Escola Politécnica da Universidade de São Paulo

Linguagens e Compiladores

5ª parte do projeto – Tradução dos comandos

Henrique Sussumu Matsui Kano
Mi Che Li Lee

4 de dezembro de 2015

Sumário

Capítulo 1

Projeto Cimplex

O projeto consiste em definir uma linguagem e a partir dessa contruir o compilador em C. O nome da linguagem é Cimplex.

1.1 A linguagem

A linguagem base do compilador está descrita abaixo, em uma primeira instância, através de uma definição informal:

- tipagem: números naturais (int), números de ponto flutuante (float), strings (string), booleano (bool), vetor (array[1]) e matriz (variavel[1][1])
- estrutura inicial de programa:

```
begin do
  -Programa-
end
```

- Operações Aritméticas: Além das quatro básicas, também é possível fazer potências:

```
int var1 = 2;
int var2 = 4;
int var3 = var2 ^ var1;
```

- Declarações:

```
int var1 = 0;
```

  Sendo possível a declaração múltipla:

```
int var1 = 1, var2, var3 = 3;
float float1 = 0.1, float2 = 1;
```

No caso de arrays e matrizes, é possível inicia-los com literais:

```
string array[20] = ["olha", "esse", "teste", "!"];
bool matrix[3][3] = [
            [false, true, true],
            [true, false, true],
            [true, true, false]
          ];
```

- Comparações:

```
var1 <= var2;
```

Se os valores comparados forem numeros, é possível compará-los com:

```
<=, >=, <, >, ==, !=
```

Se eles forem booleanos, só é possível compará-los com:

```
==, !=
```

- Condicionais:

```
if(var1 == 0) do
      -Programa-
endif
```

- Iteradores:

```
while(true) do
      i = i + 1;
endwhile
```

- Funções: Declaração:

```
function int func1(float a, float b) do
      return 1;
endfunction
```

Chamada:

```
int var_int = func1(1.0, 2.0);
```

- Entrada:

```
scan(var1, var2, var3[0]);
```

- Saída:

```
print("var1=", var1, "var2=", var2, "var3[0]=", var3
   [0]);
```

- Seleção:

```
when(var1)
  is(1) do
    print("teste1");
  break;
  is(2) do
    print("teste2");
  break;
  default do
    print("erro");
  break;
endwhen
```

- Comentários:

```
##
    Sou um comentario de bloco!
##

#Sou um comentario de linha!
```

- Estruturas heterogêneas:

```
struct Caixa do
  int inteiro;
  string nome;
endstruct
```

Para acessar uma variável da caixa:

```
  Caixa->inteiro = 20;
int var1 = Caixa->inteiro;
```

Essa struct servirá apenas como uma variável simples. Se o usuário quiser outra caixa, ele deverá criar outra caixa:

```
struct Caixa2 do
  int inteiro;
  string nome;
endstruct
```

Capítulo 2

Análise Léxica

2.1 Enunciado

1. Quais são as funções do analisador léxico nos compiladores/interpretadores?

2. Quais as vantagens e desvantagens da implementação do analisador léxico como uma fase separada do processamento da linguagem de programação em relação à sua implementação como sub-rotina que vai extraindo um átomo a cada chamada?

3. Defina formalmente, através de expressões regulares sobre o conjunto de caracteres ASCII, a 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.

4. Converta cada uma das expressões regulares, assim obtidas, em autômatos finitos equivalentes que reconheçam as correspondentes linguagens por elas definidas.

5. Crie um autômato único que aceite todas essas linguagens a partir de um mesmo estado inicial, mas que apresente um estado final diferenciado para cada uma delas.

6. Transforme o autômato assim obtido em um transdutor, que emita como saída o átomo encontrado ao abandonar cada um dos estados finais para iniciar o reconhecimento de mais um átomo do texto.

7. Converta o transdutor assim obtido em uma sub-rotina, escrita na linguagem de programação de sua preferência. Não se esqueça que o final de

cada átomo é determinado ao ser encontrado o primeiro símbolo do átomo ou do espaçador seguinte. Esse símbolo não pode ser perdido, devendo-se, portanto, tomar os cuidados de programação que forem necessários para reprocessá-los, apesar de já terem sido lidos pelo autômato.

8. Crie um programa principal que chame repetidamente a sub-rotina assim construída, e a aplique sobre um arquivo do tipo texto contendo o texto-fonte a ser analisado. Após cada chamada, esse programa principal deve imprimir as duas componentes do átomo extraído (o tipo e o valor do átomo encontrado). Faça o programa parar quando o programa principal receber do analisador léxico um átomo especial indicativo da ausência de novos átomos no texto de entrada.

9. Relate detalhadamente o funcionamento do analisador léxico assim construído, incluindo no relatório: descrição teórica do programa; descrição da sua estrutura; descrição de seu funcionamento; descrição dos testes realizados e das saídas obtidas.

10. Explique como enriquecer esse analisador léxico com um expansor de macros do tipo #DEFINE, não paramétrico nem recursivo, as que permita a qualquer macro chamar outras macros, de forma não cíclica.

2.2 Introdução

O analisador léxico tem como função ler o arquivo que contém o código escrito na linguagem origem e transformar esse conjunto de caracteres em um conjunto de tokens que possuam um significado na linguagem escrita. As atividades principais nesta fase são: conversão numérica, à identificação de palavras reservadas e criação e manutenção de tabelas de símbolos da compilação.

2.3 Subrotina

A desvantagem do analisador léxico como uma subrotina é a necessidade do envio de mensagens por parte do analisador sintático para requisitar um novo átomo. Além disso é necessário tomar muito cuidado para não tornar o algoritmo do analisador léxico ineficiente. A vantagem da fase de análise léxica como uma subrotina é o aumento da rapidez no processo de compilação como um todo, pois o analisador léxico é dedicado à tarefa de reconhecimento léxico, enquanto o analisador sintático realiza as funções mais pesadas, chamando o analisador léxico somente quando necessário.

2.4 Construção

A seguir, é mostrado o processo de construção do Analisador Léxico.

2.4.1 Expressões regulares

Legenda:
<**qualquer**>: qualquer caracter que não se encaixa nos tipos de entrada definidos
<**nova linha**>: caracter lido como '\n'

Átomos

Há 10 tipos diferentes de átomos, e cada um está representado abaixo com suas respectivas sintaxes na forma de expressões regulares.

1. String: "<qualquer>+"|'<qualquer>+'
2. Operadores aritméticos: (+ | - | * | /)
3. Operadores booleanos: (&& | || | !)
4. Número: ([0-9]+ | [0-9]+.[0-9]+)
5. Comparadores: (> | < | <= | >= | == | !=)
6. Palavras Reservadas: (begin | do | end | if | elsif | endif | while | endwhile | function | endfunction | int | char | bool | float | return | true | false | print | scan | switch | case | continue | endswitch)
7. Identificadores: [a-zA-Z]+[a-zA-Z0-9_]+
8. Separadores: ([|] | | | ,)
9. Final de comando: (;)
10. Desconhecido: <nenhuma das anteriores>
11. Atribuição: (=)

Não-átomos

As sintaxes de elementos como comentários (de linha e de bloco) não fazem parte do conjunto de tipos de átomos pois devem ser ignorados durante a análise léxica. Entretanto, eles devem ser implementados no transdutor do analisador léxico, ortanto, estes foram definidos como não-átomos, e as sintaxes estão representadas abaixo por expressões regulares:

Comentário de linha: #<qualquer>*<nova linha>

Comentário de bloco: ##<qualquer>*##

2.4.2 Autômatos Finitos

As expressões regulares dos diferentes tipos de átomos, juntamente não-átomos foram representadas abaixo na forma de autômatos finitos :

Legenda:
<dig>: [0-9]
<letra>: [a-zA-Z]
<qualquer exceto”>: qualquer símbolo lido exceto ”
<qualquer exceto’>: qualquer símbolo lido exceto ’
<desconhecido>: qualquer símbolo não descrito pelos autômatos

