

Departamento de Engenharia de Computação e Sistemas Digitais

**PCS 2056 – Linguagens e Compiladores**

**Relatório Final**

09/12/2011

Bruno Pezzolo dos Santos 5948816

Carla Guillen Gomes 5691366

**Sumário**

[1. Introdução 3](#_Toc311198496)

[2. Definição da Linguagem 4](#_Toc311198497)

[2.1. Descrição Informal 4](#_Toc311198498)

[Funções 4](#_Toc311198499)

[Repetição Condicional 5](#_Toc311198500)

[Decisão 5](#_Toc311198501)

[Entrada 5](#_Toc311198502)

[Saída 5](#_Toc311198503)

[2.2. Descrição Formal 5](#_Toc311198504)

[Notação BNF 5](#_Toc311198505)

[Notação Wirth 7](#_Toc311198506)

[2.3. Descrição Reduzida 9](#_Toc311198507)

[3. Análise Léxica 11](#_Toc311198508)

[3.1. Autômatos 12](#_Toc311198509)

[3.2. Implementação 15](#_Toc311198510)

[4. Análise Sintática 18](#_Toc311198511)

[4.1. Submáquinas do Autômato de Pilha Estruturado (APE) 18](#_Toc311198512)

[4.2. Implementação 23](#_Toc311198513)

[5. Características da MVN 25](#_Toc311198514)

[5.1. Instruções da linguagem de saída 25](#_Toc311198515)

[5.2. Pseudoinstruções da linguagem de saída 26](#_Toc311198516)

[5.3. Características gerais 26](#_Toc311198517)

[Chamada de subrotina 28](#_Toc311198518)

[Retorno de subrotina 28](#_Toc311198519)

[6. Tradução de Comandos 29](#_Toc311198520)

[6.1. Controle de fluxo 29](#_Toc311198521)

[6.2. Comandos imperativos 30](#_Toc311198522)

[6.3. Exemplo de programa traduzido 34](#_Toc311198523)

[7. Análise Semântica 36](#_Toc311198524)

[7.1. Tabela de símbolos com suporte a escopo 36](#_Toc311198525)

[7.2. Principais ações semânticas 37](#_Toc311198526)

[Rótulos 37](#_Toc311198527)

[Submáquina Programa 38](#_Toc311198528)

[Submáquina Comando 38](#_Toc311198529)

[Submáquina Expressão 40](#_Toc311198530)

[7.3. Ambiente de execução 40](#_Toc311198531)

[8. Testes 42](#_Toc311198532)

[Teste 1: input, output, if e while aninhados 42](#_Toc311198533)

[Teste 2: operações aritméticas 43](#_Toc311198534)

[Teste 3: operações lógicas 44](#_Toc311198535)

[9. Conclusão 46](#_Toc311198536)

# Introdução

Este documento descreve o compilador desenvolvido pelos alunos Bruno Pezzolo dos Santos e Carla Guillen Gomes na disciplina PCS2056 – Linguagens e Compiladores. O compilador foi desenvolvido com fins didáticos, sendo desconsideradas características como otimização de código-fonte e desempenho.

O compilador foi escrito na linguagem C e gera código objeto em linguagem simbólica composta por mnemônicos, aceita pela MVN fornecida. A MVN (Máquina de von Neumann) simula o Modelo de von Neumann como um processador simples rodando na máquina virtual Java (JVM).

A arquitetura do compilador é orientada à sintaxe, ou seja, o módulo de análise sintática gerencia as atividades através de requisições aos outros módulos, conforme representação a seguir:

Código fonte

Tokens

Código objeto

Analisador Sintático

Analisador Léxico

Tabela de símbolos

Solicita token

Analisador Semântico

As fases do desenvolvimento do compilador (definição da linguagem, análise léxica, análise sintática, definição do ambiente de execução e análise semântica) serão descritas nos próximos capítulos.

# Definição da Linguagem

A linguagem de alto nível criada é baseada na linguagem C. Os programas são formados por sequências de comandos separados por ponto e vírgula. Os programas se iniciam com a palavra **program**, seguida pelas definições de structs, declarações de variáveis, structs e/ou vetores e funções. Por fim, a estrutura **main {**<declarações> <comandos>**}** contém o programa principal a ser executado.

Um exemplo de programa está representado a seguir:

|  |
| --- |
| program  function int fatorial\_iterativo(int n) {    declare int fatorial;  fatorial = 1;   while (n > 0) {  fatorial = fatorial \* n;     n = n - 1;  }   return fatorial; }   main {  declare int fat;  fat = fatorial\_iterativo(10);  output fat; } |

## Descrição Informal

A seguir, apresenta-se a representação informal das principais funcionalidades definidas para a linguagem

### Funções

|  |
| --- |
| **function** tipo identificador **(** parâmetro\_1**,** ..., parâmetro\_n **)** **{**  declaração\_1**;**  ...  declaração\_n**;**  comando\_1**;**  ...  comando\_n**;**  **}** |

### Repetição Condicional

|  |
| --- |
| **while** **(**condição**) {**  comando\_1**;**  ...  comando\_n**;**  **}** |

### Decisão

|  |
| --- |
| **if (**condição**) {**  comando\_1**;**  ...  comando\_n**;**  **}** |
| **if (**condição**) {**  comando\_1**;**  ...  comando\_n**;**  **}** **else** **{**  comando\_1**;**  ...  comando\_n**;**  **}** |

### Entrada

|  |
| --- |
| **input** identificador **;** |

### Saída

|  |
| --- |
| **output** identificador **;** |

## Descrição Formal

Para formalização da sintaxe definida, foram usadas as notações BNF e Wirth. A descrição formal da gramática nas duas notações encontra-se a seguir.

### Notação BNF

|  |
| --- |
| <programa> ::=  program  <definições> <declarações> <funções>  main { <declarações> <comandos> }  <definições> ::= <definição> <definições> | ε  <definição> ::= typedef struct <nome\_da\_estrutura> { <declarações> }  <nome\_da\_estrutura> ::= <identificador>  <declarações> ::= <declarações\_variáveis> | <declarações\_vetores> | <declarações\_structs> | <declarações> <declarações> | ε  <declarações\_variáveis> ::= declare <declaração\_variável> ; | declare <declaração\_variável>, <declarações\_variáveis>  <declaração\_variável> ::= <tipo> <identificador>  <declarações\_vetores> ::= declare <declaração\_vetor> ; | declare <declaração\_vetor>, <declarações\_vetores>  <declaração\_vetor> ::= <tipo> <identificador>[<número>]  <declarações\_structs> ::= declare <declaração\_struct> ; | declare <declaração\_struct>, <declarações\_structs>  <declaração\_struct> ::= struct <nome\_da\_estrutura> <identificador>  <funções> ::= function <tipo> <identificador> ( <parâmetros> ) { <declarações> <comandos> }  <parâmetros> ::= <declaração\_variável> | <declaração\_variável>, <parâmetros> | ε  <comandos> ::= <comando\_atribuição> | <comando\_condicional> | <comando\_iterativo> | <comando\_entrada> | <comando\_saída> | <comando\_chamada> | <comando\_retorno> | <comandos> <comandos> | ε  <comando\_atribuição> ::= <atribuição\_variável> | <atribuição\_agregados> | <atribuição\_vetor>  <atribuição\_variável> ::= <identificador> = <expressão> ;  <atribuição\_vetor> ::= <identificador>[<número>] = <expressão> ;  <atribuição\_agregados> ::= <identificador> = { <expressões> } ;  <comando\_condicional> ::= if ( <expressão> ) { <comandos> } |  if ( <expressão> ) { <comandos> } else { <comandos> }  <comando\_iterativo> ::= while ( <expressão> ) { <comandos> }  <comando\_entrada> ::= input <identificador> ;  <comando\_saída> ::= output <identificador> ;  <comando\_chamada> ::= <identificador> ( <argumentos> ) ;  <comando\_retorno> = return <expressão> ;  <expressões> ::= <expressão> | <expressão>, <expressões>  <expressão> ::= <expressão\_lógica> | <expressão\_aritmética> | <identificador> | <elemento\_do\_vetor> | <elemento\_struct> | <chamada\_função>  <expressão\_lógica> ::= <booleano> | <expressão> <operador\_lógico> <expressão> | not <expressão\_lógica>  <expressão\_aritmética> ::=<número> | <expressão> <operador\_aritmético> <expressão>  <elemento\_do\_vetor> ::= <identificador>[<numero>]   <elemento\_struct> ::= <identificador>.<identificador>  <chamada\_função> ::= <identificador> ( <argumentos> )  <argumentos> ::= <expressões>  <tipo> ::= int | boolean  <identificador> ::= <letra> | <letra><letra\_digito>  <letra\_dígito> ::= <letra> | <dígito>  <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 | \_  <dígito> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9  <booleano> ::= true | false  <operador\_lógico> ::= > | < | == | and | or  <operador\_aritmético> ::= + | - | \* | / |

