PCS 2056 – Linguagens e Compiladores

Relatório – Compilador

Daniel Ziviani Cassettari NºUSP: 3370079
Rafael Nóbrega Pereira NºUSP: 3346001

Índice

1 Introdução 3
2 Definição da Linguagem 4
2.1 Definição Formal da Linguagem – Notação BNF 4
2.2 Definição Formal da Linguagem – Notação de Wirth 5
2.3 Simplificação da Gramática 7
3 Autômato finito equivalente à gramática léxica definida 8
4 Dependências de contexto da linguagem 9
5 Descrição do analisador léxico 11
5.1 Autômato Finito do analisador léxico 11
5.2 Funções do analisador léxico 12
5.3 Tabelas utilizadas pelo analisador léxico 13
5.4 Teste realizado no analisador léxico 14
6 Projeto do analisador sintático e das rotinas semânticas 17
6.1 Lógica do Reconhecimento sintático 17
6.2 Lógica do Programa 17
6.3 Estruturas de Dados Utilizadas 17
6.3.1 Tabela de símbolos de identificadores de função(tsif) 17
6.3.2 Tabela de símbolos de variáveis de uma função(tsvar) 18
6.3.3 PilhaIFs 18
6.3.4 PilhaControleIFs 18
6.3.5 PilhaLoops 18
6.3.6 PilhaRetorno 18
6.3.7 PilhaOperadores 18
6.3.8 PilhaOperandos 19
6.4 Funções do analisador e reconhecedor sintático 19
6.5 Descrição das principais rotinas semânticas 20
7 Linguagem de Montagem 22
7.1 Correspondência entre construções sintáticas e linguagem de montagem 23
8 Ambiente de Execução 25
8.1 Estrutura geral do ambiente de execução 25
8.2 Geração e execução do código de máquina virtual 25
8.3 Particionamento da Memória 26
9 Exemplo de compilação 27
9.1 Arquivo de saída gerado em linguagem de montagem 27
9.2 Arquivo gerado pelo montador 33
9.3 Resultado obtido na execução 34

1 Introdução

Este projeto tem como objetivo a construção de um compilador de um só passo, dirigido por sintaxe, com analisador e reconhecedor sintático baseado em autômato de pilha estruturado.

Primeiramente foi definida uma linguagem de programação e identificados os tipos de átomos. Para cada átomo foi escrita uma gramática linear representativa da sua lei de formação e um reconhecedor para o átomo. Desse modo, as gramáticas assim escritas foram unidas e convertidas em um autômato finito, o qual foi implementado dando origem ao analisador léxico propriamente dito.

Estando implementado o analisador léxico, iniciou-se a construção do reconhecedor sintático. Os passos seguidos para sua construção foram:

- Transformação da gramática na notação BNF para a notação de Wirth;
- Eliminação dos não-terminais desnecessários;
- Eliminação das recursões desnecessárias;
- Transformação da gramática assim preparada em autômato de pilha estruturado (foram construídos três autômatos, referentes a três submáquinas - programa, comando e expressão).

Estando projetado o reconhecedor sintático, iniciou-se o projeto das rotinas semânticas, com a correspondente associação às transições dos autômatos. Como as rotinas semânticas são responsáveis pela geração de código, paralelamente ao seu projeto, iniciou-se o projeto do ambiente de execução e o estudo da linguagem de montagem na qual seriam mapeados os comandos da linguagem projetada.

Para desenvolver o projeto foi escolhida a linguagem Java e como resultado da implementação foram criados dois arquivos .java:

- Compilador: contém a classe principal (Compilador) e a classe TabSimbIdentFunc com a sua subclasse TabSimbVar (implementam a tabela de símbolos de identificadores).

- Máquinas: Implementam as tabelas de transição do analisador e de reconhecedor sintático.

2 Definição da Linguagem

A linguagem adotada no projeto foi a que fora definida em aula, com algumas modificações. Abaixo encontra-se a descrição formal da linguagem através de uma gramática livre de contexto na notação BNF e na notação de Wirth.

2.1 Definição Formal da Linguagem – Notação BNF

<programa> ::= program <declarações de funções> begin <declarações> <seqüência de comandos> end

<declarações> ::= declare <lista de identificadores>; | ε

<lista de identificadores> ::= <identificador> <vetor> | <lista de identificadores>, <identificador> <vetor>

<vetor> ::= **[** <número> **]** | ε

<seqüência de comandos> ::= <comando> | <seqüência de comandos>; <comando>

<comando> ::= <atribuição> | <iterativo> | <condicional> | <entrada> | <saída> | <função> | ε

<atribuição> ::= <identificador> <vetor1>**:= <**expressão>

<vetor1> ::= **[** <expressão> **]** | ε

<iterativo> ::= while <condição> loop <seqüência de comandos> endloop

<condicional> ::= if <condição> then <seqüência de comandos> endif | if <condição> then <seqüência de comandos> else <seqüência de comandos> endif

<entrada> ::= input <lista de posições de memória>

<lista de posições de memória> ::= <lista de identificadores1>

<lista de identificadores1> ::= <identificador> <vetor1>

<saída> ::= output <lista de expressões>

<lista de expressões> ::= <expressão> | <lista de expressões>, <expressão>

<condição> ::= <expressão> <operador de comparação> <expressão>

<operador de comparação> :: = **>** | **<** | **>=** | **<=** | **==** | **!=**

<expressão> ::= <expressão> **+** <termo> | <expressão> **-** <termo> | <termo>

<termo> ::= <termo> ***** <fator> | <termo> **/** <fator> | <fator>

<fator> ::= <identificador> <vetor1> | <função> | <número> | **(**<expressão>**)**

<função> ::= <nome da função> **( )** | <nome da função> (<parâmetros>)

<parâmetros> **::=** <expressão> | <parâmetros>, <expressão>

<declarações de funções> ::= <declarações de funções> <tipo> function <nome da função> (<param>) <declarações> <seqüência de comandos> return <expressão> endfunction | <declarações de funções> void function <nome da função> (<param>) <declarações> <seqüência de comandos> endfunction | ε

<tipo> ::= int

<nome da função> ::= <identificador>

<param> ::= <lista de parâmetros> | ε

<lista de parâmetros> ::= <tipo> <identificador> | <lista de parâmetros>, <tipo> <identificador>

<número> ::= <dígito> | <número> <dígito>

<identificador> ::= <letra> | <identificador> <letra> | <identificador> <dígito>

<dígito> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

<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 | _

2.2 Definição Formal da Linguagem – Notação de Wirth

programa = “program” declarações de funções “begin” declarações seqüência de comandos “end”.

declarações = [“declare” lista de identificadores “;”].

lista de identificadores = identificador vetor { “,” identificador vetor }.

vetor = [“[” número “]”].

seqüência de comandos = comando { “;” comando }.

comando = [atribuição | iterativo | condicional | entrada | saída | função].

atribuição = identificador vetor1 “**:=**” expressão.

vetor1 = [“**[**” expressão “**]**”].

iterativo = “while” condição “loop” seqüência de comandos “endloop”.

condicional = “if” condição “then” seqüência de comandos “endif” | “if” condição “then” seqüência de comandos “else” seqüência de comandos “endif”.