String

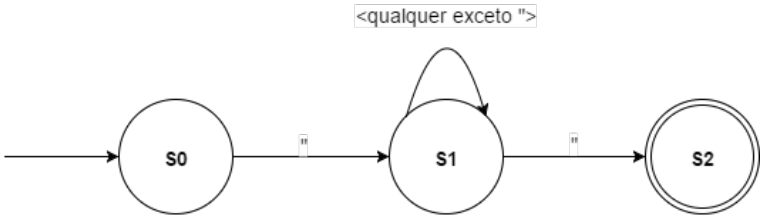

Figura 2.1: Autômato: String com aspas duplas

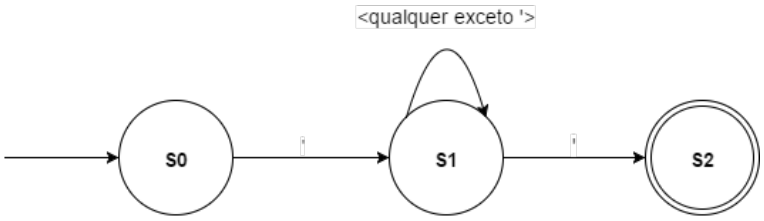

Figura 2.2: Autômato: String com aspas simples

Operadores aritméticos

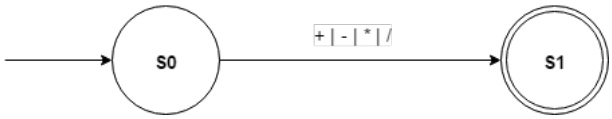

Figura 2.3: Autômato: Operadores aritméticos

Número

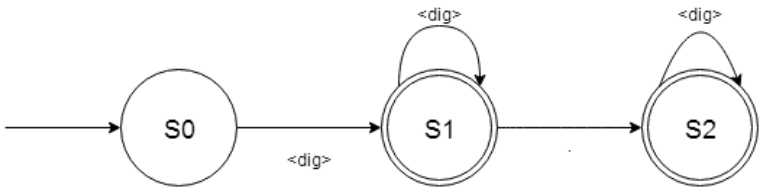

Figura 2.4: Autômato: Número

Atribuição

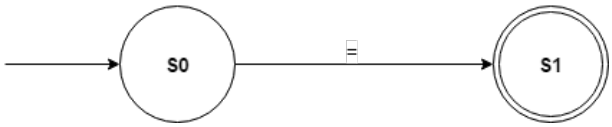

Figura 2.5: Autômato: Atribuição

Comparadores

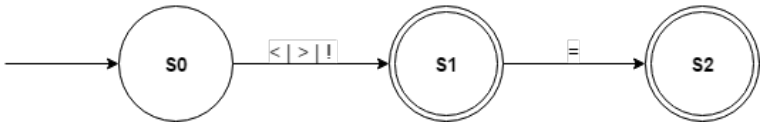

Figura 2.6: Autômato: Comparadores

Palavras reservadas

Figura 2.7: Autômato: Palavras reservadas

Identificadores

Figura 2.8: Autômato: Identificadores

Separadores

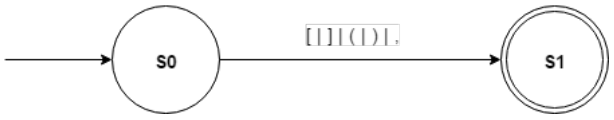

Figura 2.9: Autômato: Separadores

Final de comando

Figura 2.10: Autômato: Final de comando

Comentário de linha

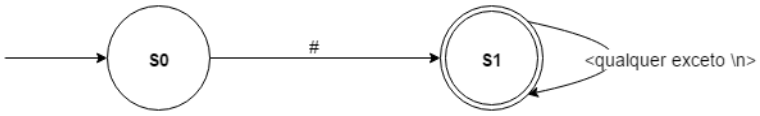

Figura 2.11: Autômato: Comentário de linha

Comentário de bloco

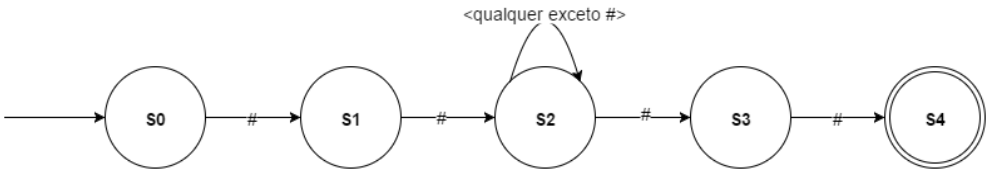

Figura 2.12: Autômato: Comentário de bloco

Desconhecido

Figura 2.13: Autômato: Desconhecido

Ao unificar os autômatos acima, tem-se um autômato que aceita todas as linguagens por elas definidas:

Autômato unificado

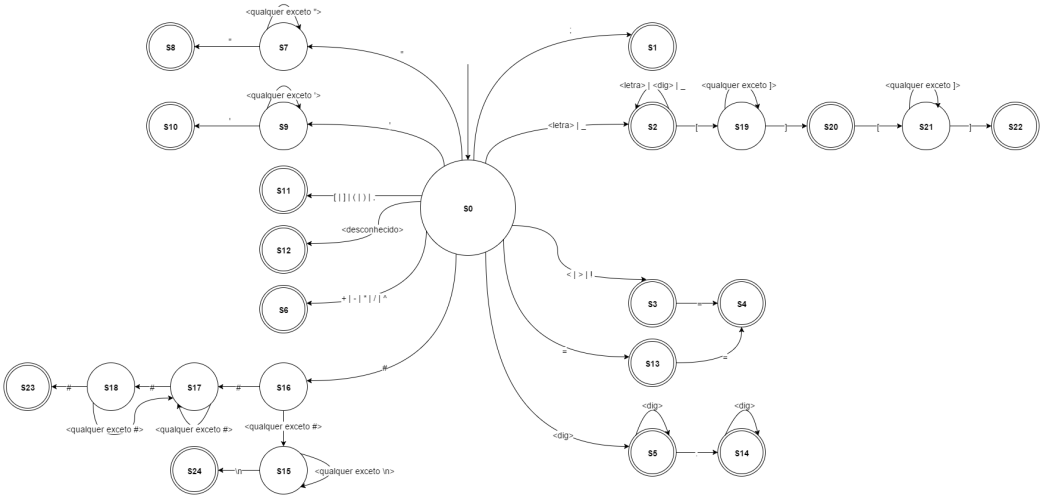

Figura 2.14: Autômato: unificado

Através do autômato unificado pode-se representar o transdutor, que emite como saída o átomo ao sair dos estados finais para iniciar o reconhecimento de mais um átomo do texto.

Transdutor

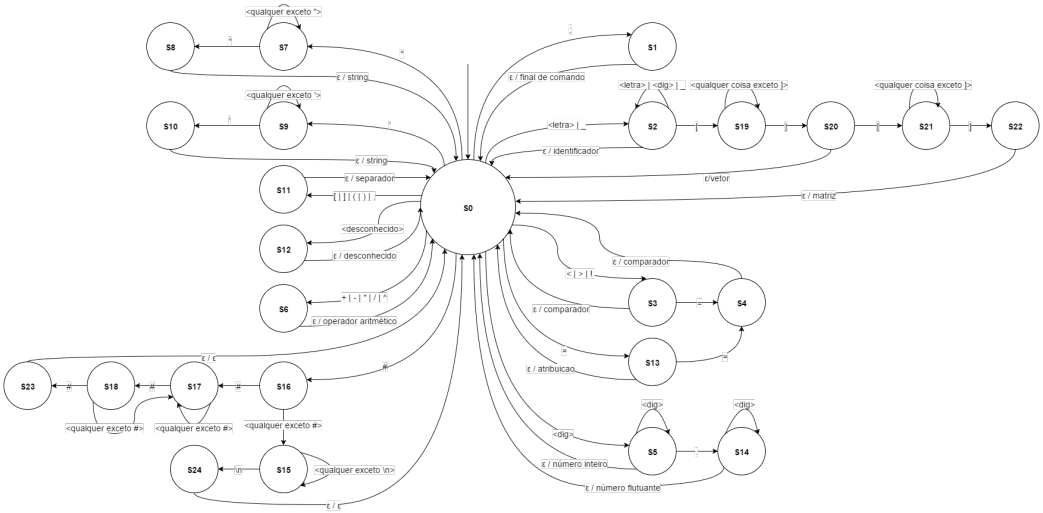

Figura 2.15: Autômato: Transdutor

2.5 Estrutura do código

Para a análise léxica foram usados alguns componentes programados:

- token
- transition_table
- automata
- analisador

2.5.1 Token

Arquivos: /utils/token.h e /utils/token.c
Auto explicativo, é uma estrutura que representa um token e guarda tanto o valor do token em characteres e o tipo do token, que se confundem com os dez tipos de linguagens obtidas em uma seção anterior.

2.5.2 Transition table

Arquivos: /transition_table.h e /transition_table.c
Contém a tabela de transição do transdutor (estado x tipo de entrada) e mais algumas funções auxiliares como um conversor de character para o tipo de entrada correspondente, a definição dos possíveis estados e os tipos de entrada e os tipos de token a ser criado dependendo do estado do autômato.

2.5.3 Automata

Arquivos: /automata.h e /automata.c
O autômato em sí, que implementa as funções simples de controlar o estado atual em função do estado atual e de uma tabela dada - no caso a tabela de transição definida (Transition table).

2.5.4 Analisador

Arquivos: /analisador.h e /analisador.c
Usando todos os componentes anteriormente citados, o analisador léxico mantém um autômato com a tabela de transição (representando o transdutor) e o arquivo com o programa com o objetivo de ler o arquivo e criar tokens. O analisador possui funções tanto para ler um token quanto para ler todos os tokens de um arquivo. Por hora as funções implementadas apenas criam os tokens e os

imprime sem retorná-los e nem salvar em alguma estrutura, o que será mudado em versões futuras. As palavras reservadas e os identificadores, em especial, possuem gramáticas em que uma (palavras reservadas) é subconjunto da outra (identificadores), por isso a distinção entre as duas é feito olhando uma tabela (vetor, no programa) de palavras reservadas.
Para criar um token, o analisador lê um caractére por vez até encontrar um outro token ou um separador ou um espaço. Porém, como é possível que o programador escreva todos os tokens juntos em uma expressão (por exemplo: "2+2"), foi necessário dois cuidados: uma boa implementação da tabela de transições, para que fosse possível detectar esse novo token e uma chamada de uma função para retornar o cabeçote de leitura do arquivo em um caractére nessa transição, pois ao acabar de ler o primeiro token, ele precisa ler mais uma entrada para então retornar ao estado inicial e criar o token, porém dessa maneira perde-se um caractére. Por isso deve-se retornar o cabeçote de leitura de um caractére ou então implementar um lookahead para perceber que o próximo estado é o estado inicial sem precisar ler o arquivo e mudar esse estado. Fora esse pequeno detalhe, a leitura do token foi uma tarefa fácil.
E a função que lê todos os tokens simplesmente descarta todos os espaços (tabs, nova linha, etc) e chama a primeira função para pegar um token, então verificando se arquivo não chegou ao fim ou se o ultimo token lido não foi "end"para parar a execução.

2.6 Testes

Arquivos: /input_file.txt
Para validar o programa feito, foi escrito um programa na linguagem definida e verificado os tokens gerados se o valor e o tipo desses estavam de acordo com o esperado. Também foram adicionadas palavras que não possuem um significado para observar o que ocorre (como é o caso da variável float, seu valor ou do print, que ainda não são reconhecidos).
É esperado que as strings reservadas sejam identificadas como RESERVED, colchetes e parenteses como SEPARADORES, variáveis como IDENTIFICADORES, sinais aritméticos como ARITH_SYMBOL, números como NUMBER, sinais de comparação (<, >, <=, ==, !=) como COMPARATOR, sinal de igual sozinho como ASSIGNMENT e conjuntos de símbolos (qualquer caracter ASCII) entre aspas simples e duplas como STRING.
O programa utilizado para testes foi escrito com o objetivo de ter a capacidade de produzir todos os tipos de token mencionados:

```
begin
  ##
```

```
  Sou um comentario de bloco!
##
float var1 = 1; #Comentario de linha
float var2 = 12.90;
float var3;
bool var_4;
char _var5;
int var6;
int matrix[2][4] = [[1, 2, 3, 4],[5, 6, 7, 8]];

function float media(int nota1, int nota2) do
  return (nota1 + nota2)/2;
endfunction

function float operacoes() do
  var1 = 43/2*5+1;
  var2 = 12 + 3 - 12 / 43 * 1;
  var3 = var1 + var2;
  return var;
endfunction

if(var1 > 1) do
  print "Maior que 1";
elsif(var1 < 1) do
  print "Menor que 1";
endif

if(var1 <= 1) do
  print "Menor igual a 1";
endif

if(var1 >= 1) do
  print "Maior igual a 1";
endif

if(var1 == true) do
  print "true";
endif

if!(var1 == false) do
  print "not false";
endif

while (true) do
  var1 = media(3,4);
  print var1;
endwhile

when(var) do
```

```
    is(1) do
      print "1";
      break;
    is(2) do
      print "2";
      break;
    is(3) do
      print "3";
      break;
    is(4) do
      print "4";
      break;
  endwhen
end
```

