ESCOLA POLITÉCNICA DA UNIVERSIDADE DE SÃO PAULO

Departamento de Engenharia de Computação e Sistemas Digitais

PCS 2056 – Linguagens e Compiladores

Relatório Final

05/12/2011

Bruno Pezzolo dos Santos 5948816
Carla Guillen Gomes 5691366

Sumário

1. Introdução 3
2. Definição da Linguagem 4
2.1. Descrição Informal 4
Funções 4
Repetição Condicional 5
Decisão 5
Entrada 5
Saída 5
2.2. Descrição Formal 5
Notação BNF 5
Notação Wirth 7
2.3. Descrição Reduzida 9
3. Análise Léxica 11
3.1. Autômatos 12
3.2. Implementação 15
4. Análise Sintática 18
4.1. Submáquinas do Autômato de Pilha Estruturado (APE) 18
4.2. Implementação 23
5. Características da MVN 25
5.1. Instruções da linguagem de saída 25
5.2. Pseudoinstruções da linguagem de saída 26
5.3. Características gerais 26
Chamada de subrotina 28
Retorno de subrotina 28
6. Tradução de Comandos 29
6.1. Controle de fluxo 29
6.2. Comandos imperativos 30
6.3. Exemplo de programa traduzido 34
7. Análise Semântica 35
7.1. Tabela de símbolos com suporte a escopo 35

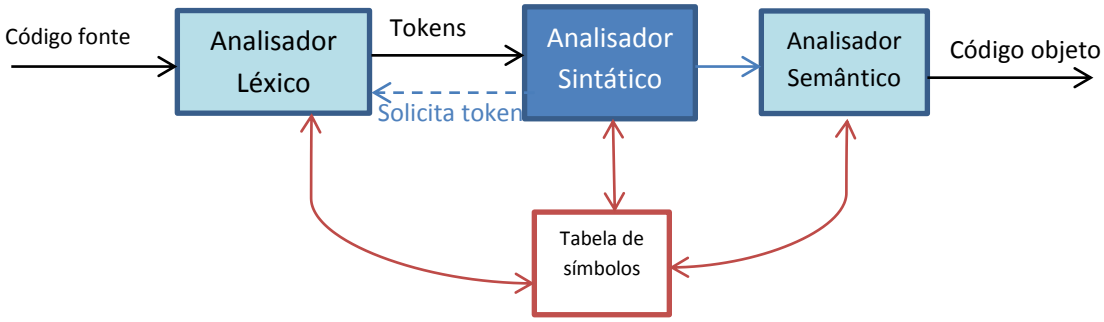
7.2. Principais ações semânticas 36
Rótulos 36
Submáquina Programa 36
Submáquina Comando 37
Submáquina Expressão 38
7.3. Ambiente de execução 39
7.4. Registro de ativação .....**Erro! Indicador não definido.**
8. Testes 40
Teste 1: input, output, if e while aninhados 40
Teste 2: operações aritméticas 41
Teste 3: operações lógicas 42
Teste 4: chamada de sub-rotina 44
9. Conclusão 45

1. 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:



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.

2. 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;
}
```

2.1. 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 ;
```

2.2. 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 = "+" | "-" | "*" | "/" .
```

2.3. 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} .
```

3. 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 numérico 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]

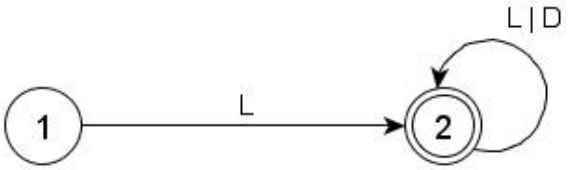
O analisador léxico implementado realiza uma simples detecção de erros, retornando *tokens* inválidos quando o conjunto de caracteres lidos não corresponde a nenhuma das expressões representadas acima.

3.1. Autômatos

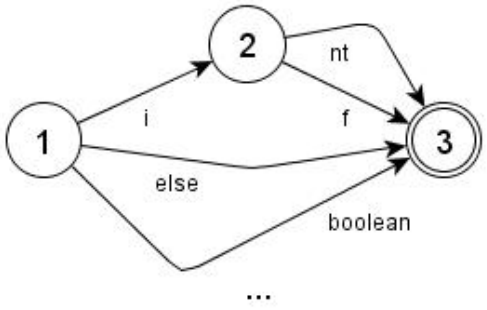
Para facilitar a representação do reconhecimento dos *tokens* pelo analisador léxico, as expressões regulares foram convertidas em autômatos finitos equivalentes, representados a seguir, onde:

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

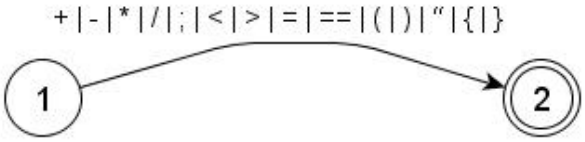
Identificador



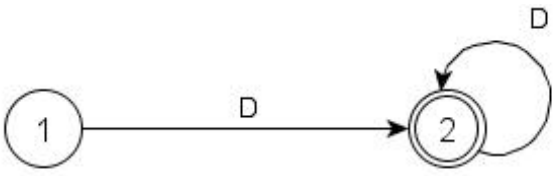
Palavra reservada



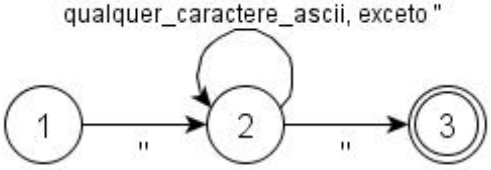
Caractere especial

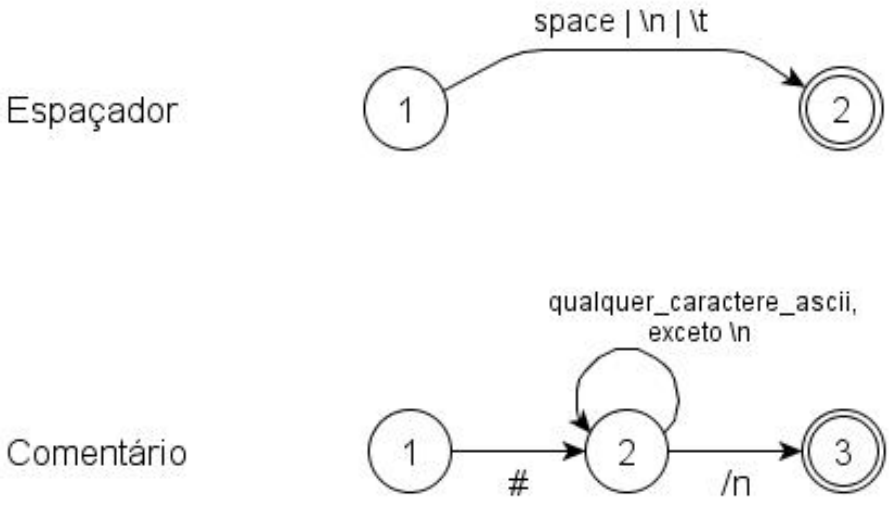


Número

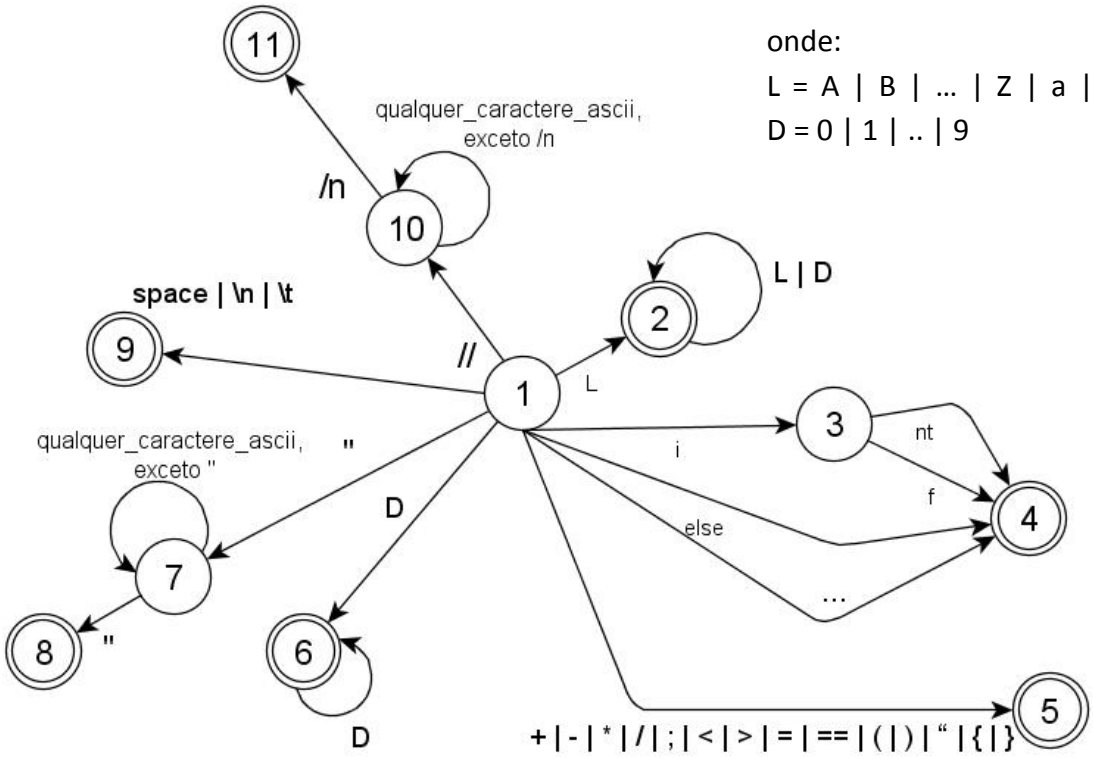


String

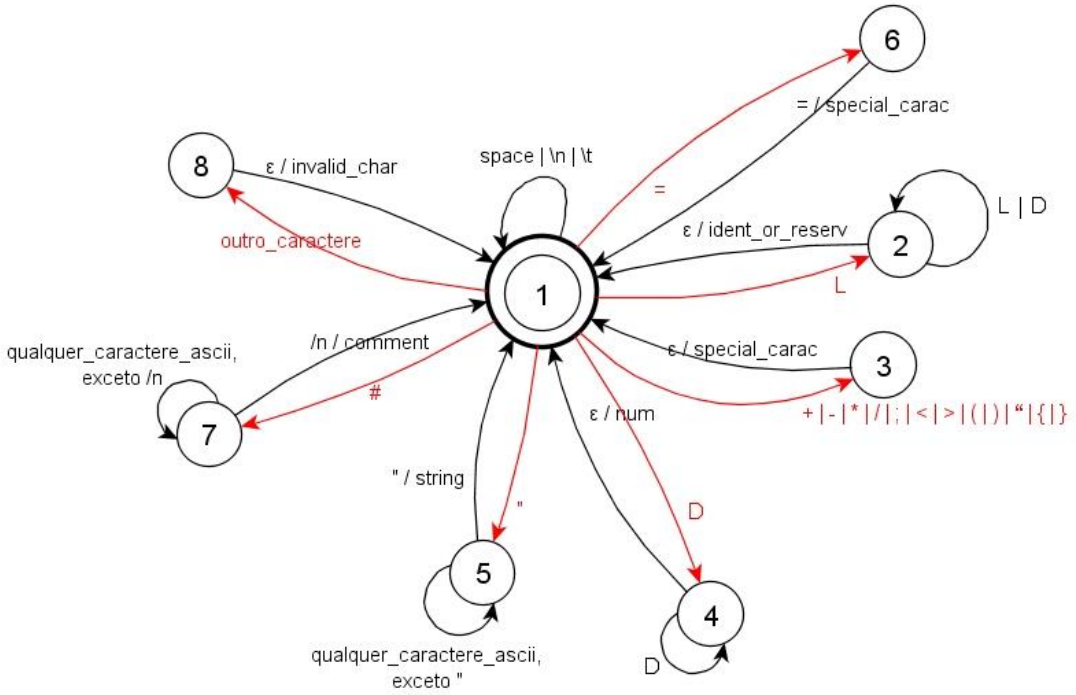




