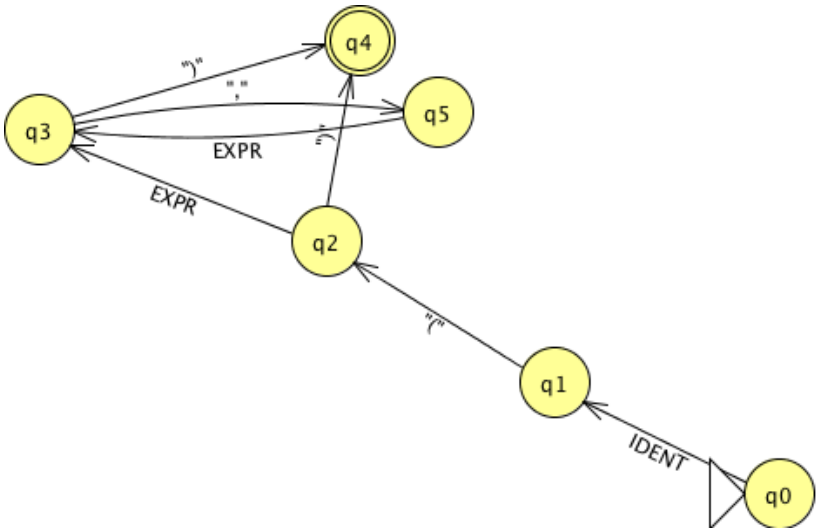


Figura 21 – Autômato FUNCTION-CALL

- IMPORTS:

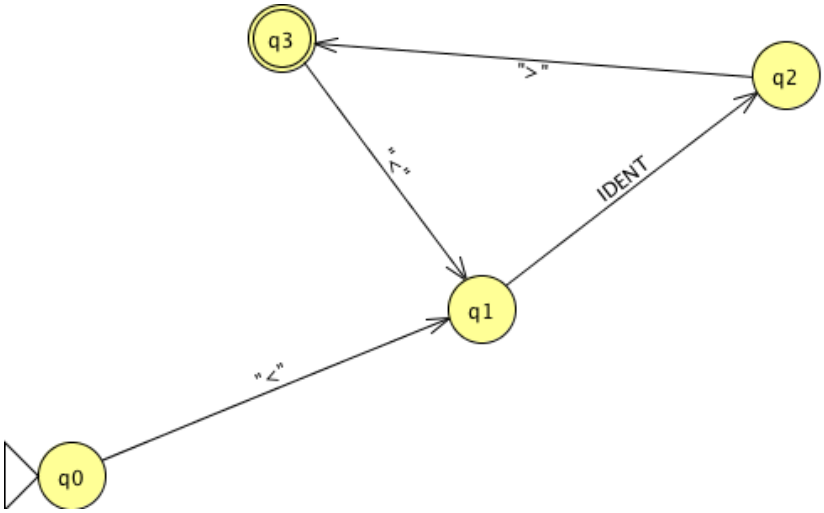


Figura 22 – Autômato IMPORTS

- INSTR-SEM-RET:

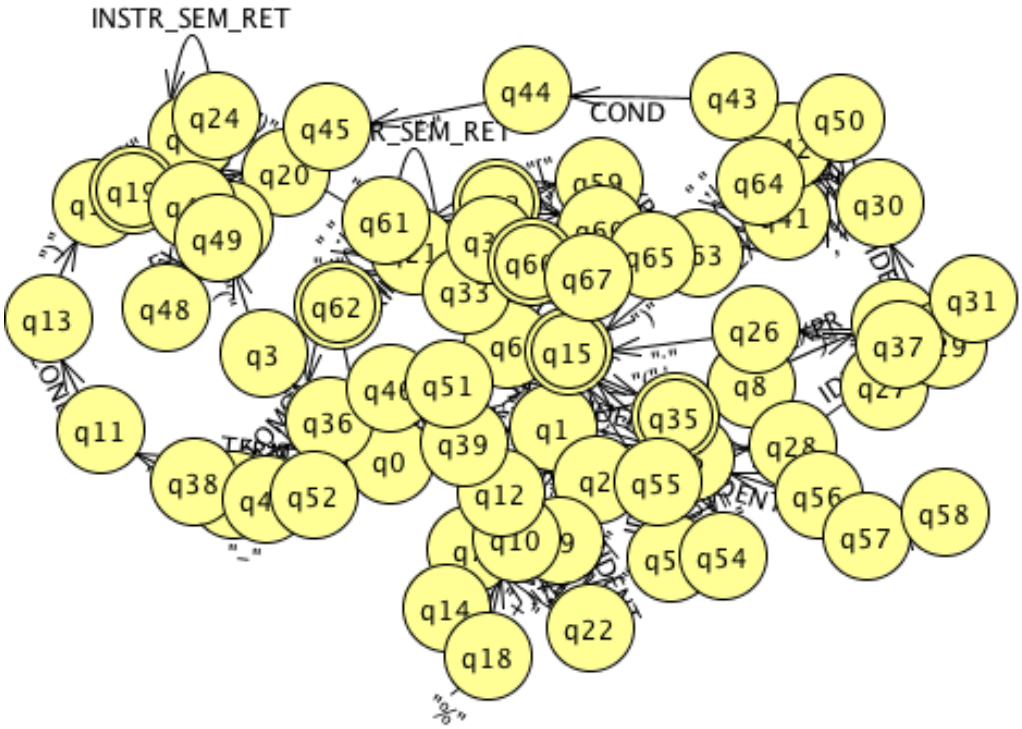


Figura 23 – Autômato INSTR-SEM-RET

- LIST-PARAMS:

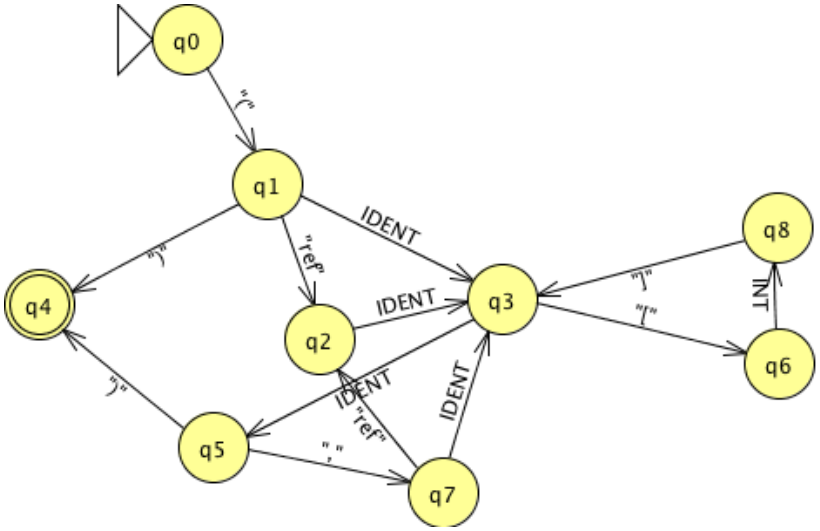


Figura 24 – Autômato LIST-PARAMS

- OPER-ATRIB:

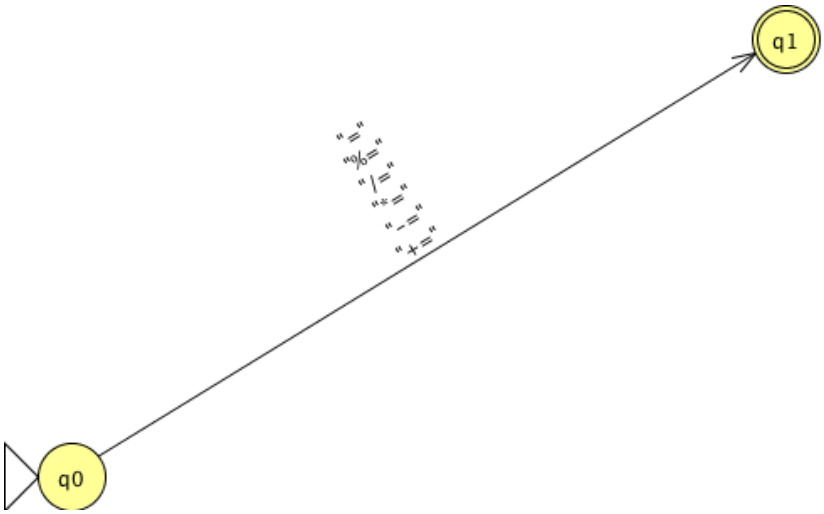


Figura 25 – Autômato OPER-ATRIB

- PROGRAM:

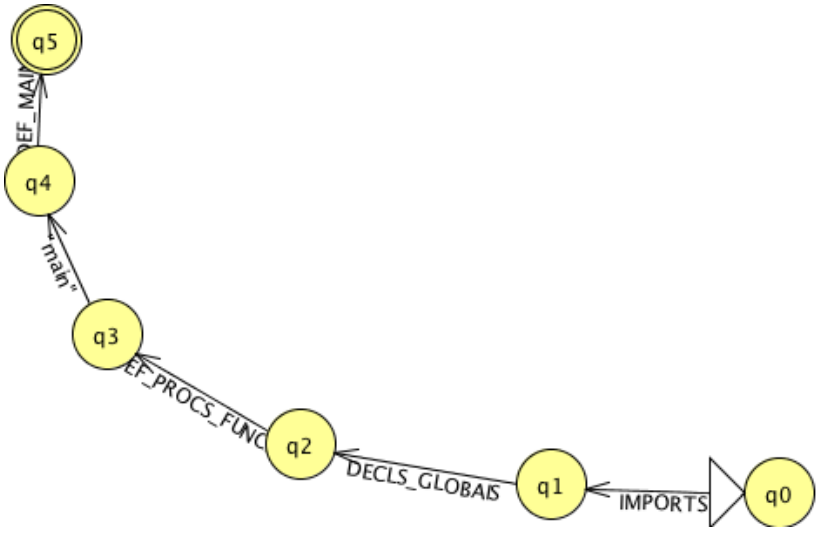


Figura 26 – Autômato PROGRAM

- TERM:

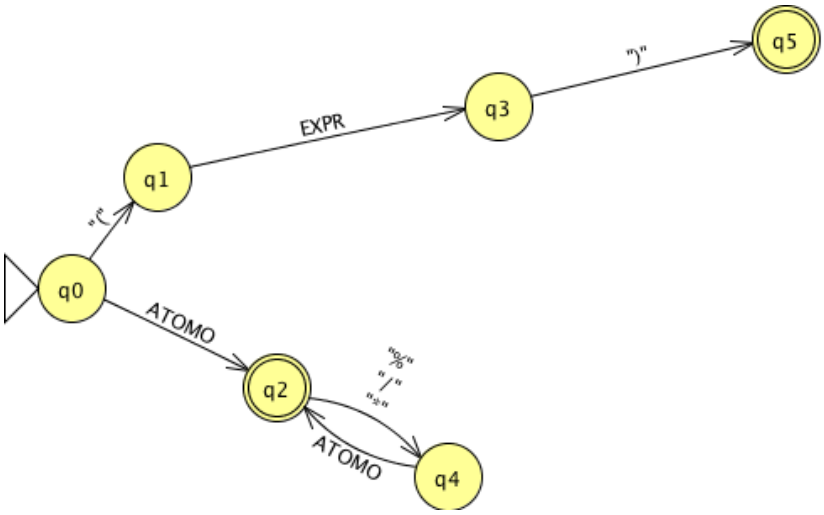


Figura 27 – Autômato TERM

- VARIDENT:

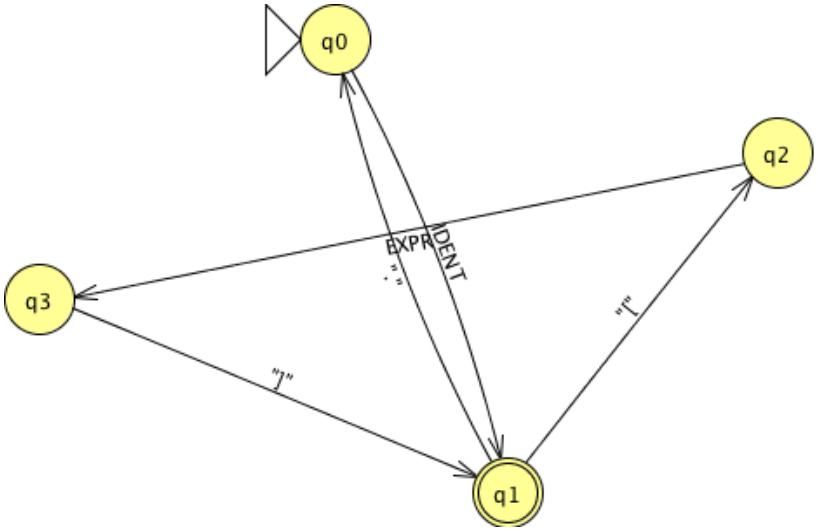


Figura 28 – Autômato VARIDENT

5 Linguagem de montagem

O compilador por nós criado terá como linguagem de saída um programa que será executado na máquina virtual chamada Máquina de von Neumann (MVN).

O Modelo de von Neumann procura oferecer uma alternativa prática, disponibilizando ações mais poderosas e ágeis em seu repertório de operações que o modelo de Turing. Isso viabiliza, codificações muito mais expressivas, compactas e eficientes. Para isso, a Máquina de von Neumann utiliza:

- Memória endereçável, usando acesso aleatório
- Programa armazenado na memória, para definir diretamente a função corrente da máquina (ao invés da Máquina de Estados Finitos)
- Dados representados na memória (ao invés da fita)
- Codificação numérica binária em lugar da unária
- Instruções variadas e expressivas para a realização de operações básicas muito frequentes (ao invés de sub-máquinas específicas)
- Maior flexibilidade para o usuário, permitindo operações de entrada e saída, comunicação física com o mundo real e controle dos modos de operação da máquina

Dessa forma, utilizaremos essa máquina para executar nosso compilador e realizar os testes necessários.

A arquitetura de Von Neumann é composta por um processador e uma memória principal. Na memória principal armazenam-se as instruções do código-fonte e os dados, sendo a divisão mostrada na figura 29 apenas ilustrativa. Além da Unidade Lógica Aritmética (ULA), responsável pelo processamento de operações lógicas e aritméticas, o processador possui um conjunto de elementos físicos de armazenamento de informações e é comum dividir esses componentes nos seguintes módulos resgistradores:

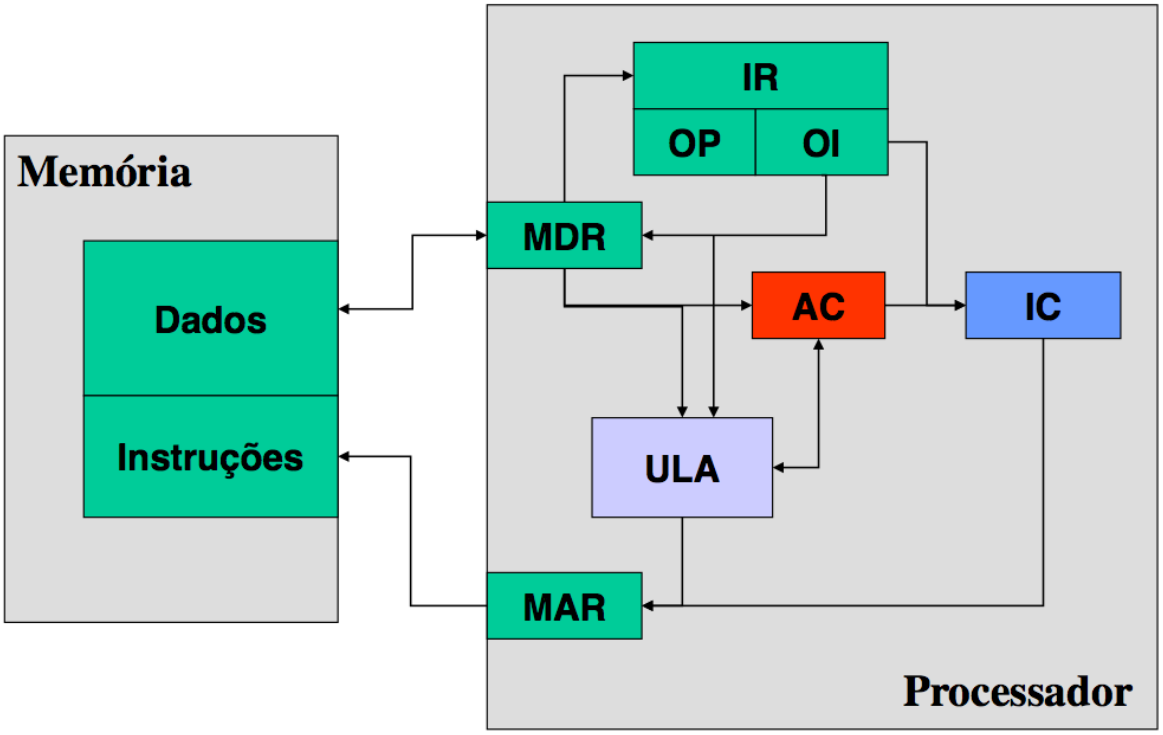
1. MAR - Registrador de endereço de memória

   Indica qual é a origem ou o destino, na memória principal, dos dados contidos no registrador de dados de memória.

2. MDR - Registrador de dados da memória

   Serve como ponte para os dados que trafegam entre a memória e os outros elementos da máquina.

Figura 29 – Arquitetura MVN



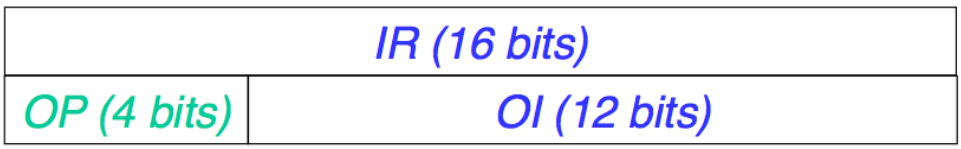
3. IC - Registrador de endereço de instrução

   Indica a cada instante qual será a próxima instrução a ser executada pelo processador.

4. IR - Registrador de instrução

   Contém a instrução atual a ser executada. é subdividido em dois outros registradores, como na figura 30.

Figura 30 – Estrutura do registro de instrução (IR)



   a) OP - Registrador de código de operação

      Parte do registrador de instrução que identifica a instrução que está sendo executada.

   b) OI - Registrador de operando de instrução

      Complementa a instrução indicando o dado ou o endereço sobre o qual ela deve agir.

5. AC - Acumulador

   Funciona como a área de trabalho para execução de operações lógicas ou aritméticas. Acumula o resultado de tais operações.

A máquina executa um programa em diversos passos, listadas abaixo:

1. Determinação da Próxima Instrução a Executar
2. Fase de Obtenção da Instrução

   Obter na memória, no endereço contido no registrador de Endereço da Próxima Instrução, o código da instrução desejada.

3. Fase de Decodificação da Instrução

   Decompor a instrução em duas partes: o código da instrução e o seu operando, depositando essas partes nos registradores de instrução e de operando, respectivamente. Selecionar, com base no conteúdo do registrador de instrução, um procedimento de execução dentre os disponíveis no repertório do simulador (passo 4).