Listing 2.1: ENTRADA.txt

E a saída obtida foi:

```
begin                    | RESERVED (4)
float                    | RESERVED (4)
var1                     | IDENTIFIER (5)
=                        | ASSIGNMENT (9)
1                        | INT (2)
;                        | END_OF_COMMAND (7)
float                    | RESERVED (4)
var2                     | IDENTIFIER (5)
=                        | ASSIGNMENT (9)
12.90                    | FLOAT (10)
;                        | END_OF_COMMAND (7)
float                    | RESERVED (4)
var3                     | IDENTIFIER (5)
;                        | END_OF_COMMAND (7)
bool                     | RESERVED (4)
var_4                    | IDENTIFIER (5)
;                        | END_OF_COMMAND (7)
char                     | IDENTIFIER (5)
_var5                    | IDENTIFIER (5)
;                        | END_OF_COMMAND (7)
int                      | RESERVED (4)
var6                     | IDENTIFIER (5)
;                        | END_OF_COMMAND (7)
int                      | RESERVED (4)
matrix[                  | UNKNOWN (8)
2][4]                    | UNKNOWN (8)
=                        | ASSIGNMENT (9)
[[1                      | UNKNOWN (8)
,                        | SEPARATOR (6)
2                        | INT (2)
,                        | SEPARATOR (6)
```

```
3                          | INT (2)
,                          | SEPARATOR (6)
4]                         | UNKNOWN (8)
,                          | SEPARATOR (6)
[5                         | UNKNOWN (8)
,                          | SEPARATOR (6)
6                          | INT (2)
,                          | SEPARATOR (6)
7                          | INT (2)
,                          | SEPARATOR (6)
8]]                        | UNKNOWN (8)
;                          | END_OF_COMMAND (7)
function                   | RESERVED (4)
float                      | RESERVED (4)
media                      | IDENTIFIER (5)
(                          | SEPARATOR (6)
int                        | RESERVED (4)
nota1                      | IDENTIFIER (5)
,                          | SEPARATOR (6)
int                        | RESERVED (4)
nota2                      | IDENTIFIER (5)
)                          | SEPARATOR (6)
do                         | RESERVED (4)
return                     | RESERVED (4)
(                          | SEPARATOR (6)
nota1                      | IDENTIFIER (5)
+                          | ARITH_SYMBOL (1)
nota2                      | IDENTIFIER (5)
)                          | SEPARATOR (6)
/                          | ARITH_SYMBOL (1)
2                          | INT (2)
;                          | END_OF_COMMAND (7)
endfunction                | RESERVED (4)
function                   | RESERVED (4)
float                      | RESERVED (4)
operacoes                  | IDENTIFIER (5)
(                          | SEPARATOR (6)
)                          | SEPARATOR (6)
do                         | RESERVED (4)
var1                       | IDENTIFIER (5)
=                          | ASSIGNMENT (9)
43                         | INT (2)
/                          | ARITH_SYMBOL (1)
2                          | INT (2)
*                          | ARITH_SYMBOL (1)
5                          | INT (2)
+                          | ARITH_SYMBOL (1)
1                          | INT (2)
;                          | END_OF_COMMAND (7)
```

```
var2                        | IDENTIFIER (5)
=                           | ASSIGNMENT (9)
12                          | INT (2)
+                           | ARITH_SYMBOL (1)
3                           | INT (2)
-                           | ARITH_SYMBOL (1)
12                          | INT (2)
/                           | ARITH_SYMBOL (1)
43                          | INT (2)
*                           | ARITH_SYMBOL (1)
1                           | INT (2)
;                           | END_OF_COMMAND (7)
var3                        | IDENTIFIER (5)
=                           | ASSIGNMENT (9)
var1                        | IDENTIFIER (5)
+                           | ARITH_SYMBOL (1)
var2                        | IDENTIFIER (5)
;                           | END_OF_COMMAND (7)
return                      | RESERVED (4)
var                         | IDENTIFIER (5)
;                           | END_OF_COMMAND (7)
endfunction                 | RESERVED (4)
if                          | RESERVED (4)
(                           | SEPARATOR (6)
var1                        | IDENTIFIER (5)
>                           | COMPARATOR (3)
1                           | INT (2)
)                           | SEPARATOR (6)
do                          | RESERVED (4)
print                       | IDENTIFIER (5)
"Maior que 1"               | STRING (0)
;                           | END_OF_COMMAND (7)
elsif                       | RESERVED (4)
(                           | SEPARATOR (6)
var1                        | IDENTIFIER (5)
<                           | COMPARATOR (3)
1                           | INT (2)
)                           | SEPARATOR (6)
do                          | RESERVED (4)
print                       | IDENTIFIER (5)
"Menor que 1"               | STRING (0)
;                           | END_OF_COMMAND (7)
endif                       | RESERVED (4)
if                          | RESERVED (4)
(                           | SEPARATOR (6)
var1                        | IDENTIFIER (5)
<=                          | COMPARATOR (3)
1                           | INT (2)
)                           | SEPARATOR (6)
```

```
do                          | RESERVED (4)
print                       | IDENTIFIER (5)
"Menor igual a 1"           | STRING (0)
;                           | END_OF_COMMAND (7)
endif                       | RESERVED (4)
if                          | RESERVED (4)
(                           | SEPARATOR (6)
var1                        | IDENTIFIER (5)
>=                          | COMPARATOR (3)
1                           | INT (2)
)                           | SEPARATOR (6)
do                          | RESERVED (4)
print                       | IDENTIFIER (5)
"Maior igual a 1"           | STRING (0)
;                           | END_OF_COMMAND (7)
endif                       | RESERVED (4)
if                          | RESERVED (4)
(                           | SEPARATOR (6)
var1                        | IDENTIFIER (5)
==                          | COMPARATOR (3)
true                        | RESERVED (4)
)                           | SEPARATOR (6)
do                          | RESERVED (4)
print                       | IDENTIFIER (5)
"true"                      | STRING (0)
;                           | END_OF_COMMAND (7)
endif                       | RESERVED (4)
if                          | RESERVED (4)
!                           | COMPARATOR (3)
(                           | SEPARATOR (6)
var1                        | IDENTIFIER (5)
==                          | COMPARATOR (3)
false                       | RESERVED (4)
)                           | SEPARATOR (6)
do                          | RESERVED (4)
print                       | IDENTIFIER (5)
"not false"                 | STRING (0)
;                           | END_OF_COMMAND (7)
endif                       | RESERVED (4)
while                       | RESERVED (4)
(                           | SEPARATOR (6)
true                        | RESERVED (4)
)                           | SEPARATOR (6)
do                          | RESERVED (4)
var1                        | IDENTIFIER (5)
=                           | ASSIGNMENT (9)
media                       | IDENTIFIER (5)
(                           | SEPARATOR (6)
3                           | INT (2)
```

```
,                          | SEPARATOR (6)
4                          | INT (2)
)                          | SEPARATOR (6)
;                          | END_OF_COMMAND (7)
print                      | IDENTIFIER (5)
var1                       | IDENTIFIER (5)
;                          | END_OF_COMMAND (7)
endwhile                   | RESERVED (4)
when                       | RESERVED (4)
(                          | SEPARATOR (6)
var                        | IDENTIFIER (5)
)                          | SEPARATOR (6)
do                         | RESERVED (4)
is                         | RESERVED (4)
(                          | SEPARATOR (6)
1                          | INT (2)
)                          | SEPARATOR (6)
do                         | RESERVED (4)
print                      | IDENTIFIER (5)
"1"                        | STRING (0)
;                          | END_OF_COMMAND (7)
break                      | RESERVED (4)
;                          | END_OF_COMMAND (7)
is                         | RESERVED (4)
(                          | SEPARATOR (6)
2                          | INT (2)
)                          | SEPARATOR (6)
do                         | RESERVED (4)
print                      | IDENTIFIER (5)
"2"                        | STRING (0)
;                          | END_OF_COMMAND (7)
break                      | RESERVED (4)
;                          | END_OF_COMMAND (7)
is                         | RESERVED (4)
(                          | SEPARATOR (6)
3                          | INT (2)
)                          | SEPARATOR (6)
do                         | RESERVED (4)
print                      | IDENTIFIER (5)
"3"                        | STRING (0)
;                          | END_OF_COMMAND (7)
break                      | RESERVED (4)
;                          | END_OF_COMMAND (7)
is                         | RESERVED (4)
(                          | SEPARATOR (6)
4                          | INT (2)
)                          | SEPARATOR (6)
do                         | RESERVED (4)
print                      | IDENTIFIER (5)
```

```
"4"                          | STRING (0)
;                            | END_OF_COMMAND (7)
break                        | RESERVED (4)
;                            | END_OF_COMMAND (7)
endwhen                      | RESERVED (4)
end                          | RESERVED (4)
```

Listing 2.2: SAIDA.txt

2.7 Expansor de Macros

Caso fosse implementado um expansor de macros seria necessário um pré-processador do código-fonte antes da leitura deste pelo analisador léxico. O pré-processador procuraria por definições de macros no código-fonte e geraria um novo texto expandido, isento de definições e de chamadas de macros, tratável pelo compilador posteriormente [2].

Capítulo 3

Análise sintática

3.1 Introdução

A descrição informal da linguagem está apresentada na Seção 1.1. A partir da descrição informal, a linguagem Cimplex está descrita abaixo em BNF e em notação e Wirth.

