Alan D. Barroso, Kenji Sakata Jr

Relatório Final

Universidade de São Paulo
Escola Politécnica

Linguagens e Compiladores
PCS2056

São Paulo
2013

1 Descrição informal da linguagem

A linguagem de alto nível criada para a construção deste compilador foi baseada nas linguagens de programção imperativas C e Pascal. Nesta seção iremos explicar informalmente as principais estruturas sintáticas reconhecidas pelo compilador.

A estrutura básica de um programa em nossa linguagem consiste de seis partes principais:

- Declarações de constantes
- Declarações de tipos
- Declarações de estruturas
- Declarações de variáveis globais
- Declarações de funções
- Programa principal

Cada uma dessas partes, com excessão do programa principal, não são obrigatórias e podem ser omitidas. No entanto, elas devem seguir a exata ordem indicada acima.

1.1 Declaração de constantes

As constantes do programa são declaradas da seguinte maneira:

```
1 const nome_da_variavel = valor_da_constante;
```

A constante poder ser um número (inteiro ou ponto flutuante), true, false ou um caracter envolto em apóstrofes.

1.2 Declarações de tipos

Os tipos definidos pelo usuário são definidos da seguinte maneira:

```
1 typedef tipo novo_nome;
```

Os novos tipos só podem ser definidos em cima dos tipos básicos da linguagem (bool, int, float, char) ou tipos já definidos acima dele.

1.3 Declaração de estruturas

As estruturas são os agregados heterogêneos da linguagem. Nessa linguagem, uma vez declarada uma estrutura, ela é considerada como um tipo a ser usado no resto do programa. Elas são declaradas da seguinte maneira:

```
1 struct nome_da_estrutura begin
2   declaracao_variavel1;
3   declaracao_variavel2;
4   ...
5   declaracao_variavelN;
6 end struct
```

Uma estrutura deve conter pelo menos uma variável. As declarações de variáveis seguem as mesmas regras que as declarações de variáveis globais.

1.4 Declarações de variáveis globais

As declarações de variáveis podem ser de dois tipos, variáveis simples ou agregados homogêneos. As variáveis simples seguem a seguinte estrutura:

```
1   tipo indentificador_variavel;
```

Já os agregados homogêneos:

```
1 tipo indentificador_variavel[inteiro];
```

É possível adicionar quantas vezes forem necessárias o párticula [inteiro], criando assim matrizes e não somente vetores.

Os tipos das variáveis podem ser: os tipos básicos da linguagem, os tipos definidos pelo programador ou as estruturas definidas anteriormente. As variáveis definidas nesse trecho são compartilhadas com todo resto do programa.

1.5 Definições de funções

As funções nessa linguagem são definidas como abaixo:

```
function tipo nome_funcao (tipo param1, tipo byref param2,
    ... tipo paramN) begin
  %
    Bloco interno
  %
end function
```

Os tipos das funções podem ser do mesmo tipo que os das variáveis além de um tipo a mais, o tipo void. Funções que possuem um tipo diferente do tipo void devem obrigatoriamente possuir um retorno. Funções do tipo void também podem possuir retorno, mas esse retorno deve ser obrigatóriamente vazio.

A estrutura interna de uma função segue a mesma estrutura que o programa principal, portanto, seus detalhes serão explicados a seguir.

1.6 Programa Principal

Primeiramente, o compilador aceita programas compostos por uma sequência de declarações de variáveis e, em seguida, uma sequência de comandos. O escopo do programa inicia-se com a sequência das palavras-chaves program seguida de begin. O programa acaba com as palavras end program.Tanto para as declarações de variáveis quanto para os comandos, o separador é o ponto e vírgula.

```
program begin
  %
    Espaco destinado a declaracao de variaveis.
  %
  declaracao_var1; declaracao_var2; ... declaracao_varN;

  %
    Espaco destinado a declaracao de comandos.
  %
  comando1; comando2; ... comandoN;
end program
```

A declaração de variáveis segue o mesmo estilo que o das variáveis globais.

Os comandos podem ser divididos em seis tipos:

- Comando de atribuição
- Comando de chamada de função

- Comando condicional
- Comando iterativo
- Comando de entrada
- Comando de saída
- Comando de retorno

1.6.1 Comando de Atribuição

O comando de atribuição associa o valor de uma expressaão a uma variável, explicitada à esquerda do comando. Note que a variável pode ser uma variável escalar, um ponteiro ou um vetor.

A avaliação das expressões segue as convenções usuais, sendo efetuada da esquerda para a direita. As expressões podem ser tanto booleanas quanto aritiméticas.

No caso das expressões aritiméticas, por definição, as potenciações possuem precedência sobre os produtos e divisões, e estes precedência sobre as somas e subtrações. Note que é possível alterar a prioridade de tais precedências graças ao uso de parênteses. As expressões aritiméticas aceitam números, identificadores de variáveis e o valor de retorno de chamadas de função.

Analogamente, no caso das expressões booleanas, a operação lógica "e"tem prioridade sobre a operação lógica "ou". As expressões booleanas aceitam tanto booleanos puros (true ou false) quanto o resultado de comparações entre expressões aritiméticas.

1.6.2 Comando de chamada de função

Funções podem ser consideradas como sub-programas, que recebem um conjunto de parâmetros e que são chamados pelo programa principal para executar uma dada ação. Há dois tipos de função implementadas via mesma estrutura sintática: rotinas e funções. Rotinas têm tipo void e executam seus comandos sem a necessidade de retornar algo no fim de sua execução. Já as funções são tipadas, como por exemplo int ou char, portanto, necessitam de pelo menos um return dentro de seu bloco principal.

Tanto as funções quanto as rotinas são chamadas através do nome da função requisitada, seguido dos parâmetros que devem ser passados para sua execução entre parênteses.

```
nome_funcao(parametro1 , parametro2, ..., parametroN);
```

1.6.3 Comando condicional

Refere-se à possibilidade de realizar um salto condicional segundo o resultado de uma expressão booleana.

Além das operações lógicas de "e"e "ou", as expressões booleanas consideram a possibilidade de realizar comparações lógicas entre partículas comparativas, através dos operadores "=="(igual a) ou "!="(diferente de). Tais partículas são ou booleanos ou o resultado de uma comparação entre expressões aritiméticas, efetuadas através dos operadores ">"(maior que), "<"(menor que), ">="(maior ou igual a) e "<="(menor ou igual a).

Há duas estruturas sintáticas possíveis para o comando condicional: a simples, na qual somente são executados os comandos referente à expressão booleana após a palavra reservada ïffor verdadeira, e a composta. Neste último caso, caso a comparação seja verdadeira, o comando que se encontra entre as palavras ẗhenë ëlseëerá executado. Caso contrário, o comando após o ëlseëerá executado.

```
if expressao_booleana then
  %
    Bloco interno
  %
else
  %
    Bloco interno
  %
end if
```

O bloco interno dos comando condicionais são equivalentes são aos blocos internos de funções e do programa principal.

1.6.4 Comando Iterativo

Este comando testa a expressão booleana após a palavra reservada ẅhileẗpara decidir se irá realizar os comandos que segue a palavra “do”. Esta ação solicitada será executada repetidamente até que a condição de teste não mais seja atendida.

```
while expressao_booleana do
  %
    Bloco interno
  %
```

```
end while
```

1.6.5 Comandos de Entrada e Saída

Os comandos de entrada e saída promovem, respectivamente, a entrada e saída de dados com relação a um meio externo. O comando de leitura captura dados e preenche o valor de uma variável, especificada após a palavra s̈can:̈ Já o comando de impressão permite a impressão do resultado de uma expressão.

```
scan variavel;

print expressao;
```

1.7 Exemplo de programa escrito na linguagem definida

```
function int fatorial_recursivo(int n) begin
  int result;

  if (n <=1)  then
    result = 1;
  else
    result = n * fatorial_recursivo(n - 1);
  end if

  return result;
end funcion

function int fatorial_iterativo(int n) begin
  int fatorial;

  fatorial = 1;
  while (n>0) do
      fatorial = fatorial * n;
      n = n - 1;
  end while

  return fatorial;
end function

program begin
  int  fatorial_10_recursivo;

  fatorial_10_recursivo = fatorial_recursivo(10);

```

```
30   print fatorial_10_recursivo;
31   print fatorial_iterativo(10);
32 end program
```