4. Fase de Execução da Instrução

   Executar o procedimento selecionado no passo 3, usando como operando o conteúdo do registrador de operando, preenchido anteriormente.

   Caso a instrução executada não seja de desvio, incrementar o registrador de endereço da próxima instrução a executar. Caso contrário, o procedimento de execução já terá atualizado convenientemente tal informação.

   a) Execução da instrução (decodificada no passo 3)

      De acordo com o código da instrução a executar (contido no registrador de instrução), executar os procedimentos de simulação correspondentes (detalhados adiante).

   b) Acerto do registrador de Endereço da Próxima Instrução para apontar a próxima instrução a ser simulada:

      Incrementar o registrador de Endereço da Próxima Instrução.

5.1 Instruções da Linguagem de Saída

As instruções da MVN podem ser resumidas pela tabela da figura 31.

A seguir, especificaremos o que é realizado pela máquina ao executar cada tipo de operação.

Figura 31 – Lista de instruções da MVN

Código (hexa)	Instrução	Operando
0	Desvio incondicional	endereço do desvio
1	Desvio se acumulador é zero	endereço do desvio
2	Desvio se acumulador é negativo	endereço do desvio
3	Deposita uma constante no acumulador	constante relativa de 12 bits
4	Soma	endereço da parcela
5	Subtração	endereço do subtraendo
6	Multiplicação	endereço do multiplicador
7	Divisão	endereço do divisor
8	Memória para acumulador	endereço-origem do dado
9	Acumulador para memória	endereço-destino do dado
A	Desvio para subprograma (função)	endereço do subprograma
B	Retorno de subprograma (função)	endereço do resultado
C	Parada	endereço do desvio
D	Entrada	dispositivo de e/s
E	Saída	dispositivo de e/s
F	Chamada de supervisor	constante (**)

(**) por ora, este operando (tipo da chamada) é irrelevante, e esta instrução nada faz.

- Registrador de instrução = 0 (desvio incondicional)

  Modifica o conteúdo do registrador de Endereço da Próxima Instrução (IC) armazenando nele o conteúdo do registrador de operando (OI)

  IC := OI

- Registrador de instrução = 1 (desvio se acumulador é zero)

  Se o conteúdo do acumulador (AC) for zero, então modifica o conteúdo do registrador de Endereço da Próxima Instrução (IC), armazenando nele o conteúdo do registrador de operando (OI)

  Se AC = 0 então IC := OI

  Se não IC := IC + 1

- Registrador de instrução = 2 (desvio se negativo)

  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 da Próxima Instrução (IC) armazenando nele o conteúdo do registrador de operando (OI)

  Se AC < 0 então IC := OI

  Se não IC := IC + 1

- Registrador de instrução = 3 (constante para acumulador)

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

- Registrador de instrução = 4 (soma)

  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

- Registrador de instrução = 5 (subtração)

  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

- Registrador de instrução = 6 (multiplicação)

  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

- Registrador de instrução = 7 (divisão inteira)

  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

- Registrador de instrução = 8 (memória para acumulador)

  Armazena no acumulador (AC) o conteúdo da posição de memória endereçada pelom registrador de operando (OI)

  AC := MEM[OI]

  IC := IC + 1

- Registrador de instrução = 9 (acumulador para memória)

  Guarda o conteúdo do acumulador (AC) na posição de memória endereçada pelo registrador de operando (OI)

  MEM[OI] := AC

  IC := IC + 1

- Registrador de instrução = A (desvio para subprograma)

  Armazena o conteúdo do registrador de Endereço da Próxima Instrução (IC), incrementado de uma unidade, no registrador de endereço de retorno (RA). Armazena no registrador de Endereço da Próxima Instrução (IC) o conteúdo do registrador de operando (OI).

  RA := IC + 1

  IC := OI

- Registrador de instrução = B (retorno de subprograma)

  Armazena no registrador de Endereço da Próxima Instrução (IC) o conteúdo do registrador de endereço de retorno (RA), e no acumulador (AC) o conteúdo da posição de memória apontada pelo registrador de operando (OI)

  AC := MEM[OI]

  IC := RA

- Registrador de instrução = C (stop)

  Modifica o conteúdo do registrador de Endereço da Próxima Instrução (IC) armazenando nele o conteúdo do registrador de operando (OI) e para o processamento

  IC := OI

- Registrador de instrução = D (input)

  Aciona o dispositivo padrão de entrada e aguardar que o usuário forneça o próximo dado a ser lido. Transfere o dado para o acumulador

  Aguarda

  AC := dado de entrada

  IC := IC + 1

- Registrador de instrução = E (output)

  Transfere o conteúdo do acumulador (AC) para o dispositivo padrão de saída. Aciona o dispositivo padrão de saída e aguardar que este termine de executar a operação de saída

dado de saída := AC

aguarda

IC := IC + 1

- Registrador de instrução = F (supervisor call)

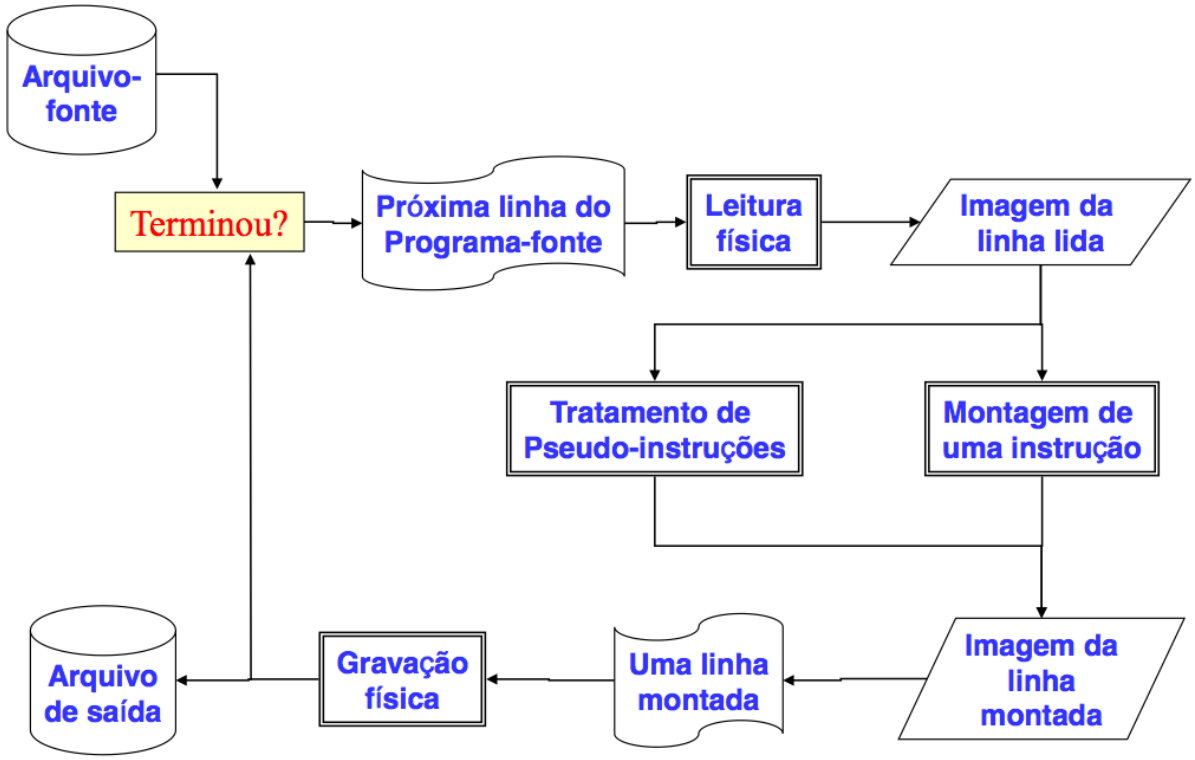
  Não implementado: por enquanto esta instrução não faz nada.

  IC := IC + 1

Escrever um programa usando diretamente codificação binária não é uma tarefa simples, e tampouco agradável. Naturalmente, se um programa é muito grande ou se lida com diversas estruturas complexas (listas, etc.), a sua codificação se torna ainda mais difícil e complexa.

Por conta disso, torna-se imprescindível construir alguma abstração que facilite a programação e a verificação dos programas. A primeira idéia, mais natural, é utilizar o modelo de máquina existente e, a partir dele, definir nomes (mnemônicos) para cada instrução da máquina. Posteriormente, verifica-se que somente isso não basta, pois é necessário lidar com os endereços dentro de um programa (rótulos, operandos, sub-rotinas), com a reserva de espaço para tabelas, com valores constantes. Enfim, é necessário definir uma linguagem simbólica.

Figura 32 – Esquema geral de um montador



Para a construção de um montador, cujo esquema geral está representado na figura 32 pressupõe-se que sejam tratadas as seguintes questões:

- definição das instruções: determinar os mnemônicos que as representam simbolicamente;
- definição das pseudo-instruções: determinar os mnemônicos que as representam, bem como sua função para o montador.

As instruções para a MVN são apresentadas na figura 33.

Figura 33 – Tabela de mnemônicos para a MVN (de 2 caracteres)

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

5.2 Pseudoinstruções da Linguagem de Saída

Programas absolutos são executáveis estritamente nas posições de memória em que foram criados, tornando difícil a manutenção e o trabalho em equipe. A utilização de programas relocáveis permitem sua execução em qualquer posição de memória, tornando possível utilizar partes de código projetadas externamente (uso de bibliotecas, por exemplo).

