Talita Ludmila de Lima

Implementação do Compilador C-

São José dos Campos - Brasil

Outubro de 2020

Talita Ludmila de Lima

Implementação do Compilador C-

Relatório apresentado à Universidade Federal de São Paulo como parte dos requisitos para aprovação na disciplina de Laboratório de Sistemas Computacionais: Compiladores.

Docente: Prof. Dr. Luiz Eduardo Galvão Martins

Universidade Federal de São Paulo - UNIFESP

Instituto de Ciência e Tecnologia - Campus São José dos Campos

São José dos Campos - Brasil

Outubro de 2020

Resumo

O objetivo deste projeto é o desenvolvimento de um compilador para linguagem C- para máquina MIPS composto por módulos de análise e síntese. Inicialmente a linguagem C- foi caracterizada para que se pudesse gerar um analisador léxico através da ferramenta Lex Flex e um analisador sintático através da ferramenta YACC-Bison com base nas definições e regras da gramática livre de contexto da linguagem C-. Baseando-se nessas definições uma árvore sintática é gerada e então percorrida em ordem pré-ordem para que se possa realizar a análise semântica, verificando a presença de incoerências passíveis de verificação antes da execução de um programa, finalizando a etapa de análise. A fase de síntese se inicia com a geração de código intermediário, que é realizado percorrendo-se a árvore sintática em pré-ordem e, simultaneamente, linearizando o código e invocando a função de geração de código Assembly, que é um código passível de transformação direta para o código executável pela máquina MIPS, finalizando a etapa de síntese. A validação do projeto ocorre através da execução de algoritmos de teste.

Palavras-chaves: compilador. linguagem c-. máquina MIPS

Lista de ilustrações

Lista de tabelas

Sumário

1 Introdução

Simultaneamente ao surgimento de sistemas computacionais capazes de armazenar programas em memória para então executá-los surgiu a necessidade de se escrever tais programas que essas máquinas fossem capazes de compreender.

Inicialmente os programas eram escritos na própria linguagem da máquina, de forma que cada máquina disponibilizava um conjunto específico de instruções, as quais os programadores deveriam conhecer. Dessa forma, não havia padrão de linguagem de máquina, fato que aumentava a carga de trabalho dos programadores, que precisavam conhecer conjuntos diferentes de instruções para máquinas diferentes.

A linguagem de montagem surgiu para amenizar essa tarefa, de forma que as instruções e endereços adotam uma forma simbólica e um montador traduz esse código simbólico para o código correspondente em linguagem de máquina. Contudo, a linguagem de máquina é mais próxima da linguagem de máquina do que da linguagem natural humana, tornando esses códigos difíceis de ler e escrever, e a linguagem de montagem ainda é dependente da máquina a ser escrita, de forma que um código de uma determinada máquina não pode ser usado em outra máquina.

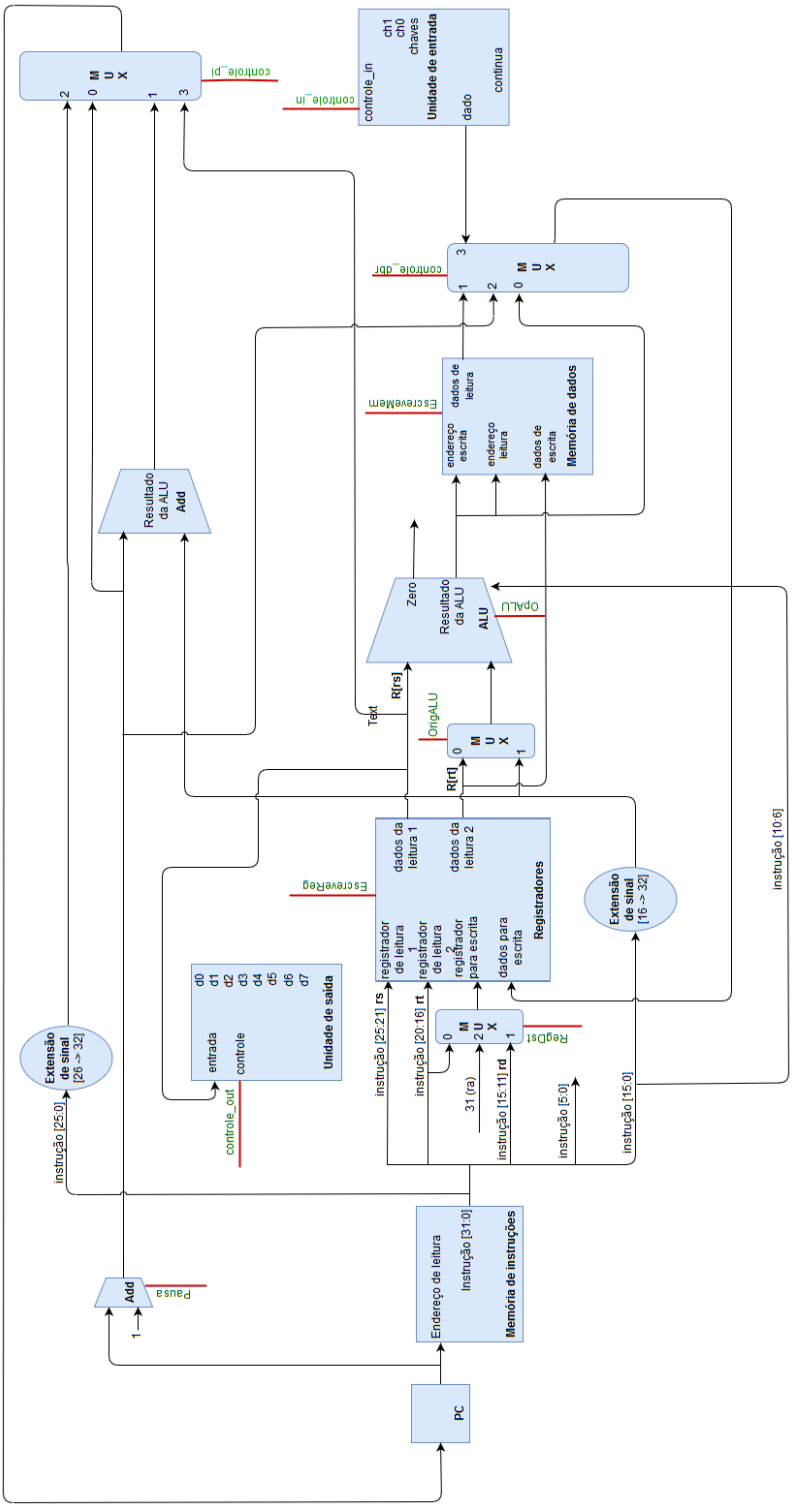
Os compiladores surgiram da necessidade de se solucionar essas questões e simplificar o processo de programação, de forma que um código possa ser escrito e compilado para qualquer máquina sem perder seu sentido (**??**). Um compilador é formado por módulos, cada módulo tem uma função específica, porém o objetivo comum é o de traduzir linguagens de acordo com regras pré-estabelecidas, simplificando e agilizando a tarefa de se programar.

Este projeto tem por objetivo familiarizar os estudantes de engenharia de computação com este tópico tão importante na área da computação, os compiladores, uma vez que compiladores demandam conhecimento em nível de software e hardware e está completamente alinhado ao que o curso se propõem.

2 O Processador

2.1 Diagrama de blocos do processador

Figura 1 – Diagrama de blocos do processador



Fonte: elaborado pela autora.

2.2 Explicação dos componentes do processador

Um MIPS, ou Microprocessador sem Estágios Intertravados de Pipeline, do inglês *Microprocessor without Interlocked Pipeline Stages*, é uma arquitetura desenvolvida pela *MIPS Technology Inc* que utiliza uma abordagem RISC.

Há diversas implementações diferentes para essa arquitetura, porém aquela que possui maior relevância e é utilizada como base para esse projeto é a arquitetura MIPS de 32 bits, monociclo, ou seja, cada instrução é executada em um único ciclo de *clock*, utilizando a arquitetura Harvard, ou seja, uma memória para armazenar dados e outra para armazenar instruções.

A implementação de um processador com arquitetura MIPS ocorre através da união das várias unidades funcionais listas a seguir, essa união se faz com base em um conjunto de instruções que se deseja atender.

2.2.1 PC

O contador de programa, do inglês *program counter*, é um registrador de uso específico cuja função é armazenar o endereço da instrução sendo executada. Trata-se de um registrador com 32 bits, que, teoricamente, seria capaz de referenciar 2^{32} posições da memória, entretanto, na arquitetura MIPS, a memória é divida em bytes, ou seja, cada palavra requer 4 posições da memória, dessa forma o PC é capaz de referenciar $2^{32}/4$ posições da memória, que equivale a 2^{30} posições da memória.

A implementação do PC consiste de um registrador com apenas uma entrada e uma saída.

2.2.2 ULA

A Unidade Lógica Aritmética, também conhecida como ULA ou ALU, do inglês *Arithmetic Logic Unit*, é o hardware que realiza operações matemáticas e lógicas no processador.

Com exceção da classe de instrução *jump*, todas as demais utilizam a ULA após a leitura dos registradores. No caso das instruções de referência à memória usa-se a ULA para o cálculo de endereço, no caso das operações lógicas ou aritméticas usa-se para a execução da operação ou comparação de desvios.

A ULA é implementada usando lógica combinacional e elementos de estado internos, possui como entradas os dois operandos e sinais de controle, possui como saídas o resultado da operação e uma *flag* para quando o resultado da operação for nulo.

2.2.3 Memória

A arquitetura MIPS implementa o conceito da arquitetura Harvard em sua memória, ou seja, há duas memórias separadas para armazenar dados e instruções, ambas as memórias são distribuídas em bytes e endereçadas a cada 4 bytes.

A implementação da memória de instrução consiste em uma matriz com 32 colunas e quantas linhas forem necessárias, com uma entrada e uma saída apenas. A implementação da memórias de dados consiste de uma matriz com 8 colunas e quantas linhas forem necessárias ao projeto, sendo que cada 4 linhas consiste em uma posição da memória, há duas entradas, uma de endereço e outra de dados, e sinais de controle para escrita e leitura que nunca são simultaneamente setados para evitar conflito na leitura/gravação de informações.

2.2.4 Extensor de bits

O campo imediato em algumas instruções contém um número de 16 bits que deve ser somado com um valor de 32 bits de um registrador, para que isso seja possível é necessário converter esse número de 16 bits em um número de 32 bits através de um extensor de bits.

A unidade de extensão de sinal possui uma entrada de 16 bits que tem o seu sinal estendido para que um resultado de 32 bits apareça na saída.

2.2.5 Banco de registradores

Um banco de registradores consiste em uma coleção de registradores que podem ser lidos ou escritos quando um número de registrador é fornecido.

O banco de registradores na arquitetura MIPS contém 32 registradores de propósito geral de 32 bits cada, os grupos de 32 bits ocorrem tão frequentemente na arquitetura MIPS que recebem o nome de palavra (*word*).

2.2.6 Unidade de Controle

Uma unidade de controle possui a instrução como entrada e é usada para determinar os sinais das linhas de controle para as unidades funcionais e dois dos multiplexadores. A implementação da unidade de controle consiste de uma porção de lógica combinacional e elementos de estado internos, recebe a instrução como entrada e possui diversos sinais de controle na saída.

2.3 Conjunto de instruções

Tabela 1 – Tipos de instruções aceitas pelo processador

R	opcode	rs	rt	rd	shamt	funct
	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits
I	opcode	rs	rt	immediate		
	6 bits	5 bits	5 bits	16 bits		
J	opcode	target				
	6 bits	26 bits				

Fonte: elaborada pela autora.

Tabela 2 – Conjunto de instruções do processador

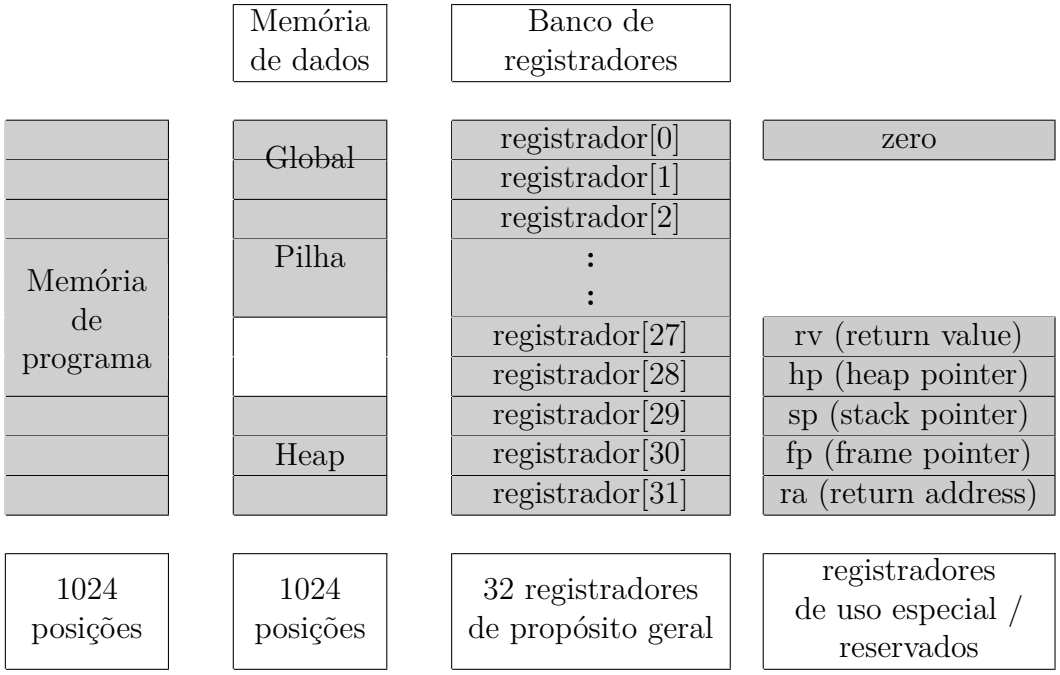
Form.	Instrução	Mnem.	Operação executada	Op	Func
R	entrada	in	R[rt] <= input	0	0
R	saida	out	output <= R[rs]	0	1
R	adição	add	R[rd] <= R[rs] + R[rt]	0	2
R	subtração	sub	R[rd] <= R[rs] - R[rt]	0	3
R	multiplicação	mul	R[rd] <= R[rs] * R[rt]	0	4
R	divisão	div	R[rd] <= R[rs] / R[rt]	0	5
R	menor que	slt	R[rd] <= (R[rs] < R[rt])	0	6
R	maior que	sgt	R[rd] <= (R[rs] > R[rt])	0	7
R	igual a	set	R[rd] <= (R[rs] == R[rt])	0	8
R	jump register	jr	PC <= R[rs]	0	9
R	and	and	R[rd] <= R[rs] & R[rt]	0	10
R	or	or	R[rd] <= R[rs] | R[rt]	0	11
R	resto	mod	R[rd] <= R[rs] % R[rt]	0	12
R	xor	xor	R[rd] <= R[rs] R̂[rt]	0	13
R	not	not	R[rd] <= ~ R[rs]	0	14
R	mover	move	R[rd] <= R[rs]	0	15
R	desloca esquerda	sll	R[rd] <= R[rs] << shamt	0	16
R	desloca direita	srl	R[rd] <= R[rs] >> shamt	0	17
J	nop	nop	não faz nada	1	0
J	jump	j	PC = IM	2	0
J	jump and link	jal	R[ra]<=PC+1 e PC <= IM	3	0
I	carregar	load	R[rt] <= M[R[rs]+IM]	4	0
I	armazenar	store	M[R[rs]+IM] <= R[rt]	5	0
I	adição imediata	addi	R[rd] <= R[rs] + IM	6	0
I	subtração imediata	subi	R[rd] <= R[rs] - IM	7	0
I	desvie se igual	beq	if(R[rs] == R[rt]) PC=IM	8	0
I	desvie se diferente	bne	if(R[rs] <> R[rt]) PC=IM	9	0
I	carregar imediato	loadi	R[rd] <= IM	10	0
I	carregar superior imediato	lui	R[rd] <= IM <<16	11	0

Fonte: elaborada pela autora.

2.4 Organização da memória

As memórias de instruções e dados consistem de 1024 posições cada, sendo a memória distribuída da seguinte forma 3:

Tabela 3 – Organização da memória do processador



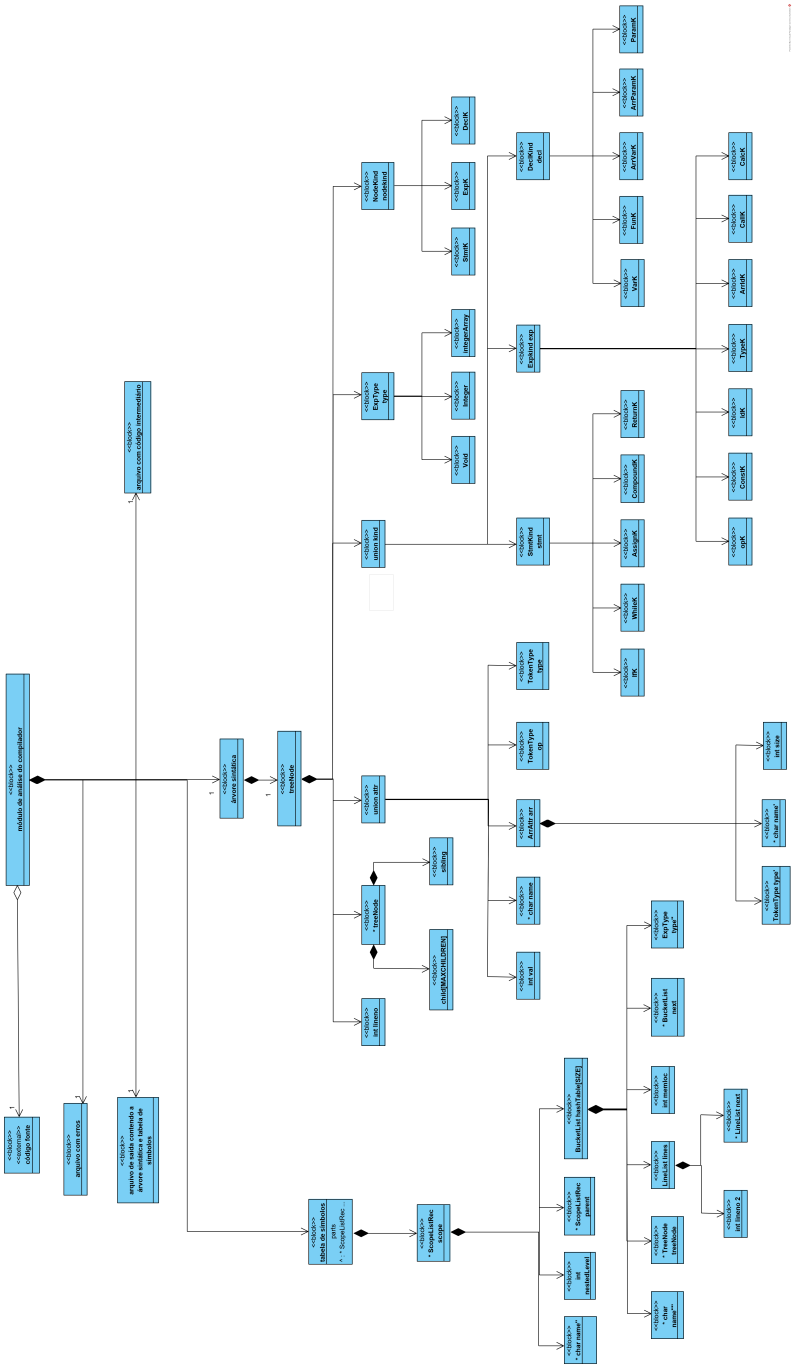
Fonte: elaborado pela autora.