Estes autômatos reconhecem as correspondentes linguagens por eles definidas. Um autômato que reconhece todas essas linguagens, partindo de um mesmo estado inicial, mas apresentando um estado diferenciado para cada tipo de átomo é mostrado a seguir.



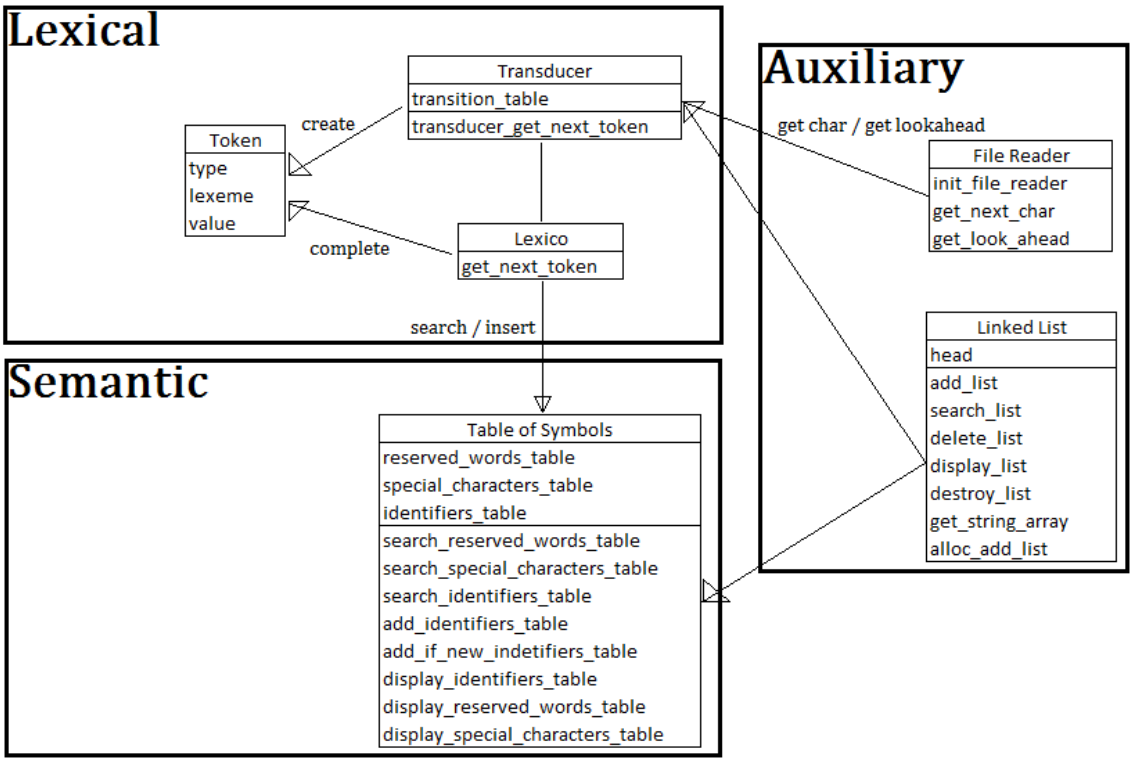
Por fim, o autômato foi convertido em transdutor, que emite como saída o átomo encontrado ao abandonar cada um dos estados finais para iniciar o reconhecimento de mais um átomo do texto.



