ESCOLA POLITÉNICA □DA UNIVERSIDADE DE SÃO PAULO

Exercício prático VI
Relatório Final
PCS
Linguagens e Compiladores

Alunos: Rafael Camargo Leite – 7629953
Vinicius Correa - 7631553

Índice

1. Sintaxe da linguagem

Para a entrega implementação final do compilador, a linguagem inicialmente definida sofreu diversas modificações para se adequar às necessidades do compilador.

1.1. Wirth

programa = {tipo identificador "(" [declaracao_variavel {"," declaracao_variavel}] ")" "{" {instrucao} "}" | tipo identificador ["[" expressao "]"] ";"}.

declaracao_variavel = tipo identificador ["[" expressao "]"].

instrucao = declaracao_variavel ";" | "return" retorno | "break" ";" | "continue" ";" | "while" laco | "if" condicional | identificador "=" expressao ";" | identificador "(" [{expressao ","} expressao] ")" ";" | "read" "(" identificador ")" ";" | "write" "(" [{expressao ","} expressao] ")" ";".

laco = "(" expressao ")" "{" {instrucao} "}".

condicional = "(" expressao ")" "{" {instrucao} ["else" {instrucao}] "}".

retorno = expressao ";".

expressao = termo_e ["|" expressao] | termo_e ["||" expressao].

termo_e = termo_igualdade ["&" termo_e] | termo_igualdade ["&&" termo_e].

termo_igualdade = termo_relacional ["==" termo_igualdade] | termo_relacional ["!=" termo_igualdade].

termo_relacional = termo_aditivo [">" termo_relacional] | termo_aditivo ["<" termo_relacional] | termo_aditivo [">=" termo_relacional] | termo_aditivo ["<=" termo_relacional].

termo_aditivo = termo_multiplicativo ["+" termo_aditivo] | termo_multiplicativo ["-" termo_aditivo].

termo_multiplicativo = termo_primario ["*" termo_multiplicativo] | termo_primario ["/" termo_multiplicativo] | termo_primario ["%" termo_multiplicativo].

termo_primario = identificador | constante_numero | constante_caractere | identificador "(" [{expressao ","} expressao] ")" | "(" expressao ")".

1.2. Símbolos terminais

Pelo Wirth construido, pode-se observar:

1. Palavras reservadas:
    a. if
    b. else
    c. while
    d. int
    e. float
    f. string
    g. return
    h. const
    i. break
    j. continue
    k. read
    l. write

2. Operadores compostos por um único char
    a. =
    b. >
    c. <
    d. !
    e. +
    f. -
    g. *
    h. /
    i. ^
    j. &
    k. |

3. Operadores compostos por dois chars
    a. ==
    b. >=

c. <=
d. !=
e. &&
f. ||

4. Delimitadores
    a. {
    b. }
    c. [
    d.]
    e. (
    f.)
    g. ,
    h. ;
    i. ' ' (espaço em branco)
    j. \n
    k. \t

5. Aspas
    a. ‘“‘

6. Ponto
    a. ‘.’

7. Comentário de linha
    a. #

1.3. Exemplo de programa

```
int func(int integer) {
	string z;
	z = "literal example";
	write(z, integer);
	return integer;
}

#comentario bla bla
int main() {
	func();
	int a;
	string xx;
	read(xx);

	a = 12;
	xx = "test literal";
	if (a>= 3) {
		int b;
		b = 4;
	}
	#outro comentario bla bla
```

```
        while(a==b) {
                float c;
                c =43;
        }

        while(1){
                a = 44;
        }

        if(a || b) {
                return a;
        }
}
```

2. Analisador Léxico

2.1. funções

De maneira geral, o Analisador léxico é responsável pela leitura e extração de tokens do código fonte.
Tokens são definidos como:

```
typedef struct token_t {
	token_class class;
	union value {
		int i_value;
		float f_value;
		char* s_value;
	} value;
} token_t;
```

onde token_class é definido como:

```
typedef enum {
	CLASS_INT, // int number
	CLASS_FLOAT, // float number
	CLASS_STRING_LIT, // literal string
	CLASS_RESERVED_WORD, // if, while, int, ...
	CLASS_IDENTIFIER, // variable name
	CLASS_SINGLE_OPERATOR, // '=', '>', '<', '!', '+', '-', '*', '/'
	CLASS_DOUBLE_OPERATOR, // "==", ">=", "<=", "!="
	CLASS_DELIMITER, //  '{', '}', '[', ']', ',', ';', ' ', '\t'
} token_class
```

Durante sua execução algumas tarefas são executadas:
Remoção de delimitadores(ex: espaços em branco) e comentários
Expansão de Macros (#define) e pré-processamento(#ifndef etc..)
Armazenamento de linha em coluna dos tokens para prover mensagens de erro ao programador
Conversões numéricas diversas
Inserções na tabela de símbolos

Na implementação adotada, o único item que não foi desenvolvido foi a expansão de macros pois, inicialmente, foi feita a opção de remover essa feature da linguagem.

2.2. Expressões regulares

Utilizando as listas de simbolos terminais(1.2), foram construídas as expressões regulares que representam o analisador léxico:

1. Reserved words:
    a. if
    b. else
    c. while
    d. int
    e. float
    f. return
    g. const
    h. break
    i. continue
    j. read
    k. write

2. Single char operators:
    a. =
    b. >
    c. <
    d. !
    e. +
    f. -
    g. *
    h. ^
    i. /
    j. &
    k. |

3. Double char operators:
    a. ==
    b. >=
    c. <=
    d. !=
    e. &&
    f. ||

4. Átomos Especiais:
    a. Identificadores/aplhanum - [a-z]+ | [0...9]* / [a...z]*

b. Inteiros - [0...9]+
c. Flutuantes - [0...9]+ | [0...9]+ '.' [0...9]+
d. Strings - “ alphanum “
e. comentários # [0...9]* / [a...z]+

2.3. Autômatico consolidado

O autômato final usado como base para o analisador léxico está mostrado na figura 1.

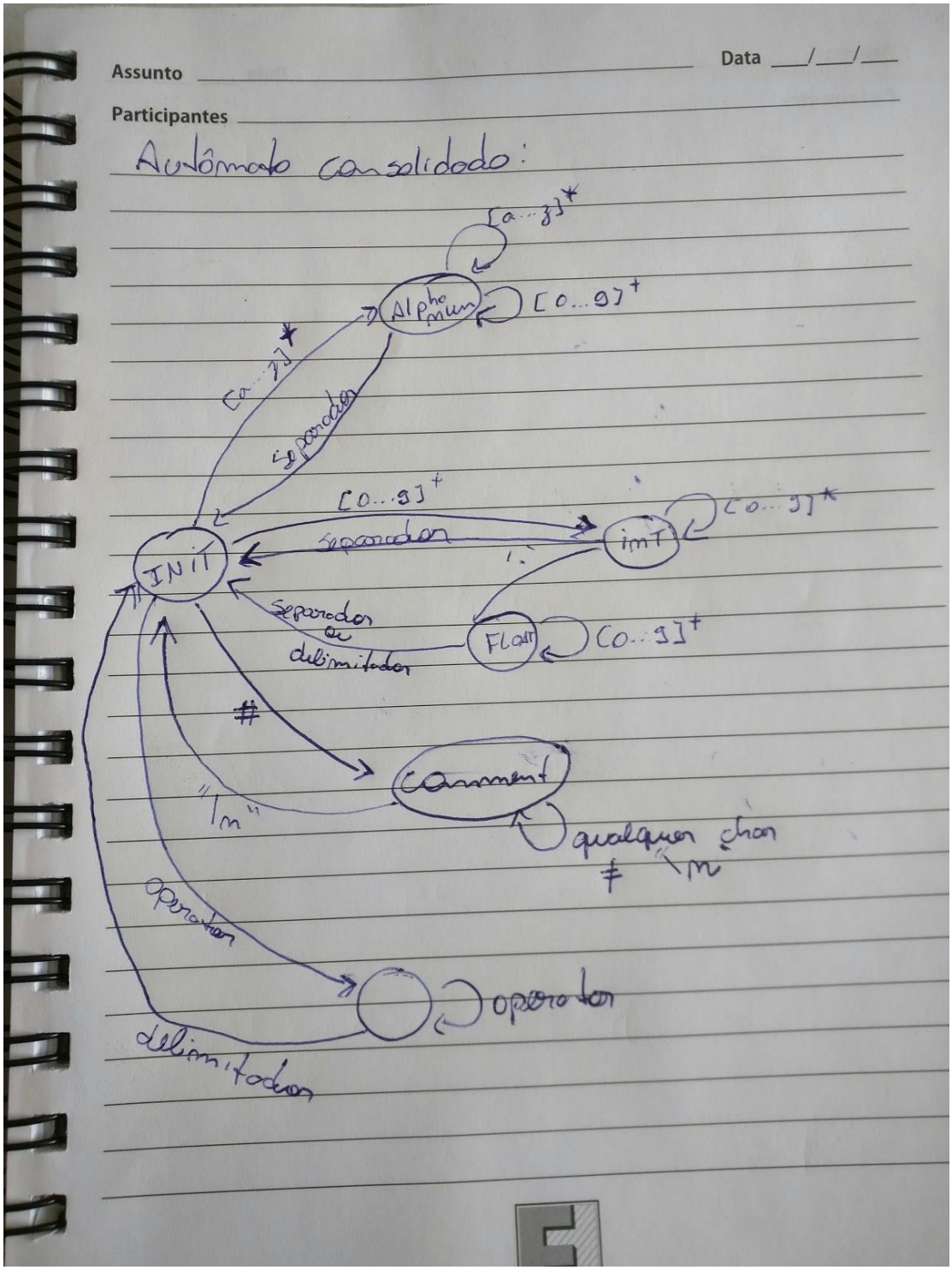


Figura 1 - Autômato finito do analisador Léxico

2.4. Funcionamento

O projeto foi desenvolvido basicamente em 2 módulos:

1. token.c / token.h
2. lex.c / lex.h

Em token.c / token.h foi feita a definição da estrutura do token e suas funcões associadas.
Um token é representado da seguinte maneira:

```
typedef struct token_t {
	token_class class;
	union value {
		int i_value;
		float f_value;
		char* s_value;
	} value;
} token_t;
```

Seu valor foi codificado como uma Union. Sendo assim, pode assumir um dos três valores mostrados.
Caso o token seja um inteiro literal, seu valor é colocado em i_value.
Caso o token seja um float literal, seu valor é colocado em f_value.
Caso o token seja uma string literal, seu valor pe colocado em s_value.
Nos demais casos, i_value é preenchido com o índice da tabela específica que guarda o valor do token.

token_class é definido da seguinte maneira:

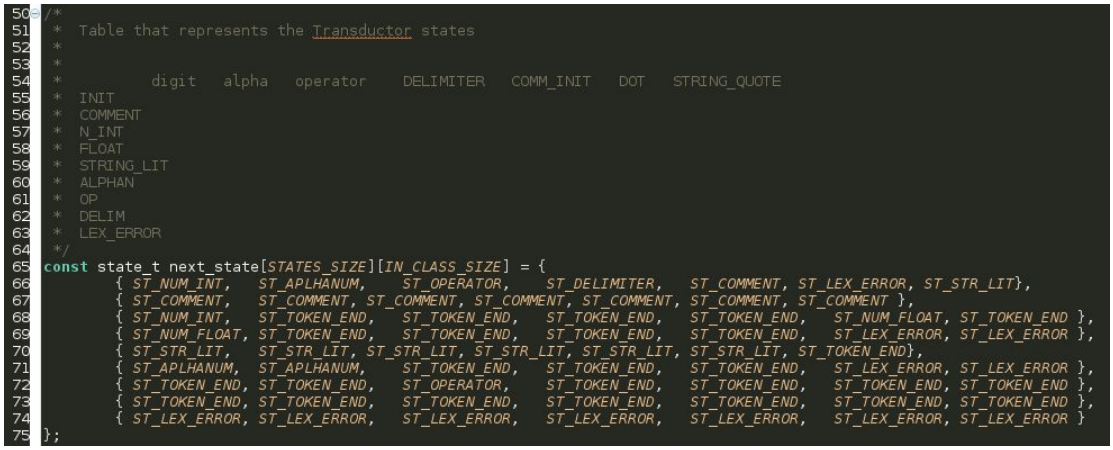
```
typedef enum {
	CLASS_INT, // int number
	CLASS_FLOAT, // float number
	CLASS_STRING_LIT, // literal string
	CLASS_RESERVED_WORD, // if, while, int, ...
	CLASS_IDENTIFIER, // variable name
	CLASS_SINGLE_OPERATOR, // '=', '>', '<', '!', '+', '-', '*', '/'
	CLASS_DOUBLE_OPERATOR, // "==", ">=", "<=", "!="
	CLASS_DELIMITER, //  '{', '}', '[', ']', ',', ';', ' ', '\t'
} token_class
```

Em tables.c foram definidas as tabelas necessárias ao analisador.
Elas são:

```
/*
 *      Reserved words
 */
const char* reserved_words[] = { "if", "else", "while", "int", "float", "string",
               "return", "const", "break", "continue", "read", "write" };

const char**  get_reserved_words() {
       return reserved_words;
}

/*
 *      Single operators
 */
const char single_operators[] = {'=', '>', '<', '!', '+', '-', '*', '/', '^', '&',
'|' };
const char* get_single_operators() {
       return single_operators;
}

/*
 *      Double operators
 */
const char* double_operators[] = {"==", ">=", "<=", "!=", "&&", "||" };
const char** get_double_operators() {
       return double_operators;
}

/*
 *      Delimiters
 */
const char delimiters[DELIMITERS_SIZE] = { '{', '}', '[', ']', '(', ')', ',', ';',
' ', '\n', '\t' };
const char* get_delimiters() {
       return delimiters;
}

/*
 *      String quote
 */
const char string_quote = '"';
const char get_string_quote() {
       return string_quote;
}

/*
 *      Dot
 */
const char dot = '.';
const char get_dot() {
       return dot;
}

/*
 *      comment begin
 */
const char commentary = '#';
const char get_commentary() {
       return commentary;
}
```

Essas são as tabelas referenciadas pelo ‘value’ do token.

Finalmente, de maneira geral, o códido em lex.c representa um automato por meio de tabelas de transição de estado.
O Screenshot abaixo representa a tabela de transição de estados:



```
/*
 *  Table that represents the Transductor states
 *
 *
 *          digit   alpha   operator    DELIMITER   COMM_INIT   DOT   STRING_QUOTE
 *  INIT
 *  COMMENT
 *  N_INT
 *  FLOAT
 *  STRING_LIT
 *  ALPHAN
 *  OP
 *  DELIM
 *  LEX_ERROR
 */
const state_t next_state[STATES_SIZE][IN_CLASS_SIZE] = {
        { ST_NUM_INT,   ST_APLHANUM,    ST_OPERATOR,    ST_DELIMITER,   ST_COMMENT, ST_LEX_ERROR, ST_STR_LIT},
        { ST_COMMENT,   ST_COMMENT, ST_COMMENT, ST_COMMENT, ST_COMMENT, ST_COMMENT, ST_COMMENT },
        { ST_NUM_INT,   ST_TOKEN_END,   ST_TOKEN_END,   ST_TOKEN_END,   ST_TOKEN_END,   ST_NUM_FLOAT, ST_TOKEN_END },
        { ST_NUM_FLOAT, ST_TOKEN_END,   ST_TOKEN_END,   ST_TOKEN_END,   ST_TOKEN_END,   ST_LEX_ERROR, ST_LEX_ERROR },
        { ST_STR_LIT,   ST_STR_LIT, ST_STR_LIT, ST_STR_LIT, ST_STR_LIT, ST_STR_LIT, ST_TOKEN_END},
        { ST_APLHANUM,  ST_APLHANUM,    ST_TOKEN_END,   ST_TOKEN_END,   ST_TOKEN_END,   ST_LEX_ERROR, ST_LEX_ERROR },
        { ST_TOKEN_END, ST_TOKEN_END,   ST_OPERATOR,    ST_TOKEN_END,   ST_TOKEN_END,   ST_TOKEN_END, ST_TOKEN_END },
        { ST_TOKEN_END, ST_TOKEN_END,   ST_TOKEN_END,   ST_TOKEN_END,   ST_TOKEN_END,   ST_TOKEN_END, ST_TOKEN_END },
        { ST_LEX_ERROR, ST_LEX_ERROR,   ST_LEX_ERROR,   ST_LEX_ERROR,   ST_LEX_ERROR,   ST_LEX_ERROR, ST_LEX_ERROR }
};
```

Figura 2 - Tabela de transição de estados - Léxico

A função principal possui um loop que começa no estado inicial e é executado até atingit o estado ST_TOKEN_END
O código abaixo é o desta função:

```
/*
 *      main function
 */
token_t* get_token(FILE *fp) {
        state_struct_t state_struct;
        create_state_struct(&state_struct, fp);

        /*
         *      Main loop
         */
        do {
                /*
                 *      Get next input
                 */
                state_struct.curr_input = getc(fp);
                if(state_struct.curr_input == EOF) {
                        break;
                }

                fprintf(stdout, "[DEBUG] curr_char = %c\n", state_struct.curr_input);

                /*
                 *      update state_struct
                 */
                state_struct.input_class =
classify_input_class(state_struct.curr_input);


state_function[*state_struct.curr_state][state_struct.input_class](&state_struct);
        } while(*state_struct.curr_state != ST_TOKEN_END);
```

```
        /*
         *      Build token to be returned
         */
        if(*state_struct.buffer_ptr != 0){
                printf("[INFO] buffer: %s\n", state_struct.buffer);
                build_token(&state_struct);
        }

        /*
         *      Free dynamic allocated memory and return
         */
        destroy_state_struct(&state_struct);

        return state_struct.token;
}
```