### Notação Wirth

|  |
| --- |
| programa = "program" {definicao} {declaracao} {funcao} "main" "{" {declaracao} {comando} "}" .  definicao = "typedef" "struct" nome\_estrutura "{" {declaracao} "}" .  declaracao = declaracao\_variaveis | declaracao\_vetores | declaracao\_structs .  declaracao\_variaveis = "declare" declaracao\_variavel{"," declaracao\_variavel  } ";"  declaracao\_variavel = tipo identificador .  declaracao\_vetores = "declare" declaracao\_vetor {"," declaracao\_vetor } ";" .  declaracao\_vetor = tipo identificador "[" numero "]" .  declaracao\_structs = "declare" declaracao\_struct {"," declaracao\_struct } ";" .  declaracao\_struct = "struct" nome\_estrutura identificador ";"  funcao = "function" tipo identificador "(" [parametros] ")" "{" {declaracao}{comando} "}" .  parametros = declaracao\_variavel {"," declaracao\_variavel} .  comando = comando\_atribuicao | comando\_condicional | comando\_iterativo | comando\_entrada | comando\_saida | comando\_chamada | comando\_retorno .  comando\_atribuicao = atribuicao\_variavel | atribuicao\_vetor | atribuicao\_agregado .  atribuicao\_variavel = identificador "=" expressao ";" .  atribuicao\_vetor = identificador "[" numero "]" "=" expressao ";" .  atribuicao\_agregado = identificador "=" "{" expressao{"," expressao} "};" .  comando\_condicional = "if" "(" expresao ")" "{" {comando} "}" ["else" "{" {comando}"}"] .  comando\_iterativo = "while" "(" expresao ")" "{" {comando}"}" .  comando\_entrada = "input" identificador ";" .  comando\_saida = "output" identificador ";" .  comando\_chamada = identificador "(" [argumentos] ")" ";" .  comando\_retorno = "return" expressao ";" .  expressao = expressao\_logica | expressao\_aritmetica | identificador | elemento\_do\_vetor | elemento\_struct | chamada\_funcao .  expressao\_logica = booleano | "not" expressao\_logica | expressao operador\_logico expressao .  expressao\_aritmetica = numero | expressao operador\_aritmetico expressao .  elemento\_do\_vetor = identificador "[" numero "]" .  elemento\_struct = identificador "." identificador .  chamada\_funcao = identificador "(" [argumentos] ")" .  argumentos = expressao {"," expressao} .  booleano = "true" | "false" .  tipo = "int" | "boolean" .  nome\_estrutura = identificador .  identificador = letra{letra|digito} .  letra = "a" | "b" | "c" | "d" | "e" | "f" | "g" | "h" | "i" | "j" | "k" | "l" | "m" | "n" | "o" | "p" | "q" | "r" | "s" | "t" | "u" | "v" | "w" | "x" | "y" | "z" | "A" | "B" | "C" | "D" | "E" | "F" | "G" | "H" | "I" | "J" | "K" | "L" | "M" | "N" | "O" | "P" | "Q" | "R" | "S" | "T" | "U" | "V" | "W" | "X" | "Y" | "Z" | "\_".  digito = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" .  numero = digito{digito} .   operador\_logico = ">" | "<" | "==" | "and" | "or" .  operador\_aritmetico = "+" | "-" | "\*" | "/" . |

## Descrição Reduzida

Para que a análise sintática fosse implementada usando um autômato de pilha estruturado, a descrição em notação de Wirth foi reduzida através do agrupamento de não-terminais. Além disso, as recursões à esquerda foram eliminadas. O resultado da redução é mostrado a seguir.

|  |
| --- |
| programa = "program" { "typedef" "struct" identificador "{" { "declare" ( ( "int" | "boolean" ) identificador [ "[" numero "]" ] {"," ( "int" | "boolean" ) identificador [ "[" numero "]" ] } | "struct" identificador identificador {"," "struct" identificador identificador } ) ";" } "}" } { "declare" ( ( "int" | "boolean" ) identificador [ "[" numero "]" ] {"," ( "int" | "boolean" ) identificador [ "[" numero "]" ] } | "struct" identificador identificador {"," "struct" identificador identificador } ) ";" } { "function" ( "int" | "boolean" ) identificador "(" [ ( "int" | "boolean" ) identificador [ "[" numero "]" ] {"," ( "int" | "boolean" ) identificador [ "[" numero "]" ] } ] ")" "{" { "declare" ( ( "int" | "boolean" ) identificador [ "[" numero "]" ] {"," ( "int" | "boolean" ) identificador [ "[" numero "]" ] } | "struct" identificador identificador {"," "struct" identificador identificador } ) ";" } {comando} "}" } "main" "{" { "declare" ( ( "int" | "boolean" ) identificador [ "[" numero "]" ] {"," ( "int" | "boolean" ) identificador [ "[" numero "]" ] } | "struct" identificador identificador {"," "struct" identificador identificador } ) ";"} {comando} "}" .  comando = identificador ( "=" ( expressao | "{" expressao{"," expressao} "}" ) | "[" numero "]" "=" expressao | "(" [ expressao {"," expressao} ] ")" ) ";" | "if" "(" expressao ")" "{" {comando} "}" ["else" "{" {comando}"}"] | "while" "(" expressao ")" "{" {comando}"}" | "input" identificador ";" | "output" identificador ";" | "return" expressao ";" .  expressao = ( "true" | "false" | "not" expressao | numero | identificador [ "[" numero "]" | "." identificador | "(" [ expressao {"," expressao} ] ")" ] ) [ ( ">" | "<" | "==" | "and" | "or" | "+" | "-" | "\*" | "/" ) expressao ] .  identificador = letra{letra|digito} .  letra = "a" | "b" | "c" | "d" | "e" | "f" | "g" | "h" | "i" | "j" | "k" | "l" | "m" | "n" | "o" | "p" | "q" | "r" | "s" | "t" | "u" | "v" | "w" | "x" | "y" | "z" | "A" | "B" | "C" | "D" | "E" | "F" | "G" | "H" | "I" | "J" | "K" | "L" | "M" | "N" | "O" | "P" | "Q" | "R" | "S" | "T" | "U" | "V" | "W" | "X" | "Y" | "Z" | "\_".  digito = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" .  numero = digito{digito} . |

# Análise Léxica

O módulo de análise léxica é responsável por converter o texto fonte em componentes chamados *tokens* ou átomos. Fornecendo, então, uma representação mais adequada para realização da análise e tradução pelos outros módulos do compilador.

Dessa maneira, o analisador léxico efetua a leitura do código fonte, e com o conhecimento das especificações da linguagem agrega os caracteres lidos do código em unidades que contenham um sentido completo para a linguagem, os *tokens*, descartando os caracteres que não possuem significado para a linguagem (por exemplo, caracteres de tabulação, na linguagem C).

Porém, mais informações são necessárias para as análises subsequentes, como o tipo de informação que está sendo representada pelo *token* (como, por exemplo, a classificação do átomo como um "identificador", "palavra reservada" ou "número"), além disso, um valor para o *token* também é necessário em outras etapas da compilação, esse valor pode ser um valor númerico do *token*, um índice em uma tabela de símbolos ou o próprio texto. Sendo também papel do analisador léxico realizar essas tarefas.

A tabela abaixo mostra os tipos de átomos reconhecidos pelo analisador léxico, definidos através de expressões regulares.

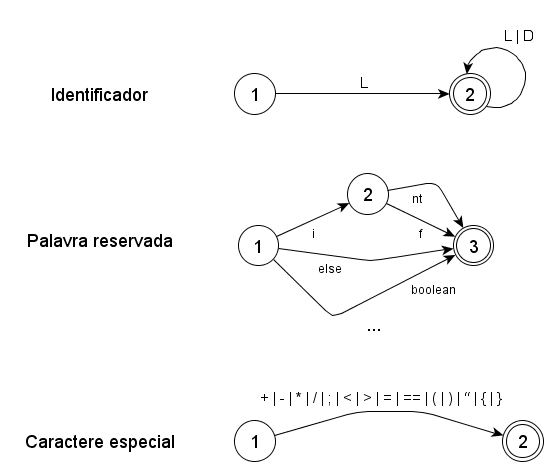
|  |  |
| --- | --- |
| **Classe do átomo** | **Expressão regular** |
| Identificador | [letra][letra | dígito]\* |
| Palavra reservada | if | else | int | boolean | string | false | true | while | main | void | return | def | and | or | not |
| Caractere especial | + | - | \* | / | ; | < | > | = | == | ( | ) | “ | { | } |
| Número | [dígito][dígito]\* |
| String | [“][qualquer\_caractere\_ascii][”] |
| Espaçador | space | \n | \t |
| Comentário | [#][qualquer\_caractere\_ascii]\*[\n] |

O analisador léxico implementado realiza uma simples detecção de erros, retornando *tokens* inválidos quando o conjunto de caracteres lidos não corresponde a nenhuma das expressões representadas acima.

## Autômatos

Para facilitar a representação do reconhecimento dos *tokens* pelo analisador léxico, as expressões regulares foram convertidas em autômatos finitos equivalentes, representados a seguir, onde:

L = A | B | … | Z | a | … | z  
D = 0 | 1 | .. | 9

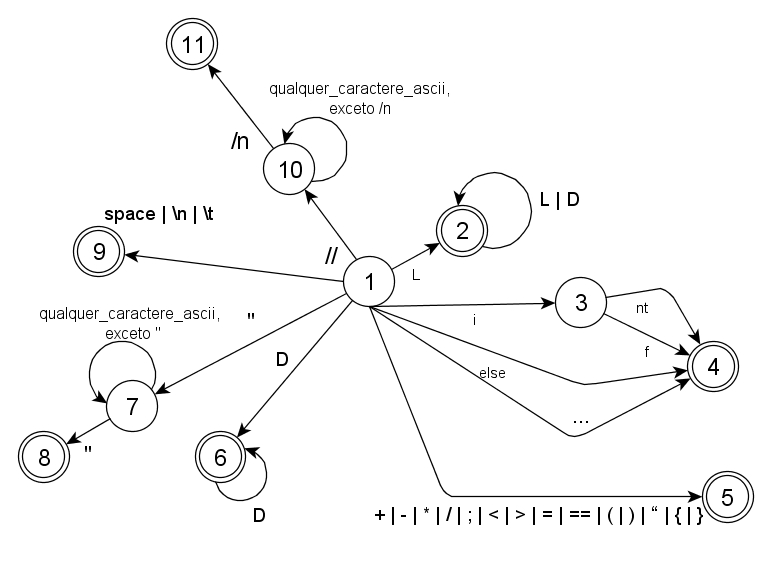






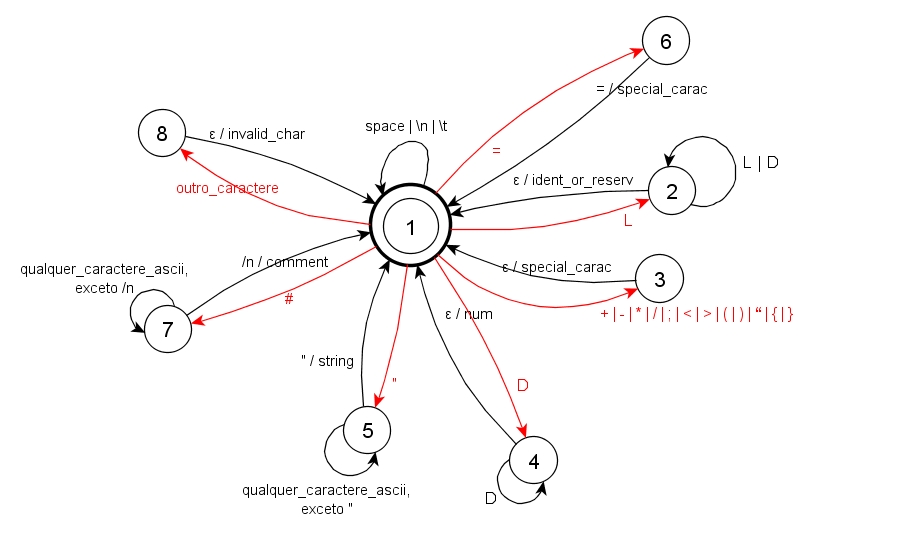


Estes autômatos reconhecem as correspondentes linguagens por eles definidas. Um autômato que reconhece todas essas linguagens, partindo de um mesmo estado inicial, mas apresentando um estado diferenciado para cada tipo de átomo é mostrado a seguir.