3 Compilador: Fase de Análise

3.1 Modelagem

3.1.1 Diagrama de blocos

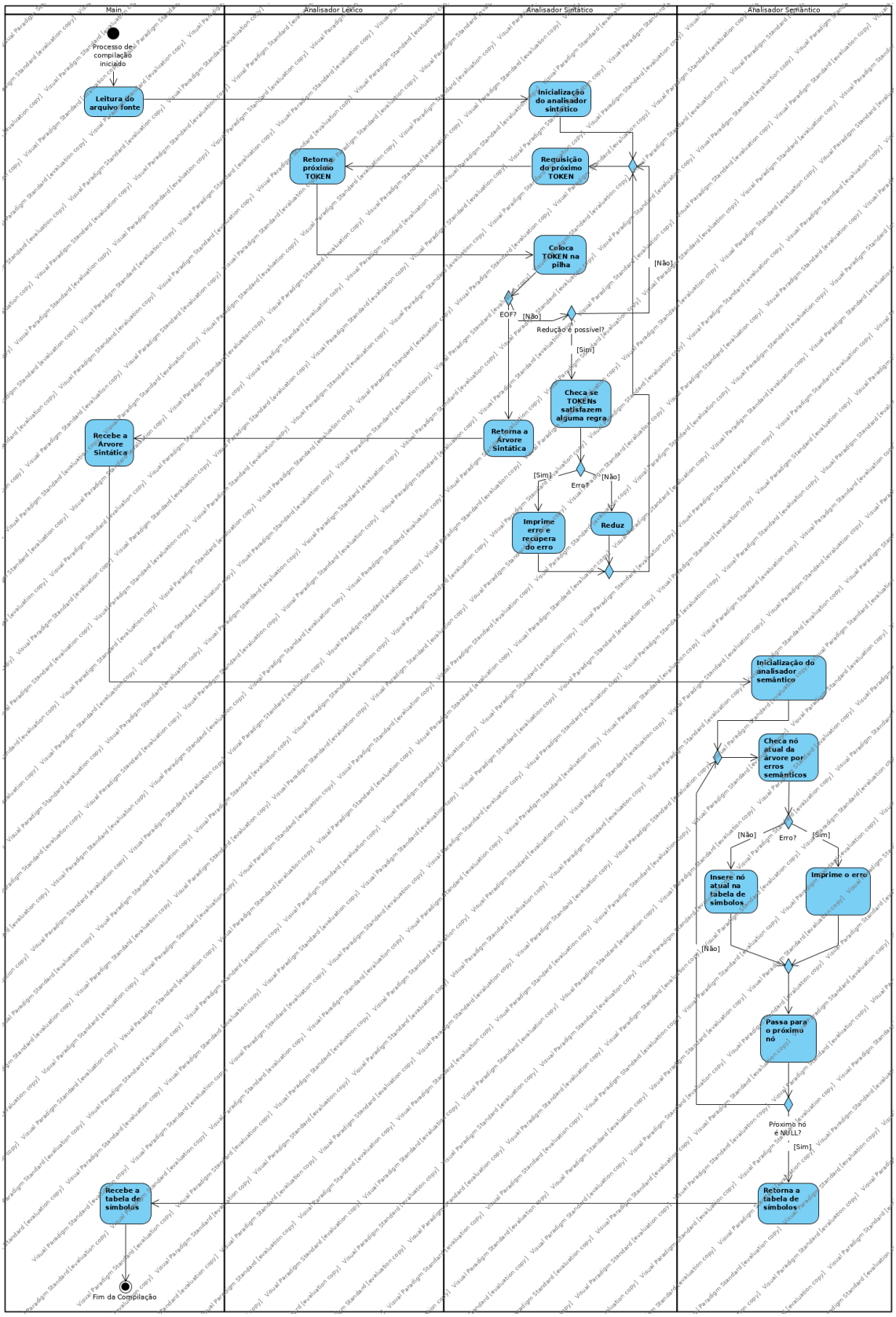
Figura 2 – Diagrama de blocos da fase de análise



Fonte: elaborado pela autora em parceria com o aluno Bruno Sampaio Leite.

3.1.2 Diagrama de Atividades

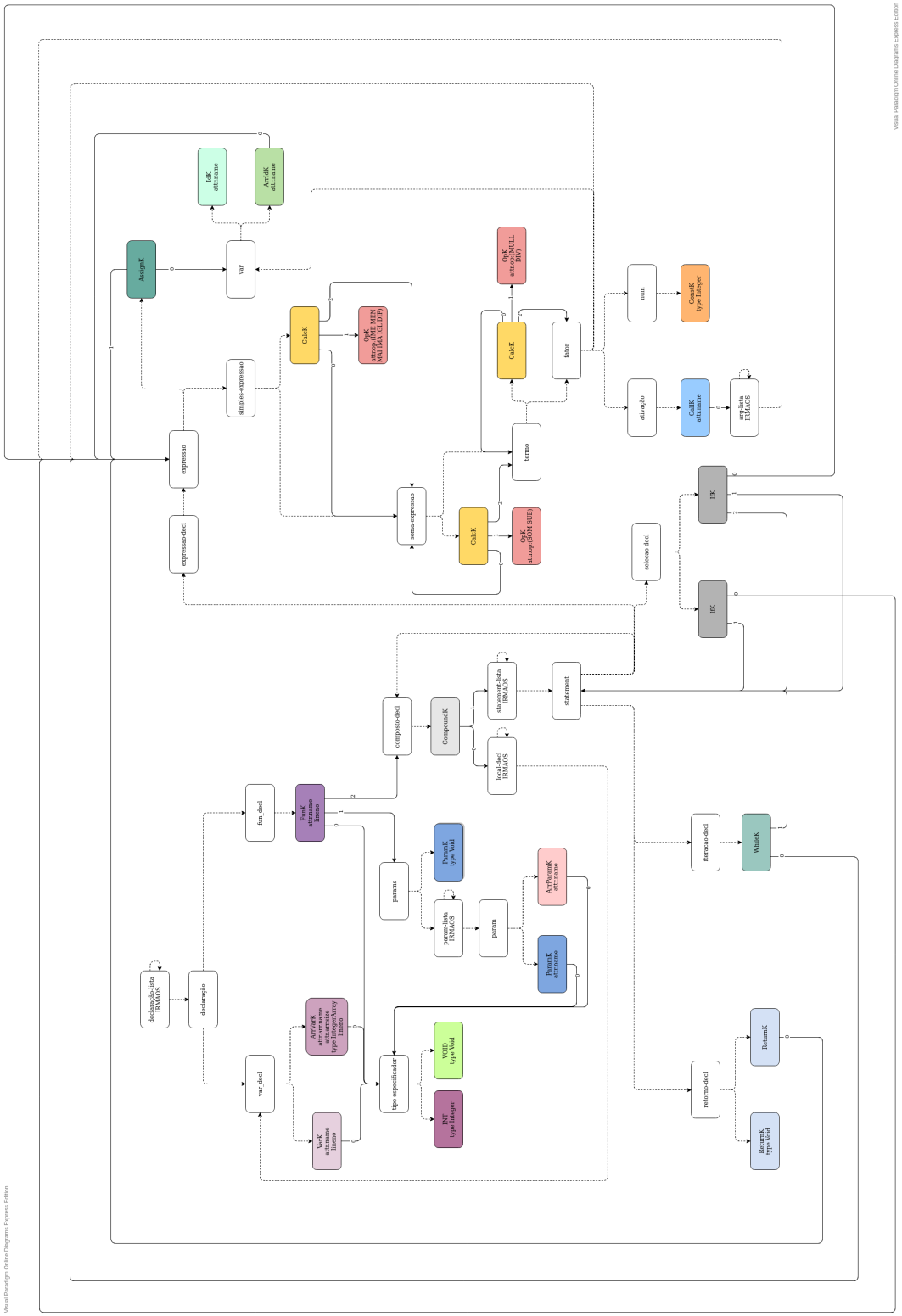
Figura 3 – Diagrama de atividades do módulo de análise



Fonte: elaborado pela autora em parceria com o aluno Bruno Sampaio Leite.

3.1.3 Outros

Figura 4 – Diagrama de construção da árvore sintática



Fonte: elaborado pela autora em parceria com o aluno Bruno Sampaio Leite.

3.2 Análise Léxica

A fase de análise léxica consiste em um sistema de varredura. Esse sistema de varredura lê um programa-fonte, que é fornecido como uma cadeia de caracteres, e a análise léxica em si consiste em organizar sequências de caracteres em unidades significativas conhecidas como marcas, ou *tokens*, ou palavra em uma linguagem natural como português, que são compostas por um ou mais caracteres agrupados (**??**).

Neste projeto utiliza-se o gerador de sistemas de varredura Lex para gerar um sistema de descrição dos *tokens* de C- através de expressões regulares. Mais especificamente, este projeto utiliza a versão Lex denominada *flex* (*Fast Lex*).

O Lex é um programa que recebe como entrada um arquivo de texto que contém as expressões regulares associadas as ações que devem ser tomadas para cada expressão, como saída ele produz um código em linguagem C definindo um procedimento chamado *yylex*. Esse arquivo de saída, *lexyy.c*, é então vinculado ao programa principal para gerar um arquivo executável (**??**).

O código scanner.l, apresentado integralmente no apêndice B, é um arquivo de entrada Lex, que divide-se em três partes: coleção de definições, coleção de regras e coleção de rotinas auxiliares. Cada parte é envolvida por chaves e as partes são separadas entre si por dois símbolos de porcentagem (%%).

Inicialmente, alguns cabeçalhos úteis são incluídos, a função *int yywrap()* é definida e um vetor chamado *tokenString* é criado, esse vetor será utilizado para armazenar temporariamente o *token* lido em *yytext*. Na sequência são definido as regras para DIGITO e LETRA e as expressões regulares, juntamente com suas ações. Na parte final ocorre a declaração da função auxiliar *TokenType getToken()*, nela o arquivo de leitura do programa Lex é definido como *source* através do nome interno *yyin* e o arquivo de escrita é definido como *listing* através do nome interno *yyout*. A rotina de varredura é, então, invocada através da chamada *yylex()*, que retorna um *token*, e o nome do *token* é escrito no arquivo de saída juntamente com a linha de ocorrência deste.

3.3 Análise Sintática

Um analisador sintático recebe do sistema de varredura o código-fonte em forma de *tokens*, a análise sintática consiste em determinar os elementos estruturais de um programa e seus relacionamentos, resultando em uma árvore de análise sintática (**??**).

Este projeto utiliza um programa gerador de analisadores sintáticos que incorpora o algoritmo de análise sintática LALR(1) (*Look Ahead - Left - Right - (1)*), conhecido como YACC () *Yet Another Compiler-Compiler*, que é uma ferramenta capaz de gerar um analisador sintático com base na gramática livre de contexto de uma determinada

linguagem. Trata-se de um programa que recebe como entrada uma especificação de linguagem e produz, como saída, um procedimento de analise sintática em linguagem C chamado *yyparse()* para essa linguagem determinada, esse procedimento retorna 0 caso nenhum erro seja identificado, ou 1 em caso de identificação de erro. Esse projeto utiliza, mais especificamente, a implementação YACC Bison disponível no GCC.

O código parser.y, apresentado integralmente no apêndice C, é um arquivo de entrada para um programa YACC-Bison. Esse arquivo de especificação da linguagem é separado em três partes separadas pelo símbolo de porcentagem %%, a primeira parte contém declarações, a segunda parte contém as regras da gramática e suas ações associadas e a terceira parte contém código complementar.

A primeira parte do código parser.y contém as inclusões de cabeçalhos úteis, a declaração de um vetor para receber uma string de *token*, declarações de variáveis úteis e das funções *int yylex()* e *int yyerror()*, responsável por escrever os erros encontrados no arquivo. Entre a primeira e segunda partes ocorre a definição dos *tokens* da linguagem.

A segunda parte contém a especificação da gramática utilizando notação estilo BNF (*Backus Naur Form*), ou seja, uma produção (ou regra gramatical) é separada em lados esquerdo e direito, o lado esquerdo contém o símbolo não terminal e o lado direito contém a expansão em termos de outros símbolos (terminais ou não).

A terceira parte contém a definição das funções auxiliares *int yyerror()*, responsável pela impressão dos erros no arquivo, *int yylex()*, responsável pela aquisição dos *tokens* e *TreeNode *parse()*, responsável por invocar o procedimento de análise sintática *yyparse()*.

3.4 Análise Semântica

O analisador semântico é responsável por executar a análise da semântica estática de uma linguagem de programação, ou seja, deve determinar os atributos, ou o comportamento de um programa, passíveis de serem determinados antes da execução do mesmo (**??**).

Através da análise semântica é possível analisar se o código-fonte possui certas incoerências, por exemplo:

- Se uma função não possui retorno;
- Se um procedimento possui retorno;
- Se há atribuição entre tipos incompatíveis;
- Se há atribuição para um vetor sem especificação do índice;
- Se há atribuição por chamada de procedimento;

- Se há utilização de um símbolo indefinido;
- Se há redefinição de função, variável ou parâmetro;
- Se há tipificação incorreta;
- Ausência da declaração da função *main()*.

Para verificar essas incoerências faz-se necessário organizar os elementos do código-fonte de uma forma diferente da árvore sintática construída, ou seja, a ideia é representar esses elementos de forma que a consulta seja simplificada. este projeto implementa uma tabela de símbolos para organizar esses elementos, essa tabela é construída percorrendo a árvore sintática.

Os códigos analyze.c e symboltable.c, integralmente disponíveis nos apêndices **??** e G, contém as funções necessárias para a verificação dessas incoerências e construção da tabela de símbolos. Além da verificação da análise semântica, a tabela de símbolos também será necessária no gerenciamento da memória de dados na geração do código Assembly.

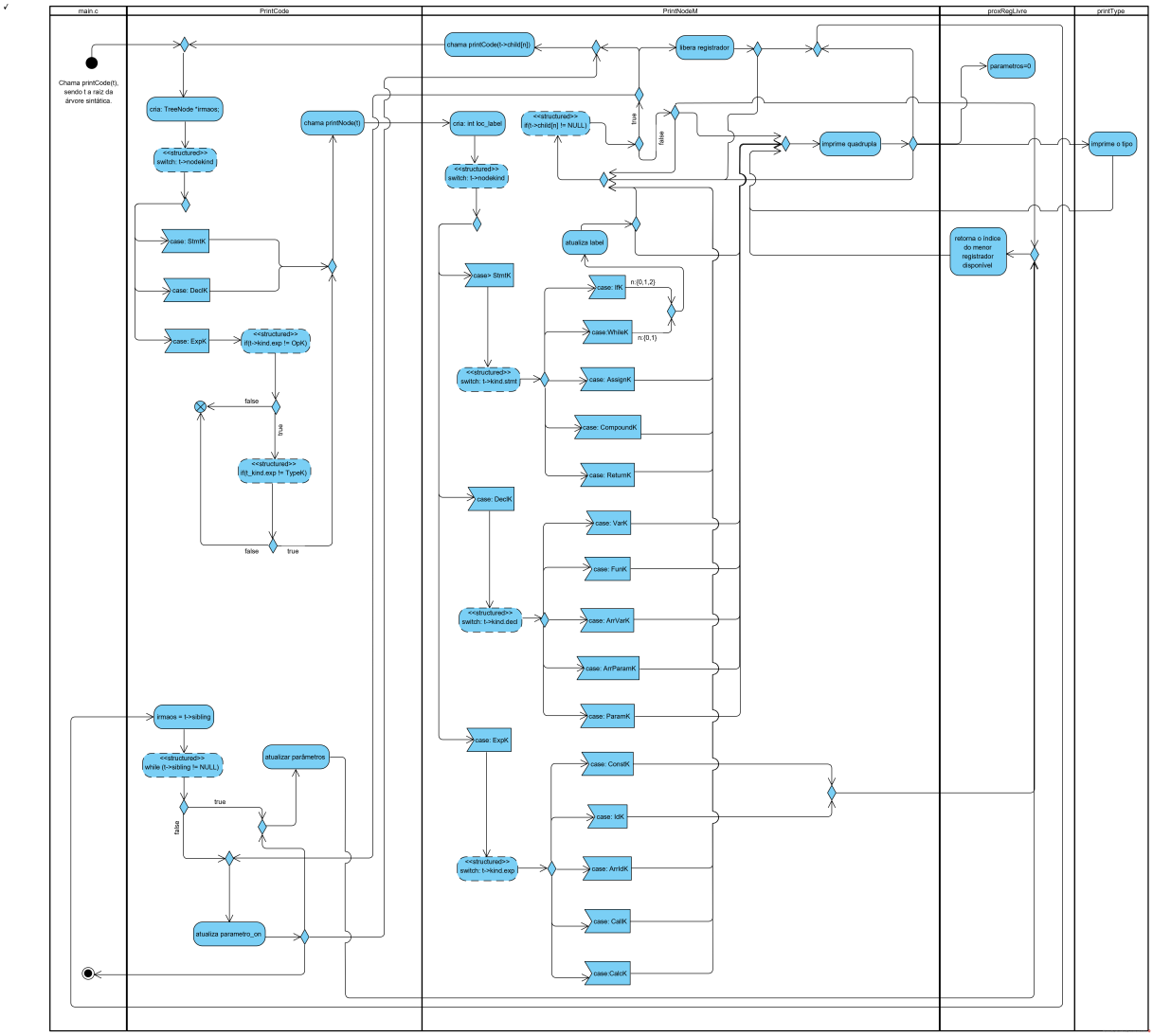
Além dos códigos anteriormente mencionados, há outros códigos auxiliares no desenvolvimento dos módulos de análise e síntese. Esses códigos estão integralmente disponíveis nos apêndices D e E. O código *main.c* é reponsável por inicializar os arquivos e gerenciar quais e quando serão chamadas as rotinas de geração do compilador, esse código está integralmente disponíveis em A.

4 Compilador: Fase de Síntese

4.1 Modelagem

4.1.1 Diagrama de Atividades

Figura 5 – Diagrama de atividades do gerador de código intermediário no módulo de síntese



Fonte: elaborado pela autora.

4.2 Geração do código intermediário

A geração do código intermediário tem por objetivo facilitar a geração do código Assembly ao linearizar a instruções em um modelo de representação mais próximo das

linhas de código Assembly.