3. Analisador Sintático

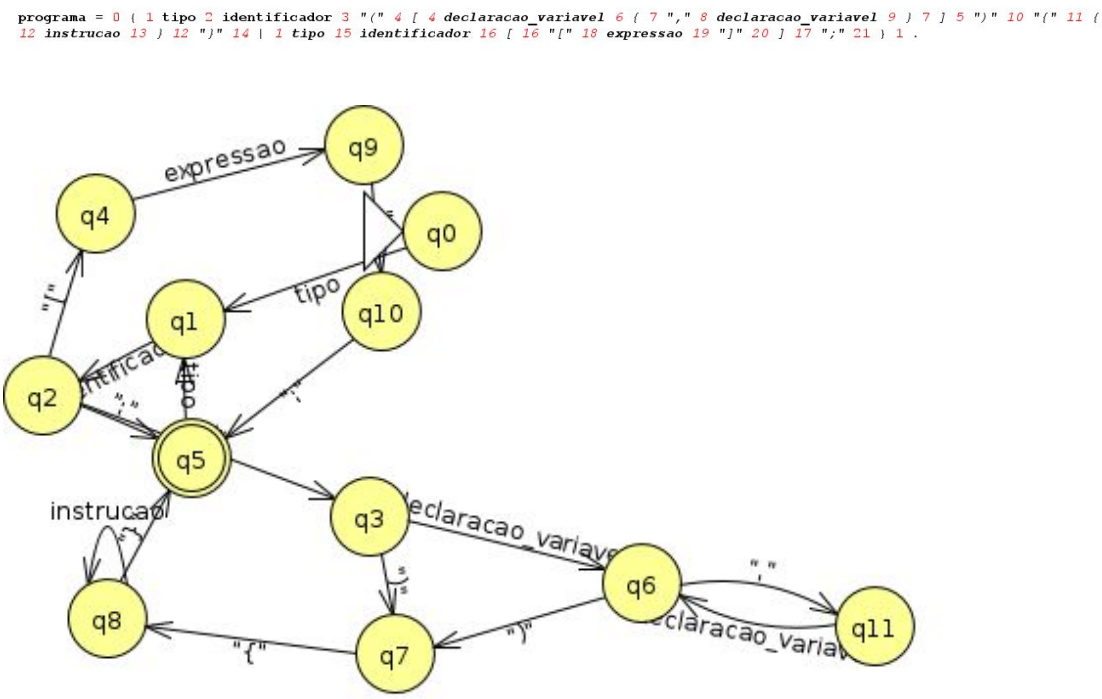
3.1. Funções

Na implementação escolhida, o primeiro ponto a ser destacado para o analisador sintático é o de que ele é o orquestrador da compilação.
Ele a função ‘analyse(FILE* fp)’, a qual faz as chamadas para conseguir os tokens do analisador léxico.
Além disso, o analisador sintático tem como objetivo verificar se o formato das sentaças é coerente com as regras de geração da gramática.
Em poucas palavras, ele deve receber os token provenientes do analisador léxico e verificar, com base na ordem de recebimento, se a estrutura sintatica apresentada é valida de acordo com a gramática da linguagem.

3.2. Sub-máquinas

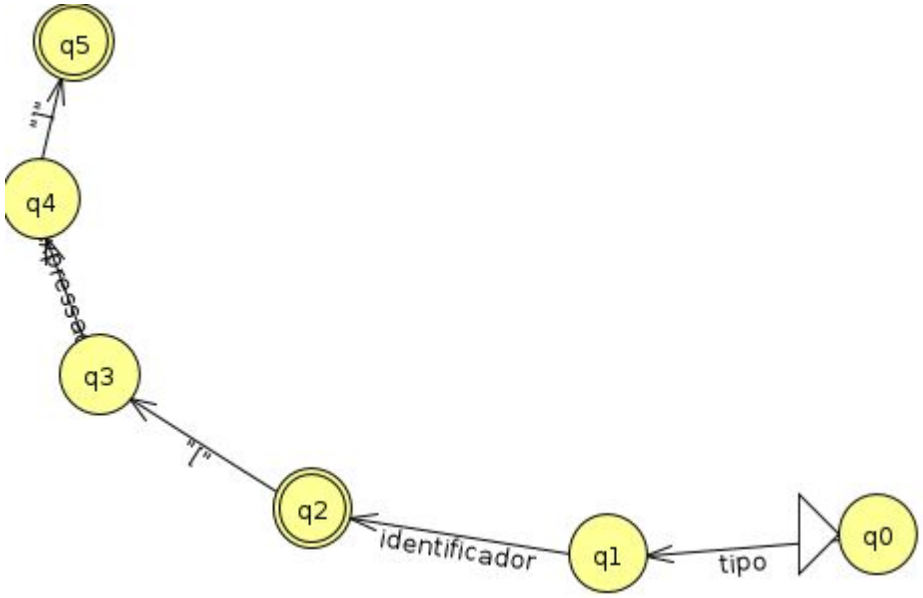
Utilizando o programa passade pelo professor para a geração do autômato de pilha estruturado, foram obtidas as seguintes sub-máquinas:

1. programa



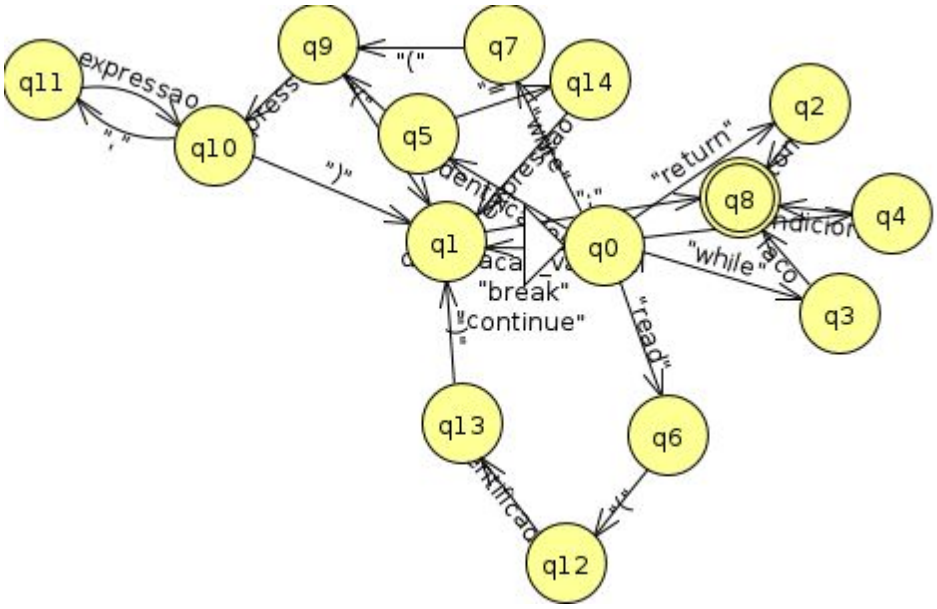
2. declaracao_variavel

```
declaracao_variavel = 0 tipo 1 identificador 2 [ 2 "[" 4 expressao 5 "]" 6 ] 3 .
```



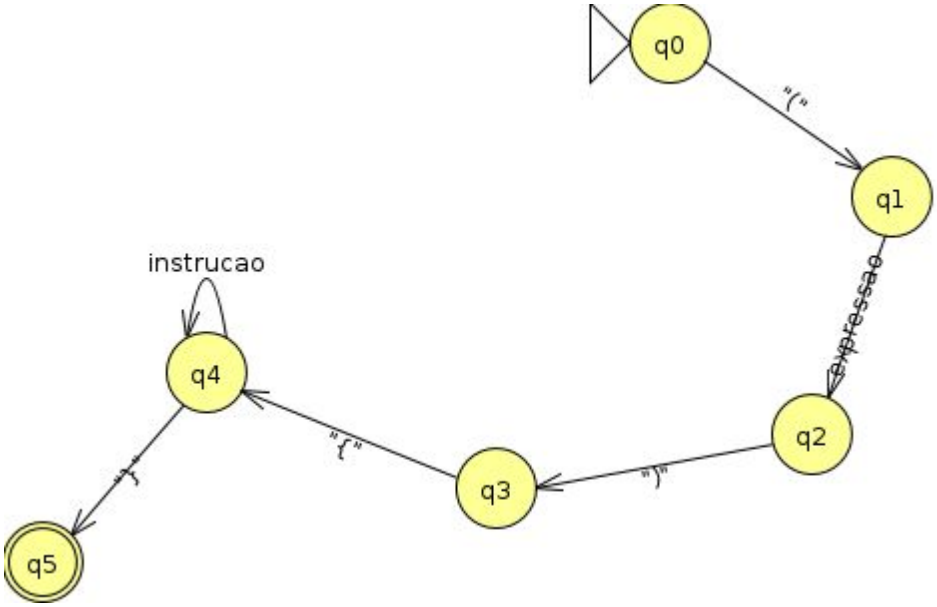
3. instrucao

```
instrucao = 0 declaracao_variavel 1 ";" 2 | 0 "return" 3 retorno 4 | 0 "break" 5 ";" 6 | 0 "continue" 7 ";" 8 | 0 "while" 9 laco
10 | 0 "if" 11 condicional 12 | 0 identificador 13 "=" 14 expressao 15 ";" 16 | 0 identificador 17 "(" 18 [ 18 { 20 expressao 21
"," 22 } 20 expressao 23 ] 19 ")" 24 ";" 25 | 0 "read" 26 "(" 27 identificador 28 ")" 29 ";" 30 | 0 "write" 31 "(" 32 [ 32 { 34
expressao 35 "," 36 } 34 expressao 37 ] 33 ")" 38 ";" 39 .
```