entrada = “input” lista de posições de memória.

lista de posições de memória = lista de identificadores1.

lista de identificadores1 = identificador vetor1 .

saída = “output” lista de expressões .

lista de expressões = expressão { “,” expressão }.

condição = expressão operador de comparação expressão.

operador de comparação = “**>**” | “**<**” | “**>=**” | “**<=**” | “**= =**” | “**!=**”.

expressão = termo { (“+” | “-”) termo }.

termo = fator { (“*” | “/”) fator }.

fator = identificador vetor1 | função | número | “(” expressão “)”.

função = nome da função “(” “)” | nome da função “(” parâmetros “)”.

parâmetros = expressão { “,” expressão }.

declarações de funções = { tipo “function” nome da função “(” param “)” declarações seqüência de comandos “return” expressão “endfunction” | “void” “function” nome da função “(” param “)” declarações seqüência de comandos “endfunction” }.

tipo = “int”.

nome da função = identificador.

param = [lista de parâmetros].
lista de parâmetros = tipo identificador { “,” tipo identificador }.

número = dígito { dígito }.

identificador = letra { letra | dígito }.

dígito = 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9.

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 | _.

2.3 Simplificação da Gramática

Eliminando não-terminais desnecessários, fatorando quando possível e deixando em função dos não-terminais: *programa, comando e expressão*, e considerando *identificador*, *lista de identificadores* e *número* como terminais, temos a seguinte gramática em notação de Wirth:

programa = “program” { (“int” “function” identificador “(” [“int” identificador { “,” “int” identificador }] “)” [“declare” lista de identificadores “;”] comando { “;” comando } “return” expressão | “void” “function” identificador “(” [“int” identificador { “,” “int” identificador }] “)” [“declare” identificador [“[” número “]”] { “,” identificador [“[” número “]”] } “;”] comando { “;” comando }) “endfunction” } “begin” [“declare” identificador [“[” número “]”] { “,” identificador [“[” número “]”] } “;”] comando { “;” comando } “end”.

comando = [identificador ([“[” expressão “]”] “:=” expressão | “(” [expressão { “,” expressão }] “)”) | “while” expressão (“>” | “<” | “>=” | “<=” | “==” | “!=”) expressão “loop” comando { “;” comando } “endloop” | “if” expressão (“>” | “<” | “>=” | “<=” | “==” | “!=”) expressão “then” comando { “;” comando } [“else” comando { “;” comando }] “endif” | “input” identificador [“[” expressão “]”] | “output” expressão { “,” expressão }].

expressão = (identificador ([“[” expressão “]”] | “(” [expressão { “,” expressão }] “)”) | número | “(” expressão “)”) { (“+” | “-” | “*” | “/”) (identificador ([“[” expressão “]”] | “(” [expressão { “,” expressão }] “)”) | número) }.

3 Autômato finito equivalente à gramática léxica definida

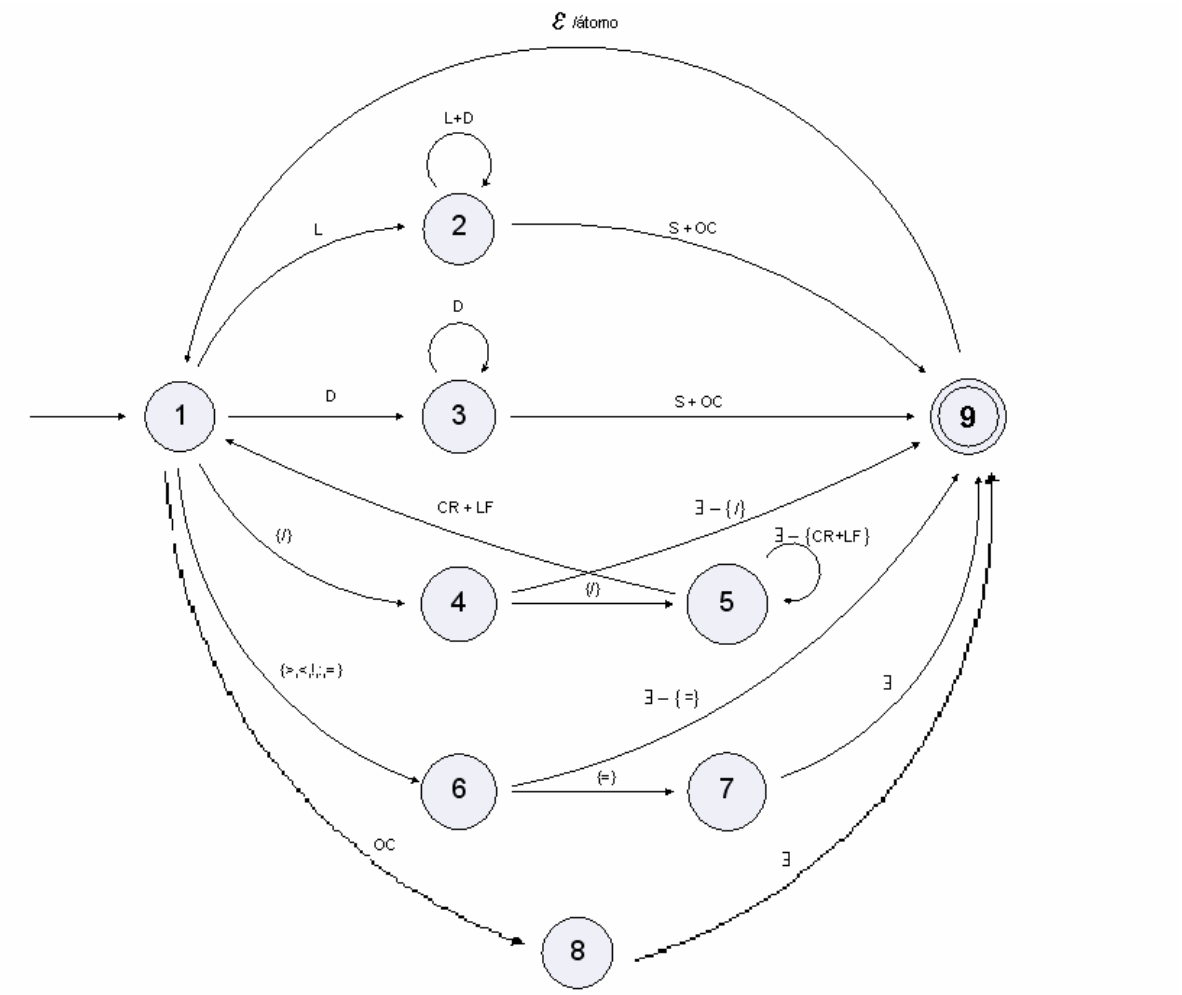

Figura 1 – Autômato finito equivalente à gramática definida

Onde:
L: letra
D: dígito
OC: outros caracteres que não L, D, /, >, <, !, :, =.
∃: qualquer caractere
CR + LF: *carrier return* ou *line feed*

4 Dependências de contexto da linguagem

Nesse tópico serão abordadas as dependências de contexto apresentadas pela linguagem descrita pela gramática em itens anteriores. Para tal, algumas observações importantes devem ser consideradas:

1) A implementação do compilador baseia-se no fato de que funções distintas de um programa devem possuir nomes distintos, caso contrário, será originado um erro.
2) Variáveis locais de uma função não podem ser acessadas pelas demais funções declaradas. Dessa forma, funções distintas podem ter variáveis locais com mesmo nome.
3) Parâmetros de funções distintas também podem ter o mesmo nome, porém o nome dos parâmetros deve ser diferente dos nomes das variáveis locais de uma determinada função, caso contrário, ocorrerá um erro de compilação.
4) Não se pode declarar variáveis globais nesta linguagem.