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.

3.2. 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

```
int      TTYPE_RESERVED_WORD (2)
a        TTYPE_IDENTIFIER (0)
=        TTYPE_SPECIAL_CHARACTER (7)
1        TTYPE_NUM (1)
;        TTYPE_SPECIAL_CHARACTER (13)
int      TTYPE_RESERVED_WORD (2)
b        TTYPE_IDENTIFIER (1)
=        TTYPE_SPECIAL_CHARACTER (7)
2        TTYPE_NUM (2)
;        TTYPE_SPECIAL_CHARACTER (13)
string   TTYPE_RESERVED_WORD (3)
s        TTYPE_IDENTIFIER (2)
=        TTYPE_SPECIAL_CHARACTER (7)
ol├í!?² TTYPE_STRING (7)
;        TTYPE_SPECIAL_CHARACTER (13)
# coment├írio qualquer
         TTYPE_COMMENT (13)
int      TTYPE_RESERVED_WORD (2)
a        TTYPE_IDENTIFIER (0)
=        TTYPE_SPECIAL_CHARACTER (7)
2        TTYPE_NUM (2)
,        TTYPE_INVALID (2)
2        TTYPE_NUM (2)
;        TTYPE_SPECIAL_CHARACTER (13)
123      TTYPE_NUM (123)
#
         TTYPE_COMMENT (123)
$        TTYPE_INVALID (123)
{        TTYPE_SPECIAL_CHARACTER (11)
typedef  TTYPE_IDENTIFIER (3)
}        TTYPE_SPECIAL_CHARACTER (12)
         TTYPE_END_OF_FILE (12)
```

tokens reconhecidos no formato: lexeme TYPE (value)

Tabelas de símbolos

```
Reserved words table

0        ¦        void
1        ¦        boolean
2        ¦        int
3        ¦        string
4        ¦        main
5        ¦        def
6        ¦        if
7        ¦        else
8        ¦        while
9        ¦        return
10       ¦        true
11       ¦        false
12       ¦        and
13       ¦        or
14       ¦        not

Special characters table

0        ¦        +
1        ¦        -
2        ¦        *
3        ¦        /
4        ¦        >
5        ¦        <
6        ¦        ==
7        ¦        =
8        ¦        (
9        ¦        )
10       ¦        "
11       ¦        {
12       ¦        }
13       ¦        ;

Identifiers table

0        ¦        a
1        ¦        b
2        ¦        s
3        ¦        typedef
```

4. 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.

4.1. 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	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	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

```
(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
```