Para que se possa exprimir um programa relocável e com possibilidade de construção em módulos, separadamente desenvolvidos, é necessário que:

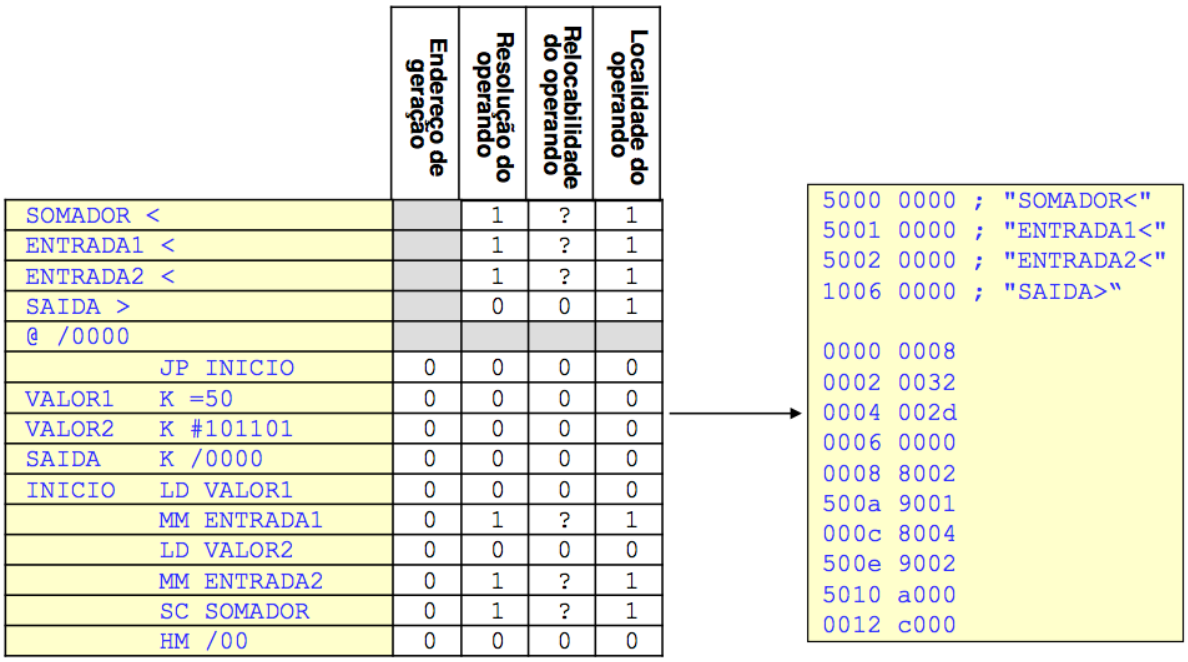
- Haja a possibilidade de representar e identificar endereços absolutos e endereços relativos;
- Um programa possa ser montado sem que os seus endereços simbólicos estejam todos resolvidos;
- Seja possível identificar, em um módulo, símbolos que possam ser referenciados simbolicamente em outros módulos.

Sendo assim, a linguagem simbólica não possui somente os mnemônicos das instruções da MVN, mas também comandos chamados de pseudo-instruções da linguagem de montagem. Na linguagem de montagem, as pseudo-instruções também são representadas por mnemônicos, listados abaixo:

- @ : Origem Absoluta. Recebe um operando numérico, define o endereço da instrução seguinte;
- K : Constante, o operando numérico tem o valor da constante (em hexadecimal). Define uma área preenchida por uma CONSTANTE de 2 bytes;
- $: Reserva de área de dados, o operando numérico define o tamanho da área a ser reservada. Define um BLOCO DE MEMóRIA com número especificado de words;
- # : Final físico do texto fonte;
- & : Origem relocável;
- > : Endereço simbólico de entrada (entry point). Define um endereço simbólico local como entry-point do programa;
- < : Endereço simbólico externo (external). Define um endereço simbólico que referencia um entry-point externo.

Na figura 34, temos um exemplo de um somador escrito em linguagem de montagem, visto na aula de Fundamentos de Eng. de Computação, e sua respectiva tradução pelos módulos Montador, *Linker* e Relocador, módulos extras porém integrados no nosso caso:

Figura 34 – Exemplo de um somador



	Endereço de geração	Resolução do operando	Relocabilidade do operando	Localidade do operando
SOMADOR <		1	?	1
ENTRADA1 <		1	?	1
ENTRADA2 <		1	?	1
SAIDA >		0	0	1
@ /0000				
JP INICIO	0	0	0	0
VALOR1 K =50	0	0	0	0
VALOR2 K #101101	0	0	0	0
SAIDA K /0000	0	0	0	0
INICIO LD VALOR1	0	0	0	0
MM ENTRADA1	0	1	?	1
LD VALOR2	0	0	0	0
MM ENTRADA2	0	1	?	1
SC SOMADOR	0	1	?	1
HM /00	0	0	0	0

```
5000 0000 ; "SOMADOR<"
5001 0000 ; "ENTRADA1<"
5002 0000 ; "ENTRADA2<"
1006 0000 ; "SAIDA>"

0000 0008
0002 0032
0004 002d
0006 0000
0008 8002
500a 9001
000c 8004
500e 9002
5010 a000
0012 c000
```

6 Ambiente de execução

6.1 Características gerais

6.1.1 Organização da memória

O ambiente de execução da MVN fornece aos programadores um tamanho limitado de memória para ser usado no geral, a ser compartilhado entre o código e as variáveis do programa. O montador aloca a memória com base nos endereços relativos especificados no código do programa. Do total, a parte inicial da memória é reservada para guardar as instruções que serão executadas pelo programa. A parte final da memória deve ser usada especialmente para o uso do registro de ativação.

De maneira mais objetiva, reserva-se uma parte do código para a área de dados, uma parte para a função principal e as subrotinas e uma parte dedicada a pilhas de variáveis e endereços que viabilizam a chamada de subrotinas.

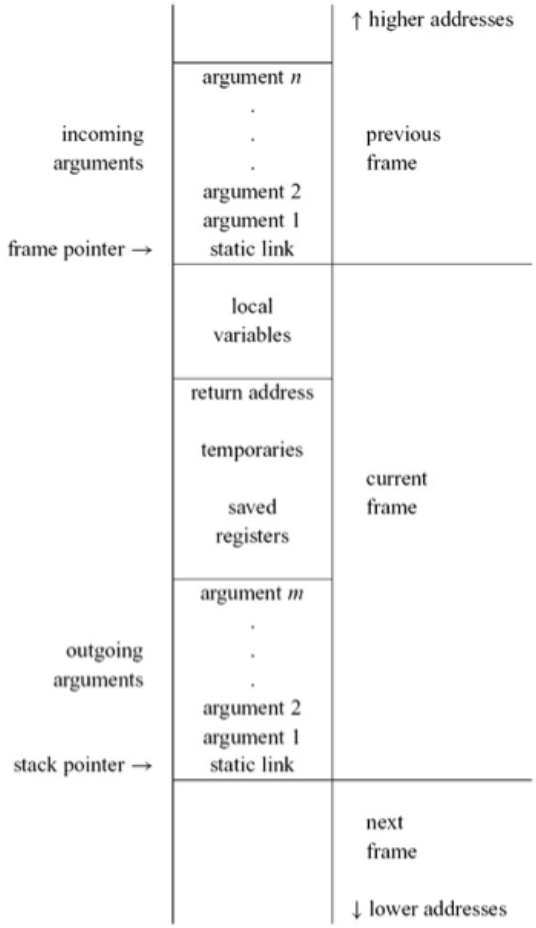
6.1.2 Registro de ativação

As funções em programas têm variáveis locais, que devem ser criadas na chamada da função e sobrevivem até que a função retorne. Elas também possuem recursão, onde cada instância da função tem seus próprios parâmetros e locais. As chamadas de funções se comportam de maneira LIFO, portanto podemos usar uma pilha como estrutura.

As operações push e pop dessa pilha não podem ser feitas individualmente para cada variável. Desa forma, manipula-se conjuntos de variáveis, e precisamos ter acesso a todas elas. Com isso, definimos dois conceitos:

- *Stack Pointer* (SP):
  - Todas as posições além do SP são lixo;
  - Todas as anteriores estão alocadas.
- *Activation Record* ou *Stack Frame*
  - área na pilha reservada para os dados de uma função (parâmetros, locais, endereço de retorno, etc).
  - esta parte da pilha foi fusionada à parte anterior, facilitando o uso da pilha e diminuindo a quantidade de dados na mesma. Esta decisão não afeta a implantação, uma vez que no caso desta linguagem o compilador tem total controle do tamanho das estruturas sendo utilizadas.

Figura 35 – Esquema do Registro de Ativação



A figura 35 ilustra a organização da pilha. O uso do registro de ativação permite entre outras coisas a chamada recursiva de funções, uma vez isso não é possível de forma nativa no ambiente da MVN. No caso da MVN, a pilha cresce para baixo e as subrotinas são executadas utilizando as seguintes instruções:

- Desvio para subprograma - mnemônico SC (0xA): armazena o endereço de instrução seguinte (atual + 1) na posição de memória apontada pelo operando. Em seguida, desvia a execução para o endereço indicado pelo operando e acrescido de uma unidade.
- Retorno de subprograma - mnemônico RS (0xB): desvia a execução para o endereço indicado pelo valor guardado na posição de memória do operando.

Foi criada por nós uma biblioteca em assembly para implementar funções auxiliares de entrada e saída de dados, além da funcionalidade de empilhar, desempilhar e ter acesso a informações contidas na pilha discutida anteriormente. Essas funções são explicadas na próxima seção.

6.2 Biblioteca desenvolvida em Assembly

A biblioteca padrão desenvolvida é dividida em dois módulos o primeiro implementa as funções básicas de empilhamento, sendo chamado de `std.asm`. O segundo módulo implementa as operações de input e output de dados, de nome `stdio.asm`.