É importante ressaltar que para cada função declarada há uma classe chamada *tsif* (tabela de símbolos de identificadores de função) contendo informações sobre o nome da função, o tipo de retorno (*void* ou *int*), o número de parâmetros da função, um vetor que armazena a ordem dos parâmetros, o número de variáveis da função e um vetor de objetos (*tsvar*). O vetor *tsvar* armazena as variáveis (tanto variáveis locais, quanto parâmetros) e seus respectivos atributos: nome da variável; tipo (variável simples ou vetor); descritor (para o caso das variáveis que são vetores, guarda o número de posições que o vetor possui), e a informação se a variável é local ou se é um parâmetro da função.

As dependências de contexto encontradas na linguagem podem ser caracterizadas por:

• As associações entre nomes definidos em declarações e nomes utilizados em comandos da linguagem.

Para o caso de variáveis de uma função, ao se encontrar um comando que utiliza uma determinada variável (identificador) é verificado se o nome da variável consta na *tsvar* da respectiva função que está sendo compilada. Caso não exista, é originado um erro: variável não declarada (pois ela nem está declarada como um parâmetro de função e nem como uma variável local à função).

Para o caso de funções verifica-se se a função chamada consta na *tsif* (ou seja, se já foi declarada ou está sendo declarada), caso não exista um erro será originado.

• Visibilidade dos nomes em função do seu escopo.

Como foi explicado anteriormente, as funções que estão sendo declaradas só podem acessar suas variáveis locais e parâmetros (que constam em sua *tsvar*).

• Verificação de tipos das variáveis

Por meio da *tsvar* associada à respectiva função que está sendo compilada, podem ser vistos quais os tipos das variáveis e se o tipo está de acordo com o que o comando espera. Para o caso de função pode-se por meio de *tsif* verificar qual o tipo de retorno (por

exemplo, para o caso de chamada de função dentro de uma expressão, o tipo de retorno não pode ser void).

• Verificação da coerência entre a declaração dos agregados (no caso desta linguagem seriam os vetores) e as referências aos seus elementos

Por meio da *tsvar* da respectiva função em que o comando que usa a variável está inserido, pode-se por meio do atributo tipo da variável verificar se esta é um vetor (se for variável simples, ocorre um erro). Além disso, com auxílio do atributo descritor, é feita a consistência se o elemento indexado existe no vetor, ou seja, se o índice do vetor está dentro dos limites estabelecidos no momento de sua declaração (0 até o número de elementos-1).

• Verificação da coerência entre os parâmetros declarados por uma função ou sub-rotina e os argumentos passados nas chamadas dessas funções ou sub-rotinas, que ocorrem no programa.

Por meio da tabela *tsif* e da *tsvar* da função, no momento da compilação em que é encontrada uma chamada para a mesma, comparam-se os tipos dos parâmetros que a função chamada deseja receber (obtido de *tsif*) com os tipos das variáveis passadas (obtido de *tsvar*). Caso o número de parâmetros passados na chamada da função seja diferente do que é esperado, ou se o tipo da variável passada é diferente do desejado pela função chamada, um erro de compilação ocorrerá.

5 Descrição do analisador léxico

5.1 Autômato Finito do analisador léxico

O autômato finito do analisador léxico foi implementado através de comandos *switch-case* da linguagem Java:

```
while ((token = fonte.read()) != -1 && continua){
	if (token == 10){ // retira o Line Feed
		token = fonte.read();
		linha ++;
	}
	tipo = verificaCarac(token);
	atomo.append((char)token);

	switch (estadoLex) {
		case 1:
			if (tipo == 1) //letra
				estadoLex = 2;
			else if (tipo == 2) //numero
				estadoLex = 3;
			else if (tipo == 3) //simbolo especial
				estadoLex = 6;
			else if (tipo == 9){ // outros simbolos
				geraAtomo(out); //precisa testar
				estadoLex = 8;
			}else if (tipo == 4) // barra
				estadoLex = 4;
			else if (tipo == 8 && token != -1) // outros caracteres
				geraAtomo(out);
			else
				atomo = new StringBuffer(); //para nao adicionar enter,esp,tab
		break;

		case 2:
			if (tipo > 2) // se nao for nem letra e nem numero
				geraAtomo(out);
		break;

		case 3:
			if (tipo != 2)
				geraAtomo(out);
		break;

		case 4:
			if (tipo == 4)
				estadoLex = 5;
			else
				geraAtomo(out);
		break;

		case 5:
			if (tipo == 5)
				estadoLex = 1;
			atomo = new StringBuffer();
		break;

		case 6:
			if (token == 61) // simbolo de igualdade
				estadoLex = 7;
			else
				geraAtomo(out);
		break;
```

```
                case 7:
                        geraAtomo(out);
                break;

                case 8:
                        geraAtomo(out);
                break;
        }

        if (! pilhaAtomos.isEmpty()){
                atomoLido = pilhaAtomos.firstElement().toString();
                System.out.print(" recebeu " + atomoLido);
                pilhaAtomos.removeElementAt(0);
                maq.sintatico(atomoLido); // Passa o atomo para o sintatico
        }
}
```

5.2 Funções do analisador léxico

private void geraAtomo(FileWriter out)

Esta função gera um átomo, coloca-o na pilha de átomos, limpa o StringBuffer, escreve o átomo no arquivo de saída de átomos, atualiza o estado atual do analisador léxico e chama a função TrataAtomo().

private void trataAtomo(String _atomo, FileWriter out)

Esta função recebe um átomo e verifica a que classe ele pertence, bem como qual é o índice deste átomo dentro dessa classe. Em posse dessas informações, o arquivo de saída de átomos é atualizado.

private int verificaCarac(int c)

Esta função recebe um caracter do arquivo de entrada e retorna: 1 para “letra”, 2 para “número”, 3 para “símbolos especiais”, 4 para “/”, 5 para “enter”, 6 para “espaço”, 7 para “tab”, 8 para “outros caracteres” e 9 para “outros símbolos”. Esta função é usada para montar os átomos.

public int identificaClasse(String _atomo)

Esta função retorna o tipo da classe de um átomo: 1 para palavras reservadas, 2 para número, 3 para identificadores, 4 para símbolos e 5 para outros átomos.

public int identificaAtomo(String _atomo)

Esta função é similar a anterior, porém ela devolve um índice que identifica unicamente o átomo. Este índice começa com as palavras reservadas (0-20), continua com os símbolos (30-49), na seqüência o 60 representa um número, o 61 representa um identificador e por último o 62 representa outros átomos.