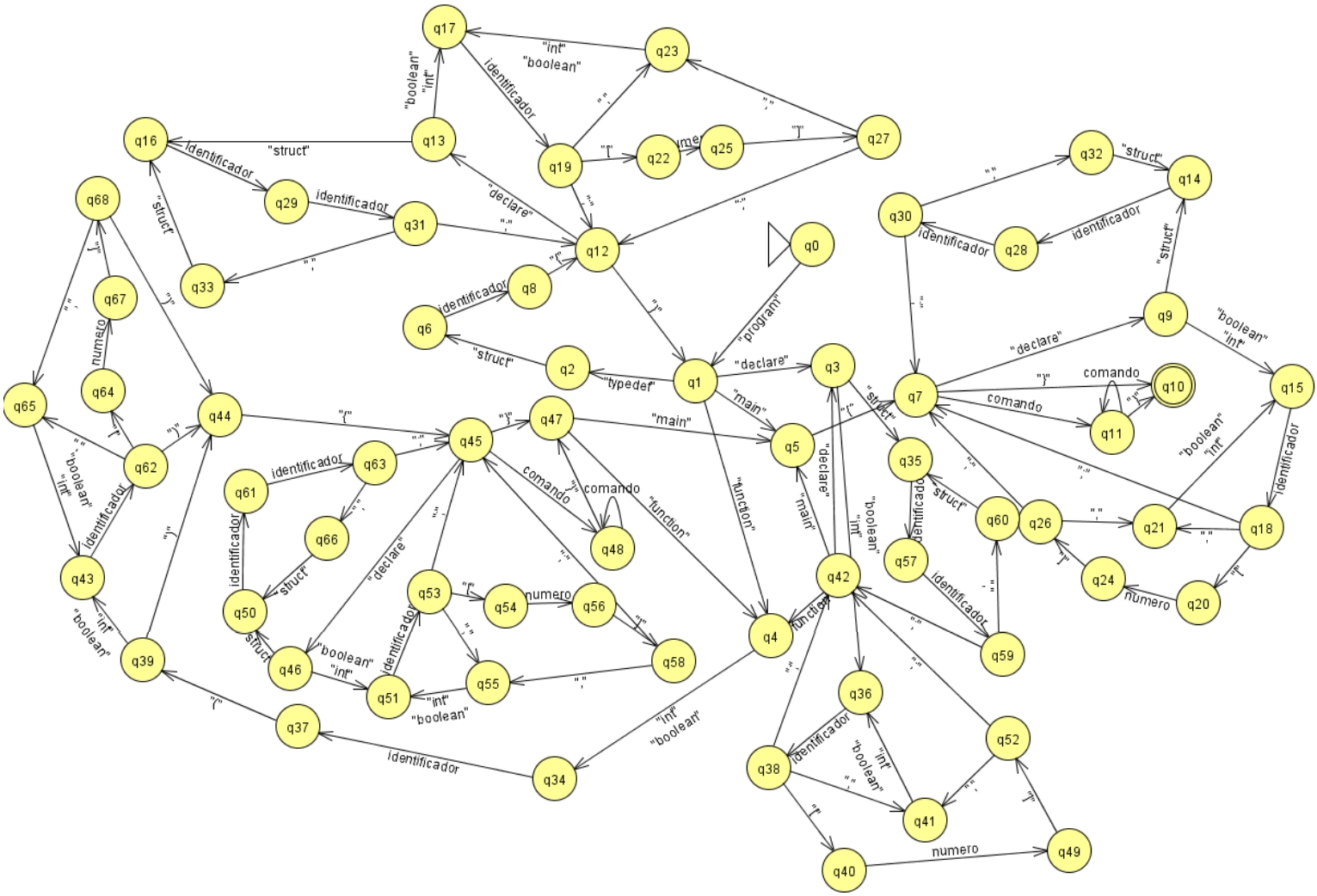
```
(16, ")") -> 17
(17, "{") -> 19
(18, expressao) -> 20
(19, "}") -> 11
(19, comando) -> 19
(20, ",") -> 18
(20, "}") -> 9
(21, expressao) -> 22
(22, ")") -> 23
(23, "{") -> 24
(24, "}") -> 25
(24, comando) -> 24
(25, "else") -> 17
```

```
(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
```