6.2.1 STD

A manipulação de pilhas é feita pela biblioteca padrão, sendo que deseja-se seguir a estrutura abaixo definida, facilitando o uso e acesso das variáveis. Vemos abaixo um exemplo do uso da biblioteca. Na linha 23 salvamos uma variável recebida por parâmetro na pilha e na linha 30 recuperamos seu valor.

Logo antes de retornar devemos executar a função `POP_CALL` ela é responsável por escrever o endereço de retorno na função em que estamos, assim aproveitando das chamadas existentes na `MVN` (funções devem ser *stateless* para tanto). Percebe-se que é possível executar a função recursivamente (linha 55 e 57). Para tanto é necessário chamar a função `PUSH_CALL` para que a mesma efetue o empilhamento e escreva o endereço de retorno atual na pilha.

```
1  ;; VARIAVEIS GLOBAIS
2  ;; comeco da pilha = FFF
3  ;; tamanho da pilha = 2FF
4  ;;    | ptr to old_stack_head  | \____ STACK_PTR
5  ;;    |       savedregist      |
6  ;;    |          ...           |
7  ;;    |       local var        |
8  ;;    |          ...           |
9  ;;    |       temporaries      |
10 ;;    |       parameters       |
11 ;;    |          ...           |
12 ;;    |      ref parameters    |  _____ OLD STACK_PTR
13 ;;    |       returnaddrs      | /      (STACK_PTR points here)
14
15 EXAMPLE_STACK_ARG    K  /0000
16 EXAMPLE_STACK        JP /000
17                      SC PRINT_STACK_ADDRS   ;; deve imprimir 0fff
18                      ;;; SALVAR ARGUMENTOS na pilha
19                      LV =0
20                      MM WORD_TO_SAVE
21                      LV EXAMPLE_STACK_ARG
22                      MM ORIGIN_PTR
23                      SC SAVE_WORD_TO_LOCAL_VAR
24                      ;;;; CORPO DA FUNCAO
25                      ;;; CARREGANDO UM VALOR DA PILHA
26                      LV =0
```

```
27                          MM WORD_TO_GET
28                          LV EXAMPLE_STACK_ARG
29                          MM STORE_PTR
30                          SC GET_WORD_LOCAL_VAR
31                          ;;;; IMPRIME
32                          LV COUNT_IS
33                          MM STRING_PTR
34                          SC P_STRING  ;; inline fct, no need to stack
35                          LD EXAMPLE_STACK_ARG
36                          MM TO_BE_PRINTED
37                          SC P_INT_ZERO
38                          SC P_LINE
39
40                          LD EXAMPLE_STACK_ARG
41                          JZ RETURN_EXAMPLE_STACK
42
43                          LD EXAMPLE_STACK_ARG
44                          -  ONE
45                          MM EXAMPLE_STACK_ARG
46
47                          LV =1
48                          MM PUSH_CALL_SIZELV
49                          LV =0
50                          MM PUSH_CALL_RET_ADDRS
51                          LV =0
52                          MM PUSH_CALL_TMP_SZ
53                          LV =0
54                          MM PUSH_CALL_PAR_SZ
55                          SC PUSH_CALL
56
57                          SC EXAMPLE_STACK  ;; chamada recursiva
58                          ;;;; FIM DO CORPO DA FUNCAO
59 RETURN_EXAMPLE_STACK LV EXAMPLE_STACK
60                          MM POP_CALL_FCT
61                          SC POP_CALL ;; trickery!
62
63                          SC PRINT_STACK_ADDRS   ;; deve imprimir 0fff
64                          RS EXAMPLE_STACK
```

Abaixo podemos ver a implementação das funções de `PUSH` e `POP`

A pilha é implementada dos valores mais altos da memória para os valores mais baixos, sendo assim, o ponteiro de pilha começa apontando para `0x0FFF`.

A pilha funciona como uma lista ligada que guarda o endereço da última célula da pilha. Sendo assim, a operação de `POP` é trivial. Estas funções fazem a gestão do endereço de retorno automaticamente, contanto que se siga a premissa de chamada (chamada da

função logo após a chamada de `PUSH_CALL` e seus parâmetros).

```
1  ;; *** PUSH_CALL ***
2  PUSH_CALL             JP /000
3                        LD PUSH_CALL  ;; get return addrs
4                        + TWO ;; return address of the callee
5                        + LOADV_CONST
6                        MM LOAD_RETURN_ADDRS
7                        LD STACK_PTR
8                        - TWO         ;; new return addrs
9                        + MOVE_CONST
10                       MM MOVE_RETURN_ADDRS
11 LOAD_RETURN_ADDRS     JP /000
12 MOVE_RETURN_ADDRS     JP /000    ;; return addrs salvo
13                       LD STACK_PTR
14                       - TWO
15                       - TWO
16                       - PUSH_CALL_SIZELV
17                       - PUSH_CALL_RET_ADDRS
18                       - PUSH_CALL_TMP_SZ
19                       - PUSH_CALL_PAR_SZ
20                       - TWO  ;; return addrs
21                       MM TMP_1
22                       LD TMP_1
23                       + MOVE_CONST
24                       MM MRKR_PC_SAVE_HEAD
25                       LD STACK_PTR
26 MRKR_PC_SAVE_HEAD     JP /000
27                       LD TMP_1
28                       MM STACK_PTR
29                       RS PUSH_CALL
30 ;; **** POP_CALL ****
31
32 POP_CALL_FCT          K /0000
33 POP_CALL              JP /000 ; retorno
34 POP_CALL_INIT         LD STACK_PTR
35                       + LOAD_CONST
36                       MM MRKR_PC_LOAD_HEAD
37 MRKR_PC_LOAD_HEAD     JP /000
38                       MM STACK_PTR
39                       LD STACK_PTR
40                       - TWO
41                       + LOAD_CONST
42                       MM LOAD_RETURN_ADDRS_2
43                       LD POP_CALL_FCT
44                       + MOVE_CONST
45                       MM MOVE_RETURN_ADDRS_2
46 LOAD_RETURN_ADDRS_2 JP /000
```

```
47 MOVE_RETURN_ADDRS_2 JP /000   ;; engana a funcao para ela pensar que ela
48                               ;; tem que retornar para esse valor
49                     RS POP_CALL
```

As rotinas de salvaguarda e carregamento dos valores locais, parâmetros, referências pode ser feita por meio das chamadas abaixo, `SAVE_WORD_TO_LOCAL_VAR` e `GET_WORD_LOCAL_VAR` respectivamente.

```
1  ;; **** SAVE_WORD_TO_LOCAL_VAR WORD_TO_SAVE ORIGIN_PTR ****
2  SAVE_WORD_TO_LOCAL_VAR       JP /000
3                               LD STACK_PTR
4                               + TWO           ;; first word
5                               + WORD_TO_SAVE
6                               + WORD_TO_SAVE  ;; WORD_TO_GET * 2
7                               + MOVE_CONST    ;;
8                               MM MOVE_WORD_LOCAL_VAR_2
9                               LD ORIGIN_PTR
10                              + LOAD_CONST
11                              MM LOAD_WORD_LOCAL_VAR_2
12 LOAD_WORD_LOCAL_VAR_2        JP /000 ;; 8FROMPTR
13 MOVE_WORD_LOCAL_VAR_2        JP /000 ;; 9TOPTR
14                              RS SAVE_WORD_TO_LOCAL_VAR
15
16 ;; **** GET_WORD_LOCAL_VAR WORD_TO_GET STORE_PTR ****
17
18
19 WORD_TO_GET          K /000
20 STORE_PTR            K /000
21
22 GET_WORD_LOCAL_VAR           JP /000
23                              LD STACK_PTR
24                              + TWO           ;; first word
25                              + WORD_TO_GET
26                              + WORD_TO_GET   ;; WORD_TO_GET * 2
27                              + LOAD_CONST    ;;
28                              MM LOAD_WORD_LOCAL_VAR
29                              LD STORE_PTR
30                              + MOVE_CONST
31                              MM MOVE_WORD_LOCAL_VAR
32 LOAD_WORD_LOCAL_VAR          JP /000 ;; 8FROMPTR
33 MOVE_WORD_LOCAL_VAR          JP /000 ;; 9TOPTR
34                              RS GET_WORD_LOCAL_VAR
```

6.2.2 STDIO

O ambiente de execução também é provido de funções de input/output:

Para a impressão de *strings* podemos utilizar a função `P_STRING`, passando o ponteiro para o começo de uma `string`. Em `CZAR` consideramos *strings* como sendo *bytes* em um vetor de *word* terminados pelo *byte* `0x0000`. Vale salientar que esta forma de armazenamento não causa problemas com outros tipos de armazenamento mais compactos, como a utilização dos dois *bytes* da *word* para armazenamento de *chars* subsequentes. Quando a função recebe a *word* `0x0030`, primeiramente ela vai imprimir `0x00` que é o caractere nulo, portanto, sem impressão e então imprimir o caractere correspondente a `0x30`.

```
1  ;; ****  P_STRING &STRING_PTR ****
2  ;;    Imprime a string apontada por STRING_PTR ate
3  ;; o caractere /000
4
5  P_STRING              JP /000            ; endereco de retorno
6  PSTRINGINIT           LD STRING_PTR
7                        MM TO_BE_PRINTED_TMP
8  LOAD_TO_BE_PRINTED    LD TO_BE_PRINTED_TMP
9                        + LOAD_CONST
10                       MM LABELLOAD
11 LABELLOAD             K /0000
12                       JZ P_STRING_END  ; se zero vamos para o final!
13                       PD /100
14                       LD TO_BE_PRINTED_TMP
15                       + TWO
16                       MM TO_BE_PRINTED_TMP
17                       JP LOAD_TO_BE_PRINTED
18 P_STRING_END          RS P_STRING
```