5.3 Tabelas utilizadas pelo analisador léxico

A primeira coisa que o compilador faz ao ser executado é criar as tabelas internas que serão utilizadas para geração do código compilado.

Tabela de palavras reservadas:

Índice	**Valor**
0	program
1	int
2	void
3	boolean
4	true
5	false
6	function
7	return
8	endFunction
9	declare
10	begin
11	while
12	loop
13	endLoop
14	input
15	output
16	if
17	then
18	else
19	endIf
20	end

Tabela de Símbolos:

Índice	**Valor**
0	<
1	>
2	!
3	=
4	:
5	;
6	,
7	+
8	-
9	*
10	/
11	(
12	)
13	[

14	]
15	<=
16	>=
17	==
18	!=
19	:=

Tabela de Classes:

Índice	**Valor**
0	Palavra Reservada (PalRes)
1	Número (Numero)
2	Identificador (Identi)
3	Símbolo (Simbol)
4	Outros (Outros)

Obs.: A implementação dessas tabelas foi feita utilizando-se vetores em Java.

5.4 Teste realizado no analisador léxico

Para testar o analisador léxico escrevemos um programa relativamente simples, que visa exemplificar o funcionamento do analisador léxico, mostrando o resultado da sua aplicação sobre o exemplo abaixo:

```
program

int function soma(int a, int b)
	declare x;
	x:= a+b;
	return x
endFunction

int function multiplica(int a, int b)
	declare x;
	x:= a * b;
	return x
endFunction

begin
	declare x, y, z[10] ;
	input y;
	x := 1;
	z[3] := ( x + y ) * ( ( 4 / 3) - 1) + 7 ;

	while x < 3 loop
		if y < 1 then
			y := 4;
		else
			y := z[3];
		endIf;
		z[9] := ( y + 9 ) * 43;
		x := x + 1;
	endLoop;
	output z[9];
end
```

Como produto do analisador léxico, foi produzida a tabela de identificadores e a tabela de átomos, com base nas tabelas internas do compilador:

Tabela de Identificadores do exemplo (observada na execução do programa):

Índice	Valor
0	soma
1	a
2	b
3	x
4	multiplica
5	y
6	z

Tabela de átomos (arquivo atomos.txt criado após a execução do programa):

program (PalRes, 0)		declare (PalRes, 9)		y (Identi, 5)
int (PalRes, 1)		x (Identi, 3)		< (Simbol, 0)
function (PalRes, 6)		, (Simbol, 6)		1 (Numero, 1)
soma (Identi, 0)		y (Identi, 5)		then (PalRes, 17)
((Simbol, 11)		, (Simbol, 6)		y (Identi, 5)
int (PalRes, 1)		z (Identi, 6)		:= (Simbol, 19)
a (Identi, 1)		[(Simbol, 13)		4 (Numero, 4)
, (Simbol, 6)		10 (Numero, 10)		; (Simbol, 5)
int (PalRes, 1)		] (Simbol, 14)		else (PalRes, 18)
b (Identi, 2)		; (Simbol, 5)		y (Identi, 5)
) (Simbol, 12)		input (PalRes, 14)		:= (Simbol, 19)
declare (PalRes, 9)		y (Identi, 5)		z (Identi, 6)
x (Identi, 3)		; (Simbol, 5)		[(Simbol, 13)
; (Simbol, 5)		x (Identi, 3)		3 (Numero, 3)
x (Identi, 3)		:= (Simbol, 19)		] (Simbol, 14)
:= (Simbol, 19)		1 (Numero, 1)		; (Simbol, 5)
a (Identi, 1)		; (Simbol, 5)		endIf (PalRes, 19)
+ (Simbol, 7)		z (Identi, 6)		; (Simbol, 5)
b (Identi, 2)		[(Simbol, 13)		z (Identi, 6)
; (Simbol, 5)		3 (Numero, 3)		[(Simbol, 13)
return (PalRes, 7)		] (Simbol, 14)		9 (Numero, 9)
x (Identi, 3)		:= (Simbol, 19)		] (Simbol, 14)
endFunction (PalRes, 8)		((Simbol, 11)		:= (Simbol, 19)
int (PalRes, 1)		x (Identi, 3)		((Simbol, 11)
function (PalRes, 6)		+ (Simbol, 7)		y (Identi, 5)
multiplica (Identi, 4)		y (Identi, 5)		+ (Simbol, 7)
((Simbol, 11)		) (Simbol, 12)		9 (Numero, 9)
int (PalRes, 1)		* (Simbol, 9)		) (Simbol, 12)
a (Identi, 1)		((Simbol, 11)		* (Simbol, 9)
, (Simbol, 6)		((Simbol, 11)		43 (Numero, 43)

int (PalRes, 1)		4 (Numero, 4)		; (Simbol, 5)
b (Identi, 2)		/ (Simbol, 10)		x (Identi, 3)
) (Simbol, 12)		3 (Numero, 3)		:= (Simbol, 19)
declare (PalRes, 9)		) (Simbol, 12)		x (Identi, 3)
x (Identi, 3)		- (Simbol, 8)		+ (Simbol, 7)
; (Simbol, 5)		1 (Numero, 1)		1 (Numero, 1)
x (Identi, 3)		) (Simbol, 12)		; (Simbol, 5)
:= (Simbol, 19)		+ (Simbol, 7)		endLoop (PalRes, 13)
a (Identi, 1)		7 (Numero, 7)		; (Simbol, 5)
* (Simbol, 9)		; (Simbol, 5)		output (PalRes, 15)
b (Identi, 2)		while (PalRes, 11)		z (Identi, 6)
; (Simbol, 5)		x (Identi, 3)		[(Simbol, 13)
return (PalRes, 7)		< (Simbol, 0)		9 (Numero, 9)
x (Identi, 3)		3 (Numero, 3)		] (Simbol, 14)
endFunction (PalRes, 8)		loop (PalRes, 12)		; (Simbol, 5)
begin (PalRes, 10)		if (PalRes, 16)		end (PalRes, 20)

Interpretação: por exemplo, o átomo program (PalRes,0*)*, indica que program é uma palavra reservada (PalRes) e seu índice no vetor de palavras reservadas é *0*.

6 Projeto do analisador sintático e das rotinas semânticas

6.1 Lógica do Reconhecimento sintático

Aplicamos o método de construção do analisador sintático baseado no autômato de pilha estruturado sobre a gramática definida. Dessa forma, o nosso analisador e reconhecedor sintático ficou definido por 3 máquinas (programa, comando e expressão). Devido ao tamanho dessas máquinas ser muito grande, elas não estão inseridas aqui neste relatório, mas elas estão desenhadas e colocadas em anexo junto com o código fonte do compilador.

Para um melhor entendimento da implementação e da definição das máquinas sugere-se a visualização do diagrama seguida da visualização do código fonte.

6.2 Lógica do Programa

Basicamente, a lógica do programa principal consiste em chamar o analisador léxico, o qual irá extrair um átomo do texto fonte, empilhá-lo na pilha de átomos; transitar pelos estados dos autômatos (submáquinas programa, expressão e comando) representados na forma de tabelas de transições, e executar as correspondentes rotinas semânticas.