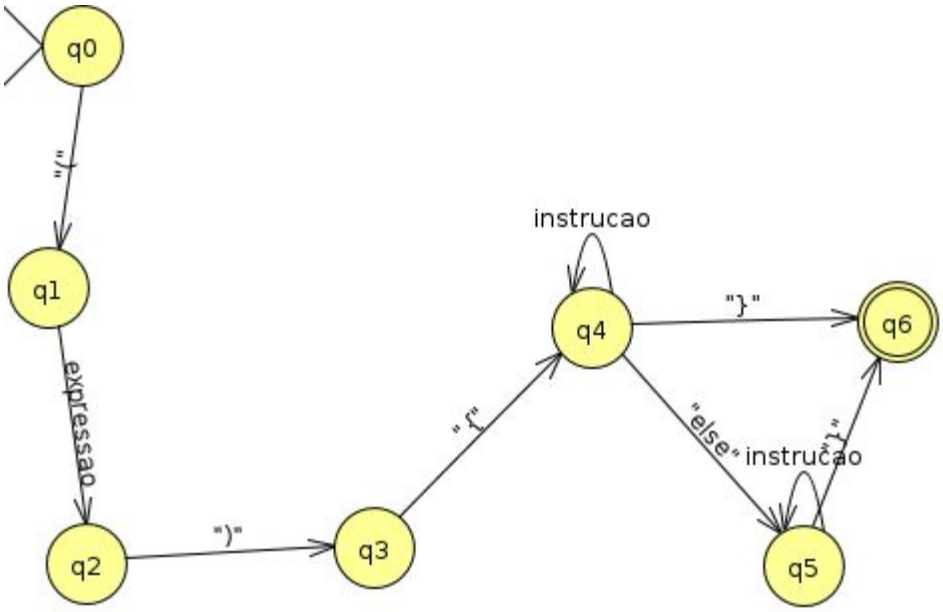
4. laco

```
laco = 0 "(" 1 expressao 2 ")" 3 "{" 4 { 5 instrucao 6 } 5 "}" 7 .
```



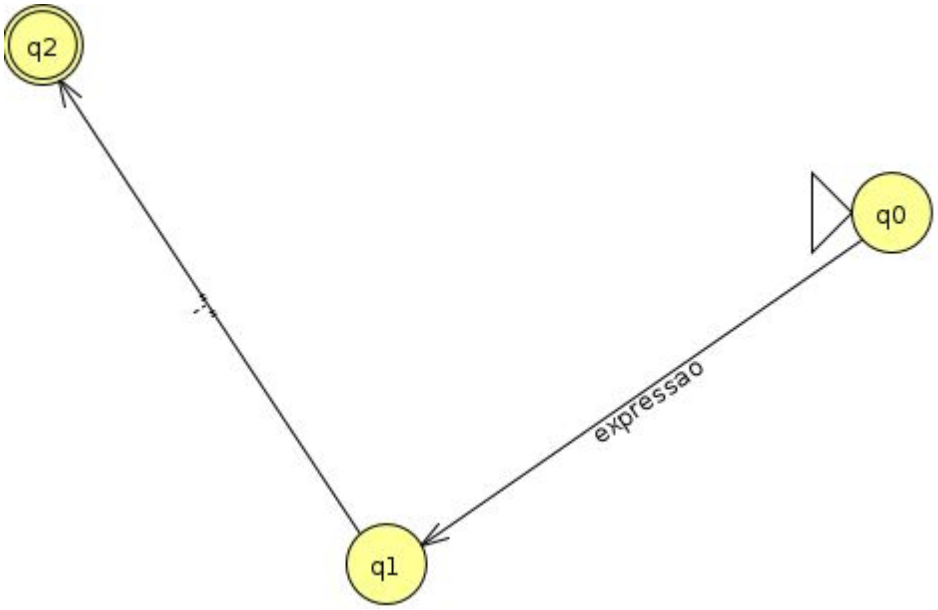
5. condicional

```
condicional = 0 "(" 1 expressao 2 ")" 3 "{" 4 { 5 instrucao 6 } 5 [ 5 "else" 8 { 9 instrucao 10 } 9 ] 7 "}" 11 .
```



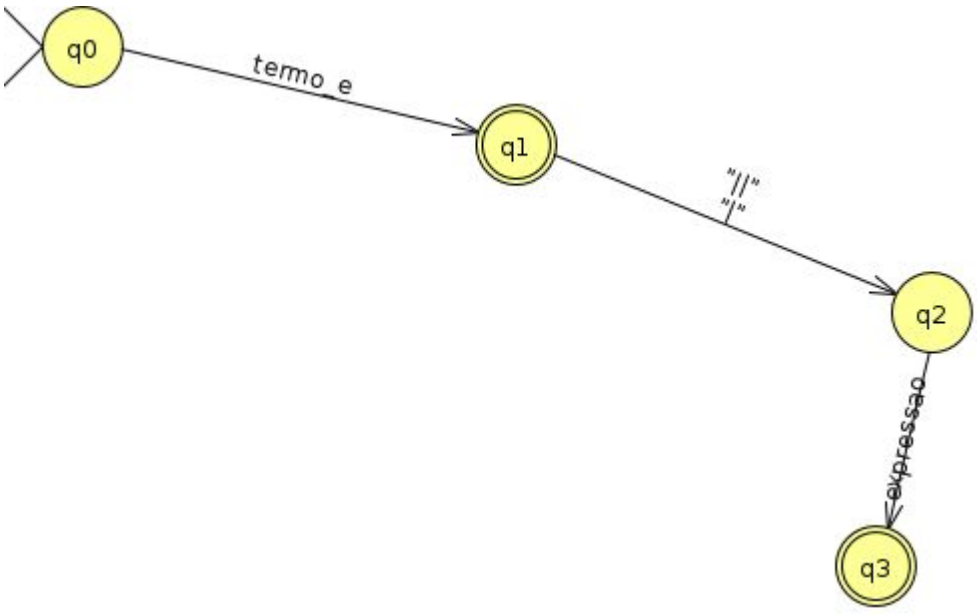
6. retorno

```
retorno = 0 expressao 1 ";" 2 .
```



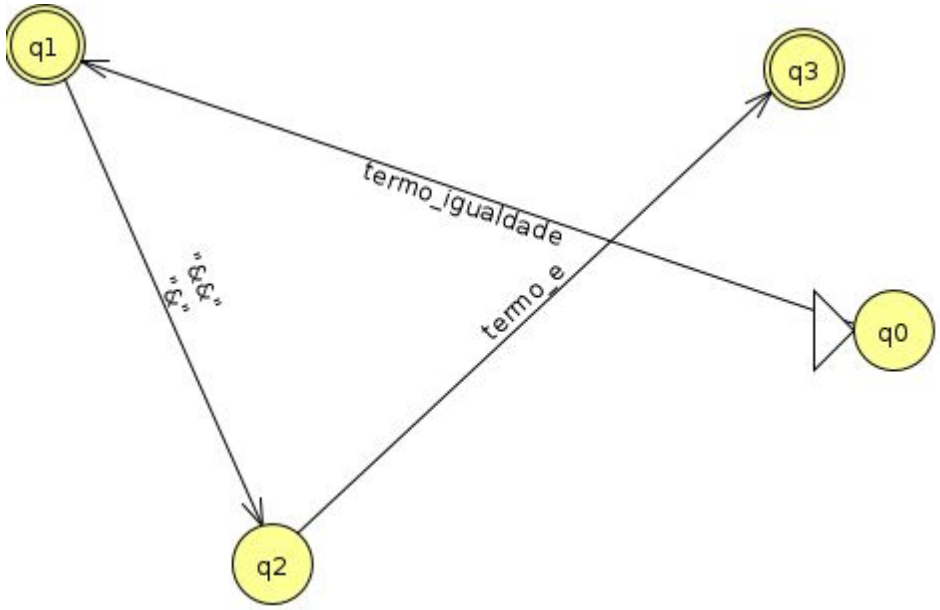
7. expressao

```
expressao = 0 termo_e 1 [ 1 "|" 3 expressao 4 ] 2 | 0 termo_e 5 [ 5 "||" 7 expressao 8 ] 6 .
```



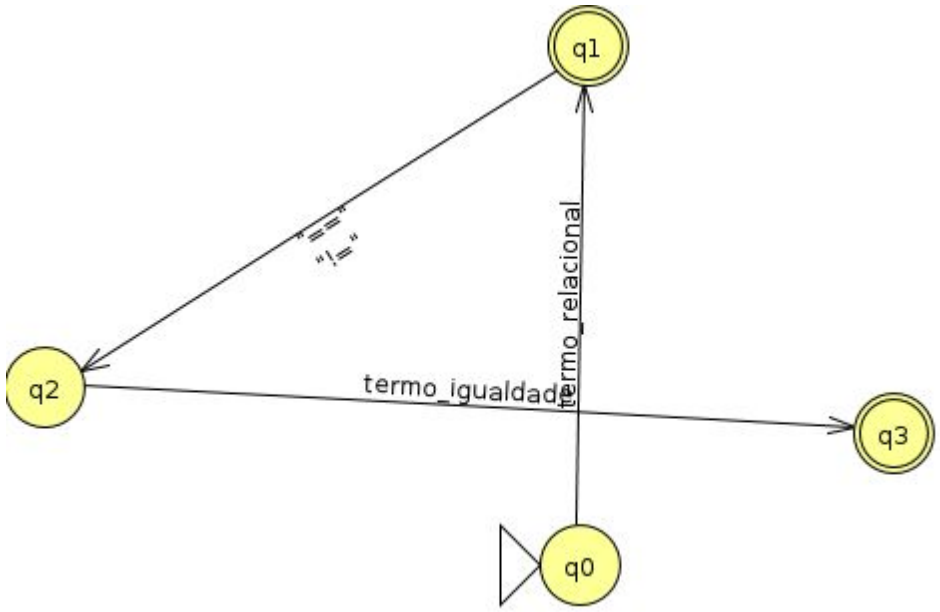
8. termo_e

```
termo_e = 0 termo_igualdade 1 [ 1 "&" 3 termo_e 4 ] 2 | 0 termo_igualdade 5 [ 5 "&&" 7 termo_e 8 ] 6 .
```