../1–linguagem/notacoes/program_example

1.8 Descrição da linguagem em BNF

```
1  <programa> ::= <declaracoes_constante> <definicoes_tipo> <
       definicoes_struct> <declaracoes_var> <declaracoes_func>
       program begin <declaracoes_var> <comandos> end program
2
3  <declaracoes_constante> ::= <declaracao_constante>
4              | <declaracao_constante> <declaracoes_constante>
5
6  <declaracao_constante> ::= const <identificador> = <constante
       > ;
7              | epsilon
8
9  <identificador> ::= <letra> | <letra><letra_dig>
10
11 <constante> ::= <numero>
12       | <booleano>
13       | <char>
14
15 <numero> ::= <inteiro>
16       | <inteiro>.<inteiro>
17       | .<inteiro>
18
19 <booleano> ::= true | false
20
21 <char> ::= '<letra_dig>' | '\<letra_dig>'
22
23 <definicoes_tipo> ::= <definicao_tipo>
24           | <definicao_tipo> <definicoes_tipo>
25
26 <definicao_tipo> ::= typedef <identificador> <tipo> ;
27           | epsilon
28
29 <tipo> ::= bool | int | char | float | <identificador>
30
31 <definicoes_struct> ::= <definicao_struct>
32           | <definicao_struct> <definicoes_struct>
33
34 <definicao_struct> ::= struct <identificador> begin <
       declaracao_var> <declaracoes_var> end struct
35           | epsilon
36
37 <declaracoes_var> ::= <declaracao_var>
38           | <declaracao_var> <declaracoes_var>
```

```
39
40 <declaracao_var> ::= <tipo> <vars> ;
41             | epsilon
42
43 <vars> ::= <declaracao_array>
44     | <declaracao_array>, <vars>
45
46 <declaracao_array> ::= <identificador>
47             | <declaracao_array>[<inteiro>]
48
49 <inteiro> ::= <digito>
50       | <digito><inteiro>
51
52 <declaracoes_func> ::= <declaracao_func>
53             | <declaracao_func> <declaracoes_func>
54
55 <declaracao_func> ::= function <tipo_funcao> <identificador>
       (<declaracoes_parametro>) begin <declaracoes_var> <
       comandos> end function
56
57 <tipo_funcao> ::= void | <tipo>
58
59 <declaracoes_parametro> ::= <declaracao_parametro>
60             | epsilon
61
62 <declaracao_parametro> ::= <tipo> <declaracao_array>
63             | <tipo> byref <declaracao_array>
64             | <tipo> <declaracao_array>, <
       declaracao_parametro>
65             | <tipo> byref <declaracao_array>, <
       declaracao_parametro>
66
67 <comandos> ::= <comando> <comandos>
68       | epsilon
69
70 <comando> ::= <comando_atribuicao>
71       | <comando_condicional>
72       | <comando_iterativo>
73       | <comando_entrada>
74       | <comando_saida>
75       | <comando_retorno>
76
77 <comando_atribuicao> ::= <var> <operador_atribuicao> <ou>;
78             | <expressao>;
79
80 <operador_atribuicao> ::= =
81             | +=
82             | -=
83             | *=
84             | /=
85
```

```
<var> ::= <array>
    | <array>.<var>

<array> ::= <identificador>
    | <identificador>[<expressao>]

<expressao> ::= <ou>
      | epsilon

<ou> ::= <e>
    | <e> or <ou>

<e> ::= <comparacao>
  | <e> and <comparacao>

<comparacao_booleana> ::= <soma>
            | <soma> <operador_comparacao> <soma>

<operador_comparacao> ::= ==
            | !=
            | <
            | >
            | <=
            | >=

<soma> ::= <multiplicacao>
    | <multiplicacao> + <soma>
    | <multiplicacao> - <soma>

<multiplicacao> ::= <potenciacao>
        | <potenciacao> * <multiplicacao>
        | <potenciacao> / <multiplicacao>

<potenciacao> ::= <operacao_unitaria>
        | <operacao_unitaria> ^ <potenciacao>

<operacao_unitaria> ::= <particula>
          | - <particula>
          | ! <particula>

<particula> ::= (expressao_aritmerica)
      | <constante>
      | <var>
      | <chamada_funcao>

<chamada_funcao> ::= <identificador>(<parametros>)

<parametros> ::= <parametro>
        | epsilon

<parametro> ::= <expressao>
```

```
137         | <expressao>, <parametro>
138
139 <comando_condicional> ::= if(<expressao>) then <
    declaracoes_var> <comandos> end if
140             | if(<expressao>) then <declaracoes_var> <
    comandos> else <comandos> end if
141
142 <comando_iterativo> ::= while(<expressao>) do <comandos> end
    while
143
144 <comando_entrada> ::= scan <lista_enderecos>;
145
146 <lista_enderecos> ::= <var>
147           | <var>, <lista_enderecos>
148
149 <comando_saida> ::= print <parametro>;
150
151
152 <comando_retorno> ::= return <expressao>;
153
154 <letra> ::= _ | a | b | c | d | e | f | g | h | i | j | k | l
      | m | n | o | p | q | r | s | t | u | v | w | x | y | z |
      A | B | C | D | E | F | G | H | I | J | K | L | M | N | O
      | P | Q | R | S | T | U | V | W | X | Y | Z
155
156 <digito> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
157
158 <letra_dig> ::= <letra>
159         | <digito>
160         | <letra><letra_dig>
161         | <dig><letra_dig>
```

../1–linguagem/notacoes/BNF.txt

1.9 Descrição da linguagem em Wirth

```
1 programa = [declaracao_constante] [definicao_tipo] {
    declaracao_struct} [declaracao_var] {declaracao_func} "
    program" "begin" [declaracao_var] {comando} "end" "program
    ".
2
3 declaracao_constante = "const" identificador "=" constante ";
    " {identificador "=" constante ";"} "end" "const".
4
5 identificador = letra {letra_dig}.
6
7 constante = numero | booleano | char.
8
9 numero = inteiro | ([inteiro] "." inteiro).
10
```

```
11 booleano = "true" | "false".
12
13 char = "'" (letra_dig | "\" letra_dig) "'".
14
15 definicao_tipo = "typedef" identificador "=" tipo ";" {
       identificador "=" tipo ";"} "end" "typedef".
16
17 tipo = "bool" | "int" | "char" | "float" | identificador.
18
19 declaracao_struct = "struct" identificador "begin" tipo vars
       ";" {tipo vars ";"} "end" "struct".
20
21 declaracao_var = "vars" tipo vars ";" {tipo vars ";"} "end" "
       vars".
22
23 vars = declaracao_array {"," declaracao_array}.
24
25 declaracao_array = identificador {"[" inteiro "]"}.
26
27 inteiro = digito {digito}.
28
29 declaracao_func = "function" tipo_funcao identificador "(" [
       declaracoes_parametro] ")" "begin" [declaracao_var] {
       comando} "end" "function".
30
31 tipo_funcao = "void" | tipo.
32
33 declaracoes_parametro = declaracao_parametro {","
       declaracao_parametro}.
34
35 declaracao_parametro = tipo ["byref"] declaracao_array.
36
37 comando = comando_atribuicao
38     | comando_chamada
39     | comando_condicional
40     | comando_iterativo
41     | comando_entrada
42     | comando_saida
43     | comando_retorno.
44
45 comando_atribuicao = [var operador_atribuicao expressao] ";".
46
47 operador_atribuicao = "=" | "+=" | "-=" | "*=" | "/=".
48
49 var = array {"." array}.
50
51 array = identificador {"[" expressao "]"}.
52
53 expressao = e {"or" e}.
54
55 e = comparacao {"and" comparacao}.
```

```
56
57 comparacao = soma {operador_comparacao soma}.
58
59 operador_comparacao = "==" | "!=" | "<" | ">" | "<=" | ">=".
60
61 soma = multiplicacao {("+" | "-") multiplicacao}.
62
63 multiplicacao = potenciacao {("*" | "/") potenciacao}.
64
65 potenciacao = operacao_unitaria {"^" operacao_unitaria}.
66
67 operacao_unitaria = ["!" | "-"] particula.
68
69 particula = ("(" expressao ")") | constante | var |
       chamada_funcao.
70
71 chamada_funcao = identificador "(" [parametros] ")".
72
73 parametros = expressao {"," expressao}.
74
75 comando_chamada = "call" chamada_funcao ";".
76
77 comando_condicional = "if" "(" [expressao] ")" "then" {
       comando} ["else" {comando}] "end" "if".
78
79 comando_iterativo = "while" "(" [expressao] ")" "do" {comando
       } "end" "while".
80
81 comando_entrada = "scan" var {"," var} ";".
82
83 comando_saida = "print" parametros ";".
84
85 comando_retorno = "return" [expressao] ";".
86
87 letra = "_" | "a" | "b" | "c" | "d" | "e" | "f" | "g" | "h" |
       "i" | "j" | "k" | "l" | "m" | "n" | "o" | "p" | "q" | "r"
       | "s" | "t" | "u" | "v" | "w" | "x" | "y" | "z" | "A" | "
      B" | "C" | "D" | "E" | "F" | "G" | "H" | "I" | "J" | "K" |
       "L" | "M" | "N" | "O" | "P" | "Q" | "R" | "S" | "T" | "U"
       | "V" | "W" | "X" | "Y" | "Z".
88
89 digito = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8"
       | "9".
90
91 letra_dig = (letra | digito) {letra_dig}.
```

../1–linguagem/notacoes/wirth.txt

2 Análise Léxica

2.1 Introdução

O primeiro módulo do compilador a ser desenvolvido foi o Analisador Léxico.

A sua principal função consiste em gerar os tokens que são recebidos pelo analisador sintático.

Cada token deve possuir uma classe - número, identificador, palavra reservada e símbolos especiais - e um valor.

O analisador léxico também é o responsável por criar a Tabela de Símbolos, a qual será utilizada pelo analisador sintático e o semântico. A tabela é preenchida graças a geração de tokens.

Além disso, ele deve permitir identificar a posição (Linha x Coluna) dos tokens na estrutura do texto. Note que comentários, espaços, <tab>s e <CR>s normalmente são ignorados pelo analisador léxico.

Como é chamado inúmeras vezes durante o processo de compilação: para cada átomo (token), ele se torna o gargalo do compilador em relação ao tempo de compilação e ao de execução.

Vantagens:

- Mantenibilidade do código é facilitada, caso o arquivo de saída intermediário seja padronizado. Assim, toda modificação sobre este módulo terá pouco ou nenhum impacto sobre os outros módulos do compilador, pois ele será visto como uma caixa preta.
- Ainda com essa visão de caixa preta, é possível utilizar o mesmo analisador léxico para diversos analisadores sintáticos que usassem os mesmos padrões de token, independentemente da sintaxe. Ele seria uma espécie de analisador léxico genérico reaproveitável para diferentes linguagens.

Desvantagens:

- Geração de um arquivo intermediário: saída do analisador léxico. No caso da sub-rotina essas informações são consumidas instantaneamentes pelo analisador sintático.

- Caso o padrão do arquivo de saída mude, não só será necessário mudar o analisador léxico, mas o sintático também. Por exemplo, se desejarmos reconhecer novos tipos de token.
- A performance é menor que o da solução com subrotina, pois tem o processamento a mais de criação do arquivo de saída e escrita sobre ele pelo módulo léxico e leitura completa do mesmo pelo sintático. No caso da sub-rotina, o token reconhecido é imediatamente lido pelo sintático.