A interação entre o analisador léxico e o reconhecedor sintático dá-se por meio de uma pilha de átomos. A função dessa pilha é receber os átomos extraídos para que possam ser posteriormente consumidos pelo analisador sintático. Idealmente, cada átomo extraído deveria ocasionar uma transição no analisador sintático, porém essa situação pode não ocorrer, ou seja, o átomo pode não ser consumido, dessa forma, o analisador sintático devolve o átomo para a pilha, para que possa ser consumido mais adiante.

A pilha de átomos também foi implementada por um vetor. A implementação de operações de look-ahead e chamadas de retorno de submáquina exigem que o átomo seja devolvido para a pilha do analisador léxico. Para consumir um átomo o último elemento deste vetor é removido.

6.3 Estruturas de Dados Utilizadas

6.3.1 Tabela de símbolos de identificadores de função(tsif)

Implementada através de uma classe da linguagem Java.

Atributos:

- **nome**: String que armazena o nome da função.
- **tipoRetorno:** inteiro que guarda o tipo de retorno da função: void = 0 ou inteiro= 1.
- **numParam**: inteiro que guarda o número de parâmetros da função.
- **vetParam**: vetor que guarda a ordem dos tipos de param: 0=var. simples e 1= vetor.
- **tsvar**: vetor de objetos que guarda as variáveis e os parâmetros da função.
- **numVar**: inteiro que guarda o número de variáveis da função.

6.3.2 Tabela de símbolos de variáveis de uma função(tsvar)

Implementada através de uma classe da linguagem Java(sub classe da classe tsif).
Atributos:

- **nome**: String que armazena o nome da variável ou parâmetro.
- **tipo:** inteiro que guarda o tipo da variável: variável simples = 0 ou vetor = 1.
- **descritor**: inteiro que guarda o número de posições do vetor, ou vale 0 p/ variável simples
- **locPar**: inteiro que guarda 0 se for uma variável local ou 1 se for um parâmetro de função

6.3.3 PilhaIFs

Implementada através de um vetor da linguagem Java. Essa pilha foi utilizada para a geração dos *labels* necessários para correta geração de código de IF's aninhados. Nela é empilhado apenas um número inteiro seqüencial (começa em zero e conforme surgem IF's ele é empilhado e incrementado).

6.3.4 PilhaControleIFs

Implementada através de um vetor da linguagem Java. Essa pilha foi utilizada para controlar a geração de código para ninhos somente de IF's e ninhos de IF's e ELSE's. Toda vez que aparece um *if* é empilhado um 1, e toda vez que aparecesse um *else,* esse número do topo da pilha é alterado para 0. Assim, no momento de se gerar o código, é checado se o valor do topo da pilha é 0 ou 1, gerando o código adequadamente.

6.3.5 PilhaLoops

Implementada através de um vetor da linguagem Java. Essa pilha foi utilizada para a geração dos *labels* necessários para a correta geração de código de WHILE's aninhados. Nela é empilhado apenas um número seqüencial. (começa em zero e conforme surgem WHILE's ele é empilhado e incrementado).

6.3.6 PilhaRetorno

Implementada através de um vetor da linguagem Java. Essa pilha foi utilizada para controlar o estado de retorno e à qual submáquina ele pertence no momento que ocorre um retorno de uma submáquina. Nela são empilhados um inteiro correspondente ao estado de retorno e um inteiro correspondente à submáquina de retorno..

6.3.7 PilhaOperadores

Implementada através de um vetor da linguagem Java. Essa pilha foi utilizada para a correta compilação de expressões. Nela são empilhados o tipo do átomo e um controle que vale 0 quando for lido do código ou vale 1 quando for inserido propositalmente no início de uma nova expressão..

6.3.8 PilhaOperandos

Implementada através de um vetor da linguagem Java. Essa pilha foi utilizada para o armazenamento dos operandos das expressões numéricas. Ela é utilizada na geração de código.

6.4 Funções do analisador e reconhecedor sintático

public int identAtomoMaqPrin(String _atomo)

Verifica se o átomo pertence a um grupo especial de átomos. Criada para facilitar a implementação do analisador sintático. Retorna 1 se o átomo pertencer ao grupo especial. Retorna 0, caso contrario.

public void executaRotina (int rs, FileWriter saida)

A classe Máquinas recebe um átomo e verifica qual deve ser o próximo estado. Realiza a transição e chama essa função com a devida rotina semântica. A função então executa a rotina solicitada e pode ou não gerar código no arquivo de saída.

A implementação dessa rotina é feita através do comando *switch-case* da linguagem Java.

public int verificaVar (String _nome)

Essa função verifica se já existe uma variável declarada com esse mesmo nome neste escopo. Para isso, ele percorre toda a *tsvar* da *tsif* ativa. Caso seja encontrada, a função retorna o índice dessa variável. Caso contrário, retorna -1.

public void InsereFunc (String _nome, int _tipo, int _numPar)

Essa função insere uma nova função na tabela de tsif. Inicialmente o programa insere um valor nessa tabela (índice 0) que representa a função principal.

public int verificaFunc (String _nome)

Essa função verifica se já existe uma função declarada com esse mesmo nome. Para isso, ele percorre toda a *tsif*. Caso seja encontrada, a função retorna o índice dessa função. Caso contrário, retorna -1.

public int existeVarOutroEscopo (String _nome)

Essa função verifica se já existe uma variável declarada com esse mesmo nome em outro escopo. Para isso, ele percorre toda a *tsvar* de todas as *tsif* existentes. Caso seja encontrada, a função retorna 1. Caso contrário, retorna 0.

6.5 Descrição das principais rotinas semânticas

As rotinas semânticas 3, 10 e 14 apesar de inicialmente previstas não foram necessárias.

RS2: Verifica se o identificador já existe na tabela de identificadores. Se já existe: erro "função já foi declarada", senão:

Verifica se o nome da função consta em alguma *tsvar* já criada.Se sim, erro: "função com mesmo nome que variável já declarada",senão:

Cria uma linha na *tsif*.

RS4: Verifica se o identificador já foi declarado nessa função. Se sim gera erro, senão,
Se ele não tiver o mesmo nome que a função, então insere.
Empilha numVar na pilha de parâmetros.

RS7: Seta o campo tipo como 1(vetor) na *tsvar* dada pelas variáveis funcao e identAux..

RS8: Seta o campo descritor usando as variáveis identAux e função.

RS9: Salva o contexto e chama a submáquina Expressão.

RS11: Limpa as variáveis de controle TipoFunc, funcao e identAux.

RS12: Salva o contexto e chama a submáquina Comando.

RS13: Cria a área de dados da memória; percorre toda a *tsif* e imprime o nome de todas as variáveis do programa; cria a área de pilha e grava zero em todas as posições.

RS30: Armazena a variável numWhile na pilha de loops e em seguida incrementa-a; recupera elemento do topo da pilha do loop e gerar o código LOOP (elemento da pilha);

RS32: Gera código apropriado para as rotinas de comparação envolvendo Whiles.