onde:  
L = A | B | … | Z | a | … | z  
D = 0 | 1 | .. | 9

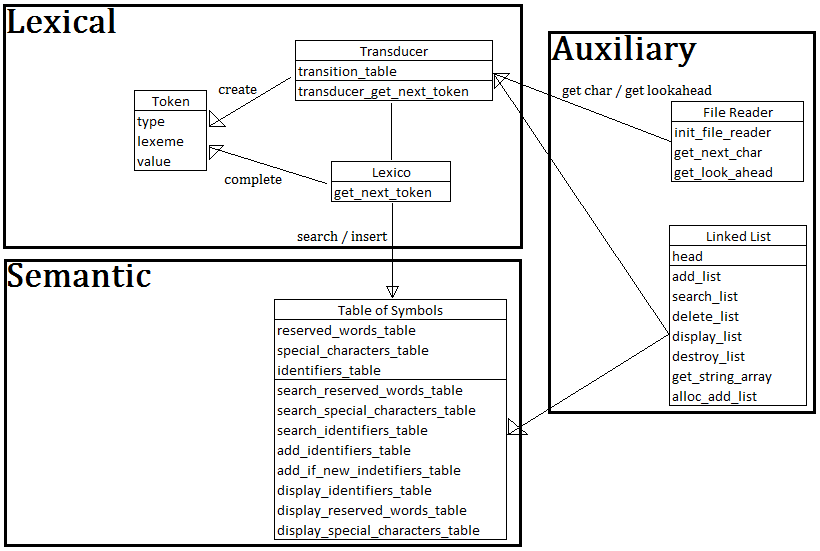
Por fim, o autômato foi convertido em transdutor, que emite como saída o átomo encontrado ao abandonar cada um dos estados finais para iniciar o reconhecimento de mais um átomo do texto.



A partir deste transdutor é possível observar uma simplificação realizada: o átomo **ident\_or\_reserv** representa um átomo que começa com uma letra seguida por letras ou dígitos. Este átomo pode ser do tipo **identificador** ou **palavra\_reservada**, porém o transdutor indefere estes dois tipos, deixando esta tarefa para uma rotina posterior, ainda no analisador léxico. Maiores detalhes da implementação são expostos na próxima seção.

## Implementação

O analisador léxico foi implementado como uma sub-rotina que é chamada pelo analisador sintático sempre que um novo *token* for necessário. A arquitetura básica da implementação do analisador léxico é a que segue:



Os arquivos token.h e token.c representam um *token* na linguagem, contendo seu tipo, lexema (ou seja, o texto que representa o *token*) e seu valor. Neles são definidos todos os tipos possíveis de tokens existentes na linguagem. Vale observar que para alguns tipos (*string, comment, ...*) o atributo *lexeme* é utilizado no lugar do *value*.

Os arquivos lexico.h e lexico.c representam o analisador léxico. Na implementação realizada, o analisador léxico faz requisições de tokens para o transdutor, que retorna um token com um tipo inicial (não necessariamente válido para a linguagem). É então, papel do analisador léxico completar o token retornado com o restante dos dados necessários, ou seja, calcular o valor dos tokens, bem como os tipos exatos de cada um, retornando o valor para quem fez a requisição. Além disso, o analisador também é responsável pelas inserções nas tabelas de identificadores, verificando a existência, ou não, de um identificador, e inserindo apropriadamente.

Os arquivos table\_of\_symbols.h e table\_of\_symbols.c representam as tabelas de símbolos existentes no compilador. Três tabelas de símbolos são utilizadas, a de palavras reservadas, a de caracteres especiais e a de identificadores. Por terem tamanho fixo, as tabelas de palavras reservadas e caracteres especiais foram implementadas através de matrizes, inicializadas com todos os lexemas possíveis. Já a tabela de identificadores, que apresenta tamanho variável, foi implementada através de uma lista ligada. Todas as tabelas apresentam funções de busca, e a tabela de identificadores apresenta também funções para inserção.

Os arquivos file\_reader.c e file\_reader.h implementam rotinas auxiliares de leitura. Dado um arquivo de entrada há funções para acesso ao caractere sendo lido e ao próximo caractere a ser lido, essas funções são get\_next\_char() e get\_look\_ahead(). O arquivo de entrada é determinado ao inicializar o file\_reader através da função void init\_file\_reader(char \*path). Esta inicialização é feita na função *main*.

Os arquivos transducer.c e transducer.h implementam o transdutor mostrado no Capítulo 3.1. Para isto, existe uma matriz de transição (denominada transition\_table) onde as linhas representam os estados atuais e, dado um caractere de entrada (representado pelas colunas), o valor da célula correspondente indica o próximo estado. Os estados são representados por números inteiros e correspondem aos estados da figura do transdutor.

A função transducer\_get\_next\_token() percorre o arquivo de entrada (através de funções auxiliares) e, a cada caractere lido, atualiza seu estado. Quando o transdutor volta para o estado 1 (inicial), ele atualiza uma variável global token preenchendo seu *type* e *lexeme* (o preenchimento do campo *value* e eventual atualização do seu tipo é feito posteriormente por rotinas do léxico). Para possibilitar as transições vazias do transdutor é usado o caractere look\_ahead. Quando, usando o lookahead, é observado que o token incompleto se tornaria inválido, o token é atualizado e o transdutor volta ao estado inicial.

Um teste foi realizado comprovando o correto funcionamento do analisador léxico. A entrada e saída correspondente são mostradas a seguir.

|  |  |
| --- | --- |
| **Entrada\_teste.txt** | |
| int a = 1; int b=2; string s = "olá!?"; # comentário qualquer int a = 2,2;  123 # $ {typedef} | |
| **Saída** | **Tabelas de símbolos** |
| https://lh3.googleusercontent.com/_TEZweKcluFI4g2BBCfnrdtsGSDhFSdF-ra0Wk-bR0iZoNPkP7S5WmP7ICV4PtFPHF7is8QTXrQyL75dyVLdH-QSzMGD4292zl4cZCTfj1shd5r1yO8 tokens reconhecidos no formato: lexeme  TYPE (value) | https://lh5.googleusercontent.com/lUGKRGWhbUoexhajo7gVG4ttRhVqvqhG7lwt45oF9CgOA5txVZZJj_CYOVVMqW4oCjPsLhCctg_f3w7QtauK5uIISnvocKsSYpP_qhH1bQUxuoYE_yU |

# Análise Sintática

A análise sintática é responsável por verificar se o código a ser analisado corresponde à gramática da linguagem-fonte.

Em compiladores orientados por sintaxe, como é o caso do compilador desenvolvido, o analisador sintático é responsável por controlar as atividades do compilador, sendo responsável por efetuar chamadas de funções léxicas para o recebimento de *tokens* conforme os mesmos são consumidos.

A partir dos *tokens* gerados pelo analisador léxico, o analisador sintático analisa a sequência de recebimentos dos *tokens* e monta a árvore sintática.

## Submáquinas do Autômato de Pilha Estruturado (APE)

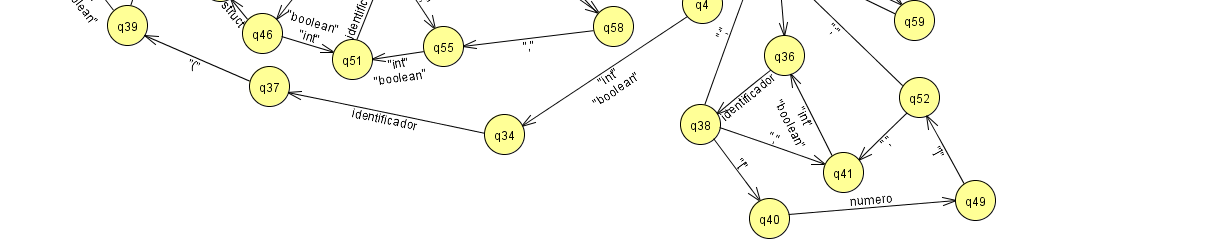
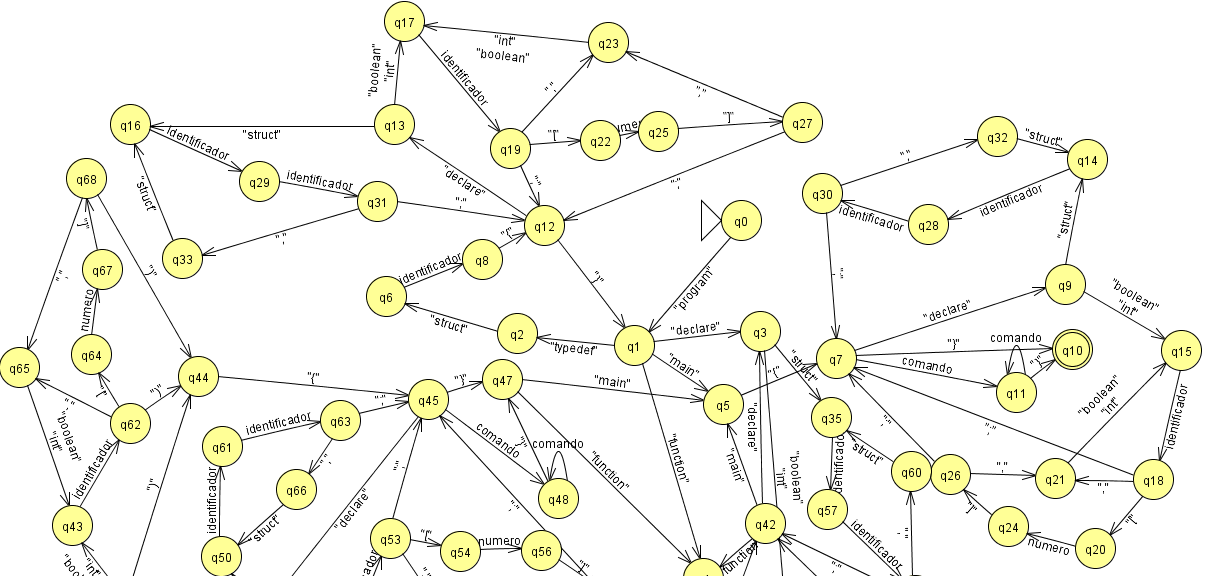
A implementação do analisador sintático do compilador desenvolvido foi baseada no autômato de pilha estruturado (APE) tornando-se necessária a tradução dos três terminais essenciais (PROGRAMA, COMANDO e EXPRESSAO) em autômatos.