2.2 Geração do Autômato

Começamos a geração do autômato, definindo as seguintes expressões regulares, as quais serão interpretadas como tokens pelo compilador.

- **Inteiro**: [0-9]+
- **Identificador**: ([a-zA-Z] | [_][a-zA-Z])[a-zA-Z0-9]*
- **Caractere**: ['][a-zA-Z]['] | ['][\][a-z][']
- **Cadeia de caractéres**: ["]([a-zA-Z] | [\]([a-z] | ["]))*["]
- **Operadores**: + - * / = < > += -= *= /= == <= >=
- **Delimitadores**: ( ) [ ]
- **Comentários**: % .* %
- **Caracteres ignorados**: \s \t \n

Em seguida, convertemos cada uma das expressões regulares em autômatos finitos equivalentes que reconheçam as correspondentes linguagens por elas definidas.

Figura 1 – Automato que reconhece um inteiro

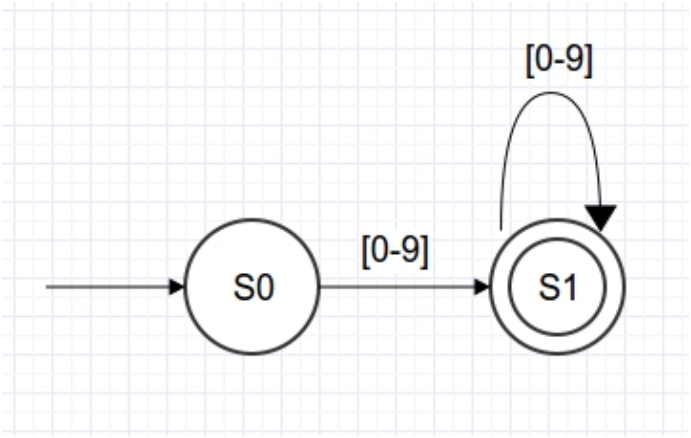


Figura 2 – Automato que reconhece um número real

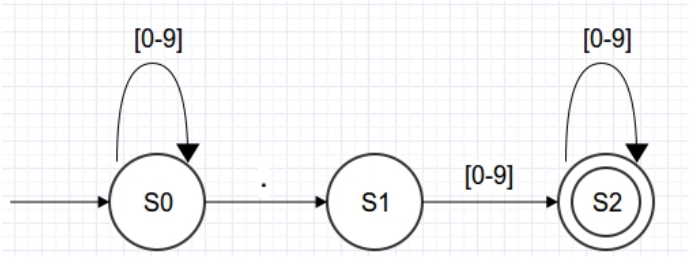


Figura 3 – Automato que reconhece um identificador

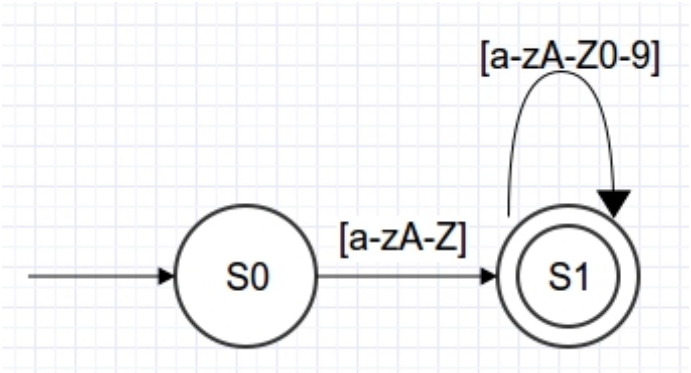


Figura 4 – Automato que reconhece um caracter

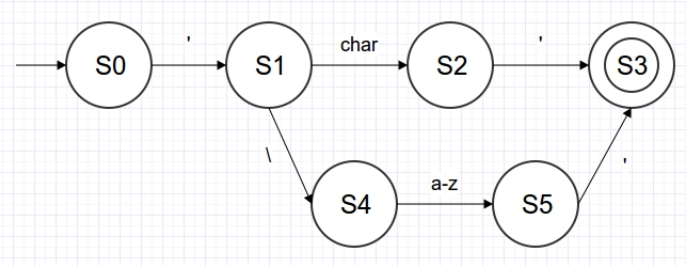


Figura 5 – Automato que reconhece uma cadeia de caractéres

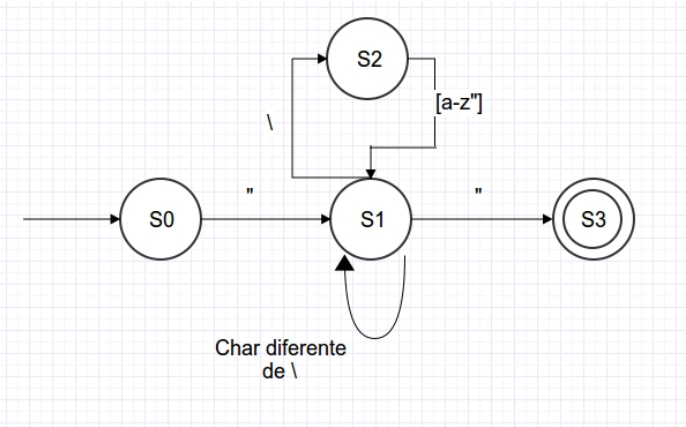


Figura 6 – Automato que reconhece operadores

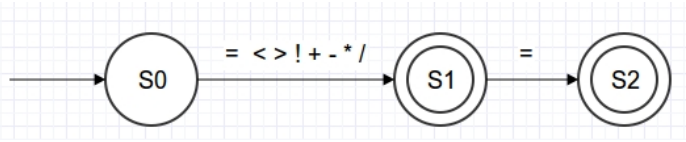


Figura 7 – Automato que reconhece delimitadores

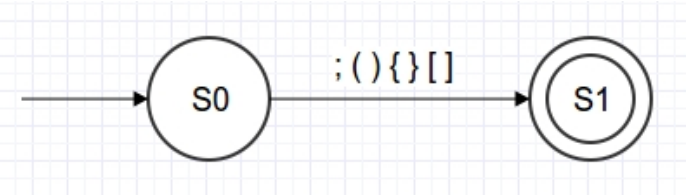
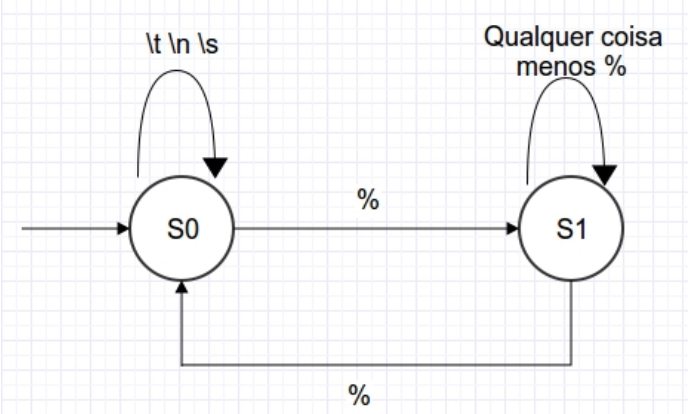
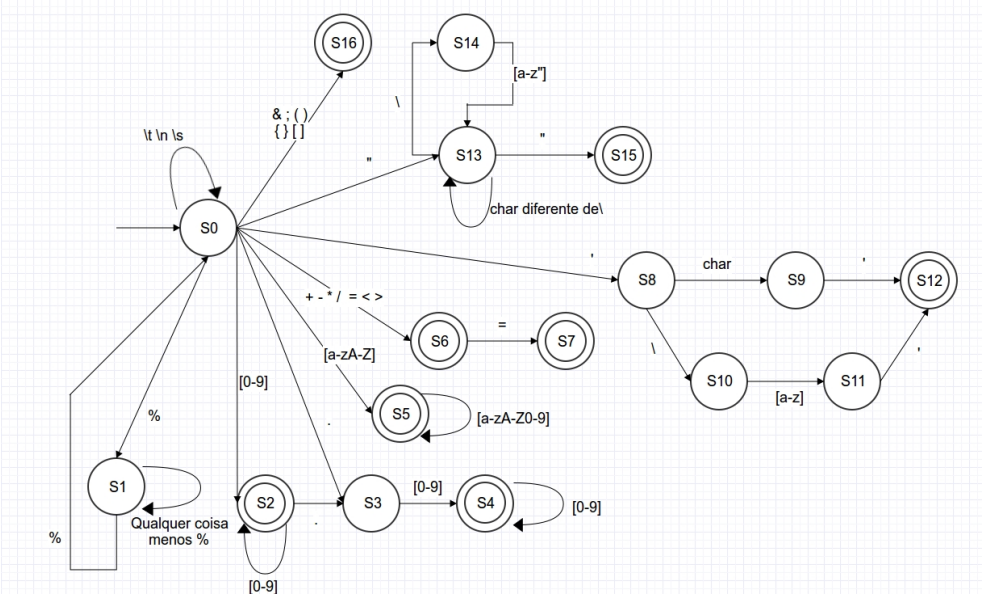


Figura 8 – Automato que reconhece comentários e caractéres ignorados



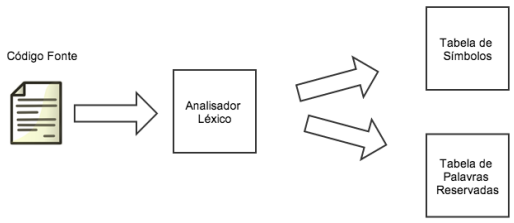
Finalmente, obtemos um autômato único, o qual aceita todas essas linguagens a partir de um mesmo estado inicial, mas que apresenta um estado final diferenciado para cada uma delas. Para obter o transdutor, basta acrescentar ϵ-transições partindo dos estados de aceitação, dando como saída o token, e tendo como destino o estado inicial da máquina S0.