Este projeto utiliza uma estrutura de quádrupla para realizar a linearização do código, sendo uma quádrupla de código intermediário representada da seguinte maneira:

Tabela 4 – Quádrupla do código intermediário

(campo 1 ,	campo 2 ,	campo 3 ,	campo 4)
ASSIGN SOM SUB MUL DIV OR MEN IGL MAI AND OR MOD XOR SLL SRL SET IME IMA DIF	registrador fonte	registrador fonte	registrador destino
LOAD STORE	registrador fonte	registrador destino	valor imediato
GOTO	-	-	alvo
LAB	-	-	identificação da label
VAR FUN PAR	nome	-	tipo
VAR_VET PAR_VET	nome	tamanho	tipo
SUBi SOMi	registrador fonte	registrador destino	valor imediato
LOADI	-	registrador destino	valor imediato
CALL	nome	qtdade de parâmetros	registrador destino
WHILE IF	registrador fonte	registrador fonte	alvo
RETURN	registrador fonte	-	registrador destino

O código codegenerate.c, integralmente disponível no apêndice H, recebe o nó raiz da árvore sintática e percorre essa árvore em pré-ordem linearizando-a, as quádruplas são geradas com base no tipo do nó e, através da função *releaseQuadList()* uma lista de quádruplas em Assembly é gerada.

4.3 Geração do código Assembly

A geração do código Assembly é imprescindível para a produção do código executável, uma vez que a tradução de Assembly para executável ocorre de um para um.

Neste projeto a geração do código Assembly ocorre paralelamente à geração do código intermediário, uma vez que a árvore é percorrida em pré-ordem para a geração do código intermediário ocorre a atualização simultânea das listas de quádruplas através da função *releaseQuadList()*. Essa função recebe como argumentos o tipo de operação, os registradores envolvidos e o nome das funções (se necessário) e gera a quádrupla correspondente na lista de quádrupla, inserindo-a na lista.

A tabela 5 apresenta o formato da quádrupla em Assembly.

Tabela 5 – Quádrupla do código assembly

Tipo da instrução:	Formato da qrádrupla Assembly
R	(opcode , rs , rt , rd , shamt , funct)
I	(opcode , rs , rt , imediato)
J	(opcode , alvo)

A tabela 6 apresenta a estrutura do nó utilizado nas listas de quádruplas que serão convertidas em código executável.

Tabela 6 – Estruturas dos nós da lista de quádruplas Assembly

Estrutura tipo R	Estrutura tipo I	Estrutura tipo J
opcode rs rt rd shamt funct	opcode rs rt imediato	opcode alvo nome
linha de ocorrência	linha de ocorrência	linha de ocorrência
ponteiro próxima quádrupla	ponteiro próxima quádrupla	ponteiro próxima quádrupla

4.4 Geração do código executável

A geração de código executável ocorre de a partir da lista de quádruplas geradas, de forma que cada quádrupla possui uma operação, registradores, imediatos ou alvos que são diretamente convertidos em binário para gerar a instrução esperada.

4.5 Gerenciamento de memória

O gerenciamento da memória se dá através da divisão desta em três partes: memória global, pilha e *heap*.

A memória global armazena as variáveis globais e, no caso de vetores, a posição do *heap* em que esse vetor se inicia.

A parte de pilha é responsável por empilhar registros de ativação, conforme a figura 7, formando quadros de pilhas. A pilha cresce para baixo na memória e o ponteiro *stack pointer*, ou sp, é responsável por gerenciar as variações do tamano da pilha.

Tabela 7 – Estrutura do registro de ativação

Registro de ativação	
endereço de vinculação	fp antigo
endereço de retorno	ra
parametros	parâmetros da função chamada
variáveis	variáveis locais

A parte de *heap* da pilha é responsável por armazenar os vetores e permitir que sejam passados por referência, ou seja, quando um vetor é passado como argumento passa-se o endereço inicial desse vetor no *heap* e as demais posições desse vetor são acessadas por meio de indexação. O *heap* cresce para cima, indo de encontro à pilha.

A tabela 8 apresenta um exemplo simplório de memória com 20 posições, um quadro de registros com 3 registros de ativação (main, call 1 e call 2), e dois vetores alocando memória no *heap*.

Tabela 8 – Estruturação da memória de dados

		var1		#0
		var2	global	#1
		var3		#2
		main		#3
	#3	call 1		#4
	#instruct main	call 1		#5
	par1	call 1		#6
	#18 (var_vet1)	call 1	pilha	#7
	var1	call 1		#8
	var2	call 1		#9
	#5	call 2		#10
fp (#11)	#instruct call 1	call 2		#11
	#16 (var_vet2)	call 2		#12
sp (#13)				#13
				#14
				#15
hp (#16)		var_vet2[0]		#16
		var_vet2[1]		#17
		var_vet1[0]	heap	#18
		var_vet1[1]		#19
		var_vet1[2]		#20

Conforme pode-se observar há alguns registradores de uso especiais, sp, para gerenciar o topo da pilha, hp, para gerenciar o topo do *heap* e fp, para gerenciar o registro de ativação atual. Outros registradores de uso especial são o rv, que armazena o valor de retorno das funções, e o zero, que contém um valor nulo para auxiliar em operações de atualização dos ponteiros.

5 Exemplos

5.1 Exemplo 1: teste.cm

5.1.1 Código fonte

```
1  int a;
2  int b[5];
3
4  int soma(int x, int y){
5      return x+y;
6  }
7
8  void main(void){
9      int c;
10
11     c = soma(a, b[3]);
12 }
```

Listing 5.1 – Código-fonte teste.cm

5.1.2 Árvore sintática e tabela de símbolos

```
1
2  Compilac o C-: teste.cm
3
4  Arvore Sintatica:
5      Declaracao de variavel:  int a;
6          Declaracao de Vetor:  int b[5];
7          Declaracao de funcao:  int soma()
8              Parametro: int x
9                Parametro: int y
10               Declaracao composta
11             Return
12                 Operador : +
13                     Id: x
14                     Op: +
15                     Id: y
16     Declaracao de funcao:  void main()
17             Parametro: void void
18         Declaracao composta
19             Declaracao de variavel:  int c;
20                     Atribuicao
21                 Id: c
22                 Chamada de funcao: soma
23                     Id: a
24                     ArrId
25                         Const: 3
26
27 Construindo a tabela de simbolos...
28
29 ---------------------
30 |  Tabela de simbolos  |
```

```
---------------------

Escopo : global
--------------------------------------------------------------------------------
Nome        Tipo              Tipo de Dado      loc    Numero das linhas
---------  ---------------  ------------      ---    ------------------
main        Funcao            Void               2     8
input       Funcao            Integer            0     0
a           Variavel          Integer            1     1
b           Vetor             Vetor de Integers  2     2
output      Funcao            Void               0     0
soma        Funcao            Integer            2     4
Qdade elementos: 2, qtdade memoria: 2
--------------------------------------------------------------------------------

Escopo : soma
--------------------------------------------------------------------------------
Nome        Tipo              Tipo de Dado      loc    Numero das linhas
---------  ---------------  ------------      ---    ------------------
x           Parametro         Integer            1     4, 5
y           Parametro         Integer            2     4, 5
Qdade elementos: 2, qtdade memoria: 2
--------------------------------------------------------------------------------

Escopo : main
--------------------------------------------------------------------------------
Nome        Tipo              Tipo de Dado      loc    Numero das linhas
---------  ---------------  ------------      ---    ------------------
c           Variavel          Integer            1     9, 11
Qdade elementos: 1, qtdade memoria: 1
--------------------------------------------------------------------------------
```

5.1.3 Código intermediário

```
(VAR, a, -, int)
(VAR_VET, *b, 5, int)
(FUN, soma, -, int)
(PAR, x, -, int)
(PAR, y, -, int)
(LOAD, x, _t1, 1)
(LOAD, y, _t2, 2)
(SOM, _t1, _t2, _t3)
(RETURN, _t3, -, rv)
(FUN, main, -, void)
(VAR, c, -, int)
(LOAD, c, _t1, 1)
(LOAD, a, _t2, 0)
(LOADI, - , _t3, 3)
(LOAD, 0, _t5, 1)
(SOM, _t5, _t3, _t4)
(LOAD, b[_t3], _t0, 0)
(PAR, -, -, _t2)
(PAR, -, -, _t0)
(CALL, soma, 2, _t3)
(ASSIGN, rv, -, _t1)
```

5.1.4 Código Assembly

```
(addi, 0, 29, 2)
```

```
2  (addi, 0, 28, 1023)
3  (addi, 29, 30, 0)
4  (addi, 0, 29, 1)
5  (addi, 0, 29, 1)
6  (subi, 28, 28, 5)
7  (load, 28, 30, 2)
8  (j, main)
9  (store, 31, 30, 0)
10 (load, 30, 1, 1)
11 (load, 30, 2, 2)
12 (add, 1, 2, 3, 0, 0)
13 (move, 3, 0, 27, 0, 0)
14 (j, soma)
15 (load, 30, 31, 0)
16 (load, 30, 30, -1)
17 (subi, 29, 29, 4)
18 (jr, soma)
19 (store, 31, 30, 0)
20 (load, 30, 1, 1)
21 (load, 0, 2, 0)
22 (loadi, 0, 3, 3)
23 (load, 0, 5, 1)
24 (add, 5, 3, 4, 0, 0)
25 (load, 4, 0, 0)
26 (store, 30, 29, 0)
27 (addi, 0, 29, 1)
28 (addi, 29, 30, 0)
29 (addi, 0, 29, 1)
30 (addi, 0, 29, 1)
31 (store, 2, 30, 1)
32 (store, 0, 30, 2)
33 (jal, soma)
34 (move, 27, 0, 1, 0, 0)
35 (store, 1, 30, 1)
36 (load, 30, 31, 0)
37 (load, 30, 30, -1)
38 (subi, 29, 29, 4)
39 (jr, main)
```

5.1.5 Código executável

```
1  000110 00000 11101 0000000000000010       I    6 0 29 2
2  000110 00000 11100 0000001111111111       I    6 0 28 1023
3  000110 11101 11110 0000000000000000       I    6 29 30 0
4  000110 00000 11101 0000000000000001       I    6 0 29 1
5  000110 00000 11101 0000000000000001       I    6 0 29 1
6  000111 11100 11100 0000000000000101       I    7 28 28 5
7  000100 11100 11110 0000000000000010       I    4 28 30 2
8  000010 00000000000000000000010010         J    2 18 main
9  000101 11111 11110 0000000000000000       I    5 31 30 0
10 000100 11110 00001 0000000000000001       I    4 30 1 1
11 000100 11110 00010 0000000000000010       I    4 30 2 2
12 000000 00011 00000 00000 00000 000010     R    0 3 0 0 0 2
13 000000 11011 00000 00000 00000 001111     R    0 27 0 0 0 15
14 000010 00000000000000000000001110         J    2 14 soma
15 000100 11110 11111 0000000000000000       I    4 30 31 0
16 000100 11110 11110 1111111111111111       I    4 30 30 -1
17 000111 11101 11101 0000000000000100       I    7 29 29 4
18 001001 00000000000000000000011111         J    9 31 soma
```

```
000101 11111 11110 0000000000000000        I    5 31 30 0
000100 11110 00001 0000000000000001        I    4 30 1 1
000100 00000 00010 0000000000000000        I    4 0 2 0
001010 00000 00011 0000000000000011        I    10 0 3 3
000100 00000 00101 0000000000000001        I    4 0 5 1
000000 00100 00000 00000 00000 000010      R    0 4 0 0 0 2
000100 00100 00000 0000000000000000        I    4 4 0 0
000101 11110 11101 0000000000000000        I    5 30 29 0
000110 00000 11101 0000000000000001        I    6 0 29 1
000110 11101 11110 0000000000000000        I    6 29 30 0
000110 00000 11101 0000000000000001        I    6 0 29 1
000110 00000 11101 0000000000000001        I    6 0 29 1
000101 00010 11110 0000000000000001        I    5 2 30 1
000101 00000 11110 0000000000000010        I    5 0 30 2
000011 00000000000000000000001000          J    3 8 soma
000000 00001 00000 00000 00000 001111      R    0 1 0 0 0 15
000101 00001 11110 0000000000000001        I    5 1 30 1
000100 11110 11111 0000000000000000        I    4 30 31 0
000100 11110 11110 1111111111111111        I    4 30 30 -1
000111 11101 11101 0000000000000100        I    7 29 29 4
001001 00000000000000000000011111          J    9 31 main
```

5.2 Exemplo 2: sort.cm

5.2.1 Código fonte

```
/* programa para ordenacao por selecao de
   uma matriz com dez elementos. */

int vet[ 10 ];

int minloc ( int a[], int low, int high )
{       int i; int x; int k;
        k = low;
        x = a[low];
        i = low + 1;
        while (i < high){
                if (a[i] < x){
                        x = a[i];
                        k = i;
                }
                i = i + 1;
        }
        return k;
}

void sort( int a[], int low, int high)
{       int i; int k;
        i = low;
        while (i < high-1){
                int t;
                k = minloc(a,i,high);
                t = a[k];
                a[k] = a[i];
                a[i] = t;
                i = i + 1;
        }
}
```

```
33
34 void main(void)
35 {
36         int i;
37         i = 0;
38         while (i < 10){
39                 vet[i] = input();
40                 i = i + 1;
41         }
42         sort(vet,0,10);
43         i = 0;
44         while (i < 10){
45                 output(vet[i]);
46                 i = i + 1;
47         }
48 }
```

Listing 5.2 – Código-gonte sort.cm

5.2.2 Árvore sintática e tabela de símbolos

```
1
2  Compilac o C-: sort.cm
3
4  Arvore Sintatica:
5      Declaracao de Vetor:  int vet[10];
6          Declaracao de funcao:  int minloc()
7              Parametro Vetor: a
8                Parametro: int low
9                Parametro: int high
10               Declaracao composta
11             Declaracao de variavel:  int i;
12                     Declaracao de variavel:  int x;
13                     Declaracao de variavel:  int k;
14                     Atribuicao
15                 Id: k
16                 Id: low
17             Atribuicao
18                 Id: x
19                 ArrId
20                     Id: low
21             Atribuicao
22                 Id: i
23                 Operador : +
24                     Id: low
25                     Op: +
26                     Const: 1
27             While
28                 Operador : <
29                     Id: i
30                     Op: <
31                     Id: high
32                 Declaracao composta
33                     If
34                         Operador : <
35                             ArrId
36                                 Id: i
37                             Op: <
```

```
38                          Id: x
39                      Declaracao composta
40                          Atribuicao
41                              Id: x
42                              ArrId
43                                  Id: i
44                          Atribuicao
45                              Id: k
46                              Id: i
47                  Atribuicao
48                      Id: i
49                      Operador : +
50                          Id: i
51                          Op: +
52                          Const: 1
53          Return
54              Id: k
55  Declaracao de funcao:  void sort()
56          Parametro Vetor: a
57            Parametro: int low
58            Parametro: int high
59            Declaracao composta
60          Declaracao de variavel:  int i;
61                  Declaracao de variavel:  int k;
62                  Atribuicao
63              Id: i
64              Id: low
65          While
66              Operador : <
67                  Id: i
68                  Op: <
69                  Operador : -
70                      Id: high
71                      Op: -
72                      Const: 1
73              Declaracao composta
74                  Declaracao de variavel:  int t;
75                              Atribuicao
76                      Id: k
77                      Chamada de funcao: minloc
78                          Id: a
79                          Id: i
80                          Id: high
81                  Atribuicao
82                      Id: t
83                      ArrId
84                          Id: k
85                  Atribuicao
86                      ArrId
87                          Id: k
88                      ArrId
89                          Id: i
90                  Atribuicao
91                      ArrId
92                          Id: i
93                      Id: t
94                  Atribuicao
95                      Id: i
96                      Operador : +
97                          Id: i
```

```
                            Op: +
                            Const: 1
    Declaracao de funcao:  void main()
            Parametro: void void
        Declaracao composta
            Declaracao de variavel:  int i;
                    Atribuicao
                Id: i
                Const: 0
            While
                Operador : <
                    Id: i
                    Op: <
                    Const: 10
                Declaracao composta
                    Atribuicao
                        ArrId
                            Id: i
                        Chamada de funcao: input
                    Atribuicao
                        Id: i
                        Operador : +
                            Id: i
                            Op: +
                            Const: 1
            Chamada de funcao: sort
                Id: vet
                Const: 0
                Const: 10
            Atribuicao
                Id: i
                Const: 0
            While
                Operador : <
                    Id: i
                    Op: <
                    Const: 10
                Declaracao composta
                    Chamada de funcao: output
                        ArrId
                            Id: i
                    Atribuicao
                        Id: i
                        Operador : +
                            Id: i
                            Op: +
                            Const: 1

Construindo a tabela de simbolos...

---------------------
|  Tabela de simbolos  |
---------------------

Escopo : global
------------------------------------------------------------------------------
Nome       Tipo             Tipo de Dado     loc    Numero das linhas
---------  ---------------  ------------     ---    ------------------
main       Funcao           Void              1     34
sort       Funcao           Void              1     21
```

```
input      Funcao           Integer           0     0
vet        Vetor            Vetor de Integers 1     4
minloc     Funcao           Integer           1     6
output     Funcao           Void              0     0
Qdade elementos: 1, qtdade memoria: 1
--------------------------------------------------------------------------------

Escopo : minloc
--------------------------------------------------------------------------------
Nome       Tipo             Tipo de Dado     loc    Numero das linhas
---------  ---------------  ------------     ---    ------------------
low        Parametro        Integer           2     6, 8, 9, 10
a          Parametro Vetor  Void              1     6, 9
i          Variavel         Integer           4     7, 10, 11
k          Variavel         Integer           6     7, 8, 18
x          Variavel         Integer           5     7, 9
high       Parametro        Integer           3     6, 11
Qdade elementos: 6, qtdade memoria: 6
--------------------------------------------------------------------------------

Escopo : sort
--------------------------------------------------------------------------------
Nome       Tipo             Tipo de Dado     loc    Numero das linhas
---------  ---------------  ------------     ---    ------------------
low        Parametro        Integer           2     21, 23
a          Parametro Vetor  Void              1     21
i          Variavel         Integer           4     22, 23, 24
k          Variavel         Integer           5     22
high       Parametro        Integer           3     21, 24
Qdade elementos: 5, qtdade memoria: 5
--------------------------------------------------------------------------------

Escopo : main
--------------------------------------------------------------------------------
Nome       Tipo             Tipo de Dado     loc    Numero das linhas
---------  ---------------  ------------     ---    ------------------
i          Variavel         Integer           1     36, 37, 38, 43, 44
Qdade elementos: 1, qtdade memoria: 1
--------------------------------------------------------------------------------
```