Para a leitura de *strings* seguimos o padrão definido anteriormente, um *byte* (*char*) por *word*:

```
1  ;; *** GETS STORE_PTR_IO ***
2  ;; Existe um problema de buffer aqui... nao vamos
3  ;; trata-lo, pois este e' um problema intri'nseco da
4  ;; MVN. (leitura e subsequente bloqueio por word)
5  LAST_CONTROL_CHAR_P_ONE    K /0021
6  ARRAY_POS_BYTE  JP /000
7  GETS            JP /000
8                  LD STORE_PTR_IO
9                  MM ARRAY_POS_BYTE
10 GETS_LOOP       GD /000
11                 MM HIGH_V
12                 SC HIGH_LOW
13                 LD HIGH_V
14                 - LAST_CONTROL_CHAR_P_ONE
15                 JN RETURN_GETS
16                 LD ARRAY_POS_BYTE
17                 + MOVE_CONST
```

```
18                  MM MOVE_HIGH_V
19                  LD HIGH_V
20 MOVE_HIGH_V      JP /000
21
22                  LD ARRAY_POS_BYTE
23                  + TWO
24                  MM ARRAY_POS_BYTE
25
26                  LD LOW_V
27                  - LAST_CONTROL_CHAR_P_ONE
28                  JN RETURN_GETS
29                  LD ARRAY_POS_BYTE
30                  + MOVE_CONST
31                  MM MOVE_LOW_V
32                  LD LOW_V
33 MOVE_LOW_V       JP /000
34
35                  LD ARRAY_POS_BYTE
36                  + TWO
37                  MM ARRAY_POS_BYTE
38
39                  JP GETS_LOOP
40
41 RETURN_GETS      LD ARRAY_POS_BYTE
42                  + MOVE_CONST
43                  MM MOVE_ZERO
44                  LV =000
45 MOVE_ZERO        JP /000
46
47                  LD ARRAY_POS_BYTE
48                  + TWO
49                  MM ARRAY_POS_BYTE
50                  RS GETS
```

A biblioteca também é capaz de realizar a leitura e escrita de valores inteiros (funções auxiliares estão disponíveis no pacote em anexo):

```
1  ;; *** READ_INT STORE_PTR_IO ***
2  ;; doesnt care about buffers, should have a trailing char at the end of the
3  ;; stream otherwise it will just discard it..
4  STORE_PTR_IO        JP /000
5  ZERO_M_ONE          K  /002F
6  NINE_P_ONE          K  /0039
7
8  LOW                 K  /0000
9  HIGH                K  /0000
10 GO_IF_NUMBER        K  /0000
11 TO_BE_TRIMMED       K  /0000
```

```
12 TBT_TMP              K   /0000
13
14 TRIM_INT             JP  /000
15                      LD  TO_BE_TRIMMED
16                      /   SHIFT_BYTE
17                      *   SHIFT_BYTE
18                      MM  TBT_TMP
19                      LD  TO_BE_TRIMMED
20                      -   TBT_TMP
21                      MM  TO_BE_TRIMMED
22                      RS  TRIM_INT
23
24 READ_INT_WORD        JP  /000
25                      GD  /000
26                      MM  TMP_3
27                      LD  TMP_3
28                      /   SHIFT_BYTE
29                      MM  TO_BE_TRIMMED
30                      SC  TRIM_INT
31                      LD  TO_BE_TRIMMED
32                      MM  HIGH
33
34                      LD  TMP_3
35                      MM  TO_BE_TRIMMED
36                      SC  TRIM_INT
37                      LD  TO_BE_TRIMMED
38                      MM  LOW
39                      RS  READ_INT_WORD
40
41 READ_INT             JP  /000
42                      LV  =0
43                      MM  TMP_4
44 READ_INT_LOOP        SC  READ_INT_WORD
45                      LD  HIGH
46                      MM  TMP_3
47                      LV  CONT1
48                      MM  GO_IF_NUMBER
49                      JP  IF_NUMBER_CONTINUE
50 CONT1                LD  LOW
51                      MM  TMP_3
52                      LV  READ_INT_LOOP
53                      MM  GO_IF_NUMBER
54                      JP  IF_NUMBER_CONTINUE
55 NOT_NUMBER           LD  STORE_PTR_IO
56                      +   MOVE_CONST
57                      MM  MOVE_READ_INT
58                      LD  TMP_4
```

```
59 MOVE_READ_INT        JP  /000
60                      RS  READ_INT
61
62 IF_NUMBER_CONTINUE   LD  TMP_3
63                      -   ZERO_M_ONE
64                      JN  NOT_NUMBER
65                      LD  NINE_P_ONE
66                      -   TMP_3
67                      JN  NOT_NUMBER
68
69                      LD  TMP_4
70                      *   TEN
71                      MM  TMP_4
72
73
74                      LD  TMP_3
75                      -   ZERO_M_ONE
76                      -   ONE
77                      +   TMP_4
78                      MM  TMP_4
79
80                      LD  GO_IF_NUMBER
81                      MM  END_READ_INT
82 END_READ_INT         JP  /000
```

A impressão de inteiros, por ser crítica e muito importante para a correção de erros, foi feita de forma simples e direta. Sem laços (unwind de `GOTO` explícito) ou complicações, resultando em uma função bem determinada e robusta.

```
1  ;; *** P_INT_ZERO TO_BE_PRINTED ***
2  ;;  Imprime um inteiro (com zeros a esquerda)
3  ;; ex:
4  ;;  INT_2 K  =345
5  ;;        LD INT_2
6  ;;        MM TO_BE_PRINTED
7  ;;        SC P_INT_ZERO
8  ;; imprime 00345
9  ;;
10 ;;
11 ;; Esta funcao esta com o loop inline
12 ;; sendo simples e robusta
13
14 P_INT_ZERO           JP  /000
15 P_INT_INIT           JP  P_INT_REAL_INIT
16 ZERO_BASE            K   /30
17 ;; bases para a conversao:
18 INT_POT_1            K   =10000
19 INT_POT_2            K   =1000
```

```
20 INT_POT_3          K =100
21 INT_POT_4          K =10
22 INT_POT_5          K =1
23 P_INT_REAL_INIT    LD TO_BE_PRINTED        ;; PRIMEIRO CHAR
24                    MM TMP_1
25                    /  INT_POT_1
26                    +  ZERO_BASE
27                    PD /100                 ;; imprime
28                    LD TMP_1
29                    /  INT_POT_1
30                    *  INT_POT_1
31                    MM TMP_2
32                    LD TMP_1
33                    -  TMP_2
34                    MM TMP_1
35                    /  INT_POT_2            ;; segundo char
36                    +  ZERO_BASE
37                    PD /100                 ;; imprime
38                    LD TMP_1
39                    /  INT_POT_2
40                    *  INT_POT_2
41                    MM TMP_2
42                    LD TMP_1
43                    -  TMP_2
44                    MM TMP_1
45                    /  INT_POT_3            ;; terceiro char
46                    +  ZERO_BASE
47                    PD /100                 ;; imprime
48                    LD TMP_1
49                    /  INT_POT_3
50                    *  INT_POT_3
51                    MM TMP_2
52                    LD TMP_1
53                    -  TMP_2
54                    MM TMP_1
55                    /  INT_POT_4            ;; quarto char
56                    +  ZERO_BASE
57                    PD /100                 ;; imprime
58                    LD TMP_1
59                    /  INT_POT_4
60                    *  INT_POT_4
61                    MM TMP_2
62                    LD TMP_1
63                    -  TMP_2
64                    MM TMP_1
65                    /  INT_POT_5            ;; quinto char
66                    +  ZERO_BASE
```

```
67                  PD  /100                 ;;  imprime
68                  LD  TMP_1
69                  /   INT_POT_5
70                  *   INT_POT_5
71                  MM  TMP_2
72                  LD  TMP_1
73                  -   TMP_2
74                  MM  TMP_1
75                  RS  P_INT_ZERO
```

7 Tradução de comandos semânticos

7.1 Tradução de estruturas de controle de fluxo

Será apresentado nas próximas seções, as traduções das estruturas de controle de fluxo que constam na nossa linguagem e foram solicitadas para essa entrega, entre elas as estruturas if, if-else e while.

Cabe ressaltar que foram utilizadas simbologias nas traduções que serão substituídas pelo compilador no momento da geração de código. Uma dessas marcações é os dois pontos no começo de uma linha que significa que os comandos devem ser colocados no início do código gerado. Outra simbologia criada é da forma XN, onde X representa uma letra maiúscula qualquer e N é o índice da instância dentro do tipo de marcação X. As opções para X são as seguintes:

- `{C0}, {C1}, ...`: Conjunto de comandos
- `{R0}, {R1}, ...`: Referência
- `{L0}, {L1}, ...`: Label ou rótulo de uma instrução criados e exportados pelo código

Há também a marcação `{N}`, utilizada para denotar que a primeira instrução do código subsequente ao comando atual deve ser adicionada no lugar da marcação. Estamos considerando substituir sempre a marcação `{N}` por uma instrução simples que só sirva para simplificar, como por exemplo somar zero ao acumulador.

Conceitos da pilha aritmética são utilizados para o cálculo de expressões booleanas, no seção 7.3 explicações mais detalhadas são apresentadas.