Figura 9 – Automato completo



2.3 Funcionamento do Módulo de Análise Léxica

Figura 10 – Arquitetura do analisador léxico



Basicamente, o analisador léxico pode ser visto como um sistema composto por quatro módulos.

2.3.1 Tabela de palavras reservadas

Implementamos esse módulo como sendo o responsável pela leitura de um arquivo externo o qual contém as palavras reservadas consideradas pela linguagem do compilador e pela poulação de uma estrutura de dados (no nosso caso, uma lista ligada - /linked_list/linked_list.c) que será consultada pelo leitor de expressões regulares de forma a identificar as palavras reservadas.

A ideia de utilizar um arquivo externo é ter uma visão de evolução potencial da linguagem, caso novas palavras reservadas passem a fazer parte da linguagem ou caso antigas palavras sejam removidas/modificadas. Nesse arquivo há uma palavra reservada por linha. O arquivo atual de entrada se chama keywords e está localizado na pasta context_stack.

Este módulo foi implementado no arquivo keyword_analyser.c dentro da pasta context_stack.

2.3.2 Tabela de símbolos

O módulo da tabela de simbolos é composto do arquivo symbol_table.c dentro da pasta context_stack.

Ele também utiliza a estrura de dados (lista ligada), no qual cada nó contém uma estrura do tipo:

```
1 typedef struct{
2     char* id;
3     int value;
4     Type type;
5 } Symbol;
```

Uma das principais funcionalidades fornecidas por este módulo ao analisador léxico é a possibilidade de inserir novos símbolos à tabela. A inserção de identificadores é única na tabela. Ele também permite, em um segundo momento, a sua consulta pelo analisador sintático.

2.3.3 Tabela de transições

Consiste em um arquivo externo que modela uma máquina de estados, resultante do transdutor dos exercícios anteriores. O arquivo /lexical_analyser/lex_machine contém:

- Primeira linha: número n de estados total da máquina
- Segunda linha: número m de estados de aceitação
- Próximas m linhas: identificador do estado (int) seguido de seu nome (char[3]) - o mesmo nome será utilizado para identificação do token retornado pelo estado
- Próxima linha: número k de estados que serão ignorados pelo módulo (comentário, espaços em branco etc.)
- Próximas k linhas: identificadores dos estados ignorados
- Próximas n linhas: representação da tabela de transições. Cada linha representa um estado σ_n e cada coluna um caracter de entrada a. Os valores das células representam o $f(\sigma_n, a)$.

Nota: transições resultando em -1 indicam um estado de saída - a máquina pára, indicando ao leitor de expressões regulares que foi encontrado :

- uma das expressões regulares definidas pela gramática da linguagem
- um erro
- uma expressão ignorada (comentário, espaços em branco etc.)

Em seguida, a máquina realiza uma ϵ-transição para o estado inicial S0.

2.3.4 Leitor de expressões regulares

O módulo de leitura de expressões regulares é o módulo central que integra e controla o fluxo de dados dos outros três módulos periféricos. Seu modo de operação obdece à seguinte rotina:

- init_lex:
  1. inicializa a máquina de estados, a partir da Tabela de transições
  2. inicializa a Tabela de Palavras Reservadas
  3. inicializa em branco a Tabela de Símbolos
- get_token: lê o código fonte e usa a máquina de estados inicializada anteriormente para tentar encontrar uma das expressões regulares definidas pela gramática. Quando a máquina pára, é realizada a leitura do buffer para gerar um token (classe, valor e posição no arquivo fonte). Caso a máquina tenha parado no estado que indica a expressão regular de um identificador, fazemos uma consulta à Tabela de Palavras Reservadas. E se for o caso deste ser uma palavra reservada, mudamos sua classe de identificador para palavra reservada (KEY). Caso contrário, inserimos o identificador na Tabela de Símbolos e o consideramos como sendo da classe de variáreis (VAR).

O reconhecedor sintático chama repetidamente a sub-rotina get_token() do analisador léxico sobre um arquivo do tipo texto contendo o texto-fonte a ser analisado. O programa termina ao receber um token especial (EOA), o qual indica o fim do arquivo; ou ao receber um token de erro.

3 Reconhecedor Sintático

3.1 Introdução

O analisador sintático é o responsável por reconhecer a estrutura da linguagem, formada a partir da gramática. Sua ação começa uma vez que o módulo léxico já reconheceu todas as partículas átomicas que compõem o código-fonte.

Na realidade, é o analisador sintático quem invoca a ação do módulo léxico em um primeiro momento a fim de identificar as partículas do texto. A medida que os tokens são lidos e repassados ao analisador sintático, ele, o analisador sintático, tenta descrever como essas partículas estão estruturadas e organizadas no texto. A estrutura é chamada de árvore de sintaxe e consiste unicamente em uma forma alternativa de representar o código-fonte, mas a qual é compreensível pelo compilador.

Em seguida, quando da análise semântica, é novamente o módulo sintático quem será o responsável por invocar o analisador semântico, o qual traduzirá a árvore de sintaxe em ações concretas, executáveis pelo computador.

Outras funções tipicamente atribuídas ao analisador sintático são detecção de erros de sintaxe, recuperação de erros, correção de erros, ativação de rotinas de síntese do código-objeto, entre outras.

Existem outras formas de delegar essas tarefas aos três módulos canônicos de um compilador clássico, mas podemos concluir que, por conta de nossas decisões de projeto, o compilador em desenvolvido é orientado à sintaxe. Isso porque o módulo sintático do compilador é o pivô central que coordena o fluxo sequencial de ações que permitem traduzir uma linguagem de alto nível em linguagem de máquina. É ele quem determina qual é a etapa do processo de compilação que está sendo executada e quem é o responsável por ela.

Assim, após definir a linguagem e desenvolver o analisador léxico, tratamos de construir o módulo de análise sintática, cujo método de reconhecimento foi baseado em autômato de pilha estruturado (APE).

3.2 Geração Automática dos Autômatos

3.2.1 Notação de Wirth Reduzida

Na primeira etapa para a criação do APE, foram utilizadas as expressões fundamentais criadas a partir da descrição reduzida em notação de Wirth. A definição das expressões seguiu o critério dos não-terminais com recursividade central, sendo possível concluir que as máquinas finais tratariam a sequência de comandos, as expressões e o programa principal.

Assim, a linguagem representada pela notação de Wirth reduzida pode ser encontrada no arquivo "wirth_reduzido.txt"que acompanha esse documento.

3.2.2 Lista de submáquinas do APE

3.2.2.1 Lista de transições

A partir da linguagem representada em notação de Wirth reduzida, utilizamos o programa do site indicado [1] para gerar automaticamente a tabela de transições as quais as submáquinas deveriam executar de forma a caracterizar a linguagem.

O programa gera diversas tipos de saídas. Cuida lembrar que somente consideramos a saída que é uma tabela de transição reduzida, de forma a otimizar o processamento do compilador e facilitar a leitura e boa compreensão da representação dos autômatos, seja ela na forma tabular ou gráfica.

Grosso modo, o programa utiliza três etapas para reduzir a tabela de transições. A ordem em que elas devem ser executadas é a seguinte:

- Eliminação das transiçõees em vazio;
- Eliminação dos estados não-acessíveis;
- Eliminação dos estados equivalentes.

A saída do tabela referente à submáquina programa:

```
initial: 0
final: 29
(0, "const") -> 1
(0, "typedef") -> 2
(0, "struct") -> 3
(0, "vars") -> 4
```

[1] http://radiant-fire-72.heroku.com/

```
(0, "function") -> 5
(0, "program") -> 6
(1, identificador) -> 7
(2, identificador) -> 23
(3, identificador) -> 30
(4, identificador) -> 53
(4, "bool") -> 53
(4, "int") -> 53
(4, "char") -> 53
(4, "float") -> 53
(5, identificador) -> 31
(5, "bool") -> 31
(5, "int") -> 31
(5, "char") -> 31
(5, "float") -> 31
(5, "void") -> 31
(6, "begin") -> 8
(7, "=") -> 9
(8, "end") -> 10
(8, "vars") -> 11
(8, comando) -> 12
(9, inteiro) -> 13
(9, float) -> 13
(9, "true") -> 13
(9, "false") -> 13
(9, char) -> 13
(10, "program") -> 29
(11, identificador) -> 15
(11, "bool") -> 15
(11, "int") -> 15
(11, "char") -> 15
(11, "float") -> 15
(12, "end") -> 10
(12, comando) -> 12
(13, ";") -> 14
(14, identificador) -> 7
(14, "end") -> 16
(15, identificador) -> 17
(16, "const") -> 18
(17, ";") -> 19
(17, "[") -> 20
(17, ",") -> 15
(18, "typedef") -> 2
(18, "struct") -> 3
(18, "vars") -> 4
(18, "function") -> 5
(18, "program") -> 6
(19, identificador) -> 15
(19, "end") -> 22
(19, "bool") -> 15
(19, "int") -> 15
```

```
(19, "char") -> 15
(19, "float") -> 15
(20, inteiro) -> 21
(21, "]") -> 17
(22, "vars") -> 12
(23, "=") -> 24
(24, identificador) -> 25
(24, "bool") -> 25
(24, "int") -> 25
(24, "char") -> 25
(24, "float") -> 25
(25, ";") -> 26
(26, identificador) -> 23
(26, "end") -> 27
(27, "typedef") -> 28
(28, "struct") -> 3
(28, "vars") -> 4
(28, "function") -> 5
(28, "program") -> 6
(30, "begin") -> 32
(31, identificador) -> 33
(32, identificador) -> 34
(32, "bool") -> 34
(32, "int") -> 34
(32, "char") -> 34
(32, "float") -> 34
(33, "(") -> 35
(34, identificador) -> 36
(35, identificador) -> 37
(35, "bool") -> 37
(35, "int") -> 37
(35, "char") -> 37
(35, "float") -> 37
(35, ")") -> 38
(36, ";") -> 39
(36, "[") -> 40
(36, ",") -> 34
(37, identificador) -> 59
(37, "byref") -> 60
(38, "begin") -> 41
(39, identificador) -> 34
(39, "end") -> 51
(39, "bool") -> 34
(39, "int") -> 34
(39, "char") -> 34
(39, "float") -> 34
(40, inteiro) -> 45
(41, "end") -> 42
(41, "vars") -> 43
(41, comando) -> 44
(42, "function") -> 58
```

```
(43, identificador) -> 46
(43, "bool") -> 46
(43, "int") -> 46
(43, "char") -> 46
(43, "float") -> 46
(44, "end") -> 42
(44, comando) -> 44
(45, "]") -> 36
(46, identificador) -> 47
(47, ";") -> 48
(47, "[") -> 49
(47, ",") -> 46
(48, identificador) -> 46
(48, "end") -> 52
(48, "bool") -> 46
(48, "int") -> 46
(48, "char") -> 46
(48, "float") -> 46
(49, inteiro) -> 50
(50, "]") -> 47
(51, "struct") -> 28
(52, "vars") -> 44
(53, identificador) -> 54
(54, ";") -> 55
(54, "[") -> 56
(54, ",") -> 53
(55, identificador) -> 53
(55, "end") -> 63
(55, "bool") -> 53
(55, "int") -> 53
(55, "char") -> 53
(55, "float") -> 53
(56, inteiro) -> 57
(57, "]") -> 54
(58, "function") -> 5
(58, "program") -> 6
(59, "[") -> 61
(59, ",") -> 62
(59, ")") -> 38
(60, identificador) -> 59
(61, inteiro) -> 64
(62, identificador) -> 37
(62, "bool") -> 37
(62, "int") -> 37
(62, "char") -> 37
(62, "float") -> 37
(63, "vars") -> 58
(64, "]") -> 59
```