RS33: Desempilha elemento do topo da pilha do loop e gerar código: JP LOOP (elemento); gera código: ENDLOOP (elemento).

RS34: Empilha o número de Ifs e em seguida incrementa-o.

RS36: Gera código apropriado para as rotinas de comparação envolvendo IFs.

RS39: Verifica se o identificador consta na *tsvar* da função corrente. Se não consta:

Verifica se existe em algum outro escopo. Gera as mensagens de erro de escopo, ou de variável não declarada, se for o caso.

Verifica se a variável é um parâmetro ou é uma variável local e empilha o operando.

RS40: Recupera o endereço empilhado e move o conteúdo de input para lá.

RS42: Imprime o valor da expressão que acabou de ser calculada.

RS44: Verifica se TipoFunc =2 está tentando fazer atribuição com nome de função.
A partir do valor de identAux, obtém-se o valor do campo descritor da linha de *tsvar* correspondente à variável (usar a variável function e identAux para chegar à linha da *tsvar*).

RS48: Verifica se o número de parâmetros passados na chamada de função está correto.

RS49: Restaura o contexto salvo na pilha de retorno.

RS75: Gera código adequado para as operações aritméticas aninhadas.

RS80: Gera código adequado para as operações aritméticas.

RS100: Gera mensagem de erro sintático.

7 Linguagem de Montagem

A linguagem de montagem que será utilizada como código objeto é a mesma vista em aula e que nos foi fornecida. Ela tem as seguintes características:

- Cada instrução é composta de um mnemônico e o seu operando.
- Entre os elementos de uma linha deve haver ao menos um espaço.
- As instruções podem ser escritas com um operando numérico ou com um operando simbólico(*label).*
- Se o primeiro caracter de uma linha for um espaço, esta linha não caracteriza um *label.*
- À direita de um ponto-e-vírgula, todo texto é ignorado (=comentário).
- Mnemônicos e significado das pseudo-instruções:
    - @ (Operando numérico: define endereço da instrução seguinte)
    - # (Final físico do texto-fonte. Operando=endereço de execução)
    - K (Constante. Operando numérico = valor da constante, em hexadecimal)
    - / Hexadecimal
    - = Decimal

OpCode	**Mnemônico**	**Ação**
0	JP	Pulo incondicional
1	JZ	Pulo se AC = 0
2	JN	Pulo se AC < 0
3	LV	Carrega valor no AC
4	+	Soma no AC
5	-	Subtrai no AC
6	*	Multiplica no AC
7	/	Divisão inteira no AC
8	LD	Carrega conteúdo de posição de memória no AC
9	MM	Guarda conteúdo do AC na posição de memória
A	SC	Chamada de subrotina
B	RS	Retorno de subrotina
C	HM	Halt
D	GD	Input
E	PD	Output
F	OS	Nop

7.1 Correspondência entre construções sintáticas e linguagem de montagem

Alto nível	**Montagem**	**Alto Nível**	**Montagem**
a := b	LD b ; MM a ;		
a := a + b	LD a ; + b ; MM a ;	a := a * b	LD a ; * b ; MM a ;
a := a - b	LD a ; - b ; MM a ;	a := a / b	LD a ; / b ; MM a ;

If (a) > (b) then
<ação1>
else
<ação2>
endif
	LD a ;
	- b ;
	JN else ;
	JZ else ;
	<ação1>
	JP endIf;
	else OS =0 ;
	<ação2>
	endIf OS =0 ;
		While (a) > (b) loop
		<ação>
		endloop;
			loop LD a ;
			- b ;
			JN endLoop ;
			JZ endLoop ;
			<ação1>
			JP loop ;
			endLoop OS =0 ;

If (a) < (b) then
<ação1>
else
<ação2>
endif
	LD b ;
	- a ;
	JN else ;
	JZ else ;
	<ação1>
	JP endIf;
	else OS =0 ;
	<ação2>
	endIf OS =0 ;
		While (a) < (b) loop
		<ação>
		endloop;
			loop LD b ;
			- a ;
			JN endLoop ;
			JZ endLoop ;
			<ação1>
			JP loop ;
			endLoop OS =0 ;

If (a) >= (b) then
<ação1>
else
<ação2>
endif
	LD a ;
	- b ;
	JN else ;
	<ação1>
	JP endIf;
	else OS =0 ;
	<ação2>
	endIf OS =0 ;
		While (a) >= (b) loop
		<ação>
		endloop;
			loop LD a ;
			- b ;
			JN endLoop ;
			<ação1>
			JP loop ;
			endLoop OS =0 ;

Alto nível	**Montagem**	**Alto Nível**	**Montagem**

If (a) <= (b) then
<ação1>
else
<ação2>
endif
	LD b ;
	- a ;
	JN else ;
	<ação1>
	JP endIf ;
	else OS =0 ;
	<ação2>
	endIf OS =0 ;
		While (a) <= (b) loop
		<ação>
		endloop;
			loop LD b ;
			- a ;
			JN endLoop ;
			<ação1>
			JP while ;
			endLoop OS =0 ;

If (a) == (b) then
<ação1>
else
<ação2>
endif
	LD a ;
	- b ;
	JZ acerto ;
	JP else ;
	acerto OS =0;
	<ação1>
	JP endIf;
	else OS =0 ;
	<ação2>
	endIf OS =0 ;
		While (a) == (b) loop
		<ação>
		endloop;
			loop LD a ;
			- b ;
			JZ acerto ;
			JP endLoop ;
			acerto OS =0;
			<ação1>
			JP loop ;
			endLoop OS =0 ;

If (a) != (b) then
<ação1>
else
<ação2>
endif
	LD a ;
	- b ;
	JZ else ;
	<ação1>
	JP endIf ;
	else OS =0 ;
	<ação2>
	endIf OS =0 ;
		While (a) != (b) loop
		<ação>
		endloop;
			loop OS = 0;
			LD a ;
			- b ;
			JZ endLoop ;
			<ação1>
			JP loop ;
			endLoop OS =0 ;

Input a;	GD =0 ; MM a ;		
Output a;	LD a ; PD =1 ;		

8 Ambiente de Execução

8.1 Estrutura geral do ambiente de execução

O ambiente de execução conta com oito registradores específicos, uma memória de 4096 posições e uma pilha, que foi implementada na máquina virtual.

Registrador	Função
MAR	Registrador de endereço de memória
MBR	Registrador de dados da memória
IC	Registrador de endereço da próxima instrução
IR	Registrador de instrução
OP	Registrador de código de operação
OI	Registrador de operando de instrução
RA	Registrador de endereço de retorno
AC	Acumulador

Quando ocorre uma chamada de subrotina, o endereço de retorno é guardado no RA. Porém, para que ocorra o encadeamento de subrotinas, este endereço também é guardado na pilha da máquina virtual. Quando ocorre o retorno de subrotina, este endereço é retirado da pilha e fica disponível no RA e o seu parâmetro de retorno fica disponível no AC.

Desta forma, para se executar uma chamada de função é necessário primeiramente atribuir aos parâmetros dessa função os valores que se deseja passar para a função, para só depois se chamar a função.

8.2 Geração e execução do código de máquina virtual

A execução do compilador tem como saída o programa de alto nível em código objeto (*assembly*). Para que este programa seja executado, ele deve ser transformado em código de máquina.