7.1.1 Estrutura de controle de fluxo: IF

```
1          {C0}                    # calculo da expressao booleana
2          SC POP_ARITH
3          JZ {L0}                 # se 0, entao pula para L0 (else)
4          {C1}                    # codigo if, C pode ser nulo ou mais
               coisas
5  {L0}    {N}                     # N executa somente a expansao
```

7.1.2 Estrutura de controle de fluxo: IF-ELSE

```
1           {C0}                    # calculo da expressao booleana
2           SC POP_ARITH
3           JZ {L0}                 # se 0, entao pula para L0 (else)
4           {C0}                    # codigo if, C pode ser nulo ou mais
              coisas
5           JP {L1}                 # codigo fim, pula para fim
6   {L0}    {C2}                    # codigo else
7   {L1}    {N}                     # N executa somente a expansao
```

7.1.3 Estrutura de controle de fluxo: WHILE

```
1   {L0}    +  ZERO                 # STUB para facilitar criacao de codigo
2           {C0}                    # calculo da expressao boleana
3           SC POP_ARITH
4           JZ {L0}
5           {C1}                    # corpo do while
6   {L0}    {N}
```

7.2 Tradução de comandos imperativos

Essa seção explica as traduções dos comandos imperativos que constam na nossa linguagem e foram solicitadas para essa entrega, entre os quais os comandos de atribuição de valor, leitura da entrada padrão, impressão na saída padrão e chamada de subrotinas, associado à definicão de novas subrotinas. As mesmas definições das marcações explicadas anteriormente são válidas para as traduções a seguir.

7.2.1 Atribuição de valor

```
1           LD {R0}
2           MM {R1}
```

7.2.2 Comando de leitura

```
1   :       STORE_PTR_IO            <
2   :       GETS                    <
3           LV {R0}
4           MM STORE_PTR_IO
5           SC GETS
```

7.2.3 Comando de impressão

```
:          STRING_PTR              <
:          P_STRING                <
           LV {R0}
           MM STRING_PTR
           SC P_STRING
```

7.2.4 Definição e chamada de subrotinas

No caso da definição de subrotinas, a tradução fica a seguinte:

```
:          PUSH_CALL               <
:          POP_CALL                <
:          LOAD_WORD_FROM_STACK    <
:          SAVE_WORD_ON_STACK      <
:          WORD_TO_BE_SAVED        <

           {C0}                    # dados da funcao
           {C1}                    # constante com o numero de dados da
                                   #  funcao em nibbles
{L0}       JP /000                 # label funcao
           LV {L0}
           SC PUSH_CALL
           {C2}
           LV {L0}
           SC POP_CALL
           RS {L0}
```

Vale salientar que as funcoes que tratam a pilha de registro de ativação foram modificadas completamente para integração mais transparente na implementacao da função.

Já quando é identificada a chamada de uma subrotina já declarada, a seguinte tradução é utilizada:

```
           {C0}                    # Copia dados para os argumentos
                                   # da funcao
           SC {R0}
```

7.3 Cálculo de expressões aritméticas e booleanas

Além do que foi solicitado como obrigatório para essa entrega, pensamos ser importante definir a forma como fizemos a implementação do cálculo de expressões para a geração de código de saída.

Como o professor Ricardo Rocha nos explicou, a MVN não tem uma implementação real de pilha, porém consegue simular a existência de uma pilha com o uso de indi-

recionamentos que definem cada uma das operações da pilha, como *push* e *pop*. Baseado nesse conceito de código alinhavado, definimos diversas funções auxiliares que realizam operações simples de forma independente. Essas funções nos permitiram realizar o cálculo de expressões de maneira mais clara e com menos erros.

Para explicar de forma mais detalhada o processo utilizado para calcular as expressões, vamos supor que lemos uma expressão `1 + 2 * 3`. A gramática que já implementamos nas etapas anteriores cria uma árvore que já considera a ordem de prioridade das operações, fazendo com que a multiplicação ocorra antes da soma. Para esse caso, o código de máquina deve primeiro empilhar o 1, em seguida o 2 e depois o 3. Ao notar que uma operação de multiplicação foi finalizada, ele retira da pilha dois operandos, no caso o 2 e o 3, realizando a multiplicação e retornando a pilha o resultado da operação, no caso 6. Em seguida, é efetuada a operação de soma com os dois operandos que estão na pilha, o 1 e o 6, adicionando novamente o resultado, 7, na pilha.

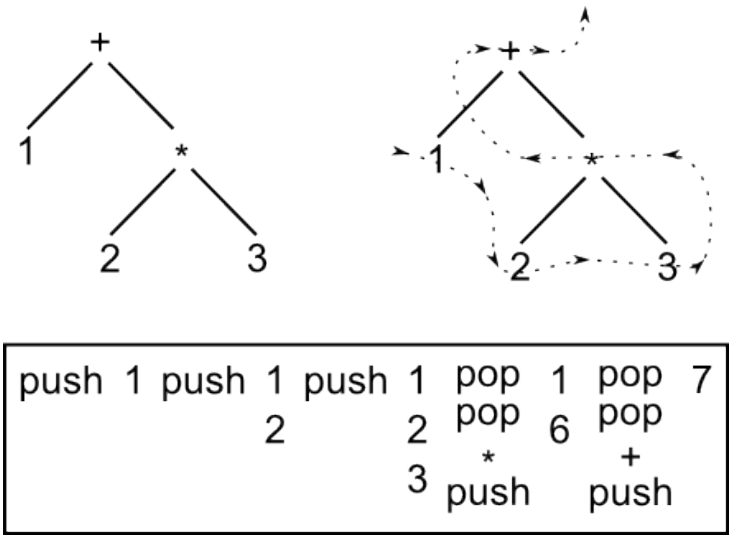


Figura 36 – Árvore da expressão e operações resultantes na pilha.

O mesmo tipo de lógica foi implementado também para operadores booleanos e permite a geração de código de forma mais simples, visto que já desenvolvemos funções auxiliares para essas operações.

Apresentamos abaixo as operações principais da pilha aritmética. Todas as outras operações da pilha se encontram no arquivo `std.asm` no final do arquivo.

```
1 ;-------------------------PUSH_ARITH-----------------------------
2 PUSH_ARITH      JP /000
3                 MM TMP_1
4                 LD ARIT_PTR_STACK
5                 +  TWO
6                 MM ARIT_PTR_STACK
7                 +  MOVE_CONST
8                 MM OP_PUSH_ARITH
```

```
9                   LD TMP_1
10  OP_PUSH_ARITH   JP /000
11                  RS PUSH_ARITH
12  ;--------------------------POP_ARITH-------------------------------
13  POP_ARITH       JP /000
14                  LD ARIT_PTR_STACK
15                  -  TWO
16                  MM ARIT_PTR_STACK
17                  +  TWO
18                  +  LOAD_CONST
19                  MM OP_POP_ARITH
20  OP_POP_ARITH    JP /000
21                  RS POP_ARITH
22  ;--------------------------SUM_ARITH-------------------------------
23  SUM_ARITH       JP /000
24                  SC POP_ARITH
25                  MM TMP_2
26                  SC POP_ARITH
27                  +  TMP_2
28                  SC PUSH_ARITH
29                  RS SUM_ARITH
30  ;--------------------------MUL_ARITH-------------------------------
31  MUL_ARITH       JP /000
32                  SC POP_ARITH
33                  MM TMP_2
34                  SC POP_ARITH
35                  *  TMP_2
36                  SC PUSH_ARITH
37                  RS MUL_ARITH
```

7.4 Arrays e Structs

Em *CZAR* **não** existem *Arrays* de tamanho dinâmico e sua criação está limitada à declaração. Sendo assim, suas dimensões internas são conhecidas pelo compilador a todo momento e seu cálculo de posição é facilitado e feito em tempo de compilação.

```
1   int i;
2
3   /*
4   * Array int[4][3][2]:
5   *
6   * [
7   *     [[0, 0], [0, 0], [0, 0]],
8   *     [[0, 0], [0, 0], [0, 0]],
9   *     [[0, 0], [0, 0], [0, 0]],
10  *     [[0, 0], [0, 0], [0, 0]]
```

```
11 * ]
12 *
13 * Preenchendo com:
14 * ----------------
15 * decl int i;
16 * decl int j;
17 * decl int k;
18 * decl int l;
19 * set i = 0;
20 * set j = 0;
21 * while (j < 4) {
22 *   set k = 0;
23 *   while (k < 3) {
24 *     set l = 0;
25 *     while (l < 2) {
26 *       set array_ex[j][k][l] = i;
27 *       set i = i + 1;
28 *       set l = l + 1;
29 *     }
30 *     set k = k + 1;
31 *   }
32 *   set j = j + 1;
33 * }
34 *
35 * Temos:
36 * ------
37 * [
38 *    [[0, 1], [2, 3], [4, 5]],
39 *    [[6, 7], [8, 9], [10, 11]],
40 *    [[12, 13], [14, 15], [16, 17]],
41 *    [[18, 19], [20, 21], [22, 23]]
42 * ]
43 * ou:
44 * ---
45 * [
46 *   0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10,
47 *   10, 11, 12, 13, 14, 15, 16, 17, 18,
48 *   19, 20, 21, 22, 23
49 * ]
50 *
51 *
52 *
53 *
54 */
55 acc = acumulado[n_dimensoes - 1] = 1; // maybe long 1L
56 for (i = n_dimensoes-2; i >= 0; i--) {
57    acc = acumulado[i+1] = dimensoes[i+1] * acc;
```

```
58 }
59 /*
60    acumulado = [6, 2, 1];
61 */
62 for (i = 0; i < n_dimensoes; i++) {
63     acumulado[i] = acumulado[i] * size_cell; // celulas podem ter
64                                              // tamanho variavel
65 }
66 for (i = 0; i < n_dimensoes; i++) {
67     fprints(str, "␣LV␣=%d␣", acumulado[i]);
68     cpy_to_lines_of_code(str);
69     fprints(str, "␣*␣ARR_DIM_%d", acumulado[i]);
70     cpy_to_lines_of_code(str);
71     fprints(str, "␣+␣␣ADDRS_ACCUMULATOR");
72     cpy_to_lines_of_code(str);
73     fprints(str, "␣MM␣ADDRS_ACCUMULATOR");
74     cpy_to_lines_of_code(str);
75 }
```

O cálculo de *structs* é resolvido em tempo de compilação. Uma vez que o tamanho de cada parte da estrutura é conhecida em tempo de compilação, é possível se fazer toda a aritmética de acesso via programação em *C*.

```
1     int deslocamento_para_celula(struct_struct* vi_struct, int
          cell_to_access) {
2         int sum_up_to_ptr = 0;
3         for (i = 0; i < cell_to_access; i++) {
4             sum_up_to_ptr += vi_struct->sizes[i];
5         }
6         return sum_up_to_ptr;
7     }
```

8 Exemplo de programa traduzido

A fim de demonstrar tudo o que foi pensado como a maneira de traduzir os comandos de alto nível da nossa linguagem CZAR, nós traduzimos um programa simples de fatorial que permite visualizar e testar a nossa tradução.