../1–linguagem/notacoes/JFLAP/programa/programa.txt

A saída referente à submáquina comando:

```
initial: 0
final: 12
(0, identificador) -> 1
(0, "call") -> 2
(0, "if") -> 3
(0, "while") -> 4
(0, "scan") -> 5
(0, "print") -> 6
(0, "return") -> 7
(1, "[") -> 8
(1, ".") -> 9
(1, "=") -> 10
(1, "+=") -> 10
(1, "-=") -> 10
(1, "*=") -> 10
(1, "/=") -> 10
(2, identificador) -> 18
(3, "(") -> 22
(4, "(") -> 28
(5, identificador) -> 14
(6, expressao) -> 13
(7, expressao) -> 11
(7, ";") -> 12
(8, expressao) -> 16
(9, identificador) -> 1
(10, expressao) -> 11
(11, ";") -> 12
(13, ";") -> 12
(13, ",") -> 6
(14, "[") -> 15
(14, ".") -> 5
(14, ";") -> 12
(14, ",") -> 5
(15, expressao) -> 17
(16, "]") -> 1
(17, "]") -> 14
(18, "(") -> 19
(19, expressao) -> 20
(19, ")") -> 11
(20, ",") -> 21
(20, ")") -> 11
(21, expressao) -> 20
(22, expressao) -> 23
(22, ")") -> 24
(23, ")") -> 24
(24, "then") -> 25
(25, comando) -> 25
(25, "else") -> 26
(25, "end") -> 27
(26, comando) -> 26
(26, "end") -> 27
```

```
(27, "if") -> 12
(28, expressao) -> 29
(28, ")") -> 30
(29, ")") -> 30
(30, "do") -> 31
(31, comando) -> 31
(31, "end") -> 32
(32, "while") -> 12
```

../1–linguagem/notacoes/JFLAP/comando/comando.txt

E, finalmente, a saída referente à submáquina expressão:

```
initial: 0
final: 3, 4, 9
(0, "!") -> 1
(0, "-") -> 1
(0, "(") -> 2
(0, inteiro) -> 3
(0, float) -> 3
(0, "true") -> 3
(0, "false") -> 3
(0, char) -> 3
(0, identificador) -> 4
(1, "(") -> 2
(1, inteiro) -> 3
(1, float) -> 3
(1, "true") -> 3
(1, "false") -> 3
(1, char) -> 3
(1, identificador) -> 4
(2, expressao) -> 8
(3, "-") -> 0
(3, "^") -> 0
(3, "*") -> 0
(3, "/") -> 0
(3, "+") -> 0
(3, "==") -> 0
(3, "!=") -> 0
(3, "<") -> 0
(3, ">") -> 0
(3, "<=") -> 0
(3, ">=") -> 0
(3, "and") -> 0
(3, "or") -> 0
(4, "-") -> 0
(4, "(") -> 5
(4, "[") -> 6
(4, ".") -> 7
(4, "^") -> 0
(4, "*") -> 0
(4, "/") -> 0
```

```
40 (4, "+") -> 0
41 (4, "==") -> 0
42 (4, "!=") -> 0
43 (4, "<") -> 0
44 (4, ">") -> 0
45 (4, "<=") -> 0
46 (4, ">=") -> 0
47 (4, "and") -> 0
48 (4, "or") -> 0
49 (5, expressao) -> 11
50 (5, ")") -> 3
51 (6, expressao) -> 10
52 (7, identificador) -> 9
53 (8, ")") -> 3
54 (9, "-") -> 0
55 (9, "[") -> 6
56 (9, ".") -> 7
57 (9, "^") -> 0
58 (9, "*") -> 0
59 (9, "/") -> 0
60 (9, "+") -> 0
61 (9, "==") -> 0
62 (9, "!=") -> 0
63 (9, "<") -> 0
64 (9, ">") -> 0
65 (9, "<=") -> 0
66 (9, ">=") -> 0
67 (9, "and") -> 0
68 (9, "or") -> 0
69 (10, "]") -> 9
70 (11, ")") -> 3
71 (11, ",") -> 12
72 (12, expressao) -> 11
```

../1–linguagem/notacoes/JFLAP/expressao/expressao.txt

3.2.2.2 Lista dos autômatos

Nesta seção listaremos simplesmente os autômatos resultantes representados de uma maneira gráfica a partir das tabelas de transições da seção anterior.

A ferramenta utilizada para gerar a modelagem gráfica é o JFLAP, obtido do site (http://www.cs.duke.edu/csed/jflap/)

Figura 11 – Autômato representando a submáquina programa

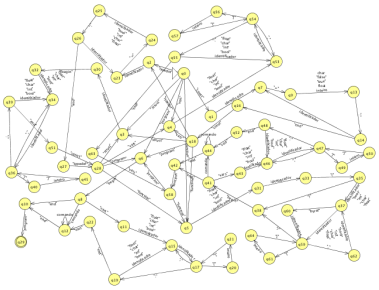


Figura 12 – Autômato representando a submáquina comando

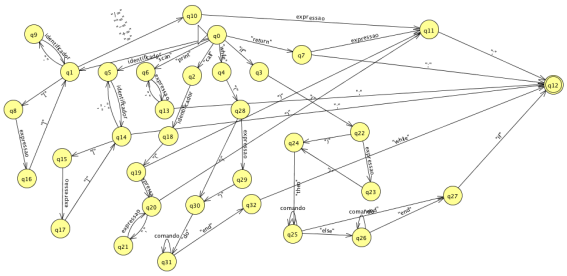
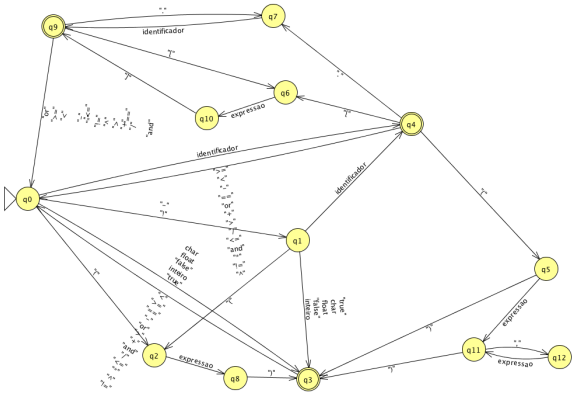


Figura 13 – Autômato representando a submáquina expressao



3.3 Comentários sobre a implementação

3.3.1 Desenvolvimento do módulo sintático

Partindo do modelo da notação de Wirth reduzida, nosso analisador sintático baseia-se na interação entre três máquinas de estados, representando as regras finais, que podem se chamar mutuamente, o que

é previsto por estarmos trabalhando com um Autômato de Pilha Estruturado (APE).

O módulo sintático, portanto, começa suas atividades com a premissa que as três máquinas de estados já estejam criadas e inicializadas. Pressupõe-se que essa inicialização seja executada no programa principal (função main) do compilador.Essa inicialização ocorre de forma análoga ao que realizamos no módulo léxico e é baseada em tabelas de transições, as quais encontram-se descritas em arquivos externos, facilitando futuras modificações.

Utiliza-se a subrotina verify_syntax(), a qual já capaz de dizer se o compilador reconhece a linguagem. Mais detalhadamente, essa verificação tem como principal atividade o lançamento da função recognize(), utilizando a máquina que representa "programa‘’ e o primeiro token lido pelo analisador léxico como parâmetros.

Grosso modo, o que esta função realiza é, dado o token passado via parâmetro, consulta-se sua tabela de transições internas e verifica-se se há alguma transição possível. Se sim, a transição é realizada, o estado do autômato é atualizado e um novo token é lido.

Cuida lembrar que há casos em que a transição chama uma nova submáquina, o que também é especificado pela tabela. Quando isso ocorre, a nova máquina é chamada através da mesma função regconize, mas passando a nova máquina como parâmetro e o mesmo token por referência. Em outros termos, não há necessidade de lookahead e, por serem armazenados em memória quando da chamada de submáquina, os tokens são consumidos sob demanda.