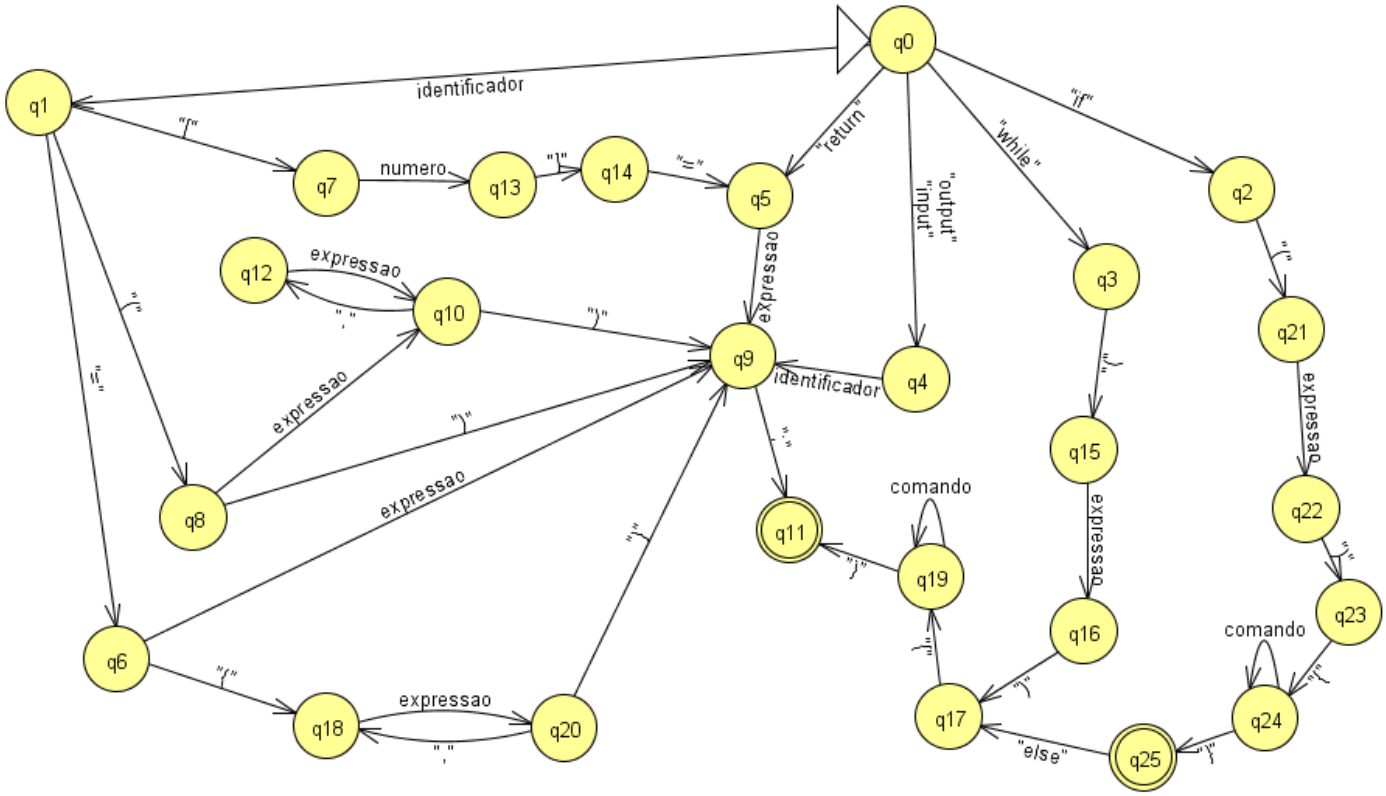
(51, identificador) -> 53 (52, ",") -> 41 (52, ";") -> 42 (53, "[") -> 54 (53, ",") -> 55 (53, ";") -> 45 (54, numero) -> 56 (55, "int") -> 51 (55, "boolean") -> 51 (56, "]") -> 58 (57, identificador) -> 59 (58, ",") -> 55 (58, ";") -> 45 (59, ",") -> 60 (59, ";") -> 42 (60, "struct") -> 35 (61, identificador) -> 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		