5.2.3 Código intermediário

```
(VAR_VET, *vet, 10, int)
(FUN, minloc, -, int)
(PAR_VET, * a, -, int)
(PAR, low, -, int)
(PAR, high, -, int)
(VAR, i, -, int)
(VAR, x, -, int)
(VAR, k, -, int)
(LOAD, k, _t1, 6)
(LOAD, low, _t2, 2)
(ASSIGN, _t2, -, _t1)
(LOAD, x, _t1, 5)
(LOAD, low, _t2, 2)
(LOAD, fp, _t4, 1)
(SOM, _t4, _t2, _t3)
(LOAD, a[_t2], _t0, 0)
(ASSIGN, _t0, -, _t1)
```

```
18 (LOAD, i, _t1, 4)
19 (LOAD, low, _t2, 2)
20 (LOADI, - , _t3, 1)
21 (SOM, _t2, _t3, _t4)
22 (ASSIGN, _t4, -, _t1)
23 (LAB, -, -, L0)
24 (LOAD, i, _t1, 4)
25 (LOAD, high, _t2, 3)
26 (MEN, _t1, _t2, _t3)
27 (WHILE, _t3, zero -, L1)
28 (GOTO, -, -, L2)
29 (LAB, -, -, 1)
30 (LOAD, i, _t1, 4)
31 (LOAD, fp, _t4, 1)
32 (SOM, _t4, _t1, _t2)
33 (LOAD, a[_t1], _t0, 0)
34 (LOAD, x, _t1, 5)
35 (MEN, _t0, _t1, _t2)
36 (IF, _t2, 0, L3)
37 (GOTO, -, -, L4)
38 (LAB, -, -, 3)
39 (LOAD, x, _t1, 5)
40 (LOAD, i, _t4, 4)
41 (LOAD, fp, _t6, 1)
42 (SOM, _t6, _t4, _t5)
43 (LOAD, a[_t4], _t0, 0)
44 (ASSIGN, _t0, -, _t1)
45 (LOAD, k, _t1, 6)
46 (LOAD, i, _t4, 4)
47 (ASSIGN, _t4, -, _t1)
48 (GOTO, -, -, L4)
49 (LAB, -, -, 4)
50 (LOAD, i, _t1, 4)
51 (LOAD, i, _t2, 4)
52 (LOADI, - , _t4, 1)
53 (SOM, _t2, _t4, _t5)
54 (ASSIGN, _t5, -, _t1)
55 (LAB, -, -, 2)
56 (LOAD, k, _t1, 6)
57 (RETURN, _t1, -, rv)
58 (FUN, sort, -, void)
59 (PAR_VET, * a, -, int)
60 (PAR, low, -, int)
61 (PAR, high, -, int)
62 (VAR, i, -, int)
63 (VAR, k, -, int)
64 (LOAD, i, _t1, 4)
65 (LOAD, low, _t2, 2)
66 (ASSIGN, _t2, -, _t1)
67 (LAB, -, -, L5)
68 (LOAD, i, _t1, 4)
69 (LOAD, high, _t2, 3)
70 (LOADI, - , _t3, 1)
71 (SUB, _t2, _t3, _t4)
72 (MEN, _t1, _t4, _t2)
73 (WHILE, _t2, zero -, L6)
74 (GOTO, -, -, L7)
75 (LAB, -, -, 6)
76 (VAR, t, -, int)
77 (LOAD, k, _t1, 5)
```

```
(LOAD, a, _t3, 1)
(LOAD, i, _t4, 4)
(LOAD, high, _t5, 3)
(PAR, -, -, _t3)
(PAR, -, -, _t4)
(PAR, -, -, _t5)
(CALL, minloc, 3, _t6)
(ASSIGN, rv, -, _t1)
(LOAD, t, _t1, -999)
(LOAD, k, _t3, 5)
(LOAD, fp, _t5, 1)
(SOM, _t5, _t3, _t4)
(LOAD, a[_t3], _t0, 0)
(ASSIGN, _t0, -, _t1)
(LOAD, k, _t1, 5)
(LOAD, fp, _t4, 1)
(SOM, _t4, _t1, _t3)
(LOAD, a[_t1], _t0, 0)
(LOAD, i, _t1, 4)
(LOAD, fp, _t4, 1)
(SOM, _t4, _t1, _t3)
(LOAD, a[_t1], _t0, 0)
(ASSIGN, _t0, -, _t0)
(LOAD, i, _t1, 4)
(LOAD, fp, _t4, 1)
(SOM, _t4, _t1, _t3)
(LOAD, a[_t1], _t0, 0)
(LOAD, t, _t1, -999)
(ASSIGN, _t1, -, _t0)
(LOAD, i, _t1, 4)
(LOAD, i, _t3, 4)
(LOADI, - , _t4, 1)
(SOM, _t3, _t4, _t5)
(ASSIGN, _t5, -, _t1)
(LAB, -, -, 7)
(FUN, main, -, void)
(VAR, i, -, int)
(LOAD, i, _t1, 1)
(LOADI, - , _t2, 0)
(ASSIGN, _t2, -, _t1)
(LAB, -, -, L8)
(LOAD, i, _t1, 1)
(LOADI, - , _t2, 10)
(MEN, _t1, _t2, _t3)
(WHILE, _t3, zero -, L9)
(GOTO, -, -, L10)
(LAB, -, -, 9)
(LOAD, i, _t1, 1)
(LOAD, 0, _t4, 0)
(SOM, _t4, _t1, _t2)
(LOAD, vet[_t1], _t0, 0)
(CALL, input, 0, _t1)
(ASSIGN, rv, -, _t0)
(LOAD, i, _t1, 1)
(LOAD, i, _t2, 1)
(LOADI, - , _t4, 1)
(SOM, _t2, _t4, _t5)
(ASSIGN, _t5, -, _t1)
(LAB, -, -, 10)
(LOAD, vet, _t1, 0)
```

```
(LOADI, - , _t2, 0)
(LOADI, - , _t3, 10)
(PAR, -, -, _t1)
(PAR, -, -, _t2)
(PAR, -, -, _t3)
(CALL, sort, 3, _t4)
(LOAD, i, _t1, 1)
(LOADI, - , _t2, 0)
(ASSIGN, _t2, -, _t1)
(LAB, -, -, L11)
(LOAD, i, _t1, 1)
(LOADI, - , _t2, 10)
(MEN, _t1, _t2, _t3)
(WHILE, _t3, zero -, L12)
(GOTO, -, -, L13)
(LAB, -, -, 12)
(LOAD, i, _t1, 1)
(LOAD, 0, _t5, 0)
(SOM, _t5, _t1, _t2)
(LOAD, vet[_t1], _t0, 0)
(PAR, -, -, _t0)
(CALL, output, 1, _t1)
(LOAD, i, _t2, 1)
(LOAD, i, _t5, 1)
(LOADI, - , _t6, 1)
(SOM, _t5, _t6, _t7)
(ASSIGN, _t7, -, _t2)
(LAB, -, -, 13)
```

5.2.4 Código Assembly

```
(addi, 0, 29, 1)
(addi, 0, 28, 1023)
(addi, 29, 30, 0)
(addi, 0, 29, 1)
(addi, 0, 29, 1)
(subi, 28, 28, 10)
(load, 28, 30, 1)
(j, main)
(store, 31, 30, 0)
(load, 30, 1, 6)
(load, 30, 2, 2)
(move, 2, 0, 1, 0, 0)
(store, 1, 30, 6)
(load, 30, 1, 5)
(load, 30, 2, 2)
(load, 30, 4, 1)
(add, 4, 2, 3, 0, 0)
(load, 3, 0, 0)
(move, 0, 0, 1, 0, 0)
(store, 1, 30, 5)
(load, 30, 1, 4)
(load, 30, 2, 2)
(loadi, 0, 3, 1)
(add, 2, 3, 4, 0, 0)
(move, 4, 0, 1, 0, 0)
(store, 1, 30, 4)
(load, 30, 1, 4)
(load, 30, 2, 3)
```

```
29 (slt, 1, 2, 3, 0, 0)
30 (bne, 3, 0, 1, 0, 0)
31 (j, lab)
32 (load, 30, 1, 4)
33 (load, 30, 4, 1)
34 (add, 4, 1, 2, 0, 0)
35 (load, 2, 0, 0)
36 (load, 30, 1, 5)
37 (slt, 0, 1, 2, 0, 0)
38 (bne, 2, 0, 3, 0, 0)
39 (j, lab)
40 (load, 30, 1, 5)
41 (load, 30, 4, 4)
42 (load, 30, 6, 1)
43 (add, 6, 4, 5, 0, 0)
44 (load, 5, 0, 0)
45 (move, 0, 0, 1, 0, 0)
46 (store, 1, 30, 5)
47 (load, 30, 1, 6)
48 (load, 30, 4, 4)
49 (move, 4, 0, 1, 0, 0)
50 (store, 1, 30, 6)
51 (j, lab)
52 (load, 30, 1, 4)
53 (load, 30, 2, 4)
54 (loadi, 0, 4, 1)
55 (add, 2, 4, 5, 0, 0)
56 (move, 5, 0, 1, 0, 0)
57 (store, 1, 30, 4)
58 (load, 30, 1, 6)
59 (move, 1, 0, 27, 0, 0)
60 (j, minloc)
61 (load, 30, 31, 0)
62 (load, 30, 30, -1)
63 (subi, 29, 29, 3)
64 (jr, minloc)
65 (store, 31, 30, 0)
66 (load, 30, 1, 4)
67 (load, 30, 2, 2)
68 (move, 2, 0, 1, 0, 0)
69 (store, 1, 30, 4)
70 (load, 30, 1, 4)
71 (load, 30, 2, 3)
72 (loadi, 0, 3, 1)
73 (sub, 2, 3, 4, 0, 0)
74 (slt, 1, 4, 2, 0, 0)
75 (bne, 2, 0, 6, 0, 0)
76 (j, lab)
77 (load, 30, 1, 5)
78 (load, 30, 3, 1)
79 (load, 30, 4, 4)
80 (load, 30, 5, 3)
81 (store, 30, 29, 0)
82 (addi, 0, 29, 1)
83 (addi, 29, 30, 0)
84 (addi, 0, 29, 1)
85 (addi, 0, 29, 5)
86 (store, 3, 30, 1)
87 (store, 4, 30, 2)
88 (store, 5, 30, 3)
```

```
89  (jal, minloc)
90  (move, 27, 0, 1, 0, 0)
91  (store, 1, 30, 5)
92  (load, 30, 1, -999)
93  (load, 30, 3, 5)
94  (load, 30, 5, 1)
95  (add, 5, 3, 4, 0, 0)
96  (load, 4, 0, 0)
97  (move, 0, 0, 1, 0, 0)
98  (store, 1, 30, -999)
99  (load, 30, 1, 5)
100 (load, 30, 4, 1)
101 (add, 4, 1, 3, 0, 0)
102 (load, 3, 0, 0)
103 (load, 30, 1, 4)
104 (load, 30, 4, 1)
105 (add, 4, 1, 3, 0, 0)
106 (load, 3, 0, 0)
107 (move, 0, 0, 0, 0, 0)
108 (store, 0, 30, 1)
109 (load, 30, 1, 4)
110 (load, 30, 4, 1)
111 (add, 4, 1, 3, 0, 0)
112 (load, 3, 0, 0)
113 (load, 30, 1, -999)
114 (move, 1, 0, 0, 0, 0)
115 (store, 0, 30, 1)
116 (load, 30, 1, 4)
117 (load, 30, 3, 4)
118 (loadi, 0, 4, 1)
119 (add, 3, 4, 5, 0, 0)
120 (move, 5, 0, 1, 0, 0)
121 (store, 1, 30, 4)
122 (load, 30, 31, 0)
123 (load, 30, 30, -1)
124 (subi, 29, 29, 3)
125 (jr, sort)
126 (store, 31, 30, 0)
127 (load, 30, 1, 1)
128 (loadi, 0, 2, 0)
129 (move, 2, 0, 1, 0, 0)
130 (store, 1, 30, 1)
131 (load, 30, 1, 1)
132 (loadi, 0, 2, 10)
133 (slt, 1, 2, 3, 0, 0)
134 (bne, 3, 0, 9, 0, 0)
135 (j, lab)
136 (load, 30, 1, 1)
137 (load, 0, 4, 0)
138 (add, 4, 1, 2, 0, 0)
139 (load, 2, 0, 0)
140 (store, 30, 29, 0)
141 (addi, 0, 29, 1)
142 (addi, 29, 30, 0)
143 (addi, 0, 29, 1)
144 (addi, 0, 29, 1)
145 (in, 1, 1, 1, 0, 0)
146 (move, 27, 0, 0, 0, 0)
147 (store, 0, 30, 0)
148 (load, 30, 1, 1)
```

```
149 (load, 30, 2, 1)
150 (loadi, 0, 4, 1)
151 (add, 2, 4, 5, 0, 0)
152 (move, 5, 0, 1, 0, 0)
153 (store, 1, 30, 1)
154 (load, 0, 1, 0)
155 (loadi, 0, 2, 0)
156 (loadi, 0, 3, 10)
157 (store, 30, 29, 0)
158 (addi, 0, 29, 1)
159 (addi, 29, 30, 0)
160 (addi, 0, 29, 1)
161 (addi, 0, 29, 1)
162 (store, 1, 30, 1)
163 (store, 2, 30, 2)
164 (store, 3, 30, 3)
165 (jal, sort)
166 (load, 30, 1, 1)
167 (loadi, 0, 2, 0)
168 (move, 2, 0, 1, 0, 0)
169 (store, 1, 30, 1)
170 (load, 30, 1, 1)
171 (loadi, 0, 2, 10)
172 (slt, 1, 2, 3, 0, 0)
173 (bne, 3, 0, 12, 0, 0)
174 (j, lab)
175 (load, 30, 1, 1)
176 (load, 0, 5, 0)
177 (add, 5, 1, 2, 0, 0)
178 (load, 2, 0, 0)
179 (store, 30, 29, 0)
180 (addi, 0, 29, 1)
181 (addi, 29, 30, 0)
182 (addi, 0, 29, 1)
183 (addi, 0, 29, 1)
184 (store, 0, 30, 1)
185 (out, 1, 1, 1, 0, 0)
186 (load, 30, 2, 1)
187 (load, 30, 5, 1)
188 (loadi, 0, 6, 1)
189 (add, 5, 6, 7, 0, 0)
190 (move, 7, 0, 2, 0, 0)
191 (store, 2, 30, 1)
192 (load, 30, 31, 0)
193 (load, 30, 30, -1)
194 (subi, 29, 29, 3)
195 (jr, main)
```