3.2 Descrições

3.2.1 BNF

```
<D> ::= 0|1|2|3|4|5|6|7|8|9
<L> ::= 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
<*> ::= @|#|$|%|&|*|(|)|[|]|<|>|=|-|^|~||||\|/|,|.|:|;|_
<operador arit> ::= +|-|*|/|^
<operador bool> ::= &&||||!
<espaco> ::= \n| |\t
<L/D/_> ::= <L>|<D>|_
<qualquer> ::= <L>|<D>|<*>|<espaco>
<ID> ::= _|<L>|<ID><L/D/_>
<tipo> ::= int|float|string|bool|<ID>
<string> ::= '<lista de caracter>'|"<lista de caracter>"
<lista de caracter> ::= <qualquer>|<lista de caracter><
   qualquer>
<comparador bool> ::= ==|!=
<comparador> ::= <|>|<=|>=|<comparador bool>
<num natural> ::= <D>|<num natural><D>
<num inteiro> ::= <num natural>|-<num natural>
<num float> ::= <num natural>.<num natural>|-<num natural>.<
   num natural>
```

```
<numero> ::= <num inteiro>|<num float>
<bool> ::= true|false
<lista de valores> ::= <R-value>|<lista de valores>,<R-value>
<array> ::= <ID>[<num natural>]
<matrix> ::= <ID>[<num natural>][<num natural>]
<acesso struct> ::= <ID>-><ID>
<R-value> ::= <numero>|<string>|<bool>|<ID>|<array>|<matrix
   >|<function call>|<acesso struct>
<L-value> ::= <ID>|<array>|<matrix>

<var value> ::= <ID>|<array>|<matrix>|<function call>|<acesso
    struct>
<int value> ::= <num inteiro>|<var value>
<float value> ::= <num float>|<var value>
<string value> ::= <string>|<var value>
<bool value> ::= <bool>|<var value>

<lista de int value> ::= <int value>|<lista de int value>,<
   int value>
<lista de float value> ::= <float value>|<lista de float
   value>,<float value>
<lista de string value> ::= <string value>|<lista de string
   value>,<string value>
<lista de bool value> ::= <bool value>|<lista de bool value
   >,<bool value>

<expressao bool> ::= <termo bool>|<expressao bool>||<termo
   bool>
<termo bool> ::= <fator bool>|<termo bool>&&<fator bool>
<fator bool> ::= (<expressao bool>)|<bool>|<var value>
<expressao arit> ::= <termo arit>|<expressao arit>+<termo
   arit>|<expressao arit>-<termo arit>
<termo arit> ::= <fator2 arit>|<termo arit>*<fator2 arit>|<
   termo arit>/<fator2 arit>
<fator2 arit> ::= <fator arit>|<fator arit>^<fator arit>
<fator arit> ::= (<expressao arit>)|<numero>|<var value>

<multideclaracao int> ::= <ID> = <int value>|<multideclaracao
    int>,<ID> = <int value>|<ID>|<multideclaracao int>,<ID>
<multideclaracao float> ::= <ID> = <float value>|<
   multideclaracao float>,<ID> = <float value>|<ID>|<
   multideclaracao float>,<ID>
<multideclaracao string> ::= <ID> = <string value>|<
   multideclaracao string>,<ID> = <string value>|<ID>|<
   multideclaracao string>,<ID>
<multideclaracao bool> ::= <ID> = <bool value>|<
   multideclaracao bool>,<ID> = <bool value>|<ID>|<
   multideclaracao bool>,<ID>
```

```
<literal array> ::= <array int>|<array float>|<array string
   >|<array bool>
<array int> ::= [<lista de int value>]
<array float> ::= [<lista de float value>]
<array string> ::= [<lista de string value>]
<array bool> ::= [<lista de bool value>]
<array list> ::= <literal array>|<array list>,<literal array>
<literal matrix> ::= [<matrix int>|<matrix float>|<matrix
   string>|<matrix bool>]
<matrix int> ::= <array int>|<matrix int>,<array int>
<matrix float> ::= <array float>|<matrix float>,<array float>
<matrix string> ::= <array string>|<matrix string>,<array
   string>
<matrix bool> ::= <array bool>|<matrix bool>,<array bool>

<declaracao> ::=
          int <ID> = <int value>|
          float <ID> = <float value>|
          string <ID> = <string value>|
          bool <ID> = <bool value>|
          int <ID> = <lista de int value>|
          float <ID> = <lista de float value>|
          string <ID> = <lista de string value>|
          bool <ID> = <lista de bool value>|
          int <array> = <array int>|
          float <array> = <array float>|
          string <array> = <array string>|
          bool <array> = <array bool>|
          int <matrix> = [<matrix int>]|
          float <matrix> = [<matrix float>]|
          string <matrix> = [<matrix string>]|
          bool <matrix> = [<matrix bool>]|
          int <multideclaracao int>|
          float <multideclaracao float>|
          string <multideclaracao string>|
          bool <multideclaracao bool>

<lista de declaracoes> ::= <declaracao>;|<lista de
   declaracoes><declaracao>;
<struct> ::= struct <ID> do <lista de declaracoes> endstruct;
<atribuicao> ::= <L-value> = <R-value>|<array> = <literal
   array>|<matrix> = <literal matrix>

<comando> ::= <atribuicao>|<condicional>|<iteracao>|<selecao
   >|<entrada>|<saida>
<condicao> ::= <expressao bool><comparador bool><expressao
   bool>|<expressao arit><comparador><expressao arit>
<expressao> ::= <expressao bool>|<expressao arit>
```

```
<lista de expressoes> ::= <expressao>|<lista de expressoes><
   expressao>
<lista de comandos> ::= E|<lista de comandos><comando>;
<parametro> ::= <tipo> <ID>|<tipo> <array>|<tipo> <matrix>
<lista de parametros> ::= <parametro>|<lista de parametros>,<
   parametro>

<programa> ::= begin<lista de comandos>end
<entrada> ::= scan(<lista de valores>);
<saida> ::= print(<lista de valores>);
<function> ::= function <tipo> <ID>(<lista de parametros>) do
    <lista de comandos> <retorno> endfunction
<function call> ::= <ID>(<lista de valores>)
<condicional> ::= if (<condicao>) do <lista de comandos> <
   else> endif
<else> ::= E|elsif (condicao>) do <lista de comandos> <else>|
   else do <lista de comandos>
<iteracao> ::= while (<condicao>) do <lista de comandos>
   endwhile
<selecao> ::= when(<ID>) <lista de cases> default <lista de
   comandos> endwhen
<lista de cases> ::= <case>|<lista de cases><case>
<case> ::= is (<R-value>) do <lista de comandos> break;
<retorno> ::= return|return <R-value>
```

Listing 3.1: bnf.txt

3.2.2 Notação de Wirth

```
D = "0"|"1"|"2"|"3"|"4"|"5"|"6"|"7"|"8"|"9".
L = "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".
SPECIAL = "@"|"#"|"$"|"%"|"""|"'"|"&"|"*"|"("|")
   "|"["|"]"|"<"|">"|"="|"-"|"^"|"~"""|"\"|"/"|","|"."|":"|";".

ESPACO = "\n" | "\t" | " ".
ANY = L | D | SPECIAL | ESPACO.
ID = (L | "_"){L | D | "_"}.
TIPO = "int"|"float"|"string"|"bool".
STRING = """{ANY}""" | "'"{ANY}"'".
INT = D{D}.
FLOAT = INT"."INT.
BOOL = "true" | "false".
COMPARADOR = "<"|">"|"<="|">="|"=="|"!=".

ELEM_VETOR = ID"["INT"]".
ELEM_MATRIZ = ID"["INT"]" "["INT"]".
```

```
PROGRAMA = "begin" COMANDOS "end".
COMANDOS = COMANDO{COMANDO}.
COMANDO = ATRIBUICAO | DECLARACAO | CONDICIONAL | ITERACAO |
   SELECAO | ENTRADA | SAIDA.
ATRIBUICAO = VARIAVEL "=" (EXPRESSAO | STRING) ";".
DECLARACAO = DECLARACAO_VARIAVEL | DECLARACAO_FUNCAO |
   DECLARACAO_STRUCT.
DECLARACAO_VARIAVEL = TIPO VARIAVEL{","VARIAVEL} ";".
DECLARACAO_FUNCAO = "function" TIPO ID"("[DECLARACAO_SIMPLES
   {","DECLARACAO_SIMPLES}]")" "do" [COMANDOS] [("return" ";"
    | "return" EXPRESSAO ";")] "endfunction".
DECLARACAO_SIMPLES = TIPO ID.
DECLARACAO_STRUCT = "struct" ID "do" {DECLARACAO_VARIAVEL |
   DECLARACAO_FUNCAO} "endstruct".
CONDICIONAL = "if" "(" CONDICAO ")" "do" COMANDOS ELSE "endif
   ".
ELSE = {"elsif" "(" CONDICAO ")" "do" COMANDOS }["else" "do"
   COMANDOS].
CONDICAO = EXPRESSAO [COMPARADOR EXPRESSAO].
ITERACAO = "while" "(" CONDICAO ")" "do" COMANDOS "endwhile".
SELECAO = "when" "(" EXPRESSAO ")" CASO{CASO} "endwhen".
CASO = "is" "(" EXPRESSAO ")" "do" COMANDOS "continue" ";".
ENTRADA = "scan" "(" VARIAVEL{","VARIAVEL} ")" ";".
SAIDA = "print" "(" ((EXPRESSAO | STRING){","(EXPRESSAO |
   STRING)}) ")" ";".

EXPRESSAO = TERMO { ("+"|"-"|"or") TERMO }.
TERMO = FATOR { ("*"|"/"|"and") FATOR }.
FATOR = ["not"] ("("EXPRESSAO")" | BOOL | VALOR_GENERICO) |
   INT | FLOAT.

ACESSO_STRUCT = ID"."ID.
CHAMADA_FUNC = ID"("[EXPRESSAO{","EXPRESSAO}]")".
VALOR_GENERICO = ID | ELEM_VETOR | ELEM_MATRIZ | CHAMADA_FUNC
    | ACESSO_STRUCT.
VARIAVEL = ID | ELEM_VETOR | ELEM_MATRIZ.
```

Listing 3.2: wirth.txt

3.2.3 Exemplo

Um exemplo de programa é o mesmo utilizado na Seção 2.6

Capítulo 4

Reconhecedor sintático

4.1 Introdução

Em um compilador, o analisador sintático tem a função de verificar se uma sequência de tokens, gerado pelo analisador léxico, faz sentido para a dada linguagem que o compilador reconhece.