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

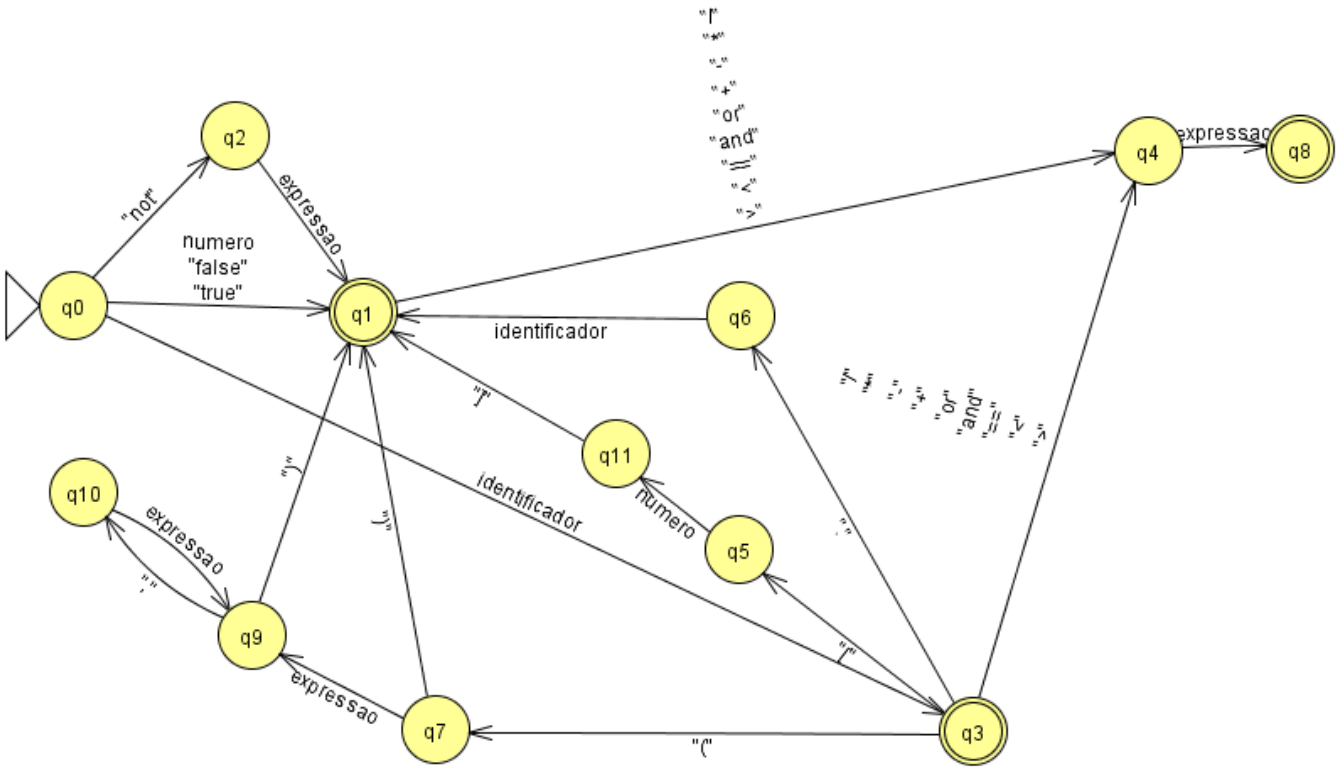
PROGRAMA


q0
q1
q2
q3
q4
q5
q6
q7
q8
q9
q10
q11
q12
q13
q14
q15
q16
q17
q18
q19
q20
q21
q22
q23
q24
q25
q26
q27
q28
q29
q30
q31
q32
q33
q34
q35
q36
q37
q38
q39
q40
q41
q42
q43
q44
q45
q46
q47
q48
q49
q50
q51
q52
q53
q54
q55
q56
q57
q58
q59
q60
q61
q62
q63
q64
q65
q66
q67
q68
"program"
"typedef"
"struct"
"declare"
"main"
"function"
"int"
"boolean"
identificador
numero
comando
"{"
"}"
";"
","

COMANDO



EXPRESSAO



4.2. 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 que se guarde a máquina atual e o seu estado corrente em uma pilha. 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 em 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.

5. 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.

5.1. 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	Parada

		desvio	
Get Data	GD	Dispositivo de E/S	Entrada
Put Data	PD	Dispositivo de E/S	Saída
Operating System	OS	Constante	Chamada de supervisor

5.2. Pseudoinstruções da linguagem de saída

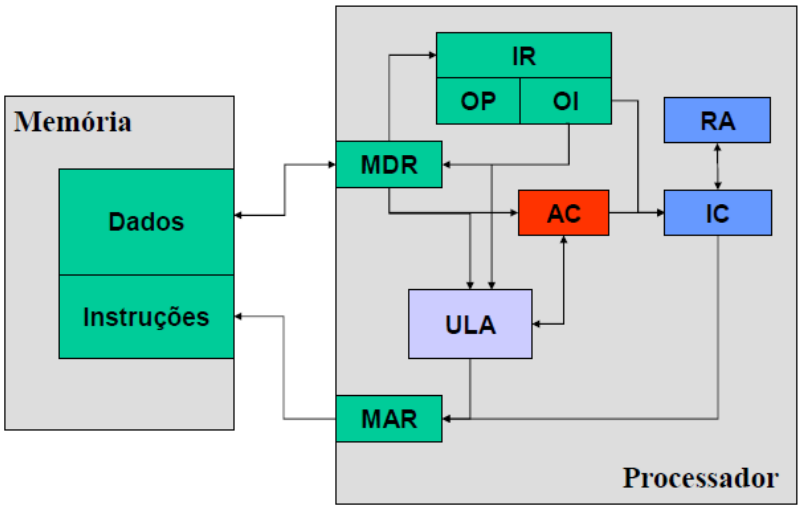
Pseudoinstrução	**Descrição**
@	Recebe um operando numérico, define o endereço da instrução seguinte, uma origem absoluta para o código a ser gerado
K	Define área preenchida por uma constante, o operando numérico tem o valor da constante de 2 bytes (em hexadecimal)
$	Define um bloco de memória com número 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

5.3. 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”.

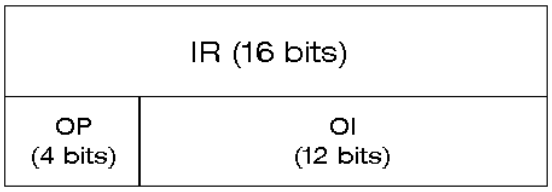
Por fim, os registradores auxiliares são utilizados em operações intermediárias. A tabela abaixo apresenta uma relação dos registradores auxiliares da MVN e suas funções. A arquitetura do simulador de MVN utilizado como ambiente de execução está representada a figura a seguir.



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 à 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, 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.

6. 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.