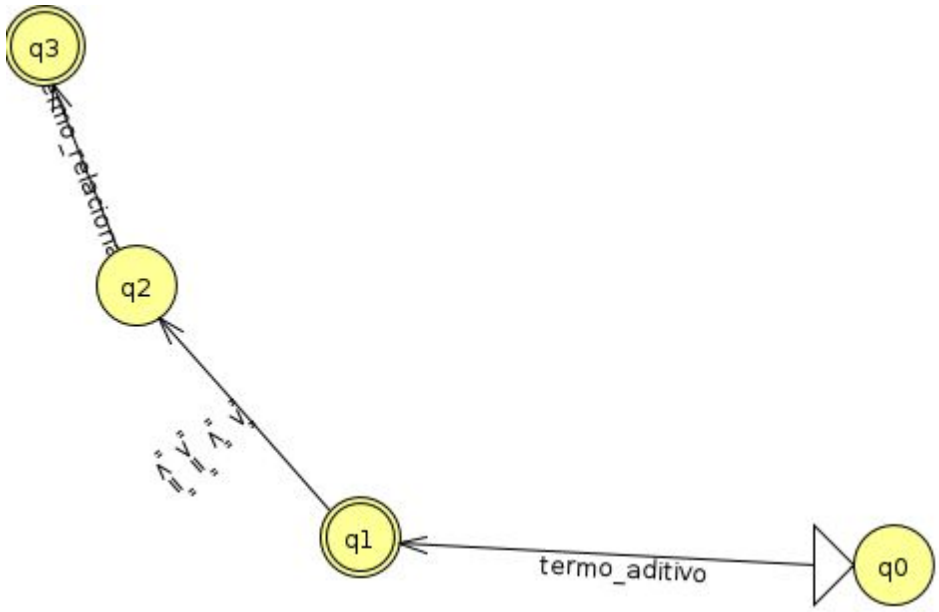
9. termo_igualdade

```
termo_igualdade = 0 termo_relacional 1 [ 1 "==" 3 termo_igualdade 4 ] 2 | 0 termo_relacional 5 [ 5 "!=" 7 termo_igualdade 8 ] 6
.
```



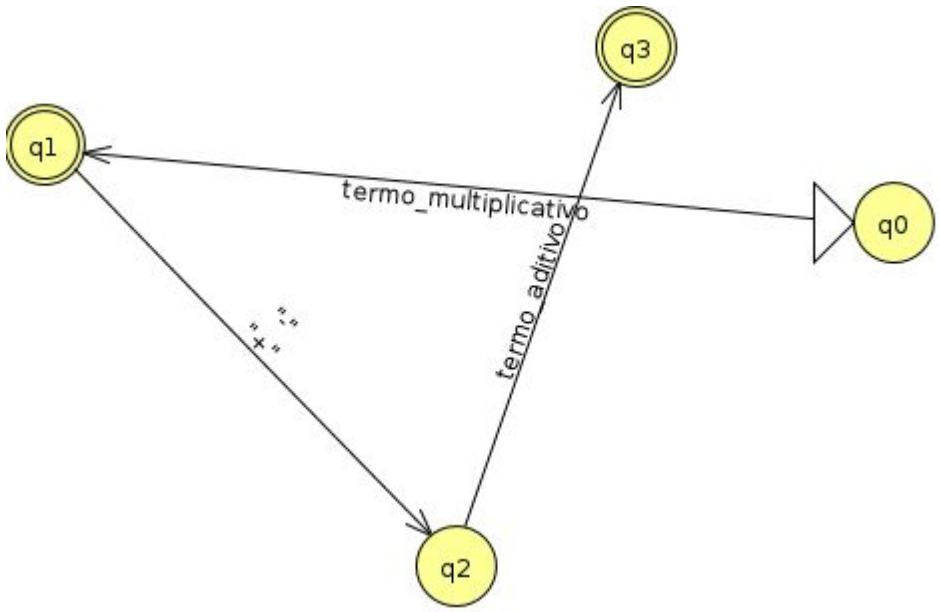
10. termo_relacional

```
termo_relacional = 0 termo_aditivo 1 [ 1 ">" 3 termo_relacional 4 ] 2 | 0 termo_aditivo 5 [ 5 "<" 7 termo_relacional 8 ] 6 | 0
termo_aditivo 9 [ 9 ">=" 11 termo_relacional 12 ] 10 | 0 termo_aditivo 13 [ 13 "<=" 15 termo_relacional 16 ] 14 .
```



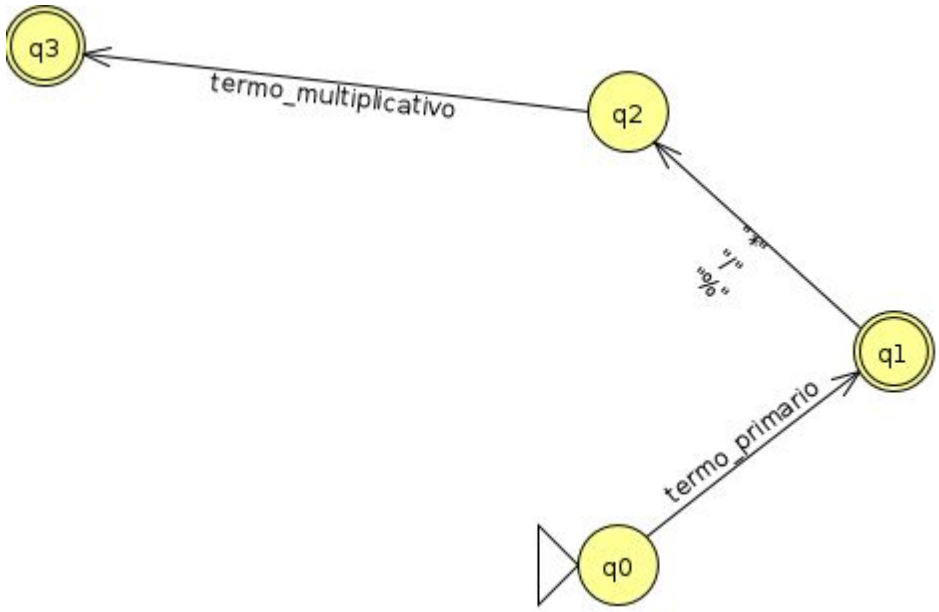
11. termo_aditivo

```
termo_aditivo = 0 termo_multiplicativo 1 [ 1 "+" 3 termo_aditivo 4 ] 2 | 0 termo_multiplicativo 5 [ 5 "-" 7 termo_aditivo 8 ] 6
.
```



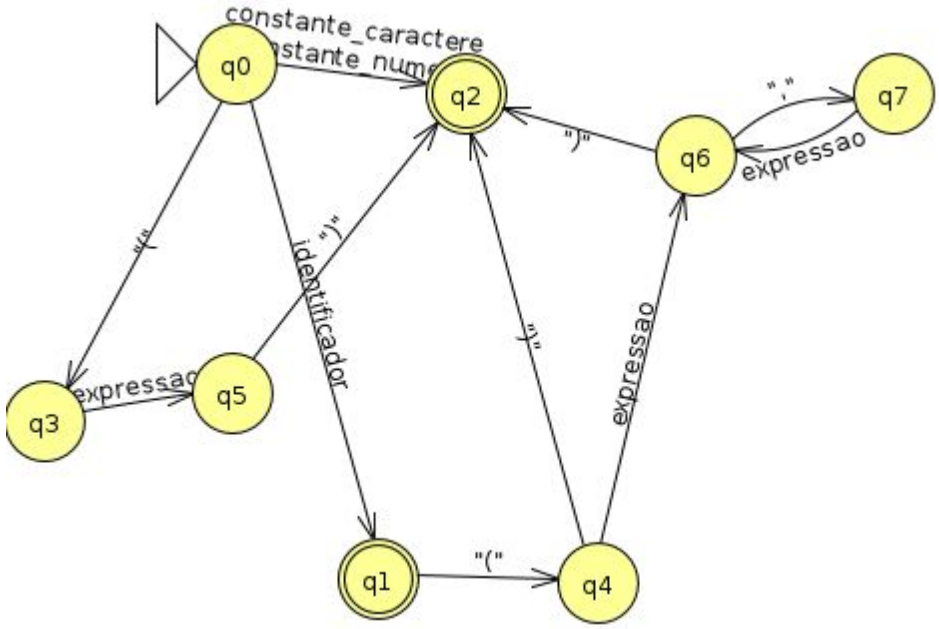
12. termo_multiplicativo

```
termo_multiplicativo = 0 termo_primario 1 [ 1 "*" 3 termo_multiplicativo 4 ] 2 | 0 termo_primario 5 [ 5 "/" 7
termo_multiplicativo 8 ] 6 | 0 termo_primario 9 [ 9 "%" 11 termo_multiplicativo 12 ] 10 .
```



13. termo_primario

```
termo_primario = 0 identificador 1 | 0 constante_numero 2 | 0 constante_caractere 3 | 0 identificador 4 "(" 5 [ 5 { 7 expressao
8 "," 9 } 7 expressao 10 ] 6 ")" 11 | 0 "(" 12 expressao 13 ")" 14 .
```



3.3. Tabela de simbolos

A table de simbolos usada pelo compilador é usada apenas dentro do analisador sintatico.
Sua estrutura está definida de seguinte maneira:

```
/*
 * Symbol table structure
 */
typedef struct symbol_table_t {
        symbol_table_entry_t* first_row;
        struct symbol_table_t* prev;
        size_t size;
} symbol_table_t;
```

onde symbol_table_entry_t é definida como:

```
/*
 *      Symbol table entry struct
 */
typedef struct symbol_table_entry_t {
        char* name;
        int type;
        int addr;
        struct symbol_table_entry* next;
} symbol_table_entry_t;
```

As linhas da tabela foram implementadas como uma lista ligada.