Para esta tradução, foi utilizado o gerador de autômatos disponível em: <http://radiant-fire-72.heroku.com/>. Ao inserir descrição reduzida da linguagem na notação Wirth, mostrada no Capítulo 2.3, obteve-se as seguintes listas de transições:

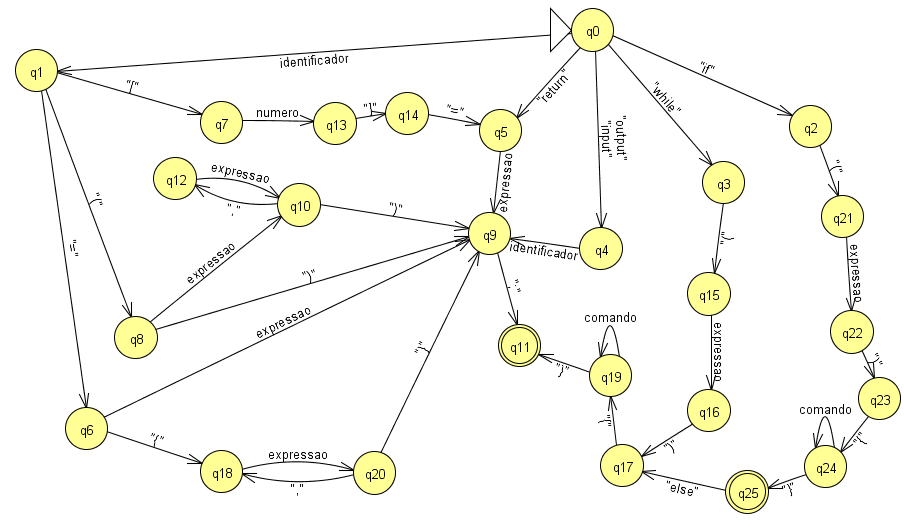
|  |  |  |
| --- | --- | --- |
| **PROGRAMA** | **COMANDO** | **EXPRESSAO** |
| initial: 0  final: 10  (0, "program") -> 1  (1, "typedef") -> 2  (1, "declare") -> 3  (1, "function") -> 4  (1, "main") -> 5  (2, "struct") -> 6  (3, "struct") -> 35  (3, "int") -> 36  (3, "boolean") -> 36  (4, "int") -> 34  (4, "boolean") -> 34  (5, "{") -> 7  (6, identificador) -> 8  (7, "declare") -> 9  (7, "}") -> 10  (7, comando) -> 11  (8, "{") -> 12  (9, "struct") -> 14  (9, "int") -> 15  (9, "boolean") -> 15  (11, "}") -> 10  (11, comando) -> 11  (12, "declare") -> 13  (12, "}") -> 1  (13, "struct") -> 16  (13, "int") -> 17  (13, "boolean") -> 17  (14, identificador) -> 28  (15, identificador) -> 18  (16, identificador) -> 29  (17, identificador) -> 19  (18, "[") -> 20  (18, ",") -> 21  (18, ";") -> 7  (19, "[") -> 22  (19, ",") -> 23  (19, ";") -> 12  (20, numero) -> 24  (21, "int") -> 15  (21, "boolean") -> 15  (22, numero) -> 25  (23, "int") -> 17  (23, "boolean") -> 17  (24, "]") -> 26  (25, "]") -> 27  (26, ",") -> 21  (26, ";") -> 7  (27, ",") -> 23  (27, ";") -> 12  (28, identificador) -> 30  (29, identificador) -> 31  (30, ",") -> 32  (30, ";") -> 7  (31, ",") -> 33  (31, ";") -> 12  (32, "struct") -> 14  (33, "struct") -> 16  (34, identificador) -> 37  (35, identificador) -> 57  (36, identificador) -> 38  (37, "(") -> 39  (38, "[") -> 40  (38, ",") -> 41  (38, ";") -> 42  (39, "int") -> 43  (39, "boolean") -> 43  (39, ")") -> 44  (40, numero) -> 49  (41, "int") -> 36  (41, "boolean") -> 36  (42, "declare") -> 3  (42, "function") -> 4  (42, "main") -> 5  (43, identificador) -> 62  (44, "{") -> 45  (45, "declare") -> 46  (45, "}") -> 47  (45, comando) -> 48  (46, "struct") -> 50  (46, "int") -> 51  (46, "boolean") -> 51  (47, "function") -> 4  (47, "main") -> 5  (48, "}") -> 47  (48, comando) -> 48  (49, "]") -> 52  (50, identificador) -> 61  (51, identificador) -> 53  (52, ",") -> 41  (52, ";") -> 42  (53, "[") -> 54  (53, ",") -> 55  (53, ";") -> 45  (54, numero) -> 56  (55, "int") -> 51  (55, "boolean") -> 51  (56, "]") -> 58  (57, identificador) -> 59  (58, ",") -> 55  (58, ";") -> 45  (59, ",") -> 60  (59, ";") -> 42  (60, "struct") -> 35  (61, identificador) -> 63  (62, "[") -> 64  (62, ",") -> 65  (62, ")") -> 44  (63, ",") -> 66  (63, ";") -> 45  (64, numero) -> 67  (65, "int") -> 43  (65, "boolean") -> 43  (66, "struct") -> 50  (67, "]") -> 68  (68, ",") -> 65  (68, ")") -> 44 | initial: 0  final: 11, 25  (0, identificador) -> 1  (0, "if") -> 2  (0, "while") -> 3  (0, "input") -> 4  (0, "output") -> 4  (0, "return") -> 5  (1, "=") -> 6  (1, "[") -> 7  (1, "(") -> 8  (2, "(") -> 21  (3, "(") -> 15  (4, identificador) -> 9  (5, expressao) -> 9  (6, expressao) -> 9  (6, "{") -> 18  (7, numero) -> 13  (8, expressao) -> 10  (8, ")") -> 9  (9, ";") -> 11  (10, ",") -> 12  (10, ")") -> 9  (12, expressao) -> 10  (13, "]") -> 14  (14, "=") -> 5  (15, expressao) -> 16  (16, ")") -> 17  (17, "{") -> 19  (18, expressao) -> 20  (19, "}") -> 11  (19, comando) -> 19  (20, ",") -> 18  (20, "}") -> 9  (21, expressao) -> 22  (22, ")") -> 23  (23, "{") -> 24  (24, "}") -> 25  (24, comando) -> 24  (25, "else") -> 17 | initial: 0  final: 1, 3, 8  (0, "true") -> 1  (0, "false") -> 1  (0, "not") -> 2  (0, numero) -> 1  (0, identificador) -> 3  (1, ">") -> 4  (1, "<") -> 4  (1, "==") -> 4  (1, "and") -> 4  (1, "or") -> 4  (1, "+") -> 4  (1, "-") -> 4  (1, "\*") -> 4  (1, "/") -> 4  (2, expressao) -> 1  (3, "[") -> 5  (3, ".") -> 6  (3, "(") -> 7  (3, ">") -> 4  (3, "<") -> 4  (3, "==") -> 4  (3, "and") -> 4  (3, "or") -> 4  (3, "+") -> 4  (3, "-") -> 4  (3, "\*") -> 4  (3, "/") -> 4  (4, expressao) -> 8  (5, numero) -> 11  (6, identificador) -> 1  (7, expressao) -> 9  (7, ")") -> 1  (9, ",") -> 10  (9, ")") -> 1  (10, expressao) -> 9  (11, "]") -> 1 |

Para uma melhor visualização dos autômatos, foi utilizado o programa JFLAP, obtendo-se as representações a seguir, onde cada figura representa uma submáquina.

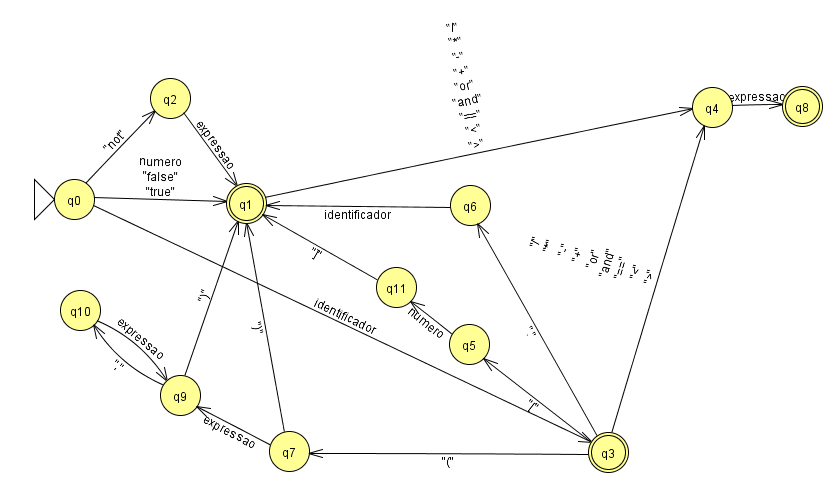
**PROGRAMA**



**COMANDO**



**EXPRESSAO**



## Implementação

O analisador sintático foi implementado como uma rotina do programa principal. A princípio, são inicializados todos os recursos utilizados pela análise sintática (o analisador léxico, tabela de transição e o autômato de pilha estruturado).