Essa transformação se dá com o auxílio do montador que nos foi fornecido. Este programa verifica se existem erros no código objeto gerado e, em caso negativo, gera o código de máquina compatível com a máquina virtual fornecida.

A máquina virtual carrega este código e executa, mostrando opcionalmente o conteúdo dos seus registradores.

8.3 Particionamento da Memória

Para possibilitar um maior controle a memória foi divida em 3 partes:

- **Área de programa:** Essa área é responsável por armazenar o código do programa. Ela começa na posição 0 da memória e se estende até a posição 1999. Portanto não deve se tentar executar programas que ocupem uma área maior do que essa.

- **Área de dados:** Essa área é responsável por armazenar os dados das variáveis. Ela começa a partir da posição 2000 e se estende até o final da memória. Não há um controle do final dessa área com o início da área de pilha. Esse controle deve ser feito pelo programador, caso contrário, as áreas correm o risco de serem sobrepostas caso o programa seja muito extenso.

- **Área de Pilha:** Essa área corresponde às 100 últimas posições da memória e auxilia no cálculo e no armazenamento de variáveis temporárias.

9 Exemplo de compilação

O compilador funciona da seguinte maneira:

- O arquivo a ser compilado deve estar no mesmo diretório do compilador e deve ter o nome "fonte.txt";

- Se durante a compilação algum problema for encontrado, o compilador exibirá uma mensagem no console informando a linha onde o erro foi encontrado, juntamente com o problema detectado;

- Se não houver nenhum erro de compilação o compilador irá gerar, também no mesmo diretório, um arquivo chamado "átomos.txt" com todo o resultado da análise léxica e um outro arquivo chamado "saida.txt" com o resultado da compilação exibida em linguagem de montagem.

- Para verificar se a compilação está correta utilizamos o montador fornecido para gerar a linguagem objeto e finalmente executamos o MVN, também fornecido, para executar o código gerado.

9.1 Arquivo de saída gerado em linguagem de montagem

Utilizando o mesmo exemplo de arquivo fonte que utilizamos para demonstrar o funcionamento do analisador léxico, obtivemos a seguinte resposta como saída do compilador (arquivo saida.txt).

```
                @ =0 ;          area de programa
                JP INICIO ;

F_soma                  OS =0 ;
                LD E1:a ;
                MM P0 ;         empilha
                LD E1:b ;
                MM P1 ;         empilha
                LD P0 ;                 desempilha
                + P1 ;          soma
                MM P0 ;         empilha
                LD P0 ;                 desempilha
                MM E1:x ;
                LD E1:x ;
                MM P0 ;         empilha
                RS F_soma ;
```

```
F_multiplica        OS =0 ;
                    LD E2:a ;
                    MM P0 ;     empilha
                    LD E2:b ;
                    MM P1 ;     empilha
                    LD P0 ;             desempilha
                    * P1 ;      multiplica
                    MM P0 ;     empilha
                    LD P0 ;             desempilha
                    MM E2:x ;
                    LD E2:x ;
                    MM P0 ;     empilha
                    RS F_multiplica ;
INICIO              OS =0 ;
                    GD =0 ;
                    MM E0:y ;
                    LV =1 ;
                    MM P0 ;             empilha
                    LD P0 ;             desempilha
                    MM E0:x ;
                    LD E0:x ;
                    MM P0 ;     empilha
                    LD E0:y ;
                    MM P1 ;     empilha
                    LD P0 ;             desempilha
                    + P1 ;      soma
                    MM P0 ;     empilha
                    LV =4 ;
                    MM P1 ;             empilha
                    LV =3 ;
                    MM P2 ;             empilha
                    LD P1 ;             desempilha
                    / P2 ;      divide
                    MM P1 ;     empilha
                    LV =1 ;
                    MM P2 ;             empilha
                    LD P1 ;             desempilha
                    - P2 ;      subtrai
                    MM P1 ;     empilha
                    LD P0 ;             desempilha
                    * P1 ;      multiplica
                    MM P0 ;     empilha
                    LV =7 ;
                    MM P1 ;             empilha
                    LD P0 ;             desempilha
                    + P1 ;      soma
                    MM P0 ;     empilha
```

```
                LD P0 ;                 desempilha
                MM E0:z3 ;
LOOP0           OS =0 ;
                LD E0:x ;
                MM P0 ;         empilha
                LD E0:y ;
                MM P1 ;         empilha
                LV =3 ;
                MM P2 ;                 empilha
                LD P2 ;                 desempilha
                - P1 ;          compara
                JZ ENDLOOP0 ;
                JN ENDLOOP0 ;
                LD E0:y ;
                MM P1 ;         empilha
                LV =1 ;
                MM P2 ;                 empilha
                LD P2 ;                 desempilha
                - P1 ;          compara
                JZ ELSE0 ;
                JN ELSE0 ;
                LV =4 ;
                MM P1 ;                 empilha
                LD P1 ;                 desempilha
                MM E0:y ;
                JP ENDIF0 ;
ELSE0           OS =0 ;
                LD E0:z3 ;
                MM P1 ;                 empilha
                LD P1 ;                 desempilha
                MM E0:y ;
ENDIF0          OS =0 ;
                LD E0:y ;
                MM P1 ;         empilha
                LV =9 ;
                MM P2 ;                 empilha
                LD P1 ;                 desempilha
                + P2 ;          soma
                MM P1 ;         empilha
                LV =43 ;
                MM P2 ;                 empilha
                LD P1 ;                 desempilha
                * P2 ;          multiplica
                MM P1 ;         empilha
                LD P1 ;                 desempilha
                MM E0:z9 ;
                LD E0:x ;
```

```
                MM P1 ;         empilha
                LV =1 ;
                MM P2 ;                 empilha
                LD P1 ;                 desempilha
                + P2 ;          soma
                MM P1 ;         empilha
                LD P1 ;                 desempilha
                MM E0:x ;
                JP LOOP0 ;
ENDLOOP0        OS =0 ;
                LD E0:z9 ;
                MM P1 ;                 empilha
                PD =1 ;
                HM /0 ;
                # /0 ;

                @ =2000 ;               area de dados
E0:x            JP =0 ;
E0:y            JP =0 ;
E0:z9           JP =0 ;
E0:z8           JP =0 ;
E0:z7           JP =0 ;
E0:z6           JP =0 ;
E0:z5           JP =0 ;
E0:z4           JP =0 ;
E0:z3           JP =0 ;
E0:z2           JP =0 ;
E0:z1           JP =0 ;
E0:z0           JP =0 ;
E1:a            JP =0 ;
E1:b            JP =0 ;
E1:x            JP =0 ;
E2:a            JP =0 ;
E2:b            JP =0 ;
E2:x            JP =0 ;

                @ =3895 ; area de pilha
P0      JP =0 ;
P1      JP =0 ;
P2      JP =0 ;
P3      JP =0 ;
P4      JP =0 ;
P5      JP =0 ;
P6      JP =0 ;
P7      JP =0 ;
P8      JP =0 ;
P9      JP =0 ;
```

```
P10    JP =0 ;
P11    JP =0 ;
P12    JP =0 ;
P13    JP =0 ;
P14    JP =0 ;
P15    JP =0 ;
P16    JP =0 ;
P17    JP =0 ;
P18    JP =0 ;
P19    JP =0 ;
P20    JP =0 ;
P21    JP =0 ;
P22    JP =0 ;
P23    JP =0 ;
P24    JP =0 ;
P25    JP =0 ;
P26    JP =0 ;
P27    JP =0 ;
P28    JP =0 ;
P29    JP =0 ;
P30    JP =0 ;
P31    JP =0 ;
P32    JP =0 ;
P33    JP =0 ;
P34    JP =0 ;
P35    JP =0 ;
P36    JP =0 ;
P37    JP =0 ;
P38    JP =0 ;
P39    JP =0 ;
P40    JP =0 ;
P41    JP =0 ;
P42    JP =0 ;
P43    JP =0 ;
P44    JP =0 ;
P45    JP =0 ;
P46    JP =0 ;
P47    JP =0 ;
P48    JP =0 ;
P49    JP =0 ;
P50    JP =0 ;
P51    JP =0 ;
P52    JP =0 ;
P53    JP =0 ;
P54    JP =0 ;
P55    JP =0 ;
P56    JP =0 ;
```

```
P57    JP =0 ;
P58    JP =0 ;
P59    JP =0 ;
P60    JP =0 ;
P61    JP =0 ;
P62    JP =0 ;
P63    JP =0 ;
P64    JP =0 ;
P65    JP =0 ;
P66    JP =0 ;
P67    JP =0 ;
P68    JP =0 ;
P69    JP =0 ;
P70    JP =0 ;
P71    JP =0 ;
P72    JP =0 ;
P73    JP =0 ;
P74    JP =0 ;
P75    JP =0 ;
P76    JP =0 ;
P77    JP =0 ;
P78    JP =0 ;
P79    JP =0 ;
P80    JP =0 ;
P81    JP =0 ;
P82    JP =0 ;
P83    JP =0 ;
P84    JP =0 ;
P85    JP =0 ;
P86    JP =0 ;
P87    JP =0 ;
P88    JP =0 ;
P89    JP =0 ;
P90    JP =0 ;
P91    JP =0 ;
P92    JP =0 ;
P93    JP =0 ;
P94    JP =0 ;
P95    JP =0 ;
P96    JP =0 ;
P97    JP =0 ;
P98    JP =0 ;
P99    JP =0 ;
```