6.1. 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

6.2. 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 MM tam_registro_ativacao SC cria_registro_ativacao LD a; MM pos_param_1 SC funcao ;

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       ;
```

6.3. Exemplo de programa traduzido

```
program

function int fatorial_iterativo(int n) {
      declare int fat;

}

main {
      declare int fat;
     declare int n;
     n = 5;
     fat = 1;
      while (n > 0) {
            fat = fat * n;
            n = n - 1;
      }
      output fat;
}
```

```
   @ /0 ; área de código
   JP main

zero        K =0 ; constante zero

main        LD k5             ;
            MM n              ;
            LD K1             ;
            MM fat            ;
while       LD zero           ;
            +  n              ;
            -  K0             ;
            JN endwhile       ;
            JZ endwhile       ;
            LD fat            ;
            *  n              ;
            MM fat            ;
            LD n              ;
            -  K1             ;
            MM fat            ;
            JP while          ;
endwhile     LD fat           ;
            MM output_number  ;
            SC output         ;
            HM /00            ;
            #  main

   @ /200   ; área de dados
K5  K =5    ; declaracao de constante
K1  K =1    ; declaracao de constante
K0  K =0    ; declaracao de constante
n   K =0    ; declaração de constante
fat  K =0   ; declaração da variável fatorial
```

7. Análise Semântica

A análise semântica do compilador desenvolvido consiste nas ações semânticas desencadeadas pelas transições do APE. As ações semânticas consistem em imprimir no arquivo de saída o código objeto gerado conforme os *tokens* são lidos e o APE é transicionado. Foram criados dois *buffers* para armazenamento do código gerado, um correspondente a área de código e outro correspondente a área de dados. Os buffers foram implementados com a 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.

7.1. 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 implementação, o compilador desenvolvido permite programas com variáveis de mesmo nome, desde que em escopos distintos.

7.2. 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, actions_on_machine_return e actions_on_machine_return. Estas matrizes mapeiam as transições do APE com as ações semânticas correspondentes. O mapeamento 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 possuem ações definidas

Rótulos

get_constant_label

Retorna um rótulo utilizado para constantes e incrementa o contador constant_counter, utilizado na geração destes rótulos. Os rótulos de constantes são constituídos pela letra **K** seguida por um inteiro. Esta rotina também imprime no buffer de dados a constante definida.

get_temp_label

Similar à get_constant_label, mas usada na declaração de variáveis temporárias.

outras

Assim como a get_constant_label e get_temp_label, foram criadas ações auxiliares para a criação de rótulos para loops, variáveis, if's e else's.

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.

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.

7.3. 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.

8. 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

```
> r
Informe o endereco do IC [0000]:
Exibir valores dos registradores a cada passo do ciclo FDE (s/n)[s]: n
5
 8
>
> r
Informe o endereco do IC [0000]:
Exibir valores dos registradores a cada passo do ciclo FDE (s/n)[s]: n
-3
 - 2 - 1 0 1 2 3 4 5 8
>
```

Teste 2: operações aritméticas

Código fonte

```
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 objeto gerado pelo compilador

```
<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 temporario
```

Execução na MVN

```
> r
Informe o endereco do IC [0000]:
Exibir valores dos registradores a cada passo do ciclo FDE (s/n)[s]: n
10
3
 1 3 7 3 0 3
>
```

Teste 3: operações lógicas

Código fonte

```
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 objeto gerado pelo compilador

```
<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 temporario
```

Execução na MVN

```
> r
Informe o endereco do IC [0000]:
Exibir valores dos registradores a cada passo do ciclo FDE (s/n)[s]: n
 1 0 0 1 0
```

Teste 4: chamada de sub-rotina

Código fonte

```
program

function int prettyprint (int number) {
    output number;
    return number;
}

main {
    declare int number;
    input number;
    number = prettyprint(number);
}
```

Código objeto gerado pelo compilador

```
<código das sub-rotinas do ambiente de execução omitido>
; Function prettyprint
F0          JP  /0000               ;
            LD  K0                  ; Carrega variavel do RA
            MM  pos_param           ;
            SC  load_ra_pos         ;
            MM  output_number       ;
            SC  output              ;
            LD  V0                  ; Returns value
            LD  K1                  ; Carrega variavel do RA
            MM  pos_param           ;
            SC  load_ra_pos         ;
            +   return_inst         ;
            MM  _F0                 ;
_F0         K   /0000               ; Guarda o endereço de retorno
; End of function prettyprint

main        JP  /0000               ;
            LD  K2                  ; Cria registro de ativacao
            MM  ra_tam              ;
            LD  K3                  ;
            MM  ra_end              ;
            SC  cria_ra             ;
            SC  F0                  ; Chama funcao
            LD  V1                  ; Atribuicao de variavel
            MM  V1                  ;
            HM  /00                 ;
            #  P                    ;

        @ /0A00
V0            K  =0         ; Declaracao de variavel
K0            K  =5         ; Declaracao de constante
K1            K  =1         ; Declaracao de constante
V1            K  =0         ; Declaracao de variavel
K2            K  =4         ; Declaracao de constante
K3            K  =4         ; Declaracao de constante
```

9. 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.

O único requisito solicitado que não foi implementado foi 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.