Por outro lado, se não houver transição possível na tabela para um dado token, a função verifica se a máquina encontra-se em estado de aceitação, retornando verdadeiro, ou não, retornando falso. O valor falso traduz o caso em que houve um erro sintático na estrutura do código-fonte e o erro é rapidamento propagado para as máquinas de estados que a chamaram, parando a execução do módulo. Já o valor verdadeiro, faz com que a máquina que a chamou continue sua execução a partir do próximo estado previsto pela tabela em que o autômato estava a esperar (resincronização); se o autômato é a raiz ("programa‘’) e já não há mais tokens a serem lidos, o compilador reconhece a sintaxe da linguagem.

3.3.2 Integração com o compilador

Devido à alta modularização configurada pela arquitetura escolhida, o programa principal que representa o compilador possui uma estrutura bastante simples.

Primeiramente, o módulo léxico é inicializado, passando o código-fonte a ser analidado como parâmetro. Em seguida, inicializa-se a Tabela de Símbolos e a Tabela de Palavras-chaves. Até aqui, não há diferenças em relação à entrega do analisador léxico.

A próxima ação, então, inicializa o módulo sintático, o que consiste basicamente na criação das máquinas de estados que representam as expressões de Wirth a partir dos arquivos externos, como descrito na seção anterior.

Finalmente, chama-se a subrotina verify_syntax(), a qual foi explicada anteriormente e é capaz de decidir se o código-fonte pertence ou não à lingugagem definida pela gramática.

4 Semântica Dinâmica

4.1 Ambiente de Execução

4.1.1 Elementos da Arquitetura de Von Neumann

O compilador da nossa linguagem terá como linguagem de saída um programa escrito especialmente para ser executado dentro de uma maquina virtual intitulada MVN.

O programa MVN é uma abstração da arquitetura de computadores conhecida como arquitetura de Von Neumann.

Em 1936, o inglês Alan M. Turing propôs um modelo de computação (Máquina de Turing), que compõe-se de:

- Uma fita infinita, composta de células, cada qual contendo um símbolo de um alfabeto finito disponível (a fita também implementa a memória externa da máquina).
- Um cursor, que pode efetuar leitura ou escrita em uma célula, ou mover-se para a direita ou para a esquerda.
- Uma máquina de estados finitos, que controla o cursor.

No entanto, a Máquina de Turing apresenta alguns problemas práticos, como:

- A Máquina de Turing se apresenta através de um formalismo poderoso, com fita infinita e apenas quatro operações triviais: ler, gravar, avançar e recuar.
- Isso faz dela um dispositivo detalhista que oferece apenas uma visão microscópica da solução do problema que pretende resolver, não permitindo ao usuário usar abstrações.
- Embora a Máquina de Turing Universal permita uma espécie de programação, o seu código é extenso e a sua velocidade final de execução, muito baixa.

A arquitetura de Von Neumann foi em uma alternativa prática viável à Máquina de Turing, disponibilizando operações mais poderosas e ágeis que o modelo de Turing. Assim, ela pode ser considerada como sua evolução natural. Isso pode ser contastado pelas seguintes características:

- 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 freqüentes (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

Note que sua principal característica é que não há divisão entre dados e o código-fonte do programa em si; ambos são escritos em memória.

O modelo da arquitetura pode ser melhor compreendido pelo esquema abaixo:

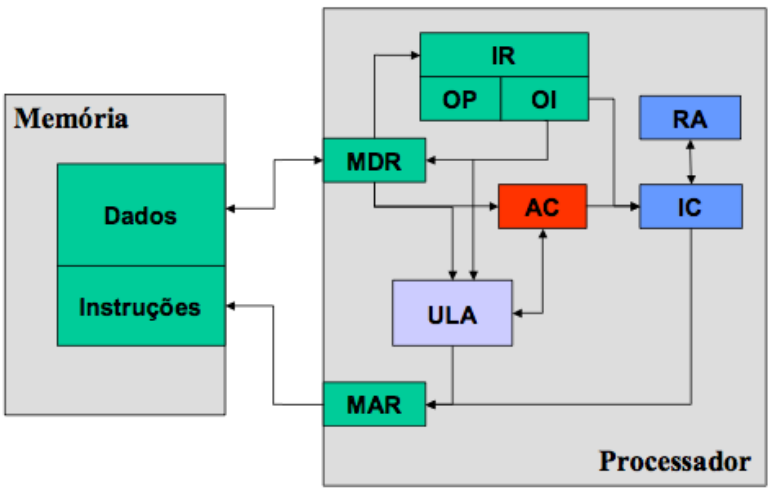


Figura 14 – Diagrama da arquitetura de Von Neumann

A arquitetura de Von Neumann é composta por um processador e uma memória principal.

Como mostrado anteriormente, na memória principal armazenam-se as intruções do código-fonte e os dados, sendo a divisão mostrada no esquema inexistente (utilizada para fins elucidativos).

Além da ULA (Unidade Lógica Aritmética), a qual é a responsável pelo processamento de operações lógicas e aritméticas, o processador possui um conjunto de elementos físicos de armazenamento de informações e é recorrente dividir esses componentes nos seguintes módulos resgistradores:

1. **MDR - Registrador de dados da memória**

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

2. **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.

3. **IC - Registrador de endereço da próxima 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.

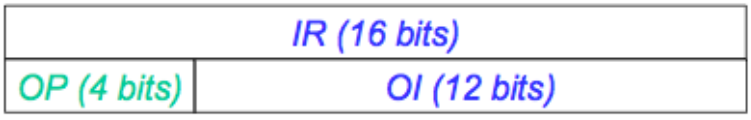


Figura 15 – Registrador de Instrução

   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. **RA - Registrador de endereço de retorno**

   Guarda o endereço da sub-rotina ou função em execução.

6. **AC - Acumulador**

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

   O conjunto de dados nos registradores contidos em cada instante constitui o estado instantâneo do processamento. Note que a máquina virtual MVN não realiza diretamente operações lógicas e não há endereçamento indireto nem indexado. Para realizar isso, é preciso realizar algumas manipulações no programa fonte de maneira conveniente.

   O funcionamento da máquina funciona em quatro fases:

   a) **Determinação da Próxima Instrução a Executar**

   b) **Fase de Obtenção da Instrução**

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

   c) **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 d).

   d) **Fase de Execução da Instrução**

      Executar o procedimento selecionado em (c), 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.

      i. **Execução da instrução (decodificada em (c))**
      De acordo com o código da instrução a executar (contido no registrador de instrução), executar os procedimentos de simulação correspondentes (detalhados adiante)

      ii. **Acerto do registrador de Endereço da Próxima Instrução para apontar a próxima instrução a ser simulada:**

Incrementar o registrador de Endereço da Próxima Instrução.

4.1.2 Instruções da MVN

As instruções da MVN podem ser resumidas pela seguinte tabela:

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 (**)

Figura 16 – Instruções da MVN

Obs.: Sistema de numeração e aritmética adotada: Binário, em complemento de dois – representa inteiros e executa operações em 16 bits. O bit mais à esquerda é o bit de sinal (1 = negativo)

A seguir descreveremos o que a máquina realiza ao executar cada tipo de operação:

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 pelo 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 = A (desvio para subprograma)

Armazena o conteúdo do registrador de Endereço da Próxima instrução (IC), incrementado de uma unidade, na posição de memória endereçada pelo registrador de operando (OI), que corresponde ao endereço do subprograma.

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

MEM[OI] := IC + 1

IC := OI + 1

Registrador de instrução = B (retorno de subprograma)

Recupera no registrador de endereço de retorno (RA) o conteúdo da posição de memória apontada pelo registrador de operando (OI), que vem a ser o endereço de retorno.

Armazena no registrador de Endereço da Próxima instrução (IC) o conteúdo do registrador de endereço de retorno (RA).

RA := 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

Para

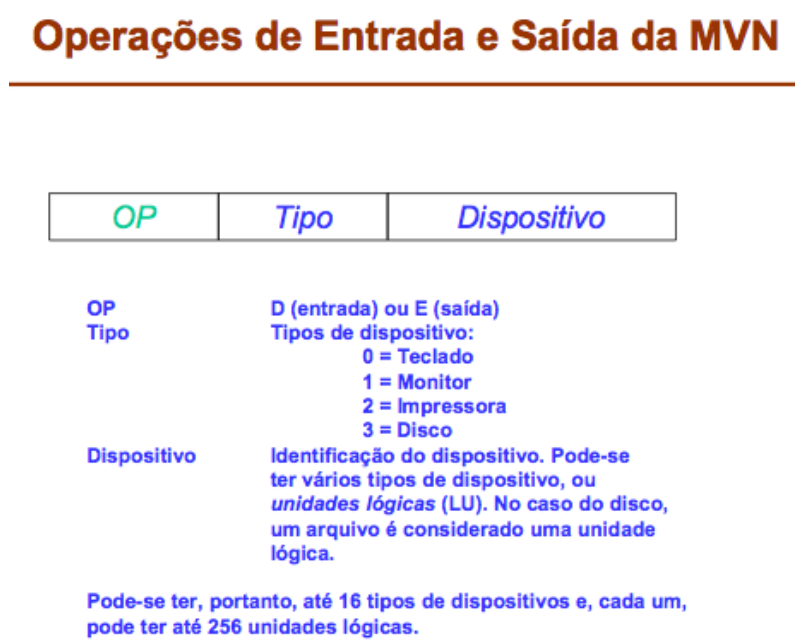


Figura 17 – Instruções de entrada e saída

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

4.1.3 Módulos Extras: Montador, Ligador e Relocador

O compilador traduz o código-fonte da linguagem de alto nível em código-objeto. Tal código não é escrito em linguagem de máquina, executável pela máquina de Von Neumann.