Então, a rotina pede o próximo *token* ao analisador léxico e, enquanto não terminar de ler o arquivo ou encontrar um erro na análise sintática, o analisador executa um passo do seu autômato de pilha estruturado. O passo do APE é sua principal função, ilustrada abaixo:

|  |
| --- |
| int spa\_step() {  int machine\_token\_type = spa\_convert\_token\_to\_machine\_type();  if (machine\_token\_type == MTTYPE\_INVALID) return 0;  return transition\_current\_machine\_with\_token(machine\_token\_type);  } |

O autômato de pilha estruturado (APE) utiliza autômatos finitos que representam os três não terminais da linguagem (PROGRAMA, COMANDO, EXPRESSAO). Primeiramente ele converte o *token* retornado pelo analisador sintático, para que ele seja reconhecido nas transições dos autômatos finitos da linguagem.

Então, o passo do APE realiza uma transição da máquina atual com o *token* convertido. Essa transição possui três possibilidades. O primeiro caso consiste na existência de uma transição na máquina corrente do estado atual com o *token* recebido. Esse caso é tratado na função “transition\_to\_next\_state(int next\_state)”:

|  |
| --- |
| void transition\_to\_next\_state (int next\_state) {  actions\_on\_state\_transition[current\_machine.machine\_id]   [current\_machine.current\_state]  [spa\_convert\_token\_to\_machine\_type()](token);  current\_machine.current\_state = next\_state;} |

Nesse caso, o mais simples deles, apenas ocorre uma mudança de estado da máquina atual. Além disso, a transição também desencadeia uma ação semântica, com uma de chamada de função guardada na matriz actions\_on\_state\_transition. As ações semânticas serão descritas com maiores detalhes no capítulo referente à análise semântica.

O segundo caso, é quando a transição de estado é uma chamada de submáquina. Esse caso é tratado na seguinte função:

|  |
| --- |
| void call\_machine(int machine\_type) {  actions\_on\_machine\_transition[current\_machine.machine\_id]  [current\_machine.current\_state]  [machine\_type](token);  current\_machine.current\_state = current\_machine.machine\_calls  [current\_machine.current\_state]  [machine\_type];  spa\_stack\_push(current\_machine, spa\_stack);  current\_machine = machines\_array[machine\_type];  current\_machine.current\_state = 0;  } |

Para que seja possível o correto funcionamento do APE, é necessário guardar, em uma pilha, a máquina atual e o estado no qual ela se encontrará após o retorno da máquina chamada. Para isso, foi implementada uma pilha em “*spa\_stack.h*” e “*spa\_stack.c*”, utilizada pelo APE para empilhar a máquina corrente antes de transicionar para a próxima. Além disso, a transição também desencadeia uma ação semântica.

O último caso existente é aquele em que a o autômato chegou ao seu estado final, mas ainda existem máquinas empilhadas. Nesse caso, o que ocorre é um retorno à máquina anterior, pela função:

|  |
| --- |
| void return\_machine() {  actions\_on\_machine\_return[current\_machine.machine\_id]  [current\_machine.current\_state](token);  current\_machine = spa\_stack\_pop(spa\_stack); } |

Nesse caso, é desencadeada uma ação semântica, e, além disso, retorna-se a máquina que se encontra no topo da pilha, para continuar a execução do APE de maneira adequada.

Todas as informações das máquinas utilizadas pelo APE se encontram nos arquivos “*machines.h*” e “*machines.c*”. Sendo estes modelos das máquinas, com suas tabelas de transição de estados, estados finais e transições que representam chamadas de máquinas.

# Características da MVN

O ambiente em questão consiste da arquitetura do computador-alvo que é, neste caso, a MVN disponibilizada. A MVN (Máquina de von Neumann) simula o Modelo de von Neumann como um processador simples composto pelos seguintes elementos: Memória, Acumulador e Registradores Auxiliares.

Também faz parte do ambiente de execução o Montador disponibilizado. Portanto, a linguagem de saída do compilador desenvolvido não é a linguagem de máquina da MVN, mas sim, uma linguagem simbólica composta por mnemônicos e que lida com rótulos, operandos e sub-rotinas para endereçamento dentro de um programa.

Esta linguagem simbólica será descrita nos itens a seguir.

## Instruções da linguagem de saída

|  |  |  |  |
| --- | --- | --- | --- |
| **Operação** | **Mnemônico** | **Operando** | **Descrição** |
| Jump | JP | Endereço/Rótulo de desvio | Desvio incondicional |
| Jump if Zero | JZ | Endereço/Rótulo de desvio | Desvio se valor no acumulador é zero |
| Jump if Negative | JN | Endereço/Rótulo de desvio | Desvio se valor no acumulador é negativo |
| Load Value | LV | Constante de 12 bits | Deposita uma constante no acumulador |
| Add | + | Endereço/Rótulo do operando | Soma o conteúdo do acumulador com o operando |
| Subtract | - | Endereço/Rótulo do subtraendo | Subtração do conteúdo do acumulado com o subtraendo |
| Multiply | \* | Endereço/Rótulo do multiplicador | Multiplicação do conteúdo do acumulado com o multiplicador |
| Divide | / | Endereço/Rótulo do divisor | Divisão do conteúdo do acumulador com o divisor |
| Load | LD | Endereço/Rótulo do dado | Copia valor contido no endereço de memória para acumulador |
| Move to Memory | MM | Endereço/Rótulo de destino do dado | Copia valor do acumulador para a memória |
| Subroutine Call | SC | Endereço//Rótulo do subprograma | Desvio para subprograma |
| Return from Subroutine | RS | Endereço/Rótulo de retorno | Retorno de subprograma |
| Halt Machine | HM | Endereço/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 |

## Pseudoinstruções da linguagem de saída

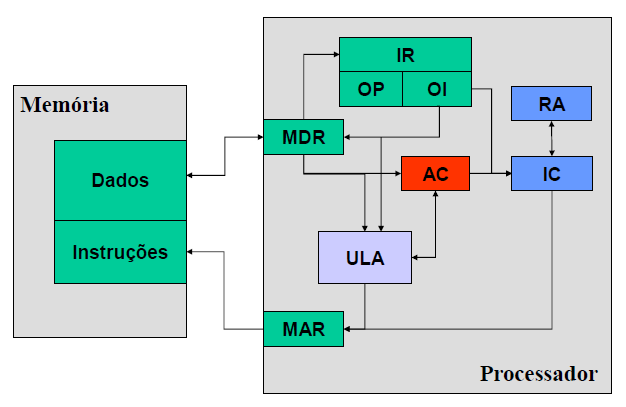
|  |  |
| --- | --- |
| **Pseudoinstrução** | **Descrição** |
| @ | Recebe um operando numérico, define o endereço da instrução seguinte, uma origem absoluta para o código a ser gerado |
| K | Define área preenchida por uma constante, o operando numérico tem o valor da constante de 2 bytes (em hexadecimal) |
| $ | Define um bloco de memória com número especificado de bytes, o operando numérico define o tamanho da área a ser reservada (em bytes) |
| # | 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 corrente) |
| > | Define endereço simbólico de entrada (Entry Point) |
| < | Define um endereço simbólico que referencia um entry-point externo |

## Características gerais

O ambiente de execução é composto pela simulação de um processador muito simples. Esse simulador apresenta um conjunto de elementos de armazenamento e dados, são eles: memória principal, acumulador e registradores auxiliares.

Primeiramente, na memória principal são armazenados as instruções dos programas e os dados utilizados por eles. Já o acumulador é um registrador especial que é utilizado para a realização das operações aritméticas e lógicas, por exemplo, a realização da instrução “+ VALOR” pode ser interpretada como “some o conteúdo indicado pelo rótulo ‘VALOR’ ao acumulador”.

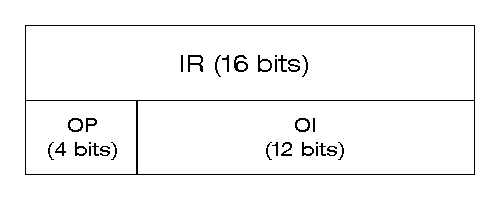
Por fim, os registradores auxiliares são utilizados em operações intermediárias. A tabela abaixo apresenta uma relação dos registradores auxiliares da MVN e suas funções. A arquitetura do simulador de MVN utilizado como ambiente de execução está representada na figura a seguir.



Por fim, os registradores auxiliares são utilizados em operações intermediárias. A tabela abaixo apresenta uma relação dos registradores auxiliares da MVN e suas funções.

|  |  |
| --- | --- |
| **Registrador Auxiliar** | **Utilização** |
| Registrador de Dados da Memória (MDR) | Utilizado para tráfego de dados entre a memória e outros elementos da MVN |
| Registrador de Endereço de Memória (MAR) | Contém a origem ou destino dos dados que se encontram no MDR |
| Registrador de Endereço de Instrução (IC) | Armazena a próxima instrução a ser executada pela máquina |
| Registrador de Instrução (IR) | Representa a instrução em execução, é composto de duas parcelas: o código de operação (OP) e o operando da instrução (OI). |

O registrador de Instrução (IR) da MVN armazena a instrução em execução corrente. A parcela OP é composta por 4 bits que codificam a instrução já a parcela OI complementa a instrução com o dado ou endereço parâmetro da instrução.



A memória possui capacidade de 4K de armazenamento e endereços de 12 bits. O acesso a variáveis é feito diretamente pelo endereço de memória, aleatoriamente. E a codificação numérica é feita em complemento de dois, com o bit mais significativo sendo indicador de sinal.

Duas instruções serão descritas em maiores detalhes para entender melhor o funcionamento do simulador, a de chamada de subrotina e de retorno de subrotina.

### Chamada de subrotina

Para realização da chamada de (ou desvio para) subrotina é necessário armazenar o endereço da próxima instrução que seria executada antes do desvio, a fim de manter a ordem de execução do programa.

Dessa forma o conteúdo do registrador de instrução (IC) é armazenado no endereço de memória passado como operando da instrução de chamada de subrotina, salvando a posição de retorno da subrotina.

Então, é colocado no IC o endereço apontado pelo operando da instrução incrementado de um. Fazendo assim o desvio de execução do programa para o início da execução da subrotina.

### Retorno de subrotina

Devido ao método como foi implementada na MVN a chamada de subrotina, o retorno da subrotina se dá simplesmente trocando o conteúdo do registrador de instrução (IC) com o conteúdo da posição de memória apontada pelo operando da instrução.

# Tradução de Comandos

Antes da implementação do analisador semântico, traduções de algumas estruturas e comandos foram feitas de forma a definir um padrão e auxiliar o desenvolvimento das ações semânticas. Tais traduções serão expostas neste capítulo.