5.2.5 Código executável

```
1 000110 00000 11101 0000000000000001       I    6 0 29 1
2 000110 00000 11100 0000001111111111       I    6 0 28 1023
3 000110 11101 11110 0000000000000000       I    6 29 30 0
4 000110 00000 11101 0000000000000001       I    6 0 29 1
5 000110 00000 11101 0000000000000001       I    6 0 29 1
6 000111 11100 11100 0000000000001010       I    7 28 28 10
7 000100 11100 11110 0000000000000001       I    4 28 30 1
8 000010 00000000000000000001111101         J    2 125 main
9 000101 11111 11110 0000000000000000       I    5 31 30 0
```

```
000100 11110 00001 0000000000000110       I    4 30 1 6
000100 11110 00010 0000000000000010       I    4 30 2 2
000000 00001 00000 00000 00000 001111     R    0 1 0 0 0 15
000101 00001 11110 0000000000000110       I    5 1 30 6
000100 11110 00001 0000000000000101       I    4 30 1 5
000100 11110 00010 0000000000000010       I    4 30 2 2
000100 11110 00100 0000000000000001       I    4 30 4 1
000000 00011 00000 00000 00000 000010     R    0 3 0 0 0 2
000100 00011 00000 0000000000000000       I    4 3 0 0
000000 00001 00000 00000 00000 001111     R    0 1 0 0 0 15
000101 00001 11110 0000000000000101       I    5 1 30 5
000100 11110 00001 0000000000000100       I    4 30 1 4
000100 11110 00010 0000000000000010       I    4 30 2 2
001010 00000 00011 0000000000000001       I    10 0 3 1
000000 00100 00000 00000 00000 000010     R    0 4 0 0 0 2
000000 00001 00000 00000 00000 001111     R    0 1 0 0 0 15
000101 00001 11110 0000000000000100       I    5 1 30 4
000100 11110 00001 0000000000000100       I    4 30 1 4
000100 11110 00010 0000000000000011       I    4 30 2 3
000000 00011 00000 00000 00000 000110     R    0 3 0 0 0 6
000000 00001 00000 00000 00000 001001     R    0 1 0 0 0 9
000010 00000000000000000000111001         J    2 57 lab
000100 11110 00001 0000000000000100       I    4 30 1 4
000100 11110 00100 0000000000000001       I    4 30 4 1
000000 00010 00000 00000 00000 000010     R    0 2 0 0 0 2
000100 00010 00000 0000000000000000       I    4 2 0 0
000100 11110 00001 0000000000000101       I    4 30 1 5
000000 00010 00000 00000 00000 000110     R    0 2 0 0 0 6
000000 00011 00000 00000 00000 001001     R    0 3 0 0 0 9
000010 00000000000000000000110011         J    2 51 lab
000100 11110 00001 0000000000000101       I    4 30 1 5
000100 11110 00100 0000000000000100       I    4 30 4 4
000100 11110 00110 0000000000000001       I    4 30 6 1
000000 00101 00000 00000 00000 000010     R    0 5 0 0 0 2
000100 00101 00000 0000000000000000       I    4 5 0 0
000000 00001 00000 00000 00000 001111     R    0 1 0 0 0 15
000101 00001 11110 0000000000000101       I    5 1 30 5
000100 11110 00001 0000000000000110       I    4 30 1 6
000100 11110 00100 0000000000000100       I    4 30 4 4
000000 00001 00000 00000 00000 001111     R    0 1 0 0 0 15
000101 00001 11110 0000000000000110       I    5 1 30 6
000010 00000000000000000000110011         J    2 51 lab
000100 11110 00001 0000000000000100       I    4 30 1 4
000100 11110 00010 0000000000000100       I    4 30 2 4
001010 00000 00100 0000000000000001       I    10 0 4 1
000000 00101 00000 00000 00000 000010     R    0 5 0 0 0 2
000000 00001 00000 00000 00000 001111     R    0 1 0 0 0 15
000101 00001 11110 0000000000000100       I    5 1 30 4
000100 11110 00001 0000000000000110       I    4 30 1 6
000000 11011 00000 00000 00000 001111     R    0 27 0 0 0 15
000010 00000000000000000000111100         J    2 60 minloc
000100 11110 11111 0000000000000000       I    4 30 31 0
000100 11110 11110 1111111111111111       I    4 30 30 -1
000111 11101 11101 0000000000000011       I    7 29 29 3
001001 00000000000000000000011111         J    9 31 minloc
000101 11111 11110 0000000000000000       I    5 31 30 0
000100 11110 00001 0000000000000100       I    4 30 1 4
000100 11110 00010 0000000000000010       I    4 30 2 2
000000 00001 00000 00000 00000 001111     R    0 1 0 0 0 15
000101 00001 11110 0000000000000100       I    5 1 30 4
```

```
000100 11110 00001 0000000000000100       I    4 30 1 4
000100 11110 00010 0000000000000011       I    4 30 2 3
001010 00000 00011 0000000000000001       I    10 0 3 1
000000 00100 00000 00000 00000 000011     R    0 4 0 0 0 3
000000 00010 00000 00000 00000 000110     R    0 2 0 0 0 6
000000 00110 00000 00000 00000 001001     R    0 6 0 0 0 9
000010 00000000000000000000001111001      J    2 121 lab
000100 11110 00001 0000000000000101       I    4 30 1 5
000100 11110 00011 0000000000000001       I    4 30 3 1
000100 11110 00100 0000000000000100       I    4 30 4 4
000100 11110 00101 0000000000000011       I    4 30 5 3
000101 11110 11101 0000000000000000       I    5 30 29 0
000110 00000 11101 0000000000000001       I    6 0 29 1
000110 11101 11110 0000000000000000       I    6 29 30 0
000110 00000 11101 0000000000000001       I    6 0 29 1
000110 00000 11101 0000000000000101       I    6 0 29 5
000101 00011 11110 0000000000000001       I    5 3 30 1
000101 00100 11110 0000000000000010       I    5 4 30 2
000101 00101 11110 0000000000000011       I    5 5 30 3
000011 00000000000000000000001000         J    3 8 minloc
000000 00001 00000 00000 00000 001111     R    0 1 0 0 0 15
000101 00001 11110 0000000000000101       I    5 1 30 5
000100 11110 00001 1111110000011001       I    4 30 1 -999
000100 11110 00011 0000000000000101       I    4 30 3 5
000100 11110 00101 0000000000000001       I    4 30 5 1
000000 00100 00000 00000 00000 000010     R    0 4 0 0 0 2
000100 00100 00000 0000000000000000       I    4 4 0 0
000000 00001 00000 00000 00000 001111     R    0 1 0 0 0 15
000101 00001 11110 1111110000011001       I    5 1 30 -999
000100 11110 00001 0000000000000101       I    4 30 1 5
000100 11110 00100 0000000000000001       I    4 30 4 1
000000 00011 00000 00000 00000 000010     R    0 3 0 0 0 2
000100 00011 00000 0000000000000000       I    4 3 0 0
000100 11110 00001 0000000000000100       I    4 30 1 4
000100 11110 00100 0000000000000001       I    4 30 4 1
000000 00011 00000 00000 00000 000010     R    0 3 0 0 0 2
000100 00011 00000 0000000000000000       I    4 3 0 0
000000 00000 00000 00000 00000 001111     R    0 0 0 0 0 15
000101 00000 11110 0000000000000001       I    5 0 30 1
000100 11110 00001 0000000000000100       I    4 30 1 4
000100 11110 00100 0000000000000001       I    4 30 4 1
000000 00011 00000 00000 00000 000010     R    0 3 0 0 0 2
000100 00011 00000 0000000000000000       I    4 3 0 0
000100 11110 00001 1111110000011001       I    4 30 1 -999
000000 00000 00000 00000 00000 001111     R    0 0 0 0 0 15
000101 00000 11110 0000000000000001       I    5 0 30 1
000100 11110 00001 0000000000000100       I    4 30 1 4
000100 11110 00011 0000000000000100       I    4 30 3 4
001010 00000 00100 0000000000000001       I    10 0 4 1
000000 00101 00000 00000 00000 000010     R    0 5 0 0 0 2
000000 00001 00000 00000 00000 001111     R    0 1 0 0 0 15
000101 00001 11110 0000000000000100       I    5 1 30 4
000100 11110 11111 0000000000000000       I    4 30 31 0
000100 11110 11110 1111111111111111       I    4 30 30 -1
000111 11101 11101 0000000000000011       I    7 29 29 3
001001 00000000000000000000011111         J    9 31 sort
000101 11111 11110 0000000000000000       I    5 31 30 0
000100 11110 00001 0000000000000001       I    4 30 1 1
001010 00000 00010 0000000000000000       I    10 0 2 0
000000 00001 00000 00000 00000 001111     R    0 1 0 0 0 15
```

```
000101 00001 11110 0000000000000001       I    5 1 30 1
000100 11110 00001 0000000000000001       I    4 30 1 1
001010 00000 00010 0000000000001010       I    10 0 2 10
000000 00011 00000 00000 00000 000110     R    0 3 0 0 0 6
000000 01001 00000 00000 00000 001001     R    0 9 0 0 0 9
000010 00000000000000000010011001         J    2 153 lab
000100 11110 00001 0000000000000001       I    4 30 1 1
000100 00000 00100 0000000000000000       I    4 0 4 0
000000 00010 00000 00000 00000 000010     R    0 2 0 0 0 2
000100 00010 00000 0000000000000000       I    4 2 0 0
000101 11110 11101 0000000000000000       I    5 30 29 0
000110 00000 11101 0000000000000001       I    6 0 29 1
000110 11101 11110 0000000000000000       I    6 29 30 0
000110 00000 11101 0000000000000001       I    6 0 29 1
000110 00000 11101 0000000000000001       I    6 0 29 1
000000 00001 00000 00000 00000 000000     R    0 1 0 0 0 0
000000 00000 00000 00000 00000 001111     R    0 0 0 0 0 15
000101 00000 11110 0000000000000000       I    5 0 30 0
000100 11110 00001 0000000000000001       I    4 30 1 1
000100 11110 00010 0000000000000001       I    4 30 2 1
001010 00000 00100 0000000000000001       I    10 0 4 1
000000 00101 00000 00000 00000 000010     R    0 5 0 0 0 2
000000 00001 00000 00000 00000 001111     R    0 1 0 0 0 15
000101 00001 11110 0000000000000001       I    5 1 30 1
000100 00000 00001 0000000000000000       I    4 0 1 0
001010 00000 00010 0000000000000000       I    10 0 2 0
001010 00000 00011 0000000000001010       I    10 0 3 10
000101 11110 11101 0000000000000000       I    5 30 29 0
000110 00000 11101 0000000000000001       I    6 0 29 1
000110 11101 11110 0000000000000000       I    6 29 30 0
000110 00000 11101 0000000000000001       I    6 0 29 1
000110 00000 11101 0000000000000001       I    6 0 29 1
000101 00001 11110 0000000000000001       I    5 1 30 1
000101 00010 11110 0000000000000010       I    5 2 30 2
000101 00011 11110 0000000000000011       I    5 3 30 3
000011 00000000000000000001000000         J    3 64 sort
000100 11110 00001 0000000000000001       I    4 30 1 1
001010 00000 00010 0000000000000000       I    10 0 2 0
000000 00001 00000 00000 00000 001111     R    0 1 0 0 0 15
000101 00001 11110 0000000000000001       I    5 1 30 1
000100 11110 00001 0000000000000001       I    4 30 1 1
001010 00000 00010 0000000000001010       I    10 0 2 10
000000 00011 00000 00000 00000 000110     R    0 3 0 0 0 6
000000 01100 00000 00000 00000 001001     R    0 12 0 0 0 9
000010 00000000000000000010111111         J    2 191 lab
000100 11110 00001 0000000000000001       I    4 30 1 1
000100 00000 00101 0000000000000000       I    4 0 5 0
000000 00010 00000 00000 00000 000010     R    0 2 0 0 0 2
000100 00010 00000 0000000000000000       I    4 2 0 0
000101 11110 11101 0000000000000000       I    5 30 29 0
000110 00000 11101 0000000000000001       I    6 0 29 1
000110 11101 11110 0000000000000000       I    6 29 30 0
000110 00000 11101 0000000000000001       I    6 0 29 1
000110 00000 11101 0000000000000001       I    6 0 29 1
000101 00000 11110 0000000000000001       I    5 0 30 1
000000 00001 00000 00000 00000 000001     R    0 1 0 0 0 1
000100 11110 00010 0000000000000001       I    4 30 2 1
000100 11110 00101 0000000000000001       I    4 30 5 1
001010 00000 00110 0000000000000001       I    10 0 6 1
000000 00111 00000 00000 00000 000010     R    0 7 0 0 0 2
```

```
000000 00010 00000 00000 00000 001111      R     0 2 0 0 0 15
000101 00010 11110 0000000000000001        I     5 2 30 1
000100 11110 11111 0000000000000000        I     4 30 31 0
000100 11110 11110 1111111111111111        I     4 30 30 -1
000111 11101 11101 0000000000000011        I     7 29 29 3
001001 00000000000000000000011111          J     9 31 main
```

5.3 Exemplo 3: gcd.cm

5.3.1 Código fonte

```
/*Um programa para calcular o mdc segundo o algoritmo de Euclides*/

int gcd(int u, int v){
        if(v == 0) return u;
        else return gcd(v, u-u/v*v);
        /* u-u/v*v == u mod v */
}

void main(void){
        int x; int y;
        x = input(); y = input();
        output(gcd(x,y));
}
```

5.3.2 Árvore sintática e tabela de símbolos

```

Compilac o C-: gcd.cm

Arvore Sintatica:
    Declaracao de funcao:  int gcd()
            Parametro: int u
              Parametro: int v
              Declaracao composta
            If
                Operador : ==
                    Id: v
                    Op: ==
                    Const: 0
                Return
                    Id: u
                Return
                    Chamada de funcao: gcd
                        Id: v
                        Operador : -
                            Id: u
                            Op: -
                            Operador : *
                                Operador : /
                                    Id: u
                                    Op: /
                                    Id: v
                                Op: *
                                Id: v
    Declaracao de funcao:  void main()
            Parametro: void void
```

```
        Declaracao composta
            Declaracao de variavel:  int x;
                    Declaracao de variavel:  int y;
                    Atribuicao
                Id: x
                Chamada de funcao: input
            Atribuicao
                Id: y
                Chamada de funcao: input
            Chamada de funcao: output
                Chamada de funcao: gcd
                    Id: x
                    Id: y

Construindo a tabela de simbolos...

---------------------
|  Tabela de simbolos  |
---------------------

Escopo : global
------------------------------------------------------------------------------
Nome       Tipo             Tipo de Dado     loc    Numero das linhas
---------  ---------------  ------------     ---    ------------------
main       Funcao           Void              0     9
input      Funcao           Integer           0     0
output     Funcao           Void              0     0
gcd        Funcao           Integer           0     3
Qdade elementos: 0, qtdade memoria: 0
------------------------------------------------------------------------------

Escopo : gcd
------------------------------------------------------------------------------
Nome       Tipo             Tipo de Dado     loc    Numero das linhas
---------  ---------------  ------------     ---    ------------------
u          Parametro        Integer           1     3, 4, 5, 5
v          Parametro        Integer           2     3, 4, 5, 5, 5
Qdade elementos: 2, qtdade memoria: 2
------------------------------------------------------------------------------

Escopo : main
------------------------------------------------------------------------------
Nome       Tipo             Tipo de Dado     loc    Numero das linhas
---------  ---------------  ------------     ---    ------------------
x          Variavel         Integer           1     10, 11, 12
y          Variavel         Integer           2     10, 11, 12
Qdade elementos: 2, qtdade memoria: 2
------------------------------------------------------------------------------
```

5.3.3 Código intermediário

```
000110 00000 11101 0000000000000000       I    6 0 29 0
000110 00000 11100 0000001111111111       I    6 0 28 1023
000110 11101 11110 0000000000000000       I    6 29 30 0
000110 00000 11101 0000000000000001       I    6 0 29 1
000110 00000 11101 0000000000000010       I    6 0 29 2
000010 00000000000000000000100110         J    2 38 main
000101 11111 11110 0000000000000000       I    5 31 30 0
000100 11110 00001 0000000000000010       I    4 30 1 2
```

```
001010 00000 00010 0000000000000000       I    10 0 2 0
000000 00011 00000 00000 00000 001000     R    0 3 0 0 0 8
000000 00000 00000 00000 00000 001001     R    0 0 0 0 0 9
000100 11110 00001 0000000000000010       I    4 30 1 2
000100 11110 00010 0000000000000001       I    4 30 2 1
000100 11110 00100 0000000000000001       I    4 30 4 1
000100 11110 00101 0000000000000010       I    4 30 5 2
000000 00110 00000 00000 00000 000101     R    0 6 0 0 0 5
000100 11110 00100 0000000000000010       I    4 30 4 2
000000 00101 00000 00000 00000 000100     R    0 5 0 0 0 4
000000 00100 00000 00000 00000 000011     R    0 4 0 0 0 3
000101 11110 11101 0000000000000000       I    5 30 29 0
000110 00000 11101 0000000000000001       I    6 0 29 1
000110 11101 11110 0000000000000000       I    6 29 30 0
000110 00000 11101 0000000000000001       I    6 0 29 1
000110 00000 11101 0000000000000010       I    6 0 29 2
000101 00001 11110 0000000000000001       I    5 1 30 1
000101 00100 11110 0000000000000010       I    5 4 30 2
000011 00000000000000000000000110         J    3 6 gcd
000000 11011 00000 00000 00000 001111     R    0 27 0 0 0 15
000010 00000000000000000000100010         J    2 34 gcd
000010 00000000000000000000100010         J    2 34 lab
000100 11110 00001 0000000000000001       I    4 30 1 1
000000 11011 00000 00000 00000 001111     R    0 27 0 0 0 15
000010 00000000000000000000100010         J    2 34 gcd
000010 00000000000000000000100010         J    2 34 lab
000100 11110 11111 0000000000000000       I    4 30 31 0
000100 11110 11110 1111111111111111       I    4 30 30 -1
000111 11101 11101 0000000000000010       I    7 29 29 2
001001 00000000000000000000011111         J    9 31 gcd
000101 11111 11110 0000000000000000       I    5 31 30 0
000100 11110 00001 0000000000000001       I    4 30 1 1
000101 11110 11101 0000000000000000       I    5 30 29 0
000110 00000 11101 0000000000000001       I    6 0 29 1
000110 11101 11110 0000000000000000       I    6 29 30 0
000110 00000 11101 0000000000000001       I    6 0 29 1
000110 00000 11101 0000000000000010       I    6 0 29 2
000000 00010 00000 00000 00000 000000     R    0 2 0 0 0 0
000000 00001 00000 00000 00000 001111     R    0 1 0 0 0 15
000101 00001 11110 0000000000000001       I    5 1 30 1
000100 11110 00001 0000000000000010       I    4 30 1 2
000101 11110 11101 0000000000000000       I    5 30 29 0
000110 00000 11101 0000000000000001       I    6 0 29 1
000110 11101 11110 0000000000000000       I    6 29 30 0
000110 00000 11101 0000000000000001       I    6 0 29 1
000110 00000 11101 0000000000000010       I    6 0 29 2
000000 00010 00000 00000 00000 000000     R    0 2 0 0 0 0
000000 00001 00000 00000 00000 001111     R    0 1 0 0 0 15
000101 00001 11110 0000000000000010       I    5 1 30 2
000100 11110 00001 0000000000000001       I    4 30 1 1
000100 11110 00010 0000000000000010       I    4 30 2 2
000101 11110 11101 0000000000000000       I    5 30 29 0
000110 00000 11101 0000000000000001       I    6 0 29 1
000110 11101 11110 0000000000000000       I    6 29 30 0
000110 00000 11101 0000000000000001       I    6 0 29 1
000110 00000 11101 0000000000000010       I    6 0 29 2
000101 00001 11110 0000000000000001       I    5 1 30 1
000101 00010 11110 0000000000000010       I    5 2 30 2
000011 00000000000000000000000110         J    3 6 gcd
000101 11110 11101 0000000000000000       I    5 30 29 0
```

```
000110 00000 11101 0000000000000001       I    6 0 29 1
000110 11101 11110 0000000000000000       I    6 29 30 0
000110 00000 11101 0000000000000001       I    6 0 29 1
000110 00000 11101 0000000000000010       I    6 0 29 2
000101 00011 11110 0000000000000001       I    5 3 30 1
000000 00001 00000 00000 00000 000001     R    0 1 0 0 0 1
000100 11110 11111 0000000000000000       I    4 30 31 0
000100 11110 11110 1111111111111111       I    4 30 30 -1
000111 11101 11101 0000000000000010       I    7 29 29 2
001001 00000000000000000000011111         J    9 31 main
```

5.3.4 Código Assembly

```
000110 00000 11101 0000000000000000       I    6 0 29 0
000110 00000 11100 0000001111111111       I    6 0 28 1023
000110 11101 11110 0000000000000000       I    6 29 30 0
000110 00000 11101 0000000000000001       I    6 0 29 1
000110 00000 11101 0000000000000010       I    6 0 29 2
000010 00000000000000000000100110         J    2 38 main
000101 11111 11110 0000000000000000       I    5 31 30 0
000100 11110 00001 0000000000000010       I    4 30 1 2
001010 00000 00010 0000000000000000       I    10 0 2 0
000000 00011 00000 00000 00000 001000     R    0 3 0 0 0 8
000000 00000 00000 00000 00000 001001     R    0 0 0 0 0 9
000100 11110 00001 0000000000000010       I    4 30 1 2
000100 11110 00010 0000000000000001       I    4 30 2 1
000100 11110 00100 0000000000000001       I    4 30 4 1
000100 11110 00101 0000000000000010       I    4 30 5 2
000000 00110 00000 00000 00000 000101     R    0 6 0 0 0 5
000100 11110 00100 0000000000000010       I    4 30 4 2
000000 00101 00000 00000 00000 000100     R    0 5 0 0 0 4
000000 00100 00000 00000 00000 000011     R    0 4 0 0 0 3
000101 11110 11101 0000000000000000       I    5 30 29 0
000110 00000 11101 0000000000000001       I    6 0 29 1
000110 11101 11110 0000000000000000       I    6 29 30 0
000110 00000 11101 0000000000000001       I    6 0 29 1
000110 00000 11101 0000000000000010       I    6 0 29 2
000101 00001 11110 0000000000000001       I    5 1 30 1
000101 00100 11110 0000000000000010       I    5 4 30 2
000011 00000000000000000000000110         J    3 6 gcd
000000 11011 00000 00000 00000 001111     R    0 27 0 0 0 15
000010 00000000000000000000100010         J    2 34 gcd
000010 00000000000000000000100010         J    2 34 lab
000100 11110 00001 0000000000000001       I    4 30 1 1
000000 11011 00000 00000 00000 001111     R    0 27 0 0 0 15
000010 00000000000000000000100010         J    2 34 gcd
000010 00000000000000000000100010         J    2 34 lab
000100 11110 11111 0000000000000000       I    4 30 31 0
000100 11110 11110 1111111111111111       I    4 30 30 -1
000111 11101 11101 0000000000000010       I    7 29 29 2
001001 00000000000000000000011111         J    9 31 gcd
000101 11111 11110 0000000000000000       I    5 31 30 0
000100 11110 00001 0000000000000001       I    4 30 1 1
000101 11110 11101 0000000000000000       I    5 30 29 0
000110 00000 11101 0000000000000001       I    6 0 29 1
000110 11101 11110 0000000000000000       I    6 29 30 0
000110 00000 11101 0000000000000001       I    6 0 29 1
000110 00000 11101 0000000000000010       I    6 0 29 2
000000 00010 00000 00000 00000 000000     R    0 2 0 0 0 0
```

```
000000 00001 00000 00000 00000 001111     R    0 1 0 0 0 15
000101 00001 11110 0000000000000001       I    5 1 30 1
000100 11110 00001 0000000000000010       I    4 30 1 2
000101 11110 11101 0000000000000000       I    5 30 29 0
000110 00000 11101 0000000000000001       I    6 0 29 1
000110 11101 11110 0000000000000000       I    6 29 30 0
000110 00000 11101 0000000000000001       I    6 0 29 1
000110 00000 11101 0000000000000010       I    6 0 29 2
000000 00010 00000 00000 00000 000000     R    0 2 0 0 0 0
000000 00001 00000 00000 00000 001111     R    0 1 0 0 0 15
000101 00001 11110 0000000000000010       I    5 1 30 2
000100 11110 00001 0000000000000001       I    4 30 1 1
000100 11110 00010 0000000000000010       I    4 30 2 2
000101 11110 11101 0000000000000000       I    5 30 29 0
000110 00000 11101 0000000000000001       I    6 0 29 1
000110 11101 11110 0000000000000000       I    6 29 30 0
000110 00000 11101 0000000000000001       I    6 0 29 1
000110 00000 11101 0000000000000010       I    6 0 29 2
000101 00001 11110 0000000000000001       I    5 1 30 1
000101 00010 11110 0000000000000010       I    5 2 30 2
000011 00000000000000000000000110         J    3 6 gcd
000101 11110 11101 0000000000000000       I    5 30 29 0
000110 00000 11101 0000000000000001       I    6 0 29 1
000110 11101 11110 0000000000000000       I    6 29 30 0
000110 00000 11101 0000000000000001       I    6 0 29 1
000110 00000 11101 0000000000000010       I    6 0 29 2
000101 00011 11110 0000000000000001       I    5 3 30 1
000000 00001 00000 00000 00000 000001     R    0 1 0 0 0 1
000100 11110 11111 0000000000000000       I    4 30 31 0
000100 11110 11110 1111111111111111       I    4 30 30 -1
000111 11101 11101 0000000000000010       I    7 29 29 2
001001 00000000000000000000011111         J    9 31 main
```