Para a construção do reconhecedor sintático, a descrição da linguagem Cimplex na notação de Wirth foi reduzida para a seguinte:

```
PROGRAMA = "begin" (COMANDO{COMANDO}) "end".

COMANDO = ((ID | (ID"["INT"]") | (ID"["INT"]" "["INT"]")) "="
    (EXPRESSAO | STRING) ";") | DECLARACAO | ("if" "("
   CONDICAO ")" "do" (COMANDO{COMANDO}) ({"elsif" "("
   CONDICAO ")" "do" (COMANDO{COMANDO}) }["else" "do" (
   COMANDO{COMANDO})]) "endif") | ("while" "(" CONDICAO ")" "
   do" (COMANDO{COMANDO}) "endwhile") | ("when" "(" EXPRESSAO
    ")" ("is" "(" EXPRESSAO ")" "do" (COMANDO{COMANDO}) "
   continue" ";"){("is" "(" EXPRESSAO ")" "do" (COMANDO{
   COMANDO}) "continue" ";")} "endwhen") | ("scan" "(" (ID |
   (ID"["INT"]") | (ID"["INT"]" "["INT"])){","(ID | (ID"["
   INT"]") | (ID"["INT"]" "["INT"]"))} ")" ";") | ("print"
   "(" ((EXPRESSAO | STRING){","(EXPRESSAO | STRING)}) ")"
   ";").

DECLARACAO = (TIPO (ID | (ID"["INT"]") | (ID"["INT"]" "["INT
   "]")){","(ID | (ID"["INT"]") | (ID"["INT"]" "["INT"]"))}
   ";") | ("function" TIPO ID"("[(TIPO ID){","(TIPO ID)}]")"
   "do" [(COMANDO{COMANDO})] [("return" ";" | "return"
   EXPRESSAO ";")] "endfunction") | ("struct" ID "do" {(TIPO
   (ID | (ID"["INT"]") | (ID"["INT"]" "["INT"])){","(ID | (
   ID"["INT"]") | (ID"["INT"]" "["INT"]"))} ";") | ("function
```

```
   " TIPO ID"("[(TIPO ID){","(TIPO ID)}]")" "do" [(COMANDO{
   COMANDO})] [("return" ";" | "return" EXPRESSAO ";")] "
   endfunction")} "endstruct").

CONDICAO = EXPRESSAO [COMPARADOR EXPRESSAO].

EXPRESSAO = ((["not"] ("("EXPRESSAO")" | BOOL | (ID | (ID"["
   INT"]") | (ID"["INT"]" "["INT"]") | (ID"("[EXPRESSAO{","
   EXPRESSAO}]")") | (ID"."ID))) | INT | FLOAT) { ("*"|"/"|"
   and") (["not"] ("("EXPRESSAO")" | BOOL | (ID | (ID"["INT
   "]") | (ID"["INT"]" "["INT"]") | (ID"("[EXPRESSAO{","
   EXPRESSAO}]")") | (ID"."ID))) | INT | FLOAT) }) {
   ("+"|"-"|"or") ((["not"] ("("EXPRESSAO")" | BOOL | (ID | (
   ID"["INT"]") | (ID"["INT"]" "["INT"]") | (ID"("[EXPRESSAO
   {","EXPRESSAO}]")") | (ID"."ID))) | INT | FLOAT) {
   ("*"|"/"|"and") (["not"] ("("EXPRESSAO")" | BOOL | (ID | (
   ID"["INT"]") | (ID"["INT"]" "["INT"]") | (ID"("[EXPRESSAO
   {","EXPRESSAO}]")") | (ID"."ID))) | INT | FLOAT) }) }.
```

Listing 4.1: reduced_wirth.txt

4.2 Autômato de Pilha estruturado

4.2.1 Lista de Transições

Utilizando o site http://mc-barau.herokuapp.com/, a descrição da linguagem em notação de Wirth foi transformada em Autômatos Finitos Determinísticos:

```
Sub-maquina PROGRAMA:
Minimized DFA:
initial: 0
final: 3
(0, "begin") -> 1
(1, COMANDO) -> 2
(2, COMANDO) -> 2
(2, "end") -> 3

Sub-maquina COMANDO:
Minimized DFA:
initial: 0
final: 2
(0, ID) -> 1
(0, DECLARACAO) -> 2
(0, "if") -> 3
(0, "while") -> 4
(0, "when") -> 5
(0, "scan") -> 6
```

```
(0, "print") -> 7
(1, "[") -> 8
(1, "=") -> 9
(3, "(") -> 21
(4, "(") -> 37
(5, "(") -> 34
(6, "(") -> 18
(7, "(") -> 10
(8, INT) -> 13
(9, EXPRESSAO) -> 11
(9, STRING) -> 11
(10, EXPRESSAO) -> 12
(10, STRING) -> 12
(11, ";") -> 2
(12, ")") -> 11
(12, ",") -> 10
(13, "]") -> 14
(14, "[") -> 15
(14, "=") -> 9
(15, INT) -> 16
(16, "]") -> 17
(17, "=") -> 9
(18, ID) -> 19
(19, "[") -> 20
(19, ")") -> 11
(19, ",") -> 18
(20, INT) -> 24
(21, CONDICAO) -> 22
(22, ")") -> 23
(23, "do") -> 25
(24, "]") -> 26
(25, COMANDO) -> 27
(26, "[") -> 28
(26, ")") -> 11
(26, ",") -> 18
(27, COMANDO) -> 27
(27, "elsif") -> 3
(27, "else") -> 29
(27, "endif") -> 2
(28, INT) -> 30
(29, "do") -> 32
(30, "]") -> 31
(31, ")") -> 11
(31, ",") -> 18
(32, COMANDO) -> 33
(33, COMANDO) -> 33
(33, "endif") -> 2
(34, EXPRESSAO) -> 35
(35, ")") -> 36
```

```
(36, "is") -> 38
(37, CONDICAO) -> 39
(38, "(") -> 40
(39, ")") -> 41
(40, EXPRESSAO) -> 42
(41, "do") -> 43
(42, ")") -> 44
(43, COMANDO) -> 45
(44, "do") -> 46
(45, COMANDO) -> 45
(45, "endwhile") -> 2
(46, COMANDO) -> 47
(47, COMANDO) -> 47
(47, "continue") -> 48
(48, ";") -> 49
(49, "is") -> 38
(49, "endwhen") -> 2

Sub-maquina DECLARACAO:
Minimized DFA:
initial: 0
final: 7
(0, TIPO) -> 1
(0, "function") -> 2
(0, "struct") -> 3
(1, ID) -> 4
(2, TIPO) -> 25
(3, ID) -> 5
(4, "[") -> 6
(4, ",") -> 1
(4, ";") -> 7
(5, "do") -> 8
(6, INT) -> 12
(8, TIPO) -> 9
(8, "function") -> 10
(8, "endstruct") -> 7
(9, ID) -> 34
(10, TIPO) -> 11
(11, ID) -> 13
(12, "]") -> 14
(13, "(") -> 15
(14, "[") -> 16
(14, ",") -> 1
(14, ";") -> 7
(15, TIPO) -> 17
(15, ")") -> 18
(16, INT) -> 19
(17, ID) -> 26
(18, "do") -> 20
```

```
(19, "]") -> 21
(20, COMANDO) -> 20
(20, "return") -> 22
(20, "endfunction") -> 8
(21, ",") -> 1
(21, ";") -> 7
(22, ";") -> 23
(22, EXPRESSAO) -> 24
(23, "endfunction") -> 8
(24, ";") -> 23
(25, ID) -> 27
(26, ",") -> 28
(26, ")") -> 18
(27, "(") -> 29
(28, TIPO) -> 17
(29, TIPO) -> 30
(29, ")") -> 31
(30, ID) -> 41
(31, "do") -> 32
(32, COMANDO) -> 32
(32, "return") -> 33
(32, "endfunction") -> 7
(33, ";") -> 36
(33, EXPRESSAO) -> 37
(34, "[") -> 35
(34, ",") -> 9
(34, ";") -> 8
(35, INT) -> 38
(36, "endfunction") -> 7
(37, ";") -> 36
(38, "]") -> 39
(39, "[") -> 40
(39, ",") -> 9
(39, ";") -> 8
(40, INT) -> 42
(41, ",") -> 43
(41, ")") -> 31
(42, "]") -> 44
(43, TIPO) -> 30
(44, ",") -> 9
(44, ";") -> 8

Sub-maquina CONDICAO:
Minimized DFA:
initial: 0
final: 1, 3
(0, EXPRESSAO) -> 1
(1, COMPARADOR) -> 2
(2, EXPRESSAO) -> 3
```

```
Sub-maquina EXPRESSAO:
Minimized DFA:
initial: 0
final: 3, 4, 10
(0, "not") -> 1
(0, "(") -> 2
(0, BOOL) -> 3
(0, ID) -> 4
(0, INT) -> 3
(0, FLOAT) -> 3
(1, "(") -> 2
(1, BOOL) -> 3
(1, ID) -> 4
(2, EXPRESSAO) -> 5
(3, "*") -> 0
(3, "/") -> 0
(3, "and") -> 0
(3, "+") -> 0
(3, "-") -> 0
(3, "or") -> 0
(4, "(") -> 6
(4, "[") -> 7
(4, ".") -> 8
(4, "*") -> 0
(4, "/") -> 0
(4, "and") -> 0
(4, "+") -> 0
(4, "-") -> 0
(4, "or") -> 0
(5, ")") -> 3
(6, EXPRESSAO) -> 13
(6, ")") -> 3
(7, INT) -> 9
(8, ID) -> 3
(9, "]") -> 10
(10, "[") -> 11
(10, "*") -> 0
(10, "/") -> 0
(10, "and") -> 0
(10, "+") -> 0
(10, "-") -> 0
(10, "or") -> 0
(11, INT) -> 12
(12, "]") -> 3
(13, ")") -> 3
(13, ",") -> 14
```

```
(14, EXPRESSAO) -> 13
```

Listing 4.2: submaquinas_geradas_online.txt

4.2.2 Lista de autômatos

A lista seguinte de autômatos foi gerada a partir do programa JFLAP (http://www.cs.duke.edu/csed/jflap/):