## **Controle de fluxo**

**If-then**

|  |  |
| --- | --- |
| **Linguagem de entrada** | **Linguagem de montagem** |
| if (<expressão>) {    <acoes> } | <expressão>   JN endif;   JZ endif;   <acoes> endif |
| if (a < b) { } | LD zero ;   - a ;   + b ;   JN endif;   JZ endif; endif |

**If-then-else**

|  |  |
| --- | --- |
| Linguagem de entrada | Linguagem de montagem |
| if (<expressão>) {    <comandos> } else {    <comandos> } | <expressão>   JN endif;   JZ endif;   <comandos>   JP endelse ; endif   <comandos> endelse |
| if (a < b) { } else { }   |  | | --- | |  | | LD zero ;   - a ;   + b ;    JN endif ;    JZ endif ;    JP endelse ; endif endelse |

**While**

|  |  |
| --- | --- |
| **Linguagem de entrada** | **Linguagem de montagem** |
| while ( <expressao> ) {    <comandos> } | while   <expressao>        JN endwhile ;        JZ endwhile ;        <comandos>        JP while ; endwhile |
| while ( a < b ) { } | while   LD zero ;        - a ;        + b ;        JN endwhile ;        JZ endwhile ;        JP while ; endwhile |

## Comandos imperativos

**Atribuição de valor**

|  |  |
| --- | --- |
| **Linguagem de entrada** | **Linguagem de montagem** |
| a = <expressao>; | <expressao> MM a ; |
| a = b | LD b ; MM a ; |

**Impressão (saída)**

|  |  |
| --- | --- |
| **Linguagem de entrada** | **Linguagem de montagem** |
| output a; | LD a ;    MM output\_number ;    SC output ; |
| **Subrotina de Output** | |
| output\_number     K     =0    ; Número para ser impresso FFFF    K     /FFFF   ; Base da representação minus\_sign    K     =45    ; Sinal de menos em ASCII ascii\_offset     K   =48    ; Offset para o código de um número na tabela ASCII o\_temp1    K =0    ; Guarda o valor da última dezena o\_temp2    K     =1    ; Indicador da dezena  output    JP    /0000    ;    LD    ten    ; Inicialização    MM    o\_temp2    ;    LD    output\_number    ;    JN    o\_negative    ; Número negativo    JP o\_start    ; Número posítivo o\_negative  LD    minus\_sign    ; Caso contrário imprime "-"    PD   /0100    ;    LD   FFFF    ; E inverte o número    - output\_number    ;    +    one    ;    MM   output\_number    ; o\_start    MM   o\_temp1    ; o\_loop    LD   output\_number    ; Carrega o número    /    o\_temp2    ;    JZ   o\_print    ; Imprime se é o número mais a esquerda.    MM   o\_temp1    ; Se não, guarda o número    LD   o\_temp2    ; Aumenta a casa decimal    \*    ten    ;    MM   o\_temp2    ;    JP   o\_loop    ; E volta para o loop o\_print    LD   o\_temp1    ; Impressão do número    +    ascii\_offset    ;    PD   /0100    ;    LD   o\_temp2    ; Verifica se é o último número    /    ten    ;    MM   o\_temp2    ;    -    one    ;    JZ   o\_end    ; Vai para o final, se imprimiu tudo    LD   o\_temp1    ;    \*    o\_temp2    ;    MM   o\_temp1    ;    LD   output\_number    ; Atualiza o número para impressão    -    o\_temp1    ;    MM   output\_number    ;    MM   o\_temp1    ;    LD   ten    ;    MM   o\_temp2    ;    JP   o\_loop    ; Imprime o próximo caractere  o\_end    RS   output    ; Final da rotina | |

**Leitura (entrada)**

|  |  |
| --- | --- |
| **Linguagem de entrada** | **Linguagem de montagem** |
| input a; | SC input ;   MM a ; |
| **Subrotina de Input** | |
| input\_number K =0 ; Variável de retorno da rotina  i\_negative K =0 ; Número digitado é negativo  i\_temp K =0 ; Variáveis temporárias  i\_temp2 K =0 ;  input JP /0000 ;  LD zero ; Inicialização  MM input\_number ;  MM i\_negative ;  MM i\_temp ;  MM i\_temp2 ;  GD /0000 ; Leitura de número negativo  MM i\_temp ; Guarda caracteres lidos em i\_temp  / sixteen ; Obtém o primeiro caracter  / sixteen ;  MM i\_temp2 ;  - ascii\_minus ; Verifica se número digitado é negativo  JZ set\_i\_negative ;  JP 1st\_char ; Não é negativo, lê como número  set\_i\_negative  LD one ; Carrega o i\_negative com FFFF  MM i\_negative ;  JP 2nd\_char ; Lê o segundo caracter  i\_loop GD /0000 ; Loop de leitura  MM i\_temp ; Guarda caracteres lidos em i\_temp  / sixteen ; Obtém o primeiro caracter  / sixteen ;  MM i\_temp2 ;  1st\_char - ascii\_cr ; Verifica se é o fim (ascii \d ou \a)  JZ i\_end ;  LD i\_temp2 ;  - ascii\_lf ;  JZ i\_end ;  LD input\_number ; Não é o último caracter  \* ten ; Aumenta uma dezena no resultado  MM input\_number ;  LD i\_temp2 ; Converte caracter lido em número  - ascii\_offset ;  + input\_number ; Soma no resultado de retorno  MM input\_number ; Atualiza o resultado de retorno  2nd\_char  LD i\_temp2 ; Obtém o segundo caracter  \* sixteen ;  \* sixteen ;  MM i\_temp2 ;  LD i\_temp ;  - i\_temp2 ;  MM i\_temp2 ;  - ascii\_cr ; Verifica se é o fim (ascii \d ou \a)  JZ i\_end ;  LD i\_temp2 ;  - ascii\_lf ;  JZ i\_end ;  LD input\_number ; Não é o último caracter  \* ten ; Aumenta uma dezena no resultado  MM input\_number ;  LD i\_temp2 ; Converte caracter lido em número  - ascii\_offset ;  + input\_number ; Soma no resultado de retorno  MM input\_number ; Atualiza o resultado de retorno  JP i\_loop ; Lê o proximo caracter  i\_end LD i\_negative ; Transforma em negativo se negativo  JZ i\_return ;  LD zero ;  - input\_number ;  MM input\_number ;  i\_return  LD input\_number ;  RS input ; | |

**Chamada de subrotina**

|  |  |
| --- | --- |
| **Linguagem de entrada** | **Linguagem de montagem** |
| funcao(a); | LD tam\_func ; Carrega o tamanho do R.A.    MM tam\_registro\_ativacao  LD end\_retorno ; Carrega o endreço de retorno do R.A.  MM end\_registro\_ativacao    SC cria\_registro\_ativacao ; Cria R.A.     LD 1 ; Carrega parâmetro da função    MM pos\_param\_1  SC store\_ra\_pos     SC função ; Executa a função |
| **Subrotina do ambiente de execução** | |
| load\_inst LD /0000 ; Instrução para o acesso indireto  store\_inst MM /0000 ; Instrução para store indireto  pos\_param K =0 ; Posição do parâmetro da função    load\_ra\_pos JP /0000 ; Ponto de entrada da subrotina  LD STOP ; Carrega topo da pilha do R.A.  - two ; Diminui um endereço na pilha do R.A.  - pos\_param ; Accumulador com o endereço correto  + load\_inst ; Here's the magic: Cria instrução nova!  MM hack ; Armazena como a PROXIMA INSTRUCAO!  hack K /0 ; Reservado para guardar a instrução recém-montada  RS load\_ra\_pos ; Thanks to Débora for this piece of gold  store\_ra\_pos JP /0000 ;  LD STOP ; Carrega topo da pilha do R.A.  - two ; Diminui um endereço na pilha do R.A.  - pos\_param ; Accumulador com o endereço correto  + store\_inst ; Here's the magic: Cria instrução nova!  MM hack2 ; Armazena como a PROXIMA INSTRUCAO!  hack2 K /0 ; Reservado para guardar a instrução recém-montada  RS store\_ra\_pos ;  ra\_tam K =0 ;  ra\_end K =0400 ;    cria\_ra JP /0000 ;  LD STOP ;  + two ;  MM STOP ;  LD zero ;  MM pos\_param ;  LD ra\_end ;  SC store\_ra\_pos ;  LD STOP ;  + ra\_tam ;  MM STOP ;  RS cria\_ra ; | |

## Exemplo de programa traduzido

|  |
| --- |
| program  main {  declare int num, int fat;  input num;  output num;  if(num < 0) {  fat = 0;  }  else {  fat = 1;  while(num > 1) {  fat = fat \* num;  num = num - 1;  }  }  output fat;  } |
| ; Rotinas do ambiente de execução ;  ; . ;  ; . ;  ; . ;  @ /0200  main JP /0000 ;  SC input ; Comando de input  MM V0 ;  LD V0 ; Comando de output  MM output\_number ;  SC output ;  I0 LD zero ; Begin if case  LD K0 ; Comparacao X < Y  - V0 ;  MM T0 ;  LD T0 ;  JN \_I0 ;  JZ \_I0 ;  LD K1 ; Atribuicao de variavel  MM V1 ;  JP E0 ;  \_I0 LD zero ; End if case/Begin else case  LD K2 ; Atribuicao de variavel  MM V1 ;  L0 LD zero ; Begin while loop  LD V0 ; Comparacao X > Y  - K2 ;  MM T1 ;  LD T1 ;  JN \_L0 ;  JZ \_L0 ;  LD V1 ;  \* V0 ;  MM T2 ;  LD T2 ; Atribuicao de variavel  MM V1 ;  LD V0 ;  - K3 ;  MM T3 ;  LD T3 ; Atribuicao de variavel  MM V0 ;  JP L0 ;  \_L0 LD zero ; End while loop  E0 LD zero ; End else case  LD V1 ; Comando de output  MM output\_number ;  SC output ;  HM /00 ;  # P ;  @ /0A00  V0 K =0 ; Declaracao de variavel  K0 K =0 ; Declaracao de constante  T0 K =0 ; Declaracao de temporario  V1 K =0 ; Declaracao de variavel  K1 K =0 ; Declaracao de constante  K2 K =1 ; Declaracao de constante  T1 K =0 ; Declaracao de temporario  T2 K =0 ; Declaracao de temporario  K3 K =1 ; Declaracao de constante  T3 K =0 ; Declaracao de temporário |

# Análise Semântica

A análise semântica do compilador desenvolvido consiste nas ações semânticas desencadeadas pelas transições do APE. As ações semânticas rsão responsáveis por algumas tarefas como impressão de código objeto no arquivo de saída conforme os *tokens* são lidos e o APE é transicionado, além disso, porém nem toda ação semântica tem como resultado a geração de código, por vezes, sua função é apenas salvar o que foi lido em alguma estrutura de dados para executar geração de código posteriormente.