O campo struct symbol_table_t* prev faz a ligação com a tabela de simbolos do escopo anterior ao atual.
A figura 3 mostra como foi construída a estrutura da tabela de simbolos.

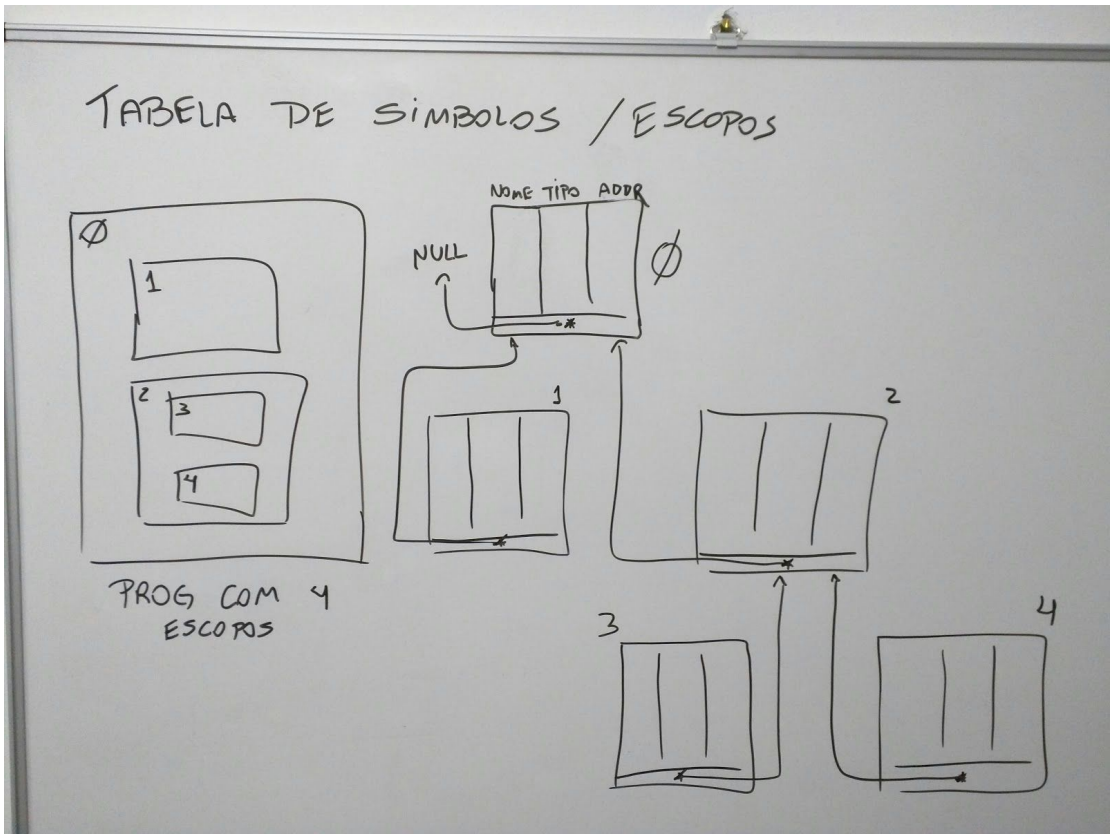


Figura 3 - Organização da tabela de simbolos

Pela figura 3 pode-se observar que cada tabela de um determinado escopo tem uma ligação com seu escopo “pai”. Isso foi feito para que a verificação de escopo pelo analisador semântico fosse feita de maneira simples.
Além disso, pode-se observar que a tabela possui nome, tipo e endereço. Nome é o identificador do item na tabela. Tipo é usado pelo analisador semântico para executar verificação de tipos. Finalmente, endereço vai guardar o endereço da variável para a geração de código posterior.

3.4. Ações semânticas

As ações semânticas implementadas foram as seguintes:

1. Verificação de escopo (Como explicado no item 3.4)
2. Verificação de tipos (Como explicado no item 3.4)

Por falta de tempo, estas foram a únicas ações de fato implementadas. A geração de código em si não foi codificada.

3.5. *Implementação*

Primeiramente, para a construção do analisador sintático, algumas alterações no código anterior do analisador léxico tiveram que ser feitas:

1. As tabelas com: palavras reservadas, delimitadores, operadores, etc.. foram retiradas do arquivo lex.c e passadas para os arquivos tables.c e tables.h. Isso foi feito para que essas tabelas pudessem ser acessadas também do analisador sintático.
2. A função main deixou de chamar a função get_token() do analisador léxico e passou a chamar a função analyse() do analisador sintático.

Depois, algumas estruturas foram criadas:

1. tabela de submáquinas:

```
/*
 *      State function table
 */
int (* const sub_machines[FSM_SIZE]) (token_t* t) = {
        fsm_program,
        fsm_var_declaration,
        fsm_instruction,
        fsm_loop,
        fsm_cond,
        fsm_return,
        fsm_expr,
        fsm_term_and,
        fsm_term_equal,
        fsm_term_relacional,
        fsm_term_add,
        fsm_term_mult,
        fsm_term_primary
};
```

2. struct que representa o estado atual da análise

```
/*
 *      Structure that represents the state of the analysis
 */
typedef struct {
        int current_sub_machine_state;
        sub_machine_t current_sub_machine;
        int get_token_flag;
} analysis_state_t;
```

3. Pilha

```
/*
 *      Stack node has the sub-machine and state
 */
typedef struct stack_node {
        sub_machine_t sub_machine;
        int state;
        struct stack_node* next;
} stack_node;

/*
 *      Stack has just a top node and size
 */
typedef struct stack {
        stack_node* top;
        int size;
} stack_t;

/*
 *      push into the stack
 */
void push(stack_t* s, sub_machine_t sub_machine, int ret_state) {
        stack_node* sn = malloc(sizeof(stack_node*));
        sn->next = s->top;
        sn->sub_machine = sub_machine;
        sn->state = ret_state;
        s->top = sn;
        s->size++;

        state.get_token_flag = 0;
}

/*
 *      Pop from stack
 *              This function will update the 'state' global variable and free the
memory
 *              allocated for the stack_node
 */
void pop(stack_t* s) {
        if(s->size > 0) {
                //update state variable
                state.current_sub_machine = s->top->sub_machine;
                state.current_sub_machine_state = s->top->state;

                //decrement stack size
                s->size--;

                // free stack top and point it to the next item
                stack_node* aux = s->top;
                s->top = s->top->next;
                free(aux);
                state.get_token_flag = 0;
        } else {
                state.current_sub_machine_state = ERROR;
        }
}

int is_empty(stack_t* s) {
        return s->size == 0;
}
```

4. função principal da compilação

```
/*
 *      Entry point for compilation
 */
```

```
int analyze(FILE* fp) {
	token_t* t;
	state.get_token_flag = 1;
	state.current_sub_machine_state = 0;
	state.current_sub_machine = 0;
	while(TRUE) {
		if(should_get_next_token()) {
			t = get_token(fp);
			if (t == NULL) {
				break;
			}
		}

		state.get_token_flag = 1;

		state.current_sub_machine_state =
sub_machines[state.current_sub_machine](t);
		if(state.current_sub_machine_state == ERROR) {
			DEBUG("Compilation error!!!!");
			return 1;
		}
	}

	DEBUG("Compilation Successfull");
	return 0;
}
```

perceba que na função acima está a chamada para o get_token do analisador léxico.

Fora isso, como mostrado na tabela de sub-maquinas, foi criada uma função para cada uma delas, sendo que foram implementadas com base nos autômatos obtidos pelo wirth da linguagem.

4. Ambiente de execução

4.1. Introdução

Primeiramente citaremos de maneira breve o que compõe um ambiente de execução de um compilador. Até agora, vimos que a análise léxica, sintática e semântica são dependentes apenas das propriedades das linguagens-fonte, independendo da máquina e seu sistema operacional. O ambiente de execução é quem fará a ligação entre código-fonte e máquina. Em outras palavras, o ambiente de execução é responsável pela porção de memória para se carregar o código gerado na compilação e para se carregar e trabalhar com os dados necessários que serão manipulados pelo código do programa.

De maneira geral, a área de dados é dividida da seguinte maneira:

1. Código
2. Estática
3. Pilha
4. Heap

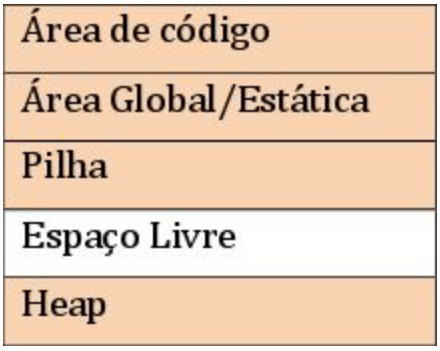


Figura 4 - organização da memório de um processo

4.2. Instruções da linguagem de saída

Código de operação	Instrução	Mnemonico
0	Jump	JP
1	Jump if Zero	JZ

2	Jump if Negative	JN
3	Load Value	LV
4	Add	+
5	Sub	-
6	Mul	*
7	Div	/
8	Load	LD
9	Move to mem	MM
A	Subroutine call (SC)	SC
B	return from SC	RS
C	Halt	HM
D	Entrada	GD
E	Saída	PD
F	Chamada de supervisor	OS

Tabela 1

4.3. Mnemonicos das pseudo-instruções

As pseudo-instruções são:

1. @: Recebe um operando numérico e define o endereço da instrução seguinte
2. K: Constante - o operando numérico tem o valor da constante
3. $: Reserva de área de dados - o perando numérico define o tamanho da área a ser reservada
4. #: Final do texto fonte

Os formatos dos operandos numéricos são os seguintes:

1. /<valor>: valor em Hexa
2. =<valor>: valor em decimal

3. @<valor>: valor em octal
4. #<valor>: valor em binário

4.4. Características Gerais da MVN

A MVN possui 8 registradores específicos, uma memória de 4096 posições e uma pilha.