5.3.5 Código executável

```
000110 00000 11101 0000000000000000       I    6 0 29 0
000110 00000 11100 0000001111111111       I    6 0 28 1023
000110 11101 11110 0000000000000000       I    6 29 30 0
000110 00000 11101 0000000000000001       I    6 0 29 1
000110 00000 11101 0000000000000010       I    6 0 29 2
000010 00000000000000000000100110         J    2 38 main
000101 11111 11110 0000000000000000       I    5 31 30 0
000100 11110 00001 0000000000000010       I    4 30 1 2
001010 00000 00010 0000000000000000       I    10 0 2 0
000000 00011 00000 00000 00000 001000     R    0 3 0 0 0 8
000000 00000 00000 00000 00000 001001     R    0 0 0 0 0 9
000100 11110 00001 0000000000000010       I    4 30 1 2
000100 11110 00010 0000000000000001       I    4 30 2 1
000100 11110 00100 0000000000000001       I    4 30 4 1
000100 11110 00101 0000000000000010       I    4 30 5 2
000000 00110 00000 00000 00000 000101     R    0 6 0 0 0 5
000100 11110 00100 0000000000000010       I    4 30 4 2
000000 00101 00000 00000 00000 000100     R    0 5 0 0 0 4
000000 00100 00000 00000 00000 000011     R    0 4 0 0 0 3
000101 11110 11101 0000000000000000       I    5 30 29 0
000110 00000 11101 0000000000000001       I    6 0 29 1
000110 11101 11110 0000000000000000       I    6 29 30 0
000110 00000 11101 0000000000000001       I    6 0 29 1
000110 00000 11101 0000000000000010       I    6 0 29 2
```

```
25 000101 00001 11110 0000000000000001       I    5 1 30 1
26 000101 00100 11110 0000000000000010       I    5 4 30 2
27 000011 00000000000000000000000110         J    3 6 gcd
28 000000 11011 00000 00000 00000 001111     R    0 27 0 0 0 15
29 000010 00000000000000000000100010         J    2 34 gcd
30 000010 00000000000000000000100010         J    2 34 lab
31 000100 11110 00001 0000000000000001       I    4 30 1 1
32 000000 11011 00000 00000 00000 001111     R    0 27 0 0 0 15
33 000010 00000000000000000000100010         J    2 34 gcd
34 000010 00000000000000000000100010         J    2 34 lab
35 000100 11110 11111 0000000000000000       I    4 30 31 0
36 000100 11110 11110 1111111111111111       I    4 30 30 -1
37 000111 11101 11101 0000000000000010       I    7 29 29 2
38 001001 00000000000000000000011111         J    9 31 gcd
39 000101 11111 11110 0000000000000000       I    5 31 30 0
40 000100 11110 00001 0000000000000001       I    4 30 1 1
41 000101 11110 11101 0000000000000000       I    5 30 29 0
42 000110 00000 11101 0000000000000001       I    6 0 29 1
43 000110 11101 11110 0000000000000000       I    6 29 30 0
44 000110 00000 11101 0000000000000001       I    6 0 29 1
45 000110 00000 11101 0000000000000010       I    6 0 29 2
46 000000 00010 00000 00000 00000 000000     R    0 2 0 0 0 0
47 000000 00001 00000 00000 00000 001111     R    0 1 0 0 0 15
48 000101 00001 11110 0000000000000001       I    5 1 30 1
49 000100 11110 00001 0000000000000010       I    4 30 1 2
50 000101 11110 11101 0000000000000000       I    5 30 29 0
51 000110 00000 11101 0000000000000001       I    6 0 29 1
52 000110 11101 11110 0000000000000000       I    6 29 30 0
53 000110 00000 11101 0000000000000001       I    6 0 29 1
54 000110 00000 11101 0000000000000010       I    6 0 29 2
55 000000 00010 00000 00000 00000 000000     R    0 2 0 0 0 0
56 000000 00001 00000 00000 00000 001111     R    0 1 0 0 0 15
57 000101 00001 11110 0000000000000010       I    5 1 30 2
58 000100 11110 00001 0000000000000001       I    4 30 1 1
59 000100 11110 00010 0000000000000010       I    4 30 2 2
60 000101 11110 11101 0000000000000000       I    5 30 29 0
61 000110 00000 11101 0000000000000001       I    6 0 29 1
62 000110 11101 11110 0000000000000000       I    6 29 30 0
63 000110 00000 11101 0000000000000001       I    6 0 29 1
64 000110 00000 11101 0000000000000010       I    6 0 29 2
65 000101 00001 11110 0000000000000001       I    5 1 30 1
66 000101 00010 11110 0000000000000010       I    5 2 30 2
67 000011 00000000000000000000000110         J    3 6 gcd
68 000101 11110 11101 0000000000000000       I    5 30 29 0
69 000110 00000 11101 0000000000000001       I    6 0 29 1
70 000110 11101 11110 0000000000000000       I    6 29 30 0
71 000110 00000 11101 0000000000000001       I    6 0 29 1
72 000110 00000 11101 0000000000000010       I    6 0 29 2
73 000101 00011 11110 0000000000000001       I    5 3 30 1
74 000000 00001 00000 00000 00000 000001     R    0 1 0 0 0 1
75 000100 11110 11111 0000000000000000       I    4 30 31 0
76 000100 11110 11110 1111111111111111       I    4 30 30 -1
77 000111 11101 11101 0000000000000010       I    7 29 29 2
78 001001 00000000000000000000011111         J    9 31 main
```

6 Conclusão

A implementação deste projeto foi bem sucedida, salvo a ausência de testes da máquina-alvo em função da modalidade de aula da disciplina ser ADE.

A próxima etapa deste projeto consiste em executar programas testes na máquina alvo para consolidar a corretude deste projeto.

6.1 Dificuldades encontradas

O grande alto grau de detalhamento das estruturas de dados requer um bom domínio sobre esse tópico, conferindo dificuldade inicial.

A ausência de conhecimento específico sobre gerenciamento de memória de dados também se apresentou como um desafio.

Contudo, as aulas ministradas pelo professor juntamente com a obra de referência foram suficiente para me orientar no caminho correto.

6.2 Destaques

Pode-se citar como destaque o grande crescimento no conhecimento sobre a comunicação entre software e máquina, um conceito bastante abstrato e que é essencial que estudantes de engenharia de computação compreendam.

APÊNDICE A – Código-fonte main.c

```c
/****************************************************/
/* File: main.c                                     */
/* Main program for TINY compiler                   */
/* Compiler Construction: Principles and Practice   */
/* Kenneth C. Louden                                */
/****************************************************/

#include "globals.h"
#include "symboltable.h"
#include "util.h"

/* set NO_PARSE to TRUE to get a scanner-only compiler */
#define NO_PARSE FALSE
/* set NO_ANALYZE to TRUE to get a parser-only compiler */
#define NO_ANALYZE FALSE

/* set NO_CODE to TRUE to get a compiler that does not
 * generate code */
#define NO_CODE FALSE

/* set NO_ASSEMBLY to TRUE to get a compiler that does not
generate assembly-code */
#define NO_ASSEMBLY FALSE

/* set NO_BINARY to TRUE to get a compiler that does not
generate assembly-code */
#define NO_BINARY FALSE

#if NO_PARSE
#include "scan.h"
#else
#include "parse.h"
#endif
#if !NO_ANALYZE
#include "analyze.h"
#endif
#if !NO_CODE
#include "codegenerate.h"
#endif
#if !NO_ASSEMBLY
#include "assemblygenerate.h"
#endif
#if !NO_BINARY
#include "bincode.h"
#endif

/* allocate global variables */
int lineno = 0;
FILE * source;
FILE * listing;
FILE * intermed;
FILE * errorfile;
FILE * assembly;
FILE * bincode;
```

```
55
56 /* allocate and set tracing flags */
57 int EchoSource = FALSE;
58 int TraceScan = FALSE;
59 int TraceParse = TRUE;
60 int TraceAnalyze = TRUE;
61 int TraceCode = FALSE;
62
63 int Error = FALSE;
64
65 Scope globalScope;
66
67 int main( int argc, char * argv[] )
68 { TreeNode * syntaxTree;
69   globalScope = (Scope) malloc(sizeof(Scope));
70   globalScope = newScope("global", Void);
71
72   char pgm[120]; /* source code file name */
73   if (argc != 2)
74     { fprintf(stderr,"usage: %s <filename>\n",argv[0]);
75       exit(1);
76     }
77   strcpy(pgm,argv[1]) ;
78   if (strchr (pgm, '.') == NULL)
79      strcat(pgm,".cm");
80   source = fopen(pgm,"r");
81   if (source==NULL)
82   { fprintf(stderr,"File %s not found\n",pgm);
83     exit(1);
84   }
85
86   errorfile = fopen("errorfile.txt", "w");
87   if(errorfile == NULL){
88     fprintf(stderr, "Could not open file to print errors.\n");
89   }
90   fprintf(errorfile, "\\----------------------------------\\\n");
91   fprintf(errorfile, "ERROS SINTATICOS DO ARQUIVO %s:\n\n", pgm);
92
93   listing = fopen("output.txt", "w"); //Print syntax tree and symbol table to file output
       .txt
94   if(listing == NULL){
95     fprintf(stderr, "Could not open output file to print.\n");
96   }
97   fprintf(listing,"\nCompilacao C-: %s\n",pgm);
98
99   intermed = fopen("intermediario.txt", "w");
100  if(intermed == NULL){
101    fprintf(stderr, "Could not open intermediate code file");
102  }
103
104  assembly = fopen("assembly.txt", "w");
105  if(assembly == NULL){
106    fprintf(stderr, "Could not open intermediate code file");
107  }
108
109  bincode = fopen("bincode.txt", "w");
110  if(bincode == NULL){
111    fprintf(stderr, "Could not open intermediate code file");
112  }
113
```

```
114 #if NO_PARSE
115   while (getToken()!=ENDFILE);
116 #else
117   syntaxTree = parse();
118   if (TraceParse) {
119     fprintf(listing,"\nArvore Sintatica:\n");
120     printTree(syntaxTree);
121   }
122 #if !NO_ANALYZE
123   if (! Error)
124   {
125     fprintf(errorfile, "  ERROS SEMANTICOS DO ARQUIVO %s:\n", pgm);
126     fprintf(errorfile, "\\-------------------------------------------\\\n");
127     if (TraceAnalyze) fprintf(listing,"\nConstruindo a tabela de simbolos...\n");
128     buildSymtab(syntaxTree);
129   }
130 #if !NO_CODE
131   #if !NO_ASSEMBLY
132    /*  if (! Error)
133     {
134       fprintf(assembly , "  CODIGO ASSEMBLY DO ARQUIVO %s:\n", pgm);
135       fprintf(assembly , "\\-------------------------------------------\\\n");
136     } */
137     #endif
138     #if !NO_BINARY
139     /* if (! Error)
140     {
141       fprintf(bincode , "  CODIGO EXECUTAVEL DO ARQUIVO %s:\n", pgm);
142       fprintf(bincode , "\\-------------------------------------------\\\n");
143     } */
144   #endif
145   if (! Error)
146   {
147     /* fprintf(intermed , "  CODIGO INTERMEDIARO DO ARQUIVO %s:\n", pgm);
148     fprintf(intermed , "\\-------------------------------------------\\\n"); */
149     printCode(syntaxTree);
150     quadModel *ptrBegin = beginQuadList;
151     getBin(ptrBegin);
152   }
153 #endif
154 #endif
155 #endif
156   fclose(source);
157   fclose(listing);
158   fclose(intermed);
159   fclose(errorfile);
160   fclose(assembly);
161   fclose(bincode);
162   return 0;
163 }
```

APÊNDICE B – Código-fonte scanner.l

```
%{
/*
 *Alunos:
 *Bruno Sampaio Leite   120213
 *Talita Ludmila de Lima    120895
*/

#include "parser.tab.h"
#include "globals.h"
#include "util.h"
#include "scan.h"

int yywrap(void){
  return 1;
}

char tokenString[MAXTOKENLEN+1];

%}

DIGITO [0-9]
LETRA [a-zA-Z]

%%

"/*"    {
        char c;
        char d;
        c = input();
        if(c!=EOF)
        {
            do
            {
                d=c;
                c = input();
                if(c==EOF) return ERR;
                if(c=='\n') lineno++;
            }while(!(d == '*' && c == '/'));
        }
    }
"else"          return ELSE;
"if"            return IF;
"int"           return INT;
"return"        return RETURN;
"void"          return VOID;
"while"         return WHILE;
{DIGITO}+                   return NUM;
{LETRA}({LETRA})*       return ID;
"+"                     return SOM;
"-"             return SUB;
"*"                             return MUL;
"/"                             return DIV;
"<"             return MEN;
"<="            return IME;
```

```
55 ">"              return MAI;
56 ">="             return IMA;
57 "=="             return IGL;
58 "!="             return DIF;
59 "="              return ATR;
60 ";"                              return PEV;
61 ","              return VIR;
62 "("                              return APR;
63 ")"                              return FPR;
64 "["              return ACL;
65 "]"              return FCL;
66 "{"              return ACH;
67 "}"              return FCH;
68 [\n]             {lineno++;}
69 [ \t]+
70 .                return ERR;
71
72 %%
73
74 TokenType getToken(void)
75 {
76   static int firstTime = TRUE;
77   TokenType currentToken;
78   if (firstTime)
79   {
80     firstTime = FALSE;
81     lineno++;
82     yyin = source;
83     yyout = listing;
84   }
85   currentToken = yylex();
86   strncpy(tokenString,yytext,MAXTOKENLEN);
87   if (TraceScan)
88   {
89     fprintf(listing,"\t%d: ",lineno);
90     printToken(0, currentToken,tokenString);
91   }
92   return currentToken;
93 }
```