Figura 4.1: PROGRAMA.jpg

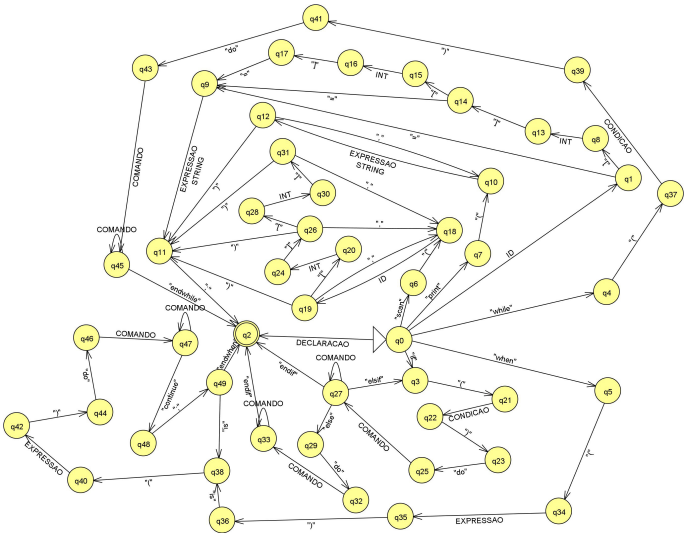

Figura 4.2: COMANDO.jpg

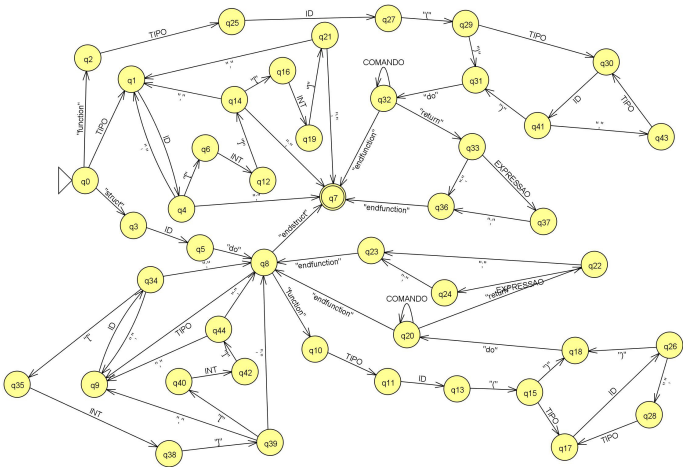

Figura 4.3: DECLARACAO.jpg

Figura 4.4: CONDICAO.jpg

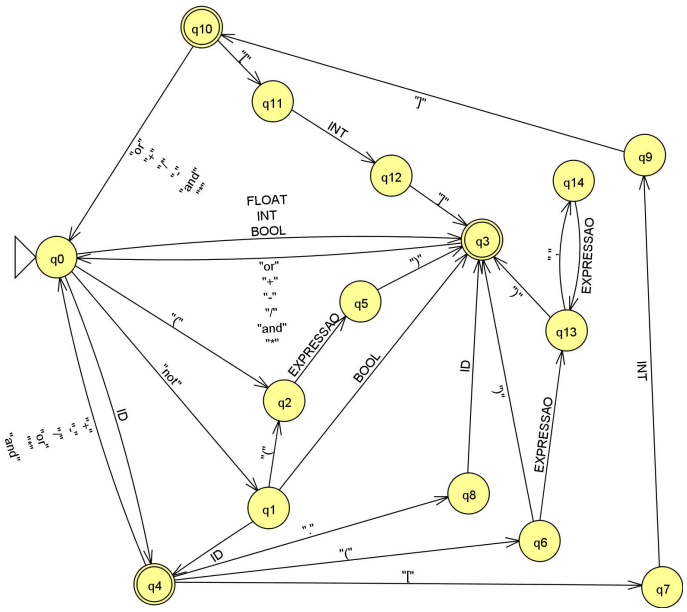

Figura 4.5: EXPRESSAO.jpg

4.3 Implementação

Com os automatos de pilha prontos, bastou fazer a implementação. Para isso, foi criado uma struct (AutomataPE) que representa um tipo de monitor. Esse monitor é responsável por gerenciar a pilha (empilhando e desempilhando quando necessário os sub-autômatos), criar os sub-autômatos chamados, e gererenciar as transições dos autômatos. Para isso, são guardadas várias tabelas auxiliares que informam pontos importantes dos autômatos, que foram identificados com um id próprio cada um:

- transitionTables: Um array de tabelas indexado pelo id da máquina em questão que informa, para cada estado da máquina e token recebido, qual o proximo estado dela;
- subMachineCall: Um array de tabelas indexado pelo id da máquina em questão que informa, para o estado atual dessa máquina, qual o id da máquina deve ser chamada. Essas tabelas acabam por ser matrizes-coluna;
- afterCallStates: Um array de tabelas indexado pelo id da máquina em questão que informa, para a máquina que acabou de ser desempilhada, para qual estado ela deve ir. São tabelas id x id;
- finalStates: Um array de tabelas indexado pelo id da máquina em questão que informa, para a máquina em questão quais estados são finais;

Com essas informações, falta apenas a lógica para fazer as transições: primeiramente, deve ser chamada a função initMachines para que sejam inicializadas as tabelas descritas anteriormente. Dessa chamada será retornado um AutomatoPE. Esse deve então ser usado para chamar a função automataPERun, que recebe também o arquivo que contém o programa e um token que será usado de buffer e deve, de inicio, conter o token inicial. Com essas informações, é possível calcular o próximo estado da máquina inicial. Desse ponto são várias as situações possíveis:

- Se esse próximo estado não existir na tabela de transição, deve-se investigar se nesse estado é possível chamar alguma máquina;
- Se na tabela de chamada de submaquinas existir uma submáquina a ser chamada, basta empilhar a submaquina atual e começar a nova submáquina;
- Se na tabela de chamada de submaquinas não existir um id de máquina válido, ainda é possível que o estado em que o autômato reside é final

- Se o estado for final, é necessário desempilhar uma máquina e recomeçar o processo para verificar se o token é reconhecido por máquinas "a cima;
- Se o estado não for final, isso significa que o reconhecimento falhou;

Por causa dessa implementação preditiva (qual deve ser o próximo estado), deve ser dado o primeiro token para começar a operação. E como uma ação de desempilhar e empilhar não envolvem o consumo de token, mas são considerados como um passo do algoritmo, deve-se usar um buffer para não perder tokens já lidos.

Capítulo 5

Ambiente de Execução

5.1 Introdução

O trabalho feito até agora visava mais o reconhecimento de uma linguagem compilada, sendo portanto um processo comum para o desenvolvimento de todo e qualquer compilador podendo ser repetida para qualquer nova linguagem independente da linguagem "alvo"(gerada pela compilação). Entretanto, sendo esse um trabalho de um compilador, cedo ou tarde é necessário estudar a linguagem gerada, se ainda não escolhida.

No caso, o código gerado será utilizada para a máquina disponibilizada, MVN (Máquina de Von Neumann)[1], que simula um processador simples composto de Memória, Acumulador e Registradores Auxiliadores.

A linguagem de saída do compilador desenvolvido utiliza uma linguagem simbólica de mnemônicos descrita nos itens 5.2 e 5.3. A linguagem de saída é assim construída para utilizar o Montador disponibilizado, que também faz parte do ambiente de execução.

5.2 Instruções da linguagem de saída

A listagem e descrição das instruções da linguagem de saída se encontram na tabela 5.1

5.3 Pseudoinstruções da linguagem de saída

A listagem e descrição das pseudoinstruções da linguagem de saída se encontram na tabela 5.2

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

Tabela 5.1: Tabela de instruções.

Pseud.	Descrição
@	Recebe um operando numérico, define o endereço da instrução seguinte
K	Reserva uma área para constante de 2 bytes (operando hexadecimal)
$	Reserva uma área de dados com tamanho especificado (operando hexadecimal)
#	Define o fim físico do texto-fonte
&	Define uma origem relocável para o código a ser gerado, o operando é o endereço em que o próximo código se localizará (relativo à origem do código atual)
>	Define endereço simbólico de entrada (Entry Point)
<	Define um endereço simbólico que referencia um entry-point externo (External)

Tabela 5.2: Tabela de pseudoinstruções.

5.4 Características Gerais

O simulador em questão apresenta algumas caracteríscas que devem ser levadas em conta para um melhor entendimento dele:

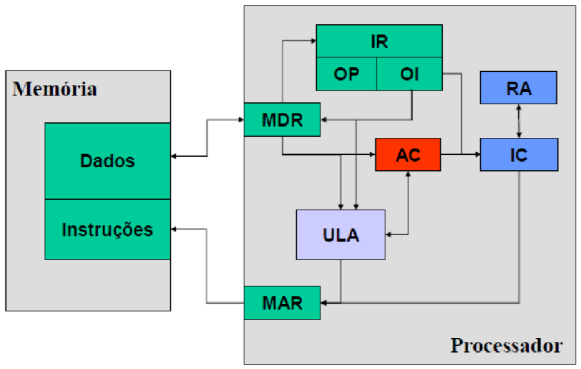

Figura 5.1: arquitetura da MVN disponibilizada

- O ambiente de execução é composto por um simulador com: memória principal, acumulador e registradores auxiliares;
- Arquitetura de Von Neumann: área de memória compartilhada entre código e dados;

- Arquitetura de acumulador: apesar de apresentar utilizar alguns registradores, apenas um está disponível para uso do programador, com exceção dos registradores de instrução (que sempre deve poder ser configurado pelo indivíduo): o acumulador. Logo, o manuseio de dados conta com vários acessos à memória;

- Words de 2 bytes/quatro hexas: tanto para instruções quanto para dados, sendo, no caso de instruções, o primeiro valor hexa reservado para identificar a instrução. Isso implica e explica a existência das 16 operações já explanadas e memória com 4095 words de tamanho (item seguinte);

- Memória com 4095 words (0xFFF) de tamanho: Uma vez que deve ser possível apontar para todos os endereços de memória para realizar operações de memória, não faz sentido ter mais que tal espaço;

- Endereçamento por bytes: Apesar de uma word formar uma instrução, muitas vezes deseja-se trabalhar com dados em forma de bytes, por isso o endereçamento de memória é feito por bytes, o que também pode gerar um certo estranhamento quando a primeira instrução ocorre no espaço zero e a segunda instrução no dois. É apenas o efeito de uma decisão de projeto

- Operações aritméticas básicas, apenas: todas as operações devem ser realizadas com operações aritméticas. Caso a linguagem compilada entenda operadores lógicos, esses devem ser implementados com operações aritméticas;