Na realidade, ele é escrito em linguagem de montagem (simbólica). Essa linguagem é bastante próxima da linguagem de máquina, mas é mais compreensível por um ser humano, por ser mais legível. Isso deve-se ao fato de que as instruções não são descritas por números hexadecimais, mas por mnemônicos os quais representam de maneira mais intuitiva o significado de cada instrução.

Os mnemônicos para a MVN estão resumidos na seguinte tabela:

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**

Figura 18 – Mnemônicos

O elemento responsável por traduzir o código-objeto em linguagem de máquina é o Montador (Assembler). Em seguida, o resultado (que não está completamente resolvido e ainda não tem seu endereço definido) é repassado para o Ligador (Linker), o qual é o responsável por resolver a modularização dos programas (uso de bibliotecas). Por exemplo, quando do uso de funções ou subrotinas de programas externos dentro da execução de um programa principal.

Finalmente, o resultado do Ligador é repassado ao Relocador, possibilitando que os programas a serem executados pela máquina de Von Neumann possam ser devidamente relocados convenientemente pelo sistema operacional na memória principal. Isso evita o problema de programas absolutos que devem ser executados estritamente nas posições de memória em que foram criados, consistindo em um risco de uso potencialmente indevido da memória.

Esse módulos extras são entidades à parte da arquitetura de Von Neumann, mas a implementação da MVN que estamos utilizando no projeto, já os possui devidamente integrados, de forma que é possível realizar a execução de um código-objeto fornecido pelo compilador que está escrito na linguagem simbólica de montagem.

4.1.4 Pseudo-instruções da Linguagem de Montagem

A linguagem simbólica do código-objeto não possui somente os mnemônicos das instruções da MVN, 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.

Assim, há comandos chamados de pseudo-instruções da linguagem de montagem. Eles são chamados dessa forma porque não representam efetivamente as instruções da máquina de Von Neumann, mas são necessários para resolver os problemas evocados anteriormente.

Na linguagem de montagem, as pseudo-instruções também são representadas por mnemônicos. São eles:

- **@** : 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.

Assim, com esses elementos, é possível obter-se o código de máquina a partir do código-objeto. A seguir, temos um exemplo de um código escrito em linguagem de montagem e sua respectiva tradução pelos módulos Montador, Logador e Relocador:

Exemplo: Somador

- Código

		Endereço de geração	Resolução do operando	Relocabilidade do operando	Localidade do operando
SOMADOR <		0	1	0	0
ENTRADA1 <		0	1	0	0
ENTRADA2 <		0	1	0	0
SAIDA >		0	0	0	0
@ /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	0	1
	LD VALOR2	0	0	0	0
	MM ENTRADA2	0	1	0	1
	SC SOMADOR	0	1	0	1
	HM /00	0	0	0	0

```
4000 0000 ; "SOMADOR<"
4001 0000 ; "ENTRADA1<"
4002 0000 ; "ENTRADA2<"
0006 0000 ; "SAIDA>"

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

Figura 19 – Exemplo: somador

4.1.5 Descrição Geral do Ambiente de Execução

4.1.5.1 Organização da memória

O ambiente de execução da MVN fornece aos programadores um total de 4Kb de memória para ser usado tanto para o código quanto para as variáveis do programa. O montador aloca a memória com base nos endereços relativos especificados no código do programa. Desses 4Kb, 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.

Em outras palavras, reserva-se uma parte do código para a área de dados, onde se encontram as variáveis, uma parte para o resto do programa, que inclui a função principal e as subrotinas e uma parte dedicada a pilhas de variáveis e endereços que viabilizam a chamada de subrotinas.

4.1.5.2 Funções de Input e Output

As funções de Input e Output serão implementadas na MVN. Serão fornecidas três funções de input e duas de output:

- GET_CHAR: Devolve um caracter lido na entrada do teclado

- GET_STRING: Pega um conjunto de caracteres, sem contar os espaços, tabs e saltos de linha
- GET_NUMBER: Captura um número no formato ASCII e devolve sob a forma de hexadecimal
- PRINT_STRING: Imprime uma cadeia de caracteres
- PRINT_NUMBER: Transforma um número hexadecimal em caracteres ASCII

4.1.5.3 Registro de ativação

As subrotinas são executadas com as seguintes instruções da MVN:

- Desvio para subprograma (função) - código SC (A): 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 (função) - código RS (B): desvia a execução para o endereço indicado pelo valor guardado na posição de memória do operando.

Para garantir a operação de chamar várias subrotinas aninhadas (ex.: recursões), é necessário empilhar as variáveis do programa, isto é, o estado da execução do programa e também a quantidade de variáveis da rotina em execução (valor guardado na área auxiliar). Para tanto, utilizamos a estrutura chamada de Registro de Ativação.

O registro de ativação nesse ambiente de execução será feito sob a forma de uma pilha, onde a cada chamada de função todos os dados do referentes a função, bem como o endereço de retorno, devem ser empilhados para serem usados. Os dados a serem empilhados no registro de ativação são:

- Endereço de retorno
- Endereço do próximo endereço da pilha
- Parâmetros da função
- Variáveis locais das função

O endereço de retorno fica localizado no primeiro endereço do bloco empilhado no registro de ativação. O segundo endereço é referente ao endereço do primeiro endereço do próximo bloco do registro de ativação. Esse endereço é usado para mudar o valor ponteiro do registro de ativação, para que a função que chamou a outra possa voltar a enxergar suas variáveis. Do terceiro endereço em diante estão localizados os parâmetros da função. Após o final dos parâmetros, estão localizadas as variáveis locais necessárias para guardar executar as operações durante a execução da função.

Os endereços das variáveis locais e dos parâmetros podem ser calculados usando o ponteiro do registro de ativação, somando dois mais os tamanhos das variáveis existentes anteriormente.

A figura a seguir ilustra a organização da pilha de ativação.

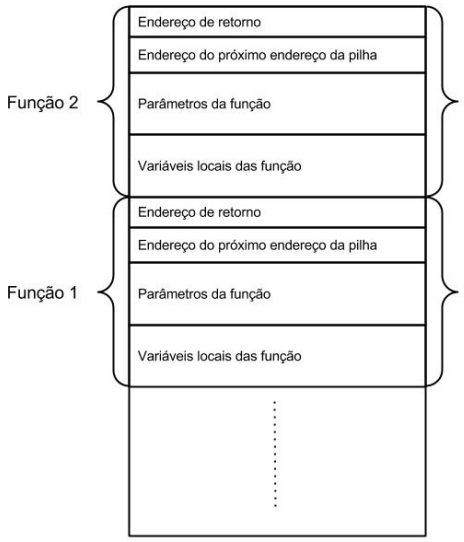


Figura 20 – Registro de Ativação

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. Com registro de ativação, realizar uma recursão significa empilhar um novo bloco à pilha e relançar a execução da função.

Para implementar o registro de ativação, tivemos de desenvolver uma bibliteca nativa ao compilador em linguagem de montagem (Assembly).

Basicamente, a biblioteca implementa uma pilha, onde os nós são as informações guardadas no registro de ativação. Uma dificuldade que encontramos foi a de copiar os dados de uma função para a pilha, pois devemos transmitir o conteúdo de um ponteiro para outro.

Para facilitar o processo de montagem, ligação e relocação utilizamos um script de nosso colega Gustavo Pacianotto Gouveia, o qual realiza automaticamente a conversão de nosso código Assembly em linguagem MVN (de máquina).

4.2 Traduções de comandos semânticos para linguagem MVN

4.2.1 Tradução dos comandos imperativos

Nota: <expressao> trata-se de um abuso de linguagem, pois na realidade precisamos guardar o valor calculado - que está no acumulador - em uma variável temporária e depois utilizá-la sempre qquando utilizamos <expressao> diretamente.

Comando	**Linguagem**	**MVN Simbólica**
Declaração de variável escalar simples	int x;	x K /0000
Declaração de variável vetorial simples	int x[n];	x $ =n
Declaração de estrutura simples	struct s begin int p1; int p2; int p3[n]; end struct s x;	x K /0000 K /0000 $ =n

Tabela 1 – Tradução dos comandos principais para a MVN: Parte I

Comando	**Linguagem**	**MVN Simbólica**
Atribuição de variável escalar ou de estrutura	x = <expressao>;	LV x + n ; n > 0 para struct MM END_ALVO <expressao> LD <expressao> MM VALOR SC GRAVA
Atribuição de variável vetorial	x[<expressao>] = <expressao>;	<expressao> LV x + <expressao> MM END_ALVO <expressao> LD <expressao> MM VALOR SC GRAVA
Acesso à variável vetorial	x[<expressao>]	<expressao> LV x + <expressao> MM END_ORIGEM SC ACESSA

Tabela 2 – Tradução dos comandos principais para a MVN: Parte II

Comando	**Linguagem**	**MVN Simbólica**
Declaração de função	function int func(int p1, char p2) begin <comandos> end function	func_end_retorno K /0000 K /0000 func_p1 K /0000 func_p2 K /0000 TMP1 K /0000 TMP2 K /0000 ... func JP /000 <comandos> RS func
Chamada de função	func(<expressao>, <expressao>)	LD parent MM parent_end_retorno LV parent_end_retorno MM END_INICIAL LV parent_tamanho MM TAMANHO SC EMPILHA <expressao> LD <expressao> MM func_p1 <expressao> LD <expressao> MM func_p2 SC func MM TMP_RETURN LD TOPO MM END_BLOCO_ORIGEM LD parent_end_retorno MM END_BLOCO_ALVO LV parent_tamanho MM TAMANHO_BLOCO SC COPIA_BLOCO LD TMP_RETURN

Tabela 3 – Tradução dos comandos principais para a MVN: Parte III

Comando	**Linguagem**	**MVN Simbólica**
Leitura (entrada)	scan x, y, z;	LV x MM END_ALVO SC SCAN_INT LV y MM END_ALVO SC SCAN_INT LV z MM END_ALVO SC SCAN_CHAR
Impressão (saída)	print x, y, z;	LV x MM END_ORIGEM SC ACESSA MM VAR SC PRINT_INT LV y MM END_ORIGEM SC ACESSA MM VAR SC PRINT_INT LV z MM END_ORIGEM SC ACESSA MM VAR SC PRINT_CHAR

Tabela 4 – Tradução dos comandos principais para a MVN: Parte IV

4.2.2 Tradução de estruturas de controle de fluxo

Comando	**Linguagem**	**MVN Simbólica**

If-then	if (<expressao>) then
	<comandos>
	end if
		TMP K /0001
		
		<expressao>
		LD <expressao>
		MM TMP
		
		LD TMP
		JZ ENDIF
		
		<comandos>
		
		ENDIF

If-then-else	if (<expressao>) then
	<comandos>
	else
	<comandos>
	end if
		TMP K /0001
		
		<expressao>
		LD <expressao>
		MM TMP
		LD TMP
		
		JZ ELSE
		<comandos>
		JP ENDIF
		
		ELSE
		<comandos>
		ENDIF

While	while (<expressao>) do
	<comandos>
	end while
		TMP K /0001
		
		WHILE
		
		<expressao>
		LD <expressao>
		MM TMP
		LD TMP
		
		JZ ENDWHILE
		<comandos>
		JP WHILE
		
		ENDWHILE

Tabela 5 – Tradução dos comandos principais para a MVN: Parte V

4.2.3 Funções auxiliares

4.2.3.1 Biblioteca auxiliar

Criamos uma biblioteca auxiliar que permite gerar o código MVN das tabelas anteriores. Ela é carregada em todos os programas compilados pelo compilador que estamos desenvolvendo.

Todas as variáveis auxiliares às quais tivemos de atribuir um valor como VALOR, TAMANHO, END_ALVO etc. são parâmetros das funções auxiliares.

As principais funções existentes na biblioteca referem-se à implementação de uma pilha (no nosso caso, uma abstração do registro de ativação). Assim, além das tradicionais EMPILHA e DESEMPILHA, temos algumas funções auxiliares que ajudam a implementá-las:

- **COPIA_BLOCO**: permite copiar o conteúdo de um bloco de código para outro. A cópia se dá de maneira conveniente para o registro de ativação, quer dizer, respeitando a estrutura convencionada.
- **ACESSA**: Coloca no acumulador o valor localizado no ponteiro do endereco de origem.
- **GRAVA**: Grava no enedereço alvo o valor do parâmetro VALOR.
- **SCAN_***: Realiza a leitura do dispotivo de entrada.
- **PRINT_***: Imprime sobre o dispositivo de entreda.

Nota: Grande parte das manipulações da biblioteca auxiliar usam a noção de ponteiros. Para maiores informações, consultar o código enviado em anexo, o qual encontra-se comentado.

4.2.3.2 Cálculo de <expressao>

Para a geração de código em linguagem MVN, será necessário uma pilha, uma lista, um contador para variáveis temporárias e um para endereços de desvio para auxílio (label).

Note que uma expressão pode ser tanto aritmética quanto booleana em nossa linguagem e a ordem de precedência dos operadores é: operações unárias (! ou -), multiplicação/divisão, adição/subtração, comparações lógicas (>, <, >=, <=, ==, !=), operação lógica E, operação lógica OU.

1. Empilha os átomos assim que são lidos para o analisador sintático até encontrar um “)”.
2. Quando encontrar o “)”, desempilha e coloque na lista auxiliar até encontrar o “(“. Se não houver “)”, vá para o passo 6.
3. Resolve-se os operadores unários "!"e -".

Linguagem	**MVN Simbólica**
! NUM/ID anterior	LV =NUM ; ou LD ID anterior JZ JUMP_EXPBOOL<contador> LV =0 MM temp<contador de variável temporária> JP FIM_EXPBOOL<contador> JUMP_EXPBOOL<contador> LV =1 MM temp<contador de variável temporária> FIM_EXPBOOL<contador>
- NUM/ID anterior	LV =NUM ; ou LD ID anterior MM temp<contador de variável temporária> LV =0 - temp<contador de variável temporária> MM temp<contador de variável temporária>

Tabela 6 – Tradução dos comandos principais para a MVN: multiplicação e divisão

Após a geração do código, deve-se substituir na lista o átomo pela variável temporária criada.

4. Resolve-se os operadores “*” e “/” percorrendo a lista procurando-os.

Linguagem	**MVN Simbólica**
NUM/ID posterior *_/ NUM/ID anterior	LV =NUM ; ou LD ID posterior *_/ <NUM/ID anterior> MM temp<contador de variável temporária>

Tabela 7 – Tradução dos comandos principais para a MVN: multiplicação e divisão

Após a geração do código, deve-se substituir na lista os três átomos pela variável temporária criada.

5. Resolve-se os operadores "+" e "-" da mesma forma que no passo 4.

Linguagem	**MVN Simbólica**
NUM/ID posterior +_- NUM/ID anterior	LV =NUM ; ou LD ID posterior +_- <NUM/ID anterior> MM temp<contador de variável temporária>

Tabela 8 – Tradução dos comandos principais para a MVN: adição e subtração

Após a geração do código, deve-se substituir na lista os três átomos pela variável temporária criada.

6. Resolve-se os operadores de comparação lógica da mesma forma que no passo 4.

Linguagem	**MVN Simbólica**
NUM/ID posterior > NUM/ID anterior	LV =NUM ; ou LD ID posterior - <NUM ou ID anterior> JN JUMP_EXPBOOL<contador> LV =1 MM temp<contador de variável temporária> JP FIM_EXPBOOL<contador> JUMP_EXPBOOL<contador> LV =0 MM temp<contador de variável temporária> FIM_EXPBOOL<contador>
NUM/ID posterior < NUM/ID anterior	LV =NUM ; ou LD ID posterior - <NUM ou ID anterior> JN JUMP_EXPBOOL<contador> LV =0 MM temp<contador de variável temporária> JP FIM_EXPBOOL<contador> JUMP_EXPBOOL<contador> LV =1 MM temp<contador de variável temporária> FIM_EXPBOOL<contador>

Tabela 9 – Tradução dos comandos principais para a MVN: comparação lógica

Linguagem	**MVN Simbólica**
NUM/ID posterior >= NUM/ID anterior	LV =NUM ; ou LD ID posterior - <NUM ou ID anterior> JN JUMP_EXPBOOL<contador> JZ JUMP_EXPBOOL<contador> LV =1 MM temp<contador de variável temporária> JP FIM_EXPBOOL<contador> JUMP_EXPBOOL<contador> LV =0 MM temp<contador de variável temporária> FIM_EXPBOOL<contador>
NUM/ID posterior <= NUM/ID anterior	LV =NUM ; ou LD ID posterior - <NUM ou ID anterior> JN JUMP_EXPBOOL<contador> JZ JUMP_EXPBOOL<contador> LV =0 MM temp<contador de variável temporária> JP FIM_EXPBOOL<contador> JUMP_EXPBOOL<contador> LV =1 MM temp<contador de variável temporária> FIM_EXPBOOL<contador>

Tabela 10 – Tradução dos comandos principais para a MVN: comparação lógica II

Linguagem	**MVN Simbólica**
NUM/ID posterior == NUM/ID anterior	LV =NUM ; ou LD ID posterior - <NUM ou ID anterior> JZ JUMP_EXPBOOL<contador> LV =0 MM temp<contador de variável temporária> JP FIM_EXPBOOL<contador> JUMP_EXPBOOL<contador> LV =1 MM temp<contador de variável temporária> FIM_EXPBOOL<contador>
NUM/ID posterior != NUM/ID anterior	LV =NUM ; ou LD ID posterior - <NUM ou ID anterior> JZ JUMP_EXPBOOL<contador> LV =1 MM temp<contador de variável temporária> JP FIM_EXPBOOL<contador> JUMP_EXPBOOL<contador> LV =0 MM temp<contador de variável temporária> FIM_EXPBOOL<contador>

Tabela 11 – Tradução dos comandos principais para a MVN: comparação lógica III

Após a geração do código, deve-se substituir na lista os três átomos pela variável temporária criada.

7. Resolve-se a operação lógica E.

Linguagem	**MVN Simbólica**
NUM/ID posterior and NUM/ID anterior	LV =NUM ; ou LD ID posterior * NUM ; ou ID anterior MM temp<contador de variável temporária>

Tabela 12 – Tradução dos comandos principais para a MVN: operação lógica E

Após a geração do código, deve-se substituir na lista os três átomos pela variável temporária criada.

8. Resolve-se a operação lógica OU.

Linguagem	**MVN Simbólica**
NUM/ID posterior or NUM/ID anterior	LV =NUM ; ou LD ID posterior + NUM ; ou ID anterior JZ JUMP_EXPBOOL<contador> LV =1 MM temp<contador de variável temporária> JP FIM_EXPBOOL<contador> JUMP_EXPBOOL<contador> LV =0 MM temp<contador de variável temporária> FIM_EXPBOOL<contador>

Tabela 13 – Tradução dos comandos principais para a MVN: operação lógica OU

Após a geração do código, deve-se substituir na lista os três átomos pela variável temporária criada.

9. Quando a lista contiver apenas um item, insira-o na pilha e volte para o passo 1, cujo intuito é o de eliminar mais parênteses.

10. Nesse passo, a pilha encontra-se com toda a expressão sem parênteses. Com isso, pode-se jogar todo o conteúdo da pilha na lista e resolvê-la com os passos 3 a 8.

11. Após resolver todos os operadores, a lista estará apenas com um átomo, sendo o resultado da expressão e o código já foi gerado.

    LV =NUM ; ou LD ID