APÊNDICE C – Código-fonte parser.y

```
%{
    #define YYPARSER

    #include "globals.h"
    #include "util.h"
    #include "scan.h"
    #include "parse.h"
    #define YYSTYPE TreeNode *
    static char * savedName; //names for use
    static int savedLineNo; //line number for use
    static TreeNode * savedTree; //tree to be returned
    static int savedNumber; //value for use

    char tokenString[MAXTOKENLEN+1];

    static int yylex(void);
    int yyerror(char * message);
%}

//Definic o dos tokens

%token ELSE IF INT RETURN VOID WHILE NUM ID SOM SUB MUL DIV MEN IME MAI IMA IGL DIF
%token ATR PEV VIR APR FPR ACL FCL ACH FCH FIM ENT TAB ERR


//Gramatica de C-
%%

programa: declaracao-lista{
            savedTree = $1;
        };

declaracao-lista: declaracao-lista declaracao{
                    YYSTYPE t = $1;
                     if (t != NULL) {
                       while (t->sibling != NULL) { t = t->sibling; }
                       t->sibling = $2;
                       $$ = $1;
                     } else {
                       $$ = $2;
                     }
                  }
                  | declaracao{
                      $$ = $1;
                  };

declaracao: var-declaracao{
                $$ = $1;
            }
            | fun-declaracao{
                $$ = $1;
            };

var-declaracao: tipo-especificador id PEV{
```

```
55                      $$ = newDeclNode(VarK);
56                      $$->child[0] = $1;
57                      $$->lineno = lineno;
58                      $$->attr.name = savedName;
59                    }
60                    | tipo-especificador id ACL num FCL PEV{
61                      $$ = newDeclNode(ArrVarK);
62                      $$->child[0] = $1;
63                      $$->lineno = lineno;
64                      $$->type = IntegerArray;
65                      $$->attr.arr.name = savedName;
66                      $$->attr.arr.size = savedNumber;
67                    };
68
69  tipo-especificador: INT{
70                          $$ = newExpNode(TypeK);
71                          $$->type = Integer;
72                      }
73                      | VOID{
74                          $$ = newExpNode(TypeK);
75                          $$->type = Void;
76                      };
77
78  fun-declaracao: tipo-especificador id {
79                                          $$ = newDeclNode(FunK);
80                                          $$->lineno = lineno;
81                                          $$->attr.name = savedName;
82                                      } APR params FPR composto-decl{
83                                          $$ = $3;
84                                          $$->child[0] = $1;
85                                          $$->child[1] = $5;
86                                          $$->child[2] = $7;
87                                      };
88
89  params: param-lista{
90              $$ = $1;
91          } | VOID{
92              $$ = newDeclNode(ParamK);
93              $$->type = Void;
94          };
95
96  param-lista: param-lista VIR param{
97                  YYSTYPE t = $1;
98                  if (t != NULL) {
99                      while (t->sibling != NULL){
100                         t = t->sibling;
101                     }
102                     t->sibling = $3;
103                     $$ = $1;
104                 } else {
105                     $$ = $3;
106                 }
107             } | param{
108                 $$ = $1;
109             };
110
111 param: tipo-especificador id{
112             $$ = newDeclNode(ParamK);
113             $$->child[0] = $1;
114             $$->attr.name = savedName;
```

```
115         } | tipo-especificador id ACL FCL{
116             $$ = newDeclNode(ArrParamK);
117             $$->child[0] = $1;
118             $$->attr.name = copyString(savedName);
119         };
120 
121 composto-decl: ACH local-declaracoes statement-lista FCH{
122                     $$ = newStmtNode(CompoundK);
123                     $$->child[0] = $2;
124                     $$->child[1] = $3;
125                 };
126 
127 local-declaracoes: local-declaracoes var-declaracao{
128                         YYSTYPE t = $1;
129                         if (t != NULL) {
130                             while (t->sibling != NULL){
131                                 t = t->sibling;
132                             }
133                             t->sibling = $2;
134                             $$ = $1;
135                         } else {
136                             $$ = $2;
137                         }
138                     } | {
139                         $$ = NULL;
140                     };
141 
142 statement-lista: statement-lista statement{
143                     YYSTYPE t = $1;
144                     if (t != NULL) {
145                         while (t->sibling != NULL) {
146                             t = t->sibling;
147                         }
148                     t->sibling = $2;
149                     $$ = $1;
150                     } else {
151                         $$ = $2;
152                     }
153                 } | {
154                     $$ = NULL;
155                 } | error {yyerrok;};
156 
157 statement: expressao-decl{
158                 $$ = $1;
159             } | composto-decl{
160                 $$ = $1;
161             } | selecao-decl{
162                 $$ = $1;
163             } | iteracao-decl{
164                 $$ = $1;
165             } | retorno-decl{
166                 $$ = $1;
167             };
168 
169 expressao-decl: expressao PEV{
170                     $$ = $1;
171                 } | PEV{
172                     $$ = NULL;
173                 };
174 
```

```
175 selecao-decl: IF APR expressao FPR statement{
176                 $$ = newStmtNode(IfK);
177                 $$->child[0] = $3;
178                 $$->child[1] = $5;
179             } | IF APR expressao FPR statement ELSE statement{
180                 $$ = newStmtNode(IfK);
181                 $$->child[0] = $3;
182                 $$->child[1] = $5;
183                 $$->child[2] = $7;
184             };
185 
186 iteracao-decl: WHILE APR expressao FPR statement{
187                     $$ = newStmtNode(WhileK);
188                     $$->child[0] = $3;
189                     $$->child[1] = $5;
190                 };
191 
192 retorno-decl: RETURN PEV{
193                 $$ = newStmtNode(ReturnK);
194                 $$->type = Void;
195             } | RETURN expressao PEV{
196                 $$ = newStmtNode(ReturnK);
197                 $$->child[0] = $2;
198             };
199 
200 expressao: var ATR expressao{
201                 $$ = newStmtNode(AssignK);
202                 $$->child[0] = $1;
203                 $$->child[1] = $3;
204             } | simples-expressao{
205                 $$ = $1;
206             };
207 
208 var: id {
209         $$ = newExpNode(IdK);
210         $$->attr.name = savedName;
211     } | id{
212         $$ = newExpNode(ArrIdK);
213         $$->attr.name = savedName;
214     } ACL expressao FCL{
215         $$ = $2;
216         $$->child[0] = $4;
217     };
218 
219 simples-expressao: soma-expressao relacional soma-expressao{
220                         $$ = newExpNode(CalcK);
221                         $$->child[0] = $1;
222                         $$->child[1] = $2;
223                         $$->child[2] = $3;
224                     } | soma-expressao{
225                         $$ = $1;
226                     };
227 
228 relacional: IME{
229                 $$ = newExpNode(OpK);
230                 $$->attr.op = IME;
231             } | MEN{
232                 $$ = newExpNode(OpK);
233                 $$->attr.op = MEN;
234             } | MAI{
```

```
235                 $$ = newExpNode(OpK);
236                 $$->attr.op = MAI;
237             } | IMA{
238                 $$ = newExpNode(OpK);
239                 $$->attr.op = IMA;
240             } | IGL{
241                 $$ = newExpNode(OpK);
242                 $$->attr.op = IGL;
243             } | DIF{
244                 $$ = newExpNode(OpK);
245                 $$->attr.op = DIF;
246             };
247 
248 soma-expressao: soma-expressao soma termo{
249                     $$ = newExpNode(CalcK);
250                     $$->child[0] = $1;
251                     $$->child[1] = $2;
252                     $$->child[2] = $3;
253                 } | termo{
254                     $$ = $1;
255                 };
256 
257 soma: SOM{
258         $$ = newExpNode(OpK);
259         $$->attr.op = SOM;
260     } | SUB{
261         $$ = newExpNode(OpK);
262         $$->attr.op = SUB;
263     };
264 
265 termo: termo mult fator{
266             $$ = newExpNode(CalcK);
267             $$->child[0] = $1;
268             $$->child[1] = $2;
269             $$->child[2] = $3;
270         } | fator{
271             $$ = $1;
272         };
273 
274 mult: MUL{
275         $$ = newExpNode(OpK);
276         $$->attr.op = MUL;
277     } | DIV{
278         $$ = newExpNode(OpK);
279         $$->attr.op = DIV;
280     };
281 
282 fator: APR expressao FPR{
283             $$ = $2;
284         } | var{
285             $$ = $1;
286         } | ativacao{
287             $$ = $1;
288         } | num{
289             $$ = newExpNode(ConstK);
290             $$->type = Integer;
291             $$->attr.val = atoi(tokenString);
292         };
293 
294 ativacao: id{
```

```
295             $$ = newExpNode(CallK);
296             $$->attr.name = savedName;
297         } APR args FPR{
298             $$ = $2;
299             $$->child[0] = $4;
300         };
301 
302 args: arg-lista{
303         $$ = $1;
304     } | {
305         $$ = NULL;
306     };
307 
308 arg-lista: arg-lista VIR expressao{
309                 YYSTYPE t = $1;
310                 if (t != NULL) {
311                 while (t->sibling != NULL) {
312                     t = t->sibling;
313                 }
314                 t->sibling = $3;
315                 $$ = $1;
316                 } else {
317                 $$ = $3;
318                 }
319             } | expressao{
320                 $$ = $1;
321             };
322 
323 id: ID {
324         savedName = copyString(tokenString);
325         savedLineNo = lineno;
326     };
327 
328 num: NUM{
329         savedNumber = atoi(tokenString);
330         savedLineNo = lineno;
331     }
332 %%
333 
334 int yyerror(char * message){
335     fprintf(errorfile, "ERRO SINTaTICO: ");
336     printToken(1, yychar, tokenString);
337     fprintf(errorfile, " LINHA: %d\n", lineno);
338     Error = TRUE;
339     return 0;
340 }
341 
342 static int yylex(void)
343 { return getToken(); }
344 
345 TreeNode * parse(void){
346     yyparse();
347     return savedTree;
348 }
```

APÊNDICE D – Código-fonte global.h

```
/****************************************************/
/* File: globals.h                                  */
/* Yacc/Bison Version                               */
/* Bruno Sampaio Leite                              */
/* Talita Ludmila de Lima                           */
/****************************************************/

#ifndef _GLOBALS_H_
#define _GLOBALS_H_

#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include <string.h>

/* Yacc/Bison generates internally its own values
 * for the tokens. Other files can access these values
 * by including the tab.h file generated using the
 * Yacc/Bison option -d ("generate header")
 *
 * The YYPARSER flag prevents inclusion of the tab.h
 * into the Yacc/Bison output itself
 */

#ifndef YYPARSER

/* the name of the following file may change */
#include "parser.tab.h"

/* ENDFILE is implicitly defined by Yacc/Bison,
 * and not included in the tab.h file
 */
#define ENDFILE 0

#endif

#ifndef FALSE
#define FALSE 0
#endif

#ifndef TRUE
#define TRUE 1
#endif

/* MAXRESERVED = the number of reserved words */
#define MAXRESERVED 8

/* Yacc/Bison generates its own integer values
 * for tokens
 */
typedef int TokenType;

extern FILE* source; /* source code text file */
extern FILE* listing; /* listing output text file */
```

```
55 extern FILE* errorfile;
56
57 extern int lineno; /* source line number for listing */
58
59 /**************************************************/
60 /***********   Syntax tree for parsing ************/
61 /**************************************************/
62
63 typedef enum { StmtK, ExpK, DeclK } NodeKind;
64 typedef enum { IfK, WhileK, AssignK, CompoundK, ReturnK } StmtKind;
65 typedef enum { OpK, ConstK, IdK, TypeK, ArrIdK, CallK, CalcK } ExpKind;
66 typedef enum { VarK, FunK, ArrVarK, ArrParamK, ParamK } DeclKind;
67
68 /* ExpType is used for type checking */
69 typedef enum {Void,Integer,IntegerArray} ExpType;
70
71 #define MAXCHILDREN 3
72 #define MAX_REG 26
73 #define MAX_MEM 1023
74
75 struct ScopeListRec;
76
77 typedef struct ArrayAttribute {
78   TokenType type;
79   char * name;
80   int size;
81 } ArrAttr;
82
83 typedef struct treeNode
84    { struct treeNode * child[MAXCHILDREN];
85      struct treeNode * sibling;
86      int lineno;
87      NodeKind nodekind;
88      union { StmtKind stmt; ExpKind exp; DeclKind decl;} kind;
89      union { TokenType op;
90              TokenType type;
91              int val;
92              char * name;
93              ArrAttr arr;} attr;
94      ExpType type; /* for type checking of exps */
95      int reg;
96      char* scopeName;
97    } TreeNode;
98
99 /**************************************************/
100 /***********   Flags for tracing       ************/
101 /**************************************************/
102
103 /* EchoSource = TRUE causes the source program to
104  * be echoed to the listing file with line numbers
105  * during parsing
106  */
107 extern int EchoSource;
108
109 /* TraceScan = TRUE causes token information to be
110  * printed to the listing file as each token is
111  * recognized by the scanner
112  */
113 extern int TraceScan;
114
```

```
115 /* TraceParse = TRUE causes the syntax tree to be
116  * printed to the listing file in linearized form
117  * (using indents for children)
118  */
119 extern int TraceParse;
120
121 /* TraceAnalyze = TRUE causes symbol table inserts
122  * and lookups to be reported to the listing file
123  */
124 extern int TraceAnalyze;
125
126 /* TraceCode = TRUE causes comments to be written
127  * to the TM code file as code is generated
128  */
129 extern int TraceCode;
130
131 /* Error = TRUE prevents further passes if an error occurs */
132 extern int Error;
133 #endif
```

APÊNDICE E – Código-fonte util.c

```h
/****************************************************/
/* File: util.h                                     */
/* Alunos:                                          */
/* Bruno Sampaio Leite                              */
/* Talita Ludmila de                                */
/****************************************************/

#ifndef _UTIL_H_
#define _UTIL_H_

/* Procedure printToken prints a token
 * and its lexeme to the listing file
 */
char* printToken(int arq, TokenType, const char* );

void printTypes(TreeNode* tree);
/* Function newStmtNode creates a new statement
 * node for syntax tree construction
 */
TreeNode * newStmtNode(StmtKind);

/* Function newExpNode creates a new expression
 * node for syntax tree construction
 */
TreeNode * newExpNode(ExpKind);

TreeNode * newDeclNode(DeclKind);
/* Function copyString allocates and makes a new
 * copy of an existing string
 */
char * copyString( char * );

/* procedure printTree prints a syntax tree to the
 * listing file using indentation to indicate subtrees
 */
void printTree( TreeNode * );

#endif
```

```h
#include "globals.h"
#include "util.h"
#include "codegenerate.h"
#include "assemblygenerate.h"

FILE * errorfile;
FILE * intermed;

/* Procedure printToken prints a token
 * and its lexeme to the listing file
 */
char* printToken(int arq, TokenType token, const char* tokenString )
{
  if(arq == 1){
    switch (token)
```

```
16      { case ELSE:
17        case IF:
18        case INT:
19        case RETURN:
20        case VOID:
21        case WHILE:
22        fprintf(errorfile, "reserved word: %s ",tokenString);
23        break;
24        case SOM:     fprintf(errorfile, "+ ");  break;
25        case SUB:    fprintf(errorfile, "- ");  break;
26        case MUL:    fprintf(errorfile, "* ");  break;
27        case DIV:     fprintf(errorfile, "/ ");  break;
28        case MEN:       fprintf(errorfile, "< ");  break;
29        case IME:       fprintf(errorfile, "<= "); break;
30        case MAI:       fprintf(errorfile, "> ");  break;
31        case IMA:       fprintf(errorfile, ">= "); break;
32        case IGL:       fprintf(errorfile, "== "); break;
33        case DIF:       fprintf(errorfile, "!= "); break;
34        case ATR:   fprintf(errorfile, "= ");  break;
35        case PEV:     fprintf(errorfile, "; ");  break;
36        case VIR:    fprintf(errorfile, ", ");  break;
37        case APR:   fprintf(errorfile, "( ");  break;
38        case FPR:   fprintf(errorfile, ") ");  break;
39        case ACL: fprintf(errorfile, "[ ");  break;
40        case FCL: fprintf(errorfile, "] ");  break;
41        case ACH:   fprintf(errorfile, "{ ");  break;
42        case FCH:   fprintf(errorfile, "} ");  break;
43        case ENDFILE:  fprintf(errorfile,"%s %s ",  "ENDFILE", "EOF"); break;
44        case NUM: fprintf(errorfile, "NUM, val = %s ",tokenString); break;
45        case ID: fprintf(errorfile, "ID, name = %s ",tokenString); break;
46        case ERR: fprintf(errorfile, "ERROR: %s ",tokenString); break;
47        default: /* should never happen */
48          fprintf(errorfile,"Unknown token: %d ",token);
49      }
50    }else if(arq == 0){
51      switch (token)
52      { case ELSE:
53        case IF:
54        case INT:
55        case RETURN:
56        case VOID:
57        case WHILE:
58        fprintf(listing, "reserved word: %s\n",tokenString);
59        break;
60        case SOM:     fprintf(listing, "+\n");  break;
61        case SUB:    fprintf(listing, "-\n");  break;
62        case MUL:    fprintf(listing, "*\n");  break;
63        case DIV:     fprintf(listing, "/\n");  break;
64        case MEN:       fprintf(listing, "<\n");  break;
65        case IME:       fprintf(listing, "<=\n"); break;
66        case MAI:       fprintf(listing, ">\n");  break;
67        case IMA:       fprintf(listing, ">=\n"); break;
68        case IGL:       fprintf(listing, "==\n"); break;
69        case DIF:       fprintf(listing, "!=\n"); break;
70        case ATR:   fprintf(listing, "=\n");  break;
71        case PEV:     fprintf(listing, ";\n");  break;
72        case VIR:    fprintf(listing, ",\n");  break;
73        case APR:   fprintf(listing, "(\n");  break;
74        case FPR:   fprintf(listing, ")\n");  break;
75        case ACL: fprintf(listing, "[\n");  break;
```

```
76        case FCL: fprintf(listing, "]\n");  break;
77        case ACH:   fprintf(listing, "{\n");  break;
78        case FCH:   fprintf(listing, "}\n");  break;
79        case ENDFILE:  fprintf(listing,"%s %s\n",  "ENDFILE", "EOF"); break;
80        case NUM: fprintf(listing, "NUM, val = %s\n",tokenString); break;
81        case ID: fprintf(listing, "ID, name = %s\n",tokenString); break;
82        case ERR: fprintf(listing, "ERROR: %s\n",tokenString); break;
83        default: /* should never happen */
84          fprintf(listing,"Unknown token: %d\n",token);
85      }
86    }else{
87      switch(token)
88      {
89        case ELSE:    return "ELSE"; break;
90        case IF:      return "IF"; break;
91        case INT:     return "INT"; break;
92        case RETURN:  return "RETURN"; break;
93        case VOID:    return "VOID"; break;
94        case WHILE:   return "WHILE"; break;
95        case SOM:     return "SOM"; break;
96        case SUB:     return "SUB";  break;
97        case MUL:     return "MUL";  break;
98        case DIV:     return "DIV";  break;
99        case MEN:     return "MEN";  break;
100       case IME:     return "IME";  break;
101       case MAI:     return "MAI";  break;
102       case IMA:     return "IMA";  break;
103       case IGL:     return "IGL";  break;
104       case DIF:     return "DIF";  break;
105       case ATR:     return "ATR";  break;
106       case PEV:     return "PEV";  break;
107       case VIR:     return "VIR";  break;
108       case APR:     return "APR";  break;
109       case FPR:     return "FPR";  break;
110       case ACL:     return "ACL";  break;
111       case FCL:     return "FCL";  break;
112       case ACH:     return "ACH";  break;
113       case FCH:     return "FCH";  break;
114       case GOTO:    return "GOTO"; break;
115       case LAB:     return "LAB";  break;
116       case ASSIGN:  return "ASSIGN"; break;
117       case STORE:   return "STORE"; break;
118       case LOADI:   return "LOADI"; break;
119       case LOAD:    return "LOAD"; break;
120       case FUN:     return "FUN"; break;
121       case VAR:     return "VAR"; break;
122       case VAR_VET: return "VAR_VET"; break;
123       case SOMi:    return "SOMi"; break;
124       case PAR_VET: return "PAR_VET"; break;
125       case PAR:     return "PAR"; break;
126       default: /* should never happen */
127         return "INDEFINIDO";
128         break;
129     }
130   }
131 }
132
133 /* Function newStmtNode creates a new statement
134  * node for syntax tree construction
135  */
```

```
136 TreeNode * newStmtNode(StmtKind kind)
137 { TreeNode * t = (TreeNode *) malloc(sizeof(TreeNode));
138   int i;
139   if (t==NULL)
140     fprintf(listing,"Out of memory error at line %d\n",lineno);
141   else {
142     for (i=0;i<MAXCHILDREN;i++) t->child[i] = NULL;
143     t->sibling = NULL;
144     t->nodekind = StmtK;
145     t->kind.stmt = kind;
146     t->lineno = lineno;
147     t->scopeName = NULL;
148   }
149   return t;
150 }
151
152 /* Function newExpNode creates a new expression
153  * node for syntax tree construction
154  */
155 TreeNode * newExpNode(ExpKind kind)
156 { TreeNode * t = (TreeNode *) malloc(sizeof(TreeNode));
157   int i;
158   if (t==NULL)
159     fprintf(listing,"Out of memory error at line %d\n",lineno);
160   else {
161     for (i=0;i<MAXCHILDREN;i++) t->child[i] = NULL;
162     t->sibling = NULL;
163     t->nodekind = ExpK;
164     t->kind.exp = kind;
165     t->lineno = lineno;
166     t->type = Void;
167     t->scopeName = NULL;
168     //t->temporary = 0;
169   }
170   return t;
171 }
172
173 /* Function newDeclNode creates a new declaration
174  * node for syntax tree construction
175  */
176 TreeNode * newDeclNode(DeclKind kind)
177 {
178   TreeNode * t = (TreeNode *) malloc(sizeof(TreeNode));
179   int i;
180   if (t==NULL)
181     fprintf(listing,"Out of memory error at line %d\n",lineno);
182   else {
183     for (i=0;i<MAXCHILDREN;i++) t->child[i] = NULL;
184     t->sibling = NULL;
185     t->nodekind = DeclK;
186     t->kind.decl = kind;
187     t->lineno = lineno;
188     t->type = Void;
189     t->scopeName = NULL;
190   }
191   return t;
192 }
193
194 /* Function copyString allocates and makes a new
195  * copy of an existing string
```

```
196  */
197 char * copyString(char * s)
198 { int n;
199   char * t;
200   if (s==NULL) return NULL;
201   n = strlen(s)+1;
202   t = malloc(n);
203   if (t==NULL)
204     fprintf(listing,"Out of memory error at line %d\n",lineno);
205   else strcpy(t,s);
206   return t;
207 }
208
209 /* Variable indentno is used by printTree to
210  * store current number of spaces to indent
211  */
212 static int indentno = 0;
213
214 /* macros to increase/decrease indentation */
215 #define INDENT indentno+=2
216 #define UNINDENT indentno-=2
217
218 /* printSpaces indents by printing spaces */
219 static void printSpaces(void)
220 { int i;
221   for (i=0;i<indentno;i++)
222     fprintf(listing," ");
223 }
224
225 /* printType print types of functions and variables */
226 void printTypes(TreeNode* tree) {
227   if (tree->child[0] != NULL) {
228     switch (tree->child[0]->type) {
229       case Integer:
230         fprintf(listing,"int");
231         break;
232       case Void:
233         fprintf(listing,"void");
234         break;
235       case IntegerArray:
236         fprintf(listing,"int array");
237         break;
238       default: return;
239     }
240   } else {
241     switch (tree->type) {
242       case Integer:
243         fprintf(listing,"int");
244         break;
245       case Void:
246         fprintf(listing,"void");
247         break;
248       case IntegerArray:
249         fprintf(listing,"int array");
250         break;
251       default: return;
252     }
253   }
254 }
255
```

```
256 /* procedure printTree prints a syntax tree to the
257  * listing file using indentation to indicate subtrees
258  */
259 void printTree( TreeNode * tree )
260 { int i;
261   INDENT;
262   // IfK, WhileK, CompoundK, ReturnK
263   while (tree != NULL) {
264     printSpaces();
265     if (tree->nodekind==StmtK) {
266       printSpaces();
267       switch (tree->kind.stmt) {
268         case IfK:
269           fprintf(listing,"If\n");
270           break;
271         case WhileK:
272           fprintf(listing, "While\n");
273           break;
274         case CompoundK:
275           fprintf(listing, "Declaracao composta\n");
276           break;
277         case ReturnK:
278           fprintf(listing, "Return\n");
279           break;
280         case AssignK:
281           fprintf(listing, "Atribuicao\n");
282           break;
283         default:
284           fprintf(listing, "Unknown stmtNode kind\n");
285           break;
286         }
287     }
288     // OpK, ConstK, AssignK, IdK, TypeK, ArrIdK, CallK, CalK
289     else if (tree->nodekind==ExpK)
290     { if (tree->kind.exp != TypeK) printSpaces();
291       switch (tree->kind.exp) {
292         case OpK:
293           fprintf(listing,"Op: ");
294           printToken(0, tree->attr.op, "\0");
295           break;
296         case ConstK:
297           fprintf(listing,"Const: %d\n",tree->attr.val);
298           break;
299         case IdK:
300           fprintf(listing,"Id: %s\n",tree->attr.name);
301           break;
302         case TypeK: break;
303         case ArrIdK:
304           fprintf(listing,"ArrId \n");
305           break;
306         case CallK:
307           fprintf(listing, "Chamada de funcao: %s\n", tree->attr.name);
308           break;
309         case CalcK:
310           fprintf(listing, "Operador : ");
311           printToken(0, tree->child[1]->attr.op, "\0");
312           break;
313         default:
314           fprintf(listing,"Unknown ExpNode kind\n");
315           break;
```

```
316       }
317     }
318     // VarK, FunK, ArrVarK, ArrParamK, ParamK
319       else if (tree->nodekind==DeclK) {
320         printSpaces();
321         switch (tree->kind.decl) {
322           case FunK :
323             fprintf(listing, "Declaracao de funcao:  ");
324             printTypes(tree);
325             fprintf(listing, " %s()\n", tree->attr.name);
326             break;
327           case VarK :
328             fprintf(listing, "Declaracao de variavel:  ");
329             printTypes(tree);
330             fprintf(listing, " %s;\n", tree->attr.name);
331             break;
332           case ArrVarK :
333             fprintf(listing, "Declaracao de Vetor:  ");
334             printTypes(tree);
335             fprintf(listing," %s[%d];\n", tree->attr.arr.name, tree->attr.arr.size);
336             break;
337           case ArrParamK :
338             fprintf(listing, "Parametro Vetor: %s\n", tree->attr.name);
339             break;
340           case ParamK :
341             fprintf(listing, "Parametro: ");
342             printTypes(tree);
343             if (tree->attr.name != NULL) {
344               fprintf(listing, " %s\n", tree->attr.name);
345             } else {
346               fprintf(listing, " void\n");
347             }
348             break;
349           default:
350             fprintf(listing, "Unknown Declaration\n");
351             break;
352           }
353         }else fprintf(listing,"Unknown node kind\n");
354       for (i=0;i<MAXCHILDREN;i++)
355         if (tree->child[i] != NULL)
356           printTree(tree->child[i]);
357       tree = tree->sibling;
358     }
359     UNINDENT;
360   }
```