Para a escrita do código, foram criados dois *buffers* de armazenamento, um correspondente a área de código e outro correspondente a área de dados. Os *buffers* foram implementados através da criação de dois arquivos de texto separados. Ao fim da compilação, os dois arquivos são integrados, gerando um único arquivo de saída.

Além disso, para que fosse possível o gerenciamento de escopo, a tabela de símbolos foi modificada. Por fim, foi necessário um registro de ativação, utilizado nas chamadas de funções para que o ambiente de execução seja guardado, evitando conflitos e garantindo o correto funcionamento dos programas.

## Tabela de símbolos com suporte a escopo

O gerenciamento de escopos foi implementado de forma que a cada novo escopo uma nova tabela de identificadores é criada. Além disso, cada tabela de identificadores possui um apontador para a tabela referente ao escopo "pai".

Quando se deseja criar um novo escopo a função abaixo, presente no arquivo table\_of\_symbols, é chamada.

|  |
| --- |
| **void** **enter\_new\_scope**() {  List \*newTable = empty\_list();  newTable->parent = identifiers\_table;  identifiers\_table = newTable;  } |

Ela cria uma nova lista ligada representando a nova tabela, associa esta nova tabela com a tabela atual e atualiza a variável global identifiers\_table para que esta aponte para a tabela recém-criada.

De forma análoga, a função exit\_current\_scope() descarta a tabela de identificadores atual e aponta a variável global para a tabela pai.

|  |
| --- |
| **void** **exit\_current\_scope**() {  List \*current\_table = identifiers\_table;  identifiers\_table = current\_table->parent;  **free**(current\_table);  } |

A busca por identificador também foi modificada de forma que ele seja procurado não só na tabela atual, mas também nas tabelas relacionadas, conforme a rotina a seguir.

|  |
| --- |
| Node \* **get\_identifier\_for\_data\_on\_all\_tables**(**char** \* data, List \* identifiers\_table) {  **if**(identifiers\_table == NULL) **return** NULL;  **int** index = search\_list(data, identifiers\_table);  **if**(index != INDEX\_NOT\_FOUND) {  Node \* identifier = get\_node\_at\_index(index, identifiers\_table);  **if**(identifier->wasDeclared) **return** identifier;  }  **return** get\_identifier\_for\_data\_on\_all\_tables(data,  identifiers\_table->parent);  } |

Com esta modificação, o compilador desenvolvido permite programas com variáveis de mesmo nome, desde que em escopos distintos.

## Principais ações semânticas

Nesta seção serão descritas algumas das ações semânticas desenvolvidas. As mesmas se encontram no arquivo semantic\_actions.c e são chamadas através de consultas às matrizes actions\_on\_state\_transition (contendo as ações desencadeadas nas trasições de estado), actions\_on\_machine\_transition (contendo as ações desencadeadas nas chamadas de máquinas) e actions\_on\_machine\_return (contendo as ações desencadeadas nos retornos de máquinas). Estas matrizes mapeiam as transições do APE com as ações semânticas correspondentes. O mapeamento, ou inicialização das tabelas, é feito com a chamada da função init\_semantic\_actions.

**dummy\_semantic\_action**

Esta ação semântica não gera código nem modifica variáveis. É uma ação executada em todas as transições do APE que não tem ações definidas, ou não devem executar qualquer ação.

### Rótulos

**get\_X\_label**

As funções do tipo **get\_X\_label**, retornam um rótulo utilizado para uma determinada função no compilador. Por exemplo, a função **get\_constant\_label** retorna um rótulo para uma constante e incrementa o contador de constantes utilizado na geração destes rótulos.

Outras funções que apresentam essa mesma finalidade são: **get\_temp\_label, get\_temp\_label\_label, get\_variable\_label, get\_loop\_label, get\_if\_label, get\_else\_label, get\_func\_label.**

Essas funções também são responsáveis por escrever no buffer de dados as declarações dessas variáveis, quando necessário.

### Submáquina Programa

**print\_main**

Ação disparada na transição da submáquina PROGRAMA, no estado 1 com o token do tipo "main". Imprime no buffer de código a inicialização da MVN, com o comando JP \000 identificado pelo rótulo main.

**end\_program**

Ação disparada nas transições que levam ao estado final da submáquina PROGRAMA. Imprime no buffer de código os comandos HM /00 ; e # P ;

**declare\_variable**

Ação disparada nas transições da submáquina PROGRAMA ao receber um identificador estando nos estados 15, 17, 36 ou 51. Caso a variável já tenha sido declarada no escopo, lança a exceção semântica ERR\_VARIABLE\_REDECLARED. Caso contrário, marca seu identificador na tabela de símbolos como declarada.

**throw\_boolean\_exception**

Ação disparada ao receber o token "boolean", pois, apesar de previsto na linguagem definida, optou-se por não suportar o tipo de dados boolean no compilador desenvolvido.

### Submáquina Comando

**push\_control\_command**

Esta ação permite rotinas de controle de fluxo aninhadas. Ela é disparada nos estados 0 e 25 da submáquina COMANDO ao receber *tokens* referentes aos comandos if, else e while. Além de solicitar um rótulo, a ação empilha o comando e seu rótulo nas pilhas command\_operator\_stack e command\_operand\_stack, respectivamente. Sua implementação está representada a seguir:

|  |
| --- |
| void push\_control\_command(Token \*token) {  char \* command = token->lexeme;  char \* label;  if(strcmp(command, "while") == 0) {  label = get\_loop\_label();  sprintf(buffer, "%s\t\t\tLD zero\t; Begin while loop\n", label);  write\_to\_code(buffer);  } else if (strcmp(command, "if") == 0) {  label = get\_if\_label();  sprintf(buffer, "%s\t\t\tLD zero\t; Begin if case\n", label);  write\_to\_code(buffer);  } else if (strcmp(command, "else") == 0) {  label = get\_else\_label();  }  stack\_push(command\_operator\_stack, command);  stack\_push(command\_operand\_stack, label);  } |

**resolve\_command**

Esta ação é chamada no retorno da submáquina COMANDO e nos estados 17 e 23 ao receber o token MTTYPE\_LEFT\_CURLY\_BRACKET. Ela desempilha o comando da pilha command\_operator\_stack e dispara diferentes ações, dependendo do comando desempilhado.

|  |
| --- |
| **void** **resolve\_command**(Token \*token) {  **char** \* command = stack\_pop(command\_operator\_stack);  **if**(**strcmp**(command, "=") == 0) resolve\_assign();  **else** **if**(**strcmp**(command, "output") == 0) resolve\_output();  **else** **if**(**strcmp**(command, "input") == 0) resolve\_input();  **else** **if**(**strcmp**(command, "while") == 0) resolve\_while();  **else** **if**(**strcmp**(command, "endwhile") == 0) resolve\_end\_while();  **else** **if**(**strcmp**(command, "if") == 0) resolve\_if();  **else** **if**(**strcmp**(command, "endif") == 0) resolve\_end\_if();  **else** **if**(**strcmp**(command, "else") == 0) resolve\_else();  **else** **if**(**strcmp**(command, "endelse") == 0) resolve\_end\_else();  } |

**resolve\_while**

Esta ação é responsável por criar um novo escopo e empilhar o comando "endwhile" na pilha command\_operator\_stack, além de escrever no buffer de código usando o rótulo presente no topo da pilha command\_operand\_stack.

**resolve\_end\_while**

Esta ação desempilha o rótulo do while em questão da pilha command\_operand\_stack e, usando este rótulo, escreve no buffer de código. Além disso, encerra o escopo corrente.

**resolve\_if**

Esta ação é responsável por criar um novo escopo e empilhar o comando "endif" na pilha command\_operator\_stack, além disso, escrever o código em MVN da função condicional no buffer de código usando o rótulo presente no topo da pilha command\_operand\_stack.

**resolve\_end\_if**

Esta ação desempilha o rótulo do comando condicional em questão da pilha command\_operand\_stack e, usando este rótulo, escreve no buffer de código o código de término correspondente. Além disso, encerra o escopo corrente.

**resolve\_else**

Esta ação é responsável por criar um novo escopo e empilhar o comando "endelse" na pilha command\_operator\_stack, além disso, escrever o código em MVN da função condicional no buffer de código usando o rótulo presente no topo da pilha command\_operand\_stack.

**resolve\_end\_else**

Esta ação desempilha o rótulo do else em questão da pilha command\_operand\_stack e, usando este rótulo, escreve no buffer de código. Além disso, encerra o escopo corrente.

### Submáquina Expressão

**push\_identifier**

Esta ação é disparada nas transições do APE a partir dos estados 0 da submáquina expressão ao receber um identificador. É feita uma verificação de declaração prévia do identificador e empilha-se o rótulo do identificador na pilha operand\_stack.

**push\_operator**

Esta ação é disparada nas transições do APE a partir dos estados 1 e 3 da submáquina expressão, ao receber um operador. Se a precedência do operador no topo da pilha operator\_stack for maior que a precedência do operador recebido, a expressão é resolvida (através da ação resolve\_expression) e a ação é repetida para o token recebido. Caso contrário, o operador é empilhado.

Nos casos em que o operador é o "and" ou "or", simplesmente chama-se a ação resolve\_expression.