Para isso, apresentamos o exemplo de programa escrito em três diferentes linguagens: (i) na nossa linguagem de alto nível CZAR; (ii) tradução para linguagem de máquina, utilizando as bibliotecas complementares *std* e *stdio*; (iii) tradução para linguagem de saída MVN.

8.1 Exemplo de programa fatorial na linguagem de alto nível

```
1  const int fat_10_rec = 6;
2  decl int retorno;
3
4  meth
5
6  int fatorial_recursivo(int n) {
7      decl int retorno = 1;
8      if (n >= 1) {
9          set retorno = n * call fatorial_recursivo (n - 1);
10     }
11     return retorno;
12 }
13
14 main () {
15     set retorno = call fatorial_recursivo(fat_10_rec);
16     call io_print_int(retorno);
17 }
```

8.2 Tradução do programa fatorial para linguagem de máquina

```
1  P_STRING                <
2  STRING_PTR              <
3  P_INT_ZERO              <
4  TO_BE_PRINTED           <
5  P_LINE                  <
6  PUSH_CALL               <
7  PRINT_STACK_ADDRS       <
8  POP_CALL                <
9  READ_INT                <
```

```
10 STORE_PTR               <
11 GETS                    <
12 STORE_PTR_IO            <
13 WORD_TO_BE_SAVED        <
14 SAVE_WORD_ON_STACK      <
15 LOAD_WORD_FROM_STACK    <
16 ORIGIN_PTR              <
17 POP_CALL_FCT            <
18 PUSH_ARITH              <
19 POP_ARITH               <
20 SUM_ARITH               <
21 SUB_ARITH               <
22 DIV_ARITH               <
23 MUL_ARITH               <
24 AND_ARITH               <
25 OR_ARITH                <
26 NOT_ARITH               <
27 GEQ_OPER_ARITH          <
28 LEQ_OPER_ARITH          <
29 DBG                     <
30 @ /0000
31 CZAR_INICIO_CODE        JP CZAR_INICIO
32 CZAR_STUB               K =1
33 ;;=========================================================
34
35 CONST_VAR_0             K =6  ; const int fat_10_rec = 6;
36 GLOBAL_VAR_0            K =0  ; decl int retorno;
37 _CONST_NUM_1            K =0001
38
39 FUNCTION_0_RETURN       K =0  ; int
40 FUNCTION_0_ARG_0        K =0  ; int n
41 FUNCTION_0_TMP_0        K =0  ; function return
42 FUNCTION_0_LOCAL_VAR_0  K =0  ; decl int retorno = 1;
43                         K =4  ; int fatorial_recursivo(int n) {
44 FUNCTION_0              JP /000 ;
45                         LV FUNCTION_0
46                         SC PUSH_CALL
47
48                         LV =1                   ; retorno = 1;
49                         MM WORD_TO_BE_SAVED
50                         LV =3                   ; FUNCTION_0_LOCAL_VAR_0
51                         SC SAVE_WORD_ON_STACK
52                         ;; (n >= 1)
53                         LV =2                   ; FUNCTION_0_ARG_0
54                         SC LOAD_WORD_FROM_STACK
55                         SC PUSH_ARITH
56                         LD _CONST_NUM_1
```

```
57                          SC PUSH_ARITH
58                          SC GEQ_OPER_ARITH
59                          SC POP_ARITH
60                          JZ FUNCTION_0_LABEL_0
61
62                          LV =2
63                          SC LOAD_WORD_FROM_STACK
64                          SC PUSH_ARITH
65                          LD _CONST_NUM_1
66                          SC PUSH_ARITH
67                          SC SUB_ARITH
68                          SC POP_ARITH
69
70                          MM FUNCTION_0_ARG_0
71                          SC FUNCTION_0           ; call
                               fatorial_recursivo (n - 1);
72                          LD FUNCTION_0_RETURN
73                          MM WORD_TO_BE_SAVED
74                          LV =1
75                          SC SAVE_WORD_ON_STACK
76
77                          LV =2                   ; n * call
                               fatorial_recursivo (n - 1);
78                          SC LOAD_WORD_FROM_STACK
79                          SC PUSH_ARITH
80                          LV =1
81                          SC LOAD_WORD_FROM_STACK
82                          SC PUSH_ARITH
83                          SC MUL_ARITH
84
85                          SC POP_ARITH ; set retorno = *
86                          MM WORD_TO_BE_SAVED
87                          LV =3
88                          SC SAVE_WORD_ON_STACK
89  FUNCTION_0_LABEL_0      *  CZAR_STUB
90                          LV =3
91                          SC LOAD_WORD_FROM_STACK
92                          SC PUSH_ARITH
93                          SC POP_ARITH
94                          MM FUNCTION_0_RETURN
95                          LV FUNCTION_0
96                          SC POP_CALL     ;; trickery!
97                          RS FUNCTION_0   ;     return retorno;
98  ;; INIT PROG ============================================
99  CZAR_INICIO             * CZAR_STUB ;; stub instruction
100                         LD CONST_VAR_0
101                         MM FUNCTION_0_ARG_0
```

```
102                         SC FUNCTION_0       ;; depois chama..
103                         LD FUNCTION_0_RETURN
104                         MM GLOBAL_VAR_0
105                         LD GLOBAL_VAR_0
106                         MM TO_BE_PRINTED
107                         SC P_INT_ZERO
108 FIM                     HM /00
109 # CZAR_INICIO_CODE
```

8.3 Tradução do programa fatorial para linguagem de saída MVN

```
1  4000 0000 ; "P_STRING<"
2  4001 0000 ; "STRING_PTR<"
3  4002 0000 ; "P_INT_ZERO<"
4  4003 0000 ; "TO_BE_PRINTED<"
5  4004 0000 ; "P_LINE<"
6  4005 0000 ; "PUSH_CALL<"
7  4006 0000 ; "PRINT_STACK_ADDRS<"
8  4007 0000 ; "POP_CALL<"
9  4008 0000 ; "READ_INT<"
10 4009 0000 ; "STORE_PTR<"
11 4010 0000 ; "GETS<"
12 4011 0000 ; "STORE_PTR_IO<"
13 4012 0000 ; "WORD_TO_BE_SAVED<"
14 4013 0000 ; "SAVE_WORD_ON_STACK<"
15 4014 0000 ; "LOAD_WORD_FROM_STACK<"
16 4015 0000 ; "ORIGIN_PTR<"
17 4016 0000 ; "POP_CALL_FCT<"
18 4017 0000 ; "PUSH_ARITH<"
19 4018 0000 ; "POP_ARITH<"
20 4019 0000 ; "SUM_ARITH<"
21 4020 0000 ; "SUB_ARITH<"
22 4021 0000 ; "DIV_ARITH<"
23 4022 0000 ; "MUL_ARITH<"
24 4023 0000 ; "AND_ARITH<"
25 4024 0000 ; "OR_ARITH<"
26 4025 0000 ; "NOT_ARITH<"
27 4026 0000 ; "GEQ_OPER_ARITH<"
28 4027 0000 ; "LEQ_OPER_ARITH<"
29 4028 0000 ; "DBG<"
30 0000 0074
31 0002 0001
32 0004 0006
33 0006 0000
34 0008 0001
35 000a 0000
```

```
000c 0000
000e 0000
0010 0000
0012 0004
0014 0000
0016 3014
5018 a005
001a 3001
501c 9012
001e 3003
5020 a013
0022 3002
5024 a014
5026 a017
0028 8008
502a a017
502c a026
502e a018
0030 1062
0032 3002
5034 a014
5036 a017
0038 8008
503a a017
503c a020
503e a018
0040 900c
0042 a014
0044 800a
5046 9012
0048 3001
504a a013
004c 3002
504e a014
5050 a017
0052 3001
5054 a014
5056 a017
5058 a022
505a a018
505c 9012
005e 3003
5060 a013
0062 6002
0064 3003
5066 a014
5068 a017
```

```
506a a018
006c 900a
006e 3014
5070 a007
0072 b014
0074 6002
0076 8004
0078 900c
007a 3666
007c a014
007e 800a
0080 9006
0082 8006
5084 9003
5086 a002
0088 c000
```

Referências

ALFRED, V.; SETHI, R.; JEFFREY, D. *Compilers: principles, techniques and tools.* [S.l.]: Addison-Wesley, 1986.

NETO, J. J. *Introdução à Compilação.* [S.l.]: LTC, 1987. (ENGENHARIA DE COMPUTAÇÃO).