9.2 Arquivo gerado pelo montador

O montador gerou o seguinte arquivo em linguagem objeto.

0000 0036		0078 9f37		00f0 e001		0f87 0000
0002 f000		007a 8f37		00f2 c000		0f89 0000
0004 87e8		007c 97e0		07d0 0000		0f8b 0000
0006 9f37		007e f000		07d2 0000		0f8d 0000
0008 87ea		0080 87d0		07d4 0000		0f8f 0000
000a 9f39		0082 9f37		07d6 0000		0f91 0000
000c 8f37		0084 87d2		07d8 0000		0f93 0000
000e 4f39		0086 9f39		07da 0000		0f95 0000
0010 9f37		0088 3003		07dc 0000		0f97 0000
0012 8f37		008a 9f3b		07de 0000		0f99 0000
0014 97ec		008c 8f3b		07e0 0000		0f9b 0000
0016 87ec		008e 5f39		07e2 0000		0f9d 0000
0018 9f37		0090 10ea		07e4 0000		0f9f 0000
001a b002		0092 20ea		07e6 0000		0fa1 0000
001c f000		0094 87d2		07e8 0000		0fa3 0000
001e 87ee		0096 9f39		07ea 0000		0fa5 0000
0020 9f37		0098 3001		07ec 0000		0fa7 0000
0022 87f0		009a 9f3b		07ee 0000		0fa9 0000
0024 9f39		009c 8f3b		07f0 0000		0fab 0000
0026 8f37		009e 5f39		07f2 0000		0fad 0000
0028 6f39		00a0 10ae		0f37 0000		0faf 0000
002a 9f37		00a2 20ae		0f39 0000		0fb1 0000
002c 8f37		00a4 3004		0f3b 0000		0fb3 0000
002e 97f2		00a6 9f39		0f3d 0000		0fb5 0000
0030 87f2		00a8 8f39		0f3f 0000		0fb7 0000
0032 9f37		00aa 97d2		0f41 0000		0fb9 0000
0034 b01c		00ac 00b8		0f43 0000		0fbb 0000
0036 f000		00ae f000		0f45 0000		0fbd 0000
0038 d000		00b0 87e0		0f47 0000		0fbf 0000
003a 97d2		00b2 9f39		0f49 0000		0fc1 0000
003c 3001		00b4 8f39		0f4b 0000		0fc3 0000
003e 9f37		00b6 97d2		0f4d 0000		0fc5 0000
0040 8f37		00b8 f000		0f4f 0000		0fc7 0000
0042 97d0		00ba 87d2		0f51 0000		0fc9 0000
0044 87d0		00bc 9f39		0f53 0000		0fcb 0000
0046 9f37		00be 3009		0f55 0000		0fcd 0000
0048 87d2		00c0 9f3b		0f57 0000		0fcf 0000
004a 9f39		00c2 8f39		0f59 0000		0fd1 0000
004c 8f37		00c4 4f3b		0f5b 0000		0fd3 0000
004e 4f39		00c6 9f39		0f5d 0000		0fd5 0000
0050 9f37		00c8 302b		0f5f 0000		0fd7 0000

0052 3004		00ca 9f3b		0f61 0000		0fd9 0000
0054 9f39		00cc 8f39		0f63 0000		0fdb 0000
0056 3003		00ce 6f3b		0f65 0000		0fdd 0000
0058 9f3b		00d0 9f39		0f67 0000		0fdf 0000
005a 8f39		00d2 8f39		0f69 0000		0fe1 0000
005c 7f3b		00d4 97d4		0f6b 0000		0fe3 0000
005e 9f39		00d6 87d0		0f6d 0000		0fe5 0000
0060 3001		00d8 9f39		0f6f 0000		0fe7 0000
0062 9f3b		00da 3001		0f71 0000		0fe9 0000
0064 8f39		00dc 9f3b		0f73 0000		0feb 0000
0066 5f3b		00de 8f39		0f75 0000		0fed 0000
0068 9f39		00e0 4f3b		0f77 0000		0fef 0000
006a 8f37		00e2 9f39		0f79 0000		0ff1 0000
006c 6f39		00e4 8f39		0f7b 0000		0ff3 0000
006e 9f37		00e6 97d0		0f7d 0000		0ff5 0000
0070 3007		00e8 007e		0f7f 0000		0ff7 0000
0072 9f39		00ea f000		0f81 0000		0ff9 0000
0074 8f37		00ec 87d4		0f83 0000		0ffb 0000
0076 4f39		00ee 9f39		0f85 0000		0ffd 0000

9.3 Resultado obtido na execução

Utilizando o programa fornecido MVN e fornecendo como entrada o valor 1, obtivemos a seguinte resposta de execução.