**resolve \_expression**

Esta ação, similar à resolve\_command desempilha o operador da pilha operator\_stack e dispara diferentes ações, dependendo do comando desempilhado, conforme representado a seguir.

|  |
| --- |
| void resolve\_expression() {  char \* o = stack\_pop(operator\_stack);  if(strcmp(o, ">") == 0) resolve\_compare\_greater\_than();  else if(strcmp(o, "<") == 0) resolve\_compare\_less\_than();  else if(strcmp(o, "==") == 0) resolve\_compare\_equal\_equal();  else if(strcmp(o, "and") == 0) resolve\_logic\_and();  else if(strcmp(o, "\_and") == 0) end\_logic\_and();  else if(strcmp(o, "or") == 0) resolve\_logic\_or();  else if(strcmp(o, "\_or") == 0) end\_logic\_or();  else if(strcmp(o, "not") == 0) resolve\_logic\_not();  else resolve\_arithmetic(o);  } |

**resolve \_compare\_equal\_equal**

Esta ação, desempilha os dois operandos da pilha operand\_stack (empilhados previamentes na ação semântica push\_operand) e, usando variáveis temporárias, escreve imprime o código referente à comparação no buffer de código.

## Ambiente de execução

Para as rotinas de input e output, além de ações semânticas, foram desenvolvidas subrotinas escritas na linguagem de montador da SVM, as subrotinas foram expostas no Capítulo 6.2.

Para suporte de chamadas de função, foram desenvolvidas rotinas para implementação do registro de ativação, que também foram expostas no Capítulo 6.2.

Todas as rotinas referentes ao ambiente de execução estão presentes no arquivo execution\_environment.asm, cujo conteúdo é inteiramente copiado para o código objeto gerado pelo compilador. Desta forma, ao utilizar os rótulos (por exemplo, output) no código gerado pelas ações semânticas, as rotinas são acessadas na MVN.

# Testes

Os testes representados a seguir mostram o correto funcionamento das diversas funcionalidades do compilador desenvolvido.

### Teste 1: input, output, if e while aninhados

|  |  |
| --- | --- |
| **Código fonte** | **Código objeto gerado pelo compilador** |
| program  main {  declare int a;  declare int b;  declare int c;  input a;  b = 5;  c = 8;  if (a == b) {  output c;  } else {  while (a < b) {  a = a+1;  output a;  if (a == b) {  output c;  }  }  }  } | <código das sub-rotinas do ambiente de execução omitido>  main JP /0000 ;  SC input ; Comando de input  MM V0 ;  LD K0 ; Atribuicao de variavel  MM V1 ;  LD K1 ; Atribuicao de variavel  MM V2 ;  I0 LD zero ; Begin if case  LD V0 ; Comparacao X == Y  - V1 ;  JZ TL0 ;  LD zero ; Nao e igual  JP TL1 ;  TL0 LD one ; E igual  TL1 MM T0 ;  LD T0 ;  JN \_I0 ;  JZ \_I0 ;  LD V2 ; Comando de output  MM output\_number ;  SC output ;  JP E0 ;  \_I0 LD zero ; End if case/Begin else case  L0 LD zero ; Begin while loop  LD V1 ; Comparacao X < Y  - V0 ;  MM T1 ;  LD T1 ;  JN \_L0 ;  JZ \_L0 ;  LD V0 ;  + K2 ;  MM T2 ;  LD T2 ; Atribuicao de variavel  MM V0 ;  LD V0 ; Comando de output  MM output\_number ;  SC output ;  I1 LD zero ; Begin if case  LD V0 ; Comparacao X == Y  - V1 ;  JZ TL2 ;  LD zero ; Nao e igual  JP TL3 ;  TL2 LD one ; E igual  TL3 MM T3 ;  LD T3 ;  JN \_I1 ;  JZ \_I1 ;  LD V2 ; Comando de output  MM output\_number ;  SC output ;  \_I1 LD zero ; End if case  JP L0 ;  \_L0 LD zero ; End while loop  E0 LD zero ; End else case  HM /00 ;  # P ;  @ /0A00  V0 K =0 ; Declaracao de variavel  V1 K =0 ; Declaracao de variavel  K0 K =5 ; Declaracao de constante  V2 K =0 ; Declaracao de variavel  K1 K =8 ; Declaracao de constante  T0 K =0 ; Declaracao de temporario  T1 K =0 ; Declaracao de temporario  K2 K =1 ; Declaracao de constante  T2 K =0 ; Declaracao de temporario  T3 K =0 ; Declaracao de temporario |
| **Execução na MVN** | |
|  | |

### Teste 2: operações aritméticas

|  |  |
| --- | --- |
| **Código fonte** | **Código objeto gerado pelo compilador** |
| program  main {  declare int a;  declare int b;  declare int c;  input a;  input b;  c = a + b;  output c;  c = a - b;  output c;  c = a \* b;  output c;  c = a / b;  output c;  } | <código das sub-rotinas do ambiente de execução omitido>  main JP /0000 ;  SC input ; Comando de input  MM V0 ;  SC input ; Comando de input  MM V1 ;  LD V0 ;  + V1 ;  MM T0 ;  LD T0 ; Atribuicao de variavel  MM V2 ;  LD V2 ; Comando de output  MM output\_number;  SC output ;  LD V0 ;  - V1 ;  MM T1 ;  LD T1 ; Atribuicao de variavel  MM V2 ;  LD V2 ; Comando de output  MM output\_number;  SC output;  LD V0 ;  \* V1 ;  MM T2 ;  LD T2 ; Atribuicao de variavel  MM V2 ;  LD V2 ; Comando de output  MM output\_number;  SC output ;  LD V0 ;  / V1 ;  MM T3 ;  LD T3 ; Atribuicao de variavel  MM V2 ;  LD V2 ; Comando de output  MM output\_number;  SC output  HM /00 ;  # P ;  @ /0A00  V0 K =0 ; Declaracao de variavel  V1 K =0 ; Declaracao de variavel  V2 K =0 ; Declaracao de variavel  T0 K =0 ; Declaracao de temporario  T1 K =0 ; Declaracao de temporario  T2 K =0 ; Declaracao de temporario  T3 K =0 ; Declaracao de temporário |
| **Execução na MVN** | |
|  | |

### Teste 3: operações lógicas

|  |  |
| --- | --- |
| **Código fonte** | **Código objeto gerado pelo compilador** |
| program  main {  declare int a;  a = 1 and 1;  output a;  a = 1 and 0;  output a;  a = 0 and 0;  output a;  a = 1 or 0;  output a;  a = 0 or 0;  output a;  } | <código das sub-rotinas do ambiente de execução omitido>  main JP /0000 ;  LD K0 ; Comeco do and logico  JZ TL0 ; Returns NO  JN TL0 ; Returns NO  LD K0 ;  JZ TL0 ; Returns NO  JN TL0 ; Returns NO  LD one ;  JP TL1 ; Returns YES  TL0 LD zero ;  TL1 MM T0 ; Fim do and logico  LD T0 ; Atribuicao de variavel  MM V0 ;  LD V0 ; Comando de output  MM output\_number ;  SC output ;  LD K0 ; Comeco do and logico  JZ TL2 ; Returns NO  JN TL2 ; Returns NO  LD K1 ;  JZ TL2 ; Returns NO  JN TL2 ; Returns NO  LD one ;  JP TL3 ; Returns YES  TL2 LD zero ;  TL3 MM T1 ; Fim do and logico  LD T1 ; Atribuicao de variavel  MM V0 ;  LD V0 ; Comando de output  MM output\_number ;  SC output ;  LD K1 ; Comeco do and logico  JZ TL4 ; Returns NO  JN TL4 ; Returns NO  LD K1 ;  JZ TL4 ; Returns NO  JN TL4 ; Returns NO  LD one ;  JP TL5 ; Returns YES  TL4 LD zero ;  TL5 MM T2 ; Fim do and logico  LD T2 ; Atribuicao de variavel  MM V0 ;  LD V0 ; Comando de output  MM output\_number ;  SC output ;  LD K0 ; Comeco do or logico  JZ TL6 ;  JN TL6 ;  JP TL7 ; Returns YES  TL6 LD K1 ;  JZ TL8 ;  JN TL8 ;  JP TL7 ; Returns YES  TL8 LD zero ;  JP TL9 ; Returns NO  TL7 LD one ;  TL9 MM T3 ; Fim do or logico  LD T3 ; Atribuicao de variavel  MM V0 ;  LD V0 ; Comando de output  MM output\_number ;  SC output ;  LD K1 ; Comeco do or logico  JZ TL10 ;  JN TL10 ;  JP TL11 ; Returns YES  TL10 LD K1 ;  JZ TL12 ;  JN TL12 ;  JP TL11 ; Returns YES  TL12 LD zero ;  JP TL13 ; Returns NO  TL11 LD one ;  TL13 MM T4 ; Fim do or logico  LD T4 ; Atribuicao de variavel  MM V0 ;  LD V0 ; Comando de output  MM output\_number ;  SC output ;  HM /00 ;  # P ;  0A00  V0 K =0 ; Declaracao de variavel  K0 K =1 ; Declaracao de constante  T0 K =0 ; Declaracao de temporario  K1 K =0 ; Declaracao de constante  T1 K =0 ; Declaracao de temporario  T2 K =0 ; Declaracao de temporario  T3 K =0 ; Declaracao de temporario  T4 K =0 ; Declaracao de temporário |
| **Execução na MVN** | |
|  | |

# Conclusão

Os objetivos foram alcançados com sucesso, conforme demonstram os testes do capítulo anterior. O compilador cumpre os seguintes requisitos:

* Permite rotinas do tipo:
  + If-then
  + If-then-else
  + While
* Permite atribuição de valor à variáveis previamente declaradas
* Fornece rotinas para leitura (entrada) e impressão (saída)
* Realiza as operações aritméticas de soma, subtração, divisão e multiplicação
* Realiza operações lógicas and, or e not

O compilador desenvolvido permite construção e chamada de sub-rotinas, com passagem de parâmetros, com criação de registro de ativação e métodos para carregar e armazenar variáveis no registro de ativação. Porém, o funcionamento não é completo, dada a forma em que as ações semânticas foram implementadas.

Os únicos requisitos solicitados que não foram implementados foram estruturas do tipo vetor e *struct*. Porém, com o conhecimento adquirido ao longo do desenvolvimento percebe-se que a implementação é simples e só não foi realizada pela limitação de tempo. Entretanto, o desenvolvimento deste compilador permitiu consolidar os conceitos vistos em sala de aula.

Uma observação a ser feita é sobre a MVN disponibilizada que apresenta comportamentos distintos quando executada muda-se o parâmetro "passo a passo". Aparentemente, a partir dos testes realizados, o input é interpretado de maneiras diferentes, devido ao buffer de leitura da MVN. Esta característica foi extremamente negativa ao desenvolvimento, custando um grande esforço para identificá-la e prejudicando os testes realizados.