- Implementação de subrotinas: O simulador já possui uma implementação para auxiliar o uso e construção de subrotinas. Ao usar a chamada de SC, o valor de endereço "mais um"é guardado no início da subrotina e para retornar usando RS, esse endereço guardado na chamada de SC é usado para retornar. Essa implementação gera o problema de funções recursivas, uma vez que a função, ao chamar a sí mesma iria perder a referência ao programa principal. Para contornar esse problema, deve-se implementar um trecho de código auto-modificável ou uma pilha, o que apresenta um outro grande problema: devido à memória reduzida da máquina, dependendo da profundidade da recursão, o programa pode acabar usando muita memória para a pilha;

- Memória reduzida: como mencionado acima, devido à memória limitada, em muitos trechos deve-se usar da auto-modificação de código para conservar memória, um artifício considerado "sujo"porém eficiente, dependendo do programador, mas que ofusca muito o código para leitores terceiros;

- Pseudo-comandos para a modularização do código final e reserva de espaço: Apesar de não funcionais (não possuem uma função no código em sí), a MVN implementa cinco pseudo símbolos que auxiliam na modularização, ao possibilitar a separação do código em arquivos com a idéia de importação e exportação de símbolos e endereços relativos para que não seja necessário calcular o endereço de cada instrução na memória, e o gerenciamento de memória, como a declaração de constantes, valores que não devem mudar durante a execução do programa, e blocos de memória, que reservam um grande bloco de bytes de uma vez;

Capítulo 6

Tradução de comandos

6.1 Controle de Fluxo

6.1.1 If-then

Linguagem de Entrada

```
if (<expressao>) do
      <comandos>
endif
```

Linguagem de Saída

```
if_1  <expressao>
            JN          endif_1
      JP      do_if_1
do_if_1 <comandos>
endif_1     <comandos-2>
```

6.1.2 If-do-elsif-do-else

Linguagem de Entrada

```
if (<expressao-1>) do
      <comandos-1>
elsif (<expressao-2>) do
      <comandos-2>
else do
      <comandos-3>
endif
```

Linguagem de Saída

```
if_1          <expressao-1>
              JN       elsif_1_1
              JP       do_if_1
do_if_1       <comandos-1>
              JP       endif_1
elsif_1_1     <expressao-2>
              JN       else_1
              JP       do_elsif_1_1
do_elsif_1_1  <comandos-2>
              JP       endif_1
else_1              <comandos-3>
              JP       endif_1
endif_1       <comandos-4>
```

6.1.3 While

Linguagem de Entrada

```
while (<expressao>) do
      <comandos>
endwhile
```

Linguagem de saída

```
while_1       <expressao>
                    JN          endwhile_1
                    JP          do_while_1
do_while_1    <comandos>
              JP          while_1
endwhile_1    <comandos-2>
```

6.2 Comandos Imperativos

6.2.1 Atribuição de valor

Linguagem de Entrada

```
a = <expressao>;
```

Linguagem de saída

```
<expressao>
MM a;
```

6.2.2 Leitura (entrada)

Para fazer a entrada de dados são feitas várias leituras de words interpretadas de dois caracteres ASCII, mesmo para números.

Linguagem de Entrada

```
int a;
scan(a);
```

Linguagem de saída

```
READ_BUFFER_LIMIT K           /0
READ_COUNT        K           /0
READ_DATA         K           /0
READ_BASE         K           /0

READ              K           /0
                  MM          READ_BUFFER_LIMIT
                  LV          /0
                  MM          READ_COUNT
                  MM          READ_DATA


READ_DO           GD          /0000
                  MM          READ_DATA
                  -           ASCII_BR
                  JZ          READ_END
                  LD          READ_DATA
                  SC          BYTIFY
                  LD          BYTIFY_SECOND
                  -           ASCII_BELL
                  JZ          READ_FIRST
                  JP          READ_ALL

READ_FIRST        LD          READ_BUFFER_LIMIT
                  MM          WRITE_BASE
                  LD          BYTIFY_FIRST
                  MM          WRITE_DATA
                  LV          /2
                  *           READ_COUNT
                  SC          WRITE
                  LV          /1
                  +           READ_COUNT
```

```
                  MM          READ_COUNT
                  JP          READ_TEST_LIMIT

READ_ALL          LD          READ_BASE
                  MM          WRITE_BASE
                  LD          READ_DATA
                  MM          WRITE_DATA
                  LV          /2
                  *           READ_COUNT
                  SC          WRITE
                  LV          /1
                  +           READ_COUNT
                  MM          READ_COUNT
                  JP          READ_TEST_LIMIT

READ_TEST_LIMIT   LD          READ_COUNT
                  -           READ_BUFFER_LIMIT
                  JZ          READ_END
                  JP          READ_DO

READ_END          RS          READ
```

6.2.3 Impressão (saída)

A impressão tem o trabalho de formatar a informação e apresentá-lo na tela. Como na linguagem há strings de caractéres e numeros, foi concluído que será necessário pelo menos dois tipos de formatação: em ASCII e em uma base qualquer. No caso, serão utilizadas as formatações ASCII e em base 16 para os números com a adição do sinal de negativo quando o número for menor que zero.

Linguagem de Entrada

```
print(a);
```

Linguagem de saída

```
PRINT_BASE              K     /0
PRINT_TYPE              K     /0

PRINT_COUNT             k     /0
PRINT_SIZE              k     /0
PRINT_DATA              k     /0
PRINT_FIRST             k     /0
PRINT_SECOND            k     /0
PRINT_THIRD             k     /0
PRINT_FOURTH            k     /0
```

```
PRINT_OFFSET            k     /0

PRINT                   K     /0

                        MM    PRINT_SIZE

                        LV    /0
                        MM    PRINT_COUNT
                        MM    PRINT_OFFSET


PRINT_BEGIN             LD    PRINT_BASE
                        +     PRINT_OFFSET
                        +     LOAD_PREFIX
                        MM    PRINT_LOAD_DATA
PRINT_LOAD_DATA         K     /0
                        MM    PRINT_DATA

                        LD    PRINT_TYPE

                        JZ    PRINT_AS_IS
                        JP    PRINT_AS_ASCII


PRINT_AS_ASCII          LV    /1
                        +     PRINT_COUNT
                        MM    PRINT_COUNT
                        -     PRINT_SIZE
                        JZ    PRINT_AS_ASCII_ODD

                        LV    /1
                        +     PRINT_COUNT
                        MM    PRINT_COUNT
                        JP    PRINT_AS_ASCII_EVEN

PRINT_AS_ASCII_ODD      LD    PRINT_DATA                ;
   SEPARA OS BYTES
                        /     SHIFT2
                        *     SHIFT2
                        PD    /0100
                        JP    PRINT_END

PRINT_AS_ASCII_EVEN     LD    PRINT_DATA
                        PD    /0100

                        LV    /2
                        +     PRINT_OFFSET
                        MM    PRINT_OFFSET
```

```
                        LD    PRINT_COUNT
                        -     PRINT_SIZE
                        JZ    PRINT_END
                        JP    PRINT_BEGIN


PRINT_AS_IS             LD    PRINT_DATA
                        JN    PRINT_NEG
                        JP    PRINT_POS
PRINT_NEG               LD    ASCII_MINUS
                        PD    /0100
                        LD    PRINT_DATA
                        *     FFFF
                        MM    PRINT_DATA

PRINT_POS               /     SHIFT3            ;Separa todos
    os numeros
                        *     SHIFT3
                        MM    PRINT_FIRST

                        LD    PRINT_DATA
                        -     PRINT_FIRST
                        /     SHIFT2
                        *     SHIFT2
                        MM    PRINT_SECOND

                        LD    PRINT_DATA
                        -     PRINT_FIRST
                        -     PRINT_SECOND
                        /     SHIFT1
                        *     SHIFT1
                        MM    PRINT_THIRD

                        LD    PRINT_DATA
                        -     PRINT_FIRST
                        -     PRINT_SECOND
                        -     PRINT_THIRD
                        MM    PRINT_FOURTH          ;
                           separado (0x1234 viraria: 0x1000,
                           0x0200, 0x0030 e 0x0004)

                        LD    PRINT_FIRST
                        /     SHIFT3
                        MM    PRINT_FIRST
                        -     A
                        JN    PRINT_SUM_NUMBER_1
                        JP    PRINT_SUM_LETTER_1
PRINT_SUM_NUMBER_1      LD    PRINT_FIRST
                        +     ASCII_0
```

```
                        MM    PRINT_FIRST
                        JP    PRINT_AS_IS_1
PRINT_SUM_LETTER_1      LD    PRINT_FIRST
                        +     HEX_LETTER_TO_ASCII
                        MM    PRINT_FIRST
PRINT_AS_IS_1           LD    PRINT_SECOND
                        /     SHIFT2
                        MM    PRINT_SECOND
                        -     A
                        JN    PRINT_SUM_NUMBER_2
                        JP    PRINT_SUM_LETTER_2
PRINT_SUM_NUMBER_2      LD    PRINT_SECOND
                        +     ASCII_0
                        MM    PRINT_SECOND
                        JP    PRINT_AS_IS_2
PRINT_SUM_LETTER_2      LD    PRINT_SECOND
                        +     HEX_LETTER_TO_ASCII
                        MM    PRINT_SECOND
PRINT_AS_IS_2           LD    PRINT_FIRST
                        *     SHIFT2
                        +     PRINT_SECOND
                        PD    /0100

                        LD    PRINT_THIRD
                        /     SHIFT1
                        MM    PRINT_THIRD
                        -     A
                        JN    PRINT_SUM_NUMBER_3
                        JP    PRINT_SUM_LETTER_3
PRINT_SUM_NUMBER_3      LD    PRINT_THIRD
                        +     ASCII_0
                        MM    PRINT_THIRD
                        JP    PRINT_AS_IS_3
PRINT_SUM_LETTER_3      LD    PRINT_THIRD
                        +     HEX_LETTER_TO_ASCII
                        MM    PRINT_THIRD
PRINT_AS_IS_3           LD    PRINT_FOURTH
                        -     A
                        JN    PRINT_SUM_NUMBER_4
                        JP    PRINT_SUM_LETTER_4
PRINT_SUM_NUMBER_4      LD    PRINT_FOURTH
                        +     ASCII_0
                        MM    PRINT_FOURTH
                        JP    PRINT_AS_IS_4
PRINT_SUM_LETTER_4      LD    PRINT_FOURTH
                        +     HEX_LETTER_TO_ASCII
                        MM    PRINT_FOURTH
PRINT_AS_IS_4           LD    PRINT_THIRD
                        *     SHIFT2
```

```
                        +       PRINT_FOURTH
                        PD      /0100

                        LV      /1
                        +       PRINT_COUNT
                        MM      PRINT_COUNT
                        -       PRINT_SIZE
                        JZ      PRINT_END

                        LV      /2
                        +       PRINT_OFFSET
                        MM      PRINT_OFFSET

                        JP      PRINT_BEGIN


PRINT_END               RS      PRINT
```

6.2.4 Chamada de subrotina

A chamada de subrotina inclui os métodos do tratamento dos registros de ativação. Para cada registro de ativação, em ordem decrescente de endereços, temos: parâmetros da função chamada, endereço do último frame pointer (base pointer do registro de ativação anterior), endereço de retorno da função, resultado da função e variáveis locais e temporárias. O frame pointer sempre aponta para a posição do endereço de retorno da função no registro de ativação.