Os Registradores são mostrados na tabela 2

Registrador	Função
MAR	Registrador de endereço de memória
MBR	Registrador de dados da memória
IC	Registrador de endereço de instrução (próxima instrução)
IR	Registrador de instrução (instrução corrente)
OP	Código de operação - parte do registrador de instrução que identifica a instrução que está sendo executada
OI	Operando de instrução - complementa a instrução indicando o dado ou o endereço sobre o qual ela deve agir.
RA	Registrador de endereço de retorno
AC	Acumulador

Table 2

A Arquitetura da MVN pode ser observada na figura 2

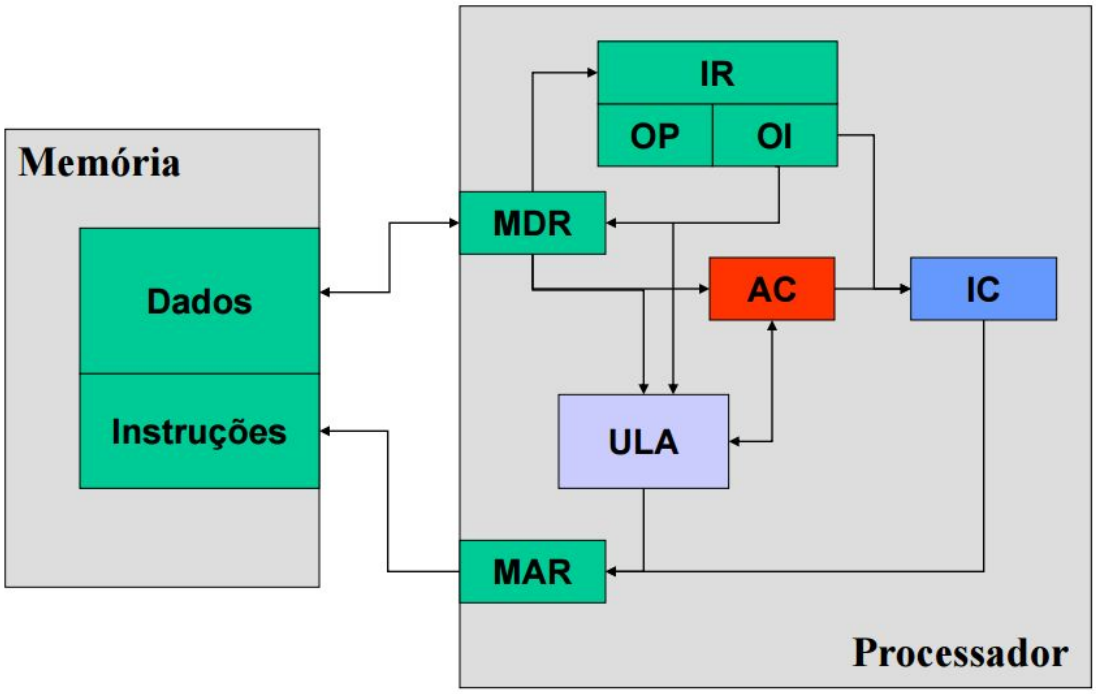


Figura 5 - Arquitetura MVN

5. Tradução dos comandos

5.1. Constantes

```
K_0 >
K_1 >
K_2 >
K_10 >
K_256 >
K_ASCII_0 >
K_LD >
K_MM >
K_ENTER >
K_MENOS >

                    &       /0

K_0                 k       /0
K_1                 k       /1
K_2                 k       /2
K_10        k       /A
K_256       k       /100
K_ASCII_0   k       /30
K_LD        k       /8000
K_MM        k       /9000
K_ENTER     k       /0A
K_MENOS     k       /2D
# CONST
```

5.2. Rotina PRINT

```
PRINT >
PRINT_ADDR >

K_2 <
K_LD <
K_ENTER <

                    &       /0


PRINT_ADDR  k       /0
PRINT       k       /0
LOOP        LD      PRINT_ADDR
                    +       K_LD
                    MM      LD_STR
LD_STR      k       /0
                    JZ      RS_PRINT
                    PD      /100
                    LD      PRINT_ADDR
                    +       K_2
                    MM      PRINT_ADDR
                    JP      LOOP

RS_PRINT    LD      K_ENTER
                    PD      /100
                    RS      PRINT
# PRINT
```

5.3. Rotina SCAN

```
SCAN_ADDR >
SCAN >

K_256 <
K_ENTER <
K_MM <
K_2 <
K_0 <

                        &       /0

SCANNED         k       /0
SCAN_ADDR       k       /0
SCAN            k       /0
LOOP            GD      /0
                        MM      SCANNED
                        /       K_256
                        -       K_ENTER
                        JZ      RS_SCAN
                        LD      SCAN_ADDR
                        +       K_MM
                        MM      MM_SCANNED0
                        LD      SCANNED
                        /       K_256
MM_SCANNED0     K       /0
                        LD      SCAN_ADDR
                        +       K_2
                        MM      SCAN_ADDR
                        LD      SCANNED
                        *       K_256
                        /       K_256
                        -       K_ENTER
                        JZ      RS_SCAN
                        LD      SCAN_ADDR
                        +       K_MM
                        MM      MM_SCANNED1
                        LD      SCANNED
                        *       K_256
                        /       K_256
MM_SCANNED1     K       /0
                        LD      SCAN_ADDR
                        +       K_2
                        MM      SCAN_ADDR
                        JP      LOOP
RS_SCAN         LD      SCAN_ADDR
                        +       K_MM
                        MM      MM_0
                        LD      K_0
MM_0            k       /0
                        RS      SCAN
# SCAN
```

5.4. Rotina PUSH (stack)

```
PUSH_ADDR >
PUSH_SIZE >
PUSH >

K_2 <
K_LD <
K_MM <

TOP <

                    &       /0

PUSH_TOP        k       /0      ; Quando parar de copiar.
PUSH_ADDR       k       /0      ; Endereco da variável a ser copiada.
PUSH_SIZE       k       /0      ; Tamanho da variável.
PUSH            k       /0
                        LD      PUSH_ADDR
                        +       PUSH_SIZE
                        MM      PUSH_TOP

LOOP            LD      PUSH_ADDR       ; Carrega valor apontado.
                        +       K_LD
                        MM      LD_PARAM
                        LD      TOP     ; Grava no endereco apontado.
                        +       K_MM
                        MM      MM_TOP
LD_PARAM        k       /0
MM_TOP          k       /0
                        LD      TOP
                        +       K_2
                        MM      TOP
                        LD      PUSH_ADDR
                        +       K_2
                        MM      PUSH_ADDR
                        -       PUSH_TOP
                        JZ      RS_PUSH
                        JP      LOOP
RS_PUSH         RS      PUSH
# PUSH
```

5.5. Rotina POP (stack)

```
POP_ADDR >
POP_SIZE >
POP >

K_2 <
K_LD <
K_MM <

TOP <

                    &       /0

POP_BOTTOM      k       /0      ; Quando parar de copiar.
POP_ADDR        k       /0      ; Endereco para onde deve-se copiar.
POP_SIZE        k       /0      ; Tamanho da variável.
POP                     k       /0
                        LD      POP_ADDR
                        MM      POP_BOTTOM
                        +       POP_SIZE
                        -       K_2
                        MM      POP_ADDR

LOOP            LD      TOP     ; Carrega valor apontado.
                        -       K_2
                        MM      TOP
                        +       K_LD
                        MM      LD_TOP
                        LD      POP_ADDR        ; Grava no endereco apontado.
                        +       K_MM
                        MM      MMPARAM
LD_TOP          k       /0
MMPARAM         k       /0
                        LD      POP_ADDR
                        -       POP_BOTTOM
                        JZ      RS_POP
                        LD      POP_ADDR
                        -       K_2
                        MM      POP_ADDR
                        JP      LOOP
RS_POP          RS      POP
# POP
```

5.6. Rotina ATOI

```
ATOI_STR >
ATOI >

K_0 <
K_LD <
K_MENOS <
K_1 <
K_2 <
K_ASCII_0 <
K_10 <

                    &       /0

NEG_FLAG        k       /0
RESULT          k       /0
NUM                     k       /0
ATOI_STR        k       /0      ; Endereco da string a ser convertida.
ATOI            k       /0
                        LD      K_0
                        MM      RESULT
                        LD      ATOI_STR
                        +       K_LD
                        MM      LD_NEG
LD_NEG          k       /0
                        -       K_MENOS
                        JZ      ISNEG
                        LD      K_0
                        MM      NEG_FLAG
                        JP      LOOP
ISNEG           LD      K_1
                        MM      NEG_FLAG
                        LD      ATOI_STR
                        +       K_2
                        MM      ATOI_STR

LOOP            LD      ATOI_STR
                        +       K_LD
                        MM      LD_ATOI
LD_ATOI         k       /0
                        JZ      RS_ATOI
                        -       K_ASCII_0
                        MM      NUM
                        LD      RESULT
                        *       K_10
                        +       NUM
                        MM      RESULT
                        LD      ATOI_STR
                        +       K_2
                        MM      ATOI_STR
                        JP      LOOP

RS_ATOI         LD      NEG_FLAG
                        JZ      ATOI_END
                        LD      K_0
                        -       RESULT
                        MM      RESULT
ATOI_END        LD      RESULT
                        RS      ATOI

# ATOI
```