Linguagem de Entrada

```
begin do
      int param_1;
      int param_2;
      param_1 = 256;
      param_1 = 768;
      sub(param_1, param_2);
end
```

Linguagem de Saída

```
INIT        JP    MAIN
TWO         K     /0002
ZERO        K     /0000
LD_INSTR    LD    /0000       ; para fabricar instrucao de LD
MM_INSTR    MM    /0000       ; para fabricar instrucao de MM

PARAM_1     K     /0300       ; parametro 1 da funcao
PARAM_2     K     /0100       ; parametro 2 da funcao
```

```
FP          K     /0400       ; frame pointer
SP          K     /0400       ; stack pointer

; subrotina SUB --------------------------------------
F_PARAM_1   K     /0000
F_PARAM_2   K     /0000
F_RESULT    K     /0000
; corpo da subrotina
SUB         K     /0000
            LD    SUB    ; endereco de retorno
            MM    PUSH_P
            SC    PUSH_RA     ; push ADDR na pilha de ra

            LD    SP
            MM    FP          ; fp aponta para o ADDR na
               pilha de ra

            LV    =6
            MM    GET_P
            SC    GET_RA      ; carrega param 1 partir de
               endereco relativo no ra
            MM    F_PARAM_1

            LV    =4
            MM    GET_P
            SC    GET_RA      ; carrega param 2 a partir de
               endereco relativo no ra
            MM    F_PARAM_2

            LD    F_PARAM_1   ; carrega o primeiro param
            -     F_PARAM_2   ;
            MM    RTRN_RSLT   ; subtrai e guarda para return

            JP    RETURN

; fim SUB ----------------------------------------------

; RETURN ---------------------------------------
RTRN_RSLT   K     /0000
RTRN_ADDR   K     /0000
; corpo da subrotina
RETURN      LD    RTRN_RSLT
            MM    PUSH_P
            SC    PUSH_RA     ; push RESULT na pilha de ra

            LV    =0
            MM    GET_P
            SC    GET_RA
```

```
            MM    RTRN_ADDR   ; endereco de retorno
               recuperado da pilha

            LD    TWO
            MM    GET_P
            SC    GET_RA
            MM    SP          ; dropa a ra atual
            MM    FP          ; volta a fp antiga

            RS    RTRN_ADDR   ; retorna para o escopo
               anterior
; fim RETURN ----------------------------------------

; subrotina PUSH_RA ----------------------------------
PUSH_P      K     /0000
; corpo da subrotina
PUSH_RA     K     /0000
            LD    MM_INSTR    ; composicao do comando de
               carregar na pilha
            +     SP
            -     TWO
            MM    _NEW_INSTR1
            LD    PUSH_P
_NEW_INSTR1 K     /0000       ; carrega na pilha de fato
            LD    SP
            -     TWO
            MM    SP          ; sp, aponte para o vazio
            RS    PUSH_RA
; fim PUSH_RA ----------------------------------------

; subrotina GET_RA -----------------------------------
GET_P       K     /0000       ; endereco relativo na ra
; corpo da subrotina
GET_RA      K     /0000
            LD    LD_INSTR
            +     FP          ;
            +     GET_P       ;
            MM    _NEW_INSTR2 ;
_NEW_INSTR2 K     /0000       ; carrega valor da ra no
   acumulador
            RS    GET_RA
; fim GET_RA -----------------------------------------

; corpo do main --------------------------------------
MAIN        LD    PARAM_1
            MM    PUSH_P
            SC    PUSH_RA     ; push PARAM_1 na pilha de ra

            LD    PARAM_2
```

```
            MM    PUSH_P
            SC    PUSH_RA     ; push PARAM_2 na pilha de ra

            LD    FP
            MM    PUSH_P
            SC    PUSH_RA     ; push old FP na pilha de ra

            SC    SUB         ; chama SUB
;fim main ------------------------------------------------
```

6.3 Estruturas de Dados

6.3.1 Struct

A struct, sendo uma variável que possui outras variáveis em seu interior, será representado na mvn por espaços de memória concatenados, sendo que em seu cabeçalho haverá uma informação de quantos campos ele possui.

Linguagem de Entrada

```
struct tipoEsquisito do
    int structInt;
    bool structBool;
    string structString;
endstruct
```

Linguagem de saída

```
var_tipoEsquisito                          K     /3
var_tipoEsquisito.structInt   $     =2
var_tipoEsquisito.structBool  $     =2
var_tipoEsquisito.structBool  $     =100
```

6.3.2 Array

Será criado um bloco de memória com o número necessário de bytes mais dois, sendo que a primeira word irá conter quantas "casas"a array possui.

Linguagem de Entrada

```
int intArray[10];
```

Linguagem de saída

```
intArray          $           =22
```

6.4 Exemplo de programa traduzido

Linguagem de Entrada

```
begin do
      int fat;
      int a;
      scan(a);
      read(a);

      if(a < 0) do
            fat = 0;
      else do
            fat = 1;
            while(num > 1) do
                  fat = fat * a;
                  a = a - 1;
            endwhile
      endif

      print(a);
end
```

Linguagem de saída

```
INIT              JP          MAIN

A                 K           /0000
FAT               K           /0000
MAIN              SC          READ
                  SC          PRINT
if_1              LD          A
                  JN          else_1
                  JP          endif_1
do_if_1           LV          =0
                  MM          FAT
                  JP          endif_1
else_1            LV          =1
                  MM          FAT
while_1           LD          A
                  JN          endwhile_1
                  JP          do_while_1
do_while_1        LD          FAT
                  *           A
                  MM          FAT
                  LD          A
                  -           ONE
```

```
                    MM          A
                    JP          while_1
endwhile_1          JP          endif_1
endif_1             LD          FAT
                    SC          PRINT
END                 HM          INIT

# INIT
```

Capítulo 7

Ações semânticas

Para enfim transformar o código da linguagem em um código legível para a MVN, foram injetadas funções semânticas no autômato de pilha construído no reconhecedor sintático. Também foram desenvolvidas funções auxiliares para a geração de código. As funções auxiliares incluem:

- get_x_label: geração de rótulos para a MVN, como rótulo de "if", "end_while"e outros que foram previstos no capítulo anterior.
- resolve_x: geração de código para os comandos básicos de "while"e "if". Essas utilizam as funções a cima.
- expression_print: geração de código para expressões. utiliza o algoritmo Shunting yard adaptado para gerar a expressão adequada. Usa uma abordagem ingênua, utilizando variáveis temporárias para guardar constantes (números) e resultados intermediários presentes na expressão

Para inserir as ações no autômato de pilha foi construída uma matriz de três dimensões (máquina atual x estado atual x token lido) que indica para cada transição ou saída de máquina qual a ação (função) deve ser tomada.

7.1 Geração de código

Para a fase de geração de código, foram implementadas para comandos while, print, scan e expressoes

Porém, como não foi possível implementar comandos de decisão à tempo, a capacidade da linguagem ficou muito limitada.

7.1.1 Comandos

O problema da implementação dos comandos de decisão estava na possibilidade de existir múltiplos caminhos ("elsif"s e "else") que dificultava a escrita do código final, pois para escrever o código era necessário saber se existiam tais condições para escrever os rótulos de jump nas confições desses. A estratégia adotada foi o uso de uma pilha para descobrir se existiam tais caminhos alternativos e ao ler a palavra reservada "endif"começar a desempilhar os comandos da pilha até um "if"escrevendo o código de MVN. E para a implementação de expressões e condições foram utilizadas duas pilhas para operandos e operadores utilizando o algoritmo a cima comentado.

7.1.2 Variáveis

As variáveis foram guardadas em uma estrutura chamada "Scope"que possuía uma tabela (no caso, uma lista de strings), uma lista de escopos (Scope) filhos e um ponteiro ao pai para possibilitar a procura de variáveis já definidas.

Para a geração de variáveis foi adicionada na tabela de símbolos uma função para imprimir em um arquivo todas as variáveis que esta possui. Essa função é então chamada no fim do programa possibilitando separar uma área de programa e dados. Porém, como só foram implementadas variáveis inteiras, todas as variáveis foram consideradas como tal e foram reservadas uma word para cada variável.

7.1.3 Saída

A saída resultante do programa exemplo dado foi:

```
begin   OS    =0
    JN    elsif_1_1
    JP    do_if_1
do_if_1   OS    =0
    JP    endif_1
    JN    elsif_1_2
    OS    =0
    JN    empty
    JP    do_elsif_1_1
    JP    do_elsif_1_1
do_elsif_1_1    OS    =0
    OS    =0
    JP    endif_1
    JN    else_1
    OS    =0
    JN    empty
    JP    do_elsif_1_2
```

```
    JP    do_elsif_1_2
do_elsif_1_2    OS    =0
    OS    =0
    JP    endif_1
    OS    =0
endif_1   OS    =0
    LV    /0300
    MM    _TEMP_1
    LV    =4
    SC    READ    ; scan()
    LV    /FFFF
    MM    _TEMP_7
    LV    /0300
    MM    _TEMP_1
    LV    =4
    SC    PRINT   ; print()
    JN    else_21_1
    JP    do_if_2
do_if_2   OS    =0
    JP    endif_2
    OS    =0
while_1   OS    =0
    JN    endwhile_1
    JP    do_while_1
do_while_1    OS    =0
endwhile_1    OS    =0
endif_2   OS    =0
end   OS    =0
```

Referências Bibliográficas

[1] R. Ricardo S. Jaime A. Reginaldo, B. Anarosa. Introdução máquina de von neumann.

[2] João José Neto. *Introdução à Compilação*. Escola Politécnica da USP, 1 edition, 1986.