5.7. Rotina ITOA

```
ITOA >
ITOA_NUM >
ITOA_ADDR >

K_0 <
K_10 <
K_ASCII_0 <
K_MM <
K_2 <
K_MENOS <
STRREV_ADDR <
STRREV <

                    &       /0

ITOA_FLAG       k       /0
ITOA_MOD        k       /0

ITOA_ADDR       k       /0
ITOA_ADDRCP     k       /0
ITOA_NUM        k       /0
ITOA            k       /0
                LD      ITOA_ADDR
                MM      ITOA_ADDRCP
                LD      ITOA_NUM            ; Verifica se precisa de '-'.
                JN      ITOA_ISNEG
                LV      /0
                MM      ITOA_FLAG
                JP      ITOA_LOOP
ITOA_ISNEG      LV      /0
                -       ITOA_NUM
                MM      ITOA_NUM
                LV      /1
                MM      ITOA_FLAG

ITOA_LOOP       LD      ITOA_NUM
                /       K_10
                *       K_10
                MM      ITOA_MOD
                LD      ITOA_NUM
                -       ITOA_MOD
                +       K_ASCII_0
                MM      ITOA_MOD            ; Temos ASCII do último
algarismo.
                LD      ITOA_ADDR
                +       K_MM
                MM      MM_ITOABUF0
                LD      ITOA_MOD
MM_ITOABUF0     k       /0
                LD      ITOA_ADDR
                +       K_2
                MM      ITOA_ADDR
                LD      ITOA_NUM
                /       K_10
                MM      ITOA_NUM
                JZ      CHECK_FLAG
                JP      ITOA_LOOP

CHECK_FLAG      LD      ITOA_FLAG
                JZ      RS_ITOA
                LD      ITOA_ADDR
                +       K_MM
                MM      MM_ITOABUF1
                LD      K_MENOS
MM_ITOABUF1     k       /0
```

```
                    LD      ITOA_ADDR
                    +       K_2
                    MM      ITOA_ADDR
                    JP      RS_ITOA

RS_ITOA       LD    ITOA_ADDR
                    +       K_MM
                    MM      MM_ITOABUF2
                    LD      K_0
MM_ITOABUF2   k     /0
                    LD      ITOA_ADDRCP
                    MM      STRREV_ADDR
                    SC      STRREV
                    RS      ITOA

# ITOA
```

5.8. Rotina STRLEN

```
STRLEN_ADDR >
STRLEN >

K_0 <
K_1 <
K_2 <
K_LD <

                    &       /0

RESULT        k     /0
STRLEN_ADDR   k     /0
STRLEN        k     /0
                    LD      K_0
                    MM      RESULT

LOOP          LD    STRLEN_ADDR
                    +       K_LD
                    MM      LD_STRLEN
LD_STRLEN     k     /0
                    JZ      RS_STRLEN
                    LD      STRLEN_ADDR
                    +       K_2
                    MM      STRLEN_ADDR
                    LD      RESULT
                    +       K_1
                    MM      RESULT
                    JP      LOOP

RS_STRLEN     LD    RESULT
                    RS      STRLEN
# STRLEN
```

5.9. Rotina STRREV(String reverse)

```
STRREV_ADDR >
STRREV >

K_2 <
K_LD <
K_MM <

STRLEN <
STRLEN_ADDR <

                    &       /0

TMP                 k       /0
STRREV_ADDR     k   /0
STRREV_END      k   /0
STRREV          k   /0
                    LD      STRREV_ADDR
                    MM      STRLEN_ADDR
                    SC      STRLEN
                    *       K_2
                    +       STRREV_ADDR
                    -       K_2
                    MM      STRREV_END      ; Aponta para o último caractere (não
nulo) da string.

LOOP            LD      STRREV_END
                    -       STRREV_ADDR
                    JN      RS_STRREV
                    JZ      RS_STRREV       ; Pára cópia se apontam para o mesmo ou
se ultrapassaram.
                    LD      STRREV_ADDR
                    +       K_LD
                    MM      LD_0
LD_0            k   /0
                    MM      TMP     ; Guarda caractere do começo.
                    LD      STRREV_END
                    +       K_LD
                    MM      LD_1
                    LD      STRREV_ADDR
                    +       K_MM
                    MM      MM_0
LD_1            k   /0
MM_0            k   /0                      ; Sobrescreve caractere do começo com
caractere do final
                    LD      STRREV_END
                    +       K_MM
                    MM      MM_1
                    LD      TMP
MM_1            k   /0                      ; Escreve caractere guardado (do começo)
no final.
                    LD      STRREV_ADDR
                    +       K_2
                    MM      STRREV_ADDR
                    LD      STRREV_END
                    -       K_2
                    MM      STRREV_END
                    JP      LOOP    ; Atualiza ponteiros e tenta novamente.

RS_STRREV       RS      STRREV
# STRREV
```

5.10. Tabela da pilha de operadores e operandos

Topo da pilha de operadores	Topo da pilha de operandos	Segunda posição da pilha de operandos	Código objeto a ser gerado
||	A	B	LD A JZ TEST_B JP TRUE TEST_B LD B JZ FALSE TRUE LV =1 MM TEMPi JP FIM FALSE LV =0 MM TEMPi FIM
&&	A	B	LD A JZ FALSE TEST_B LD B JZ FALSE TRUE LV =1 MM TEMPi JP FIM FALSE LV =0 MM TEMPi FIM
==	A	B	LD A - B JZ TRUE FALSE LV =0 MM TEMPi JP FIM TRUE LV =1 MM TEMPi

			FIM		
!=	A	B		LD	A
				-	B
				JZ	FALSE
			TRUE	LV	=1
				MM	TEMPi
				JP	FIM
			FALSE	LV	=0
				MM	TEMPi
			FIM		
<	A	B		LD	A
				-	B
				JN	TRUE
			FALSE	LV	=0
				MM	TEMPi
				JP	FIM
			TRUE	LV	=1
				MM	TEMPi
			FIM		
<=	A	B		LD	A
				-	B
				JN	TRUE
				JZ	TRUE
			FALSE	LV	=0
				MM	TEMPi
				JP	FIM
			TRUE	LV	=1
				MM	TEMPi
			FIM		
>	A	B		LD	B
				-	A
				JN	TRUE
			FALSE	LV	=0
				MM	TEMPi
				JP	FIM
			TRUE	LV	=1
				MM	TEMPi
			FIM		
>=	A	B		LD	B
				-	A
				JN	TRUE
				JZ	TRUE
			FALSE	LV	=0

			MM TEMPi JP FIM TRUE LV =1 MM TEMPi FIM
+	A	B	LD A + B MM TEMPi
-	A	B	LD A - B MM TEMPi
*	A	B	LD A * B MM TEMPi
/	A	B	LD A / B MM TEMPi
%	A	B	LD A / B * B MM TEMPi LD A - TEMPi MM TEMPi
!	A	VAZIO	LD A JZ GO_TRUE GO_FALSE LV =0 MM TEMPi JP FIM GO_TRUE LV =1 MM TEMPi FIM
- Obs: número negativo	A	VAZIO	LV =0 - A MM TEMPi
IF	A	VAZIO	LD A JZ FIM ; comandos FIM
IF {} ELSE	A	VAZIO	LD A JZ ELSE ;comandos JP FIM ELSE ;comandos FIM
WHILE	A	VAZIO	LOOP LD A JZ FIM ;comandos

			JP LOOP FIM
ATRIBUIÇÃO (=)	A	B	LD A MM B
ARRAY ([])	A	B	LD B + A ;offset MM TEMPi
STRUCT (.)	A	B	LD B + A ;offset MM TEMPi
PRINT	A	VAZIO	LV A MM PRINT_ADDR SC PRINT
CHAMADA DE SUBROTINA (Identificador)	A, B, ... , N Obs: depende da declaração da rotina Obs2: Número de parametros	VAZIO	**Caller** Empilha Variáveis locais Empilha N Empilha ... Empilha B Empilha A Chama Subrotina **Callee** Desempilha A Desempilha B Desempilha ... Desempilha N Empilha Endereço de Retorno
RETORNO DE SUBROTINA (return)	A ou VAZIO	VAZIO	**Callee** Desempilha endereço de retorno Empilha retorno (se tiver) Sai da Subrotina **Caller** Desempilha retorno Desempilha variáveis locais
SCAN	A	VAZIO	LV A MM SCAN_ADDR SC SCAN
ATOI	A	VAZIO	LV A; MM ATOI_ADDR SC ATOI MM TEMPI;
ITOA	A	B	MM A LV B ;variável resposta MM ITOA_ADDR SC ITOA
EMPILHA	A	VAZIO	LV A

(PUSH)			MM PUSH_ADDR LV =2 ;coloca tamanho alocado para a variavel MM PUSH_SIZE SC PUSH
DESEMPILHA (POP)	A	VAZIO	LV A MM POP_ADDR LV =2 ;coloca tamanho alocado para a variavel MM POP_SIZE SC POP