APÊNDICE F – Código-fonte analyze.c

```c
/****************************************************/
/* File: analyze.c                                  */
/* Semantic analyzer implementation                 */
/*  Alunos:                                         */
/*  Bruno Sampaio Leite   120213                    */
/*  Talita Ludmila de Lima    120895                */
/****************************************************/

#include "globals.h"
#include "analyze.h"
#include "util.h"
#include "symboltable.h"

FILE * errorfile;

/*
 * Func o responsavel por criar um no do tipo FunK
 * e inserir na arvore sintatica representando a
 * func o int input() na qual um valor inteiro e
 * inserido pelo usuario.
 */

static void insertInputFunc(void) {
  TreeNode * fun_declaration = newDeclNode(FunK);
  fun_declaration->type = Integer;

  TreeNode * type_specifier = newExpNode(TypeK);
  type_specifier->attr.type = INT;

  TreeNode * compound_stmt = newStmtNode(CompoundK);
  compound_stmt->child[0] = NULL;
  compound_stmt->child[1] = NULL;

  fun_declaration->lineno = 0;
  fun_declaration->attr.name = "input";
  fun_declaration->child[0] = type_specifier;
  fun_declaration->child[1] = NULL;
  fun_declaration->child[2] = compound_stmt;

  /* Insert input function*/
  st_insert("global", "input", Integer, fun_declaration, 0);
}

/*
 * Func o responsavel por criar um no do tipo FunK
 * e inserir na arvore sintatica representando a
 * func o int output() que imprime o valor do argumento.
 */

static void insertOutputFunc(void) {

  TreeNode * fun_declaration = newDeclNode(FunK);
  fun_declaration->type = Void;

```

```
55    TreeNode * type_specifier = newDeclNode(FunK);
56    type_specifier->attr.type = VOID;
57
58    TreeNode * params = newDeclNode(ParamK);
59    params->attr.name = "arg";
60    params->child[0] = newExpNode(TypeK);
61    params->child[0]->attr.type = INT;
62
63    TreeNode * compound_stmt = newStmtNode(CompoundK);
64    compound_stmt->child[0] = NULL;
65    compound_stmt->child[1] = NULL;
66
67    fun_declaration->lineno = 0;
68    fun_declaration->attr.name = "output";
69    fun_declaration->child[0] = type_specifier;
70    fun_declaration->child[1] = params;
71    fun_declaration->child[2] = compound_stmt;
72
73    /* Insert output function*/
74    st_insert("global", "output", Void, fun_declaration, 0);
75  }
76
77  /* contador para as posicoes de memoria de variaveis */
78  static int location = 0;
79
80  /* Procedure traverse is a generic recursive
81   * syntax tree traversal routine:
82   * it applies preProc in preorder and postProc
83   * in postorder to tree pointed to by t
84   */
85  static void traverse( TreeNode * t, void (* preProc) (TreeNode *), void (* postProc) (
      TreeNode *) ){
86    if (t != NULL)
87    { preProc(t);
88      { int i;
89        for (i=0; i < MAXCHILDREN; i++)
90          traverse(t->child[i],preProc,postProc);
91      }
92      postProc(t);
93      traverse(t->sibling,preProc,postProc);
94    }
95  }
96
97  /* nullProc is a do-nothing procedure to
98   * generate preorder-only or postorder-only
99   * traversals from traverse
100  */
101 static void nullProc(TreeNode * t)
102 { if (t==NULL) return;
103   else return;
104 }
105
106 /* Func o auxiliar da func o traverse() */
107
108 static void popAfterInsertProc(TreeNode * t) {
109
110   if (t->nodekind == StmtK) {
111     if (t->kind.stmt == CompoundK) {
112       popScope();
113     }
```

```
114   }
115
116   if (t==NULL) return;
117   else return;
118 }
119
120 /* Funcoes responsaveis por imprimir mensagens de erro no aruivo de erros */
121
122 static void typeError(TreeNode * t, char * message)
123 { fprintf(errorfile,"Erro de tipo na linha %d: %s\n",t->lineno,message);
124   Error = TRUE;
125   //exit(-1);
126 }
127
128 static void symbolError(TreeNode * t, char * message) {
129   fprintf(errorfile,"ERRO SEMANTICO: Linha: %d: %s\n", t->lineno, message);
130   Error = TRUE;
131   //exit(-1);
132 }
133
134 // this is needed to check parameters
135 static int isFirstCompoundK = 0;
136 static int locationCounter = 1;
137
138 /* Procedimento que insere elementos na tabela de simbolos */
139
140 static void insertNode( TreeNode * t)
141 { switch (t->nodekind)
142   {
143     //Statement case
144     case StmtK:{
145       switch (t->kind.stmt) {
146         case CompoundK: {
147           if (!isFirstCompoundK) { // verifica e cria escopo se necessario
148             Scope scope = newScope(currScope()->name, currScope()->type);
149             scope->parent = currScope();
150             pushScope(scope);
151             t->scopeName = currScope()->name;
152           }
153           isFirstCompoundK = 0;
154           break;
155         }
156         case ReturnK:
157           if(t->child[0] == NULL){
158             /* verifica se o retorno e indevidamente nulo */
159               if(currScope()->type != Void){
160                 typeError(t, "Retorno esperado!");
161               }
162             }else{
163               /* verifica se ha retorno indevido */
164               if(currScope()->type == Void){
165                 typeError(t, "Retorno vazio esperado!");
166               }
167             }
168             /* atualiza nome do escopo no no */
169             t->scopeName = currScope()->name;
170           break;
171         case AssignK: {
172           /* verifica a coerencia dos tipos em uma atribuic o */
173           if (t->child[0]->type == IntegerArray) {
```

```
174             if(t->child[0]->child[0] == NULL){
175               typeError(t->child[0], "Atribuicao para uma variavel do tipo vetor");
176             }
177           }
178           /* verifica atribuic o indevida */
179           if (t->child[0]->attr.arr.type == Void) {
180             typeError(t->child[0], "Atribuicao para uma variavel do tipo VOID");
181           }
182           /* verifica se ha retorno para ser atribuido */
183           if(t->child[1]->kind.exp == CallK){
184             if(st_lookup_scope(t->child[1]->attr.name) != NULL){
185               if(st_lookup_scope(t->child[1]->attr.name)->type == Void){
186                 typeError(t->child[1], "Atribuicao de VOID");
187               }
188             }
189           }
190           /* atualiza nome do escopo no no */
191           t->scopeName = currScope()->name;
192           break;
193         }
194
195         default:
196           break;
197       }
198
199       break;
200     }
201     //Expression case
202     case ExpK: {
203       switch (t->kind.exp){
204         case IdK:
205         case ArrIdK:
206         case CallK: {
207           // check undeclation / func o n o declarada
208           if (st_lookup_all_scope(t->attr.name) == NULL){
209             symbolError(t, "Simbolo nao definido");
210           } else {
211             BucketList list = st_lookup_all_scope(t->attr.name);
212             t->type = list->type;
213             insertLines(t->attr.name, t->lineno);
214             t->scopeName = currScope()->name;
215           }
216           break;
217         }
218         default:
219           break;
220       }
221       break;
222     }
223
224     //Declaration case
225     case DeclK: {
226       switch (t->kind.decl) {
227         case FunK: {
228           // initialize location counter
229           locationCounter = 0;
230           /* Look up scope list to check scope existence / func o ja declarada*/
231           if (st_lookup_scope(t->attr.name) != NULL) {
232             symbolError(t, "Redefinicao de funcao");
233             break;
```

```
234           }
235           //verifica se o escopo atual e o global e cria um novo escopo
236           if (strcmp(currScope()->name, "global") == 0) {
237             t->scopeName = currScope()->name;
238             st_insert(currScope()->name, t->attr.name, t->child[0]->type, t,
                  locationCounter++);
239           }
240
241           Scope scope = newScope(t->attr.name, t->child[0]->type);
242           scope->parent = currScope();
243           pushScope(scope);
244           isFirstCompoundK = 1;
245           break;
246         }
247
248         //Variable case
249         case VarK: {
250           /* Look up to check variable existence / variavel ja declarada*/
251           if (st_lookup(t->attr.name) != NULL) {
252             symbolError(t, "Redefinicao de variavel");
253             break;
254           }
255           // func o com nome ja existe
256           if(st_lookup_scope(t->attr.name) != NULL){
257             symbolError(t, "Declaracao Invalida");
258             break;
259           }
260
261           // Type Checking : Type should not be void / variavel como void
262           if (t->child[0]->type == Void) {
263             symbolError(t, "Variavel nao deveria ser do tipo VOID");
264             break;
265           }
266           t->scopeName = currScope()->name;
267           st_insert(currScope()->name, t->attr.name, t->child[0]->type, t,
                locationCounter++);
268           break;
269         }
270
271         //Array case
272         case ArrVarK: {
273
274           // Type Checking : Type should not be void / vetor como void
275           if (t->child[0]->type == Void) {
276             symbolError(t, "Tipo invalido");
277             break;
278           }
279           // func o ja declarada
280           if(st_lookup_scope(t->attr.arr.name) != NULL){
281             symbolError(t, "Declaracao invalida");
282             break;
283           }
284           //vetor ja declarado
285           /*  Look up to check array variable existence  */
286           if (st_lookup(t->attr.arr.name) != NULL) {
287             symbolError(t, "Vetor ja foi declarado");
288             break;
289           }
290           t->scopeName = currScope()->name;
291           st_insert(currScope()->name, t->attr.arr.name, t->type, t, locationCounter++);
```

```
292           locationCounter = locationCounter + t->attr.arr.size;
293           break;
294         }
295
296         //ArrayParamenter case
297         case ArrParamK: {
298
299           // Type Checking : Type should not be void / argumento n o pode ser void
300           if (t->child[0]->type == Void) {
301             symbolError(t, "Tipo invalido");
302             break;
303           }
304           // func o com esse nome ja existe
305           if(st_lookup_scope(t->attr.name) != NULL){
306             symbolError(t, "Declaracao invalida");
307             break;
308           }
309
310           /*  Look up to check array parameter existence / parametro ja existente */
311           if (st_lookup(t->attr.name) != NULL) {
312             symbolError(t, "Redefinicao de um parametro vetor");
313             break;
314           }
315           t->scopeName = currScope()->name;
316           st_insert(currScope()->name, t->attr.name, t->type, t, locationCounter++);
317           break;
318         }
319
320         //Parameter case
321         case ParamK: {
322
323           if (t->attr.name != NULL) {
324             /*  Look up to check parameter existence  */
325             if(t->child[0]->type == Void){
326               symbolError(t, "Tipo invalido");
327               break;
328             }
329
330             if(st_lookup_scope(t->attr.name) != NULL){
331               symbolError(t, "Declaracao invalida");
332               break;
333             }
334
335             if (st_lookup(t->attr.name) != NULL) {
336               symbolError(t, "Redefinicao de parametro");
337               break;
338             }
339             t->scopeName = currScope()->name;
340             st_insert(currScope()->name, t->attr.name, t->child[0]->type, t,
                  locationCounter++);
341           }
342           break;
343         }
344         default:
345           break;
346       }
347       break;
348       t->scopeName = currScope()->name;
349     }
350     default:
```

```
351         break;
352       }
353
354 }
355
356 /* Function buildSymtab constructs the symbol
357  * table by preorder traversal of the syntax tree
358  */
359 void buildSymtab(TreeNode * syntaxTree)
360 {
361   globalScope = newScope("global", Void);
362   // push global scope
363   pushScope(globalScope);
364
365   insertInputFunc();
366   insertOutputFunc();
367
368   traverse(syntaxTree, insertNode, popAfterInsertProc);
369   popScope();
370
371   if(st_lookup_scope("main") == NULL){
372     fprintf(errorfile, "ERRO SEMANTICO: Main nao declarada!\n");
373   }
374
375   if (TraceAnalyze)
376   {
377     printSymTab(listing);
378   }
379 }
```

APÊNDICE G – Código-fonte symboltable.c

```
1  /****************************************************/
2  /* File: symtab.h                                   */
3  /* Alunos:                                          */
4  /* Bruno Sampaio Leite                              */
5  /* Talita Ludmila de Lima                           */
6  /****************************************************/
7
8  #ifndef _SYMTAB_H_
9  #define _SYMTAB_H_
10
11 #include "globals.h"
12
13 #define SIZE 211
14
15 static int hash ( char * key );
16
17 typedef struct LineListRec{
18   int lineno;
19   struct LineListRec * next;
20 } * LineList;
21
22 typedef struct BucketListRec {
23   char * name;
24   TreeNode * treeNode;
25   LineList lines;
26   int memloc; /* memory location for variable */
27   struct BucketListRec * next;
28   ExpType type;
29 } * BucketList;
30
31 /* Procedure st_insert inserts line numbers and
32  * memory locations into the symbol table
33  * loc = memory location is inserted only the
34  * first time, otherwise ignored
35  */
36 void st_insert(char * scopeName, char * name, ExpType type, TreeNode * streeNode, int loc
       );
37
38 /* Function st_lookup returns the memory
39  * location of a variable or -1 if not found
40  */
41 BucketList st_lookup(char * name);
42
43 /* Procedure printSymTab prints a formatted
44  * listing of the symbol table contents
45  * to the listing file
46  */
47
48 void printSymTab(FILE * listing);
49
50 /*Function st_lookup_memPos returns the memory
51  * location of a variable or NULL if not found
52  */
53
```

```
54 int st_lookup_memPos(char *varName, char *scopeName);
55
56 int st_isGlobal(char *varName, char *scopeName);
57
58 /*
59  *Scope definitions
60 */
61
62 //Scope List
63 typedef struct ScopeListRec
64 {
65   char * name; // function name
66   int nestedLevel;
67   int qtdeElementos;
68   int posMem;
69   struct ScopeListRec *parent;
70   BucketList hashTable[SIZE]; /* the hash table */
71   ExpType type;
72 } * Scope;
73
74 // global scope to cover function definitions
75 Scope globalScope;
76
77 //Scope List to output
78 static Scope scopeList[SIZE];
79 static int sizeOfList = 0;
80
81 //Stack to deal with scope
82 static Scope scopeStack[SIZE];
83 static int topScope = 0;
84
85 Scope newScope(char * scopeName, ExpType type);
86 void popScope(void);
87 void pushScope(Scope scope);
88 void insertScopeToList(Scope scope);
89 Scope currScope();
90 Scope st_lookup_scope(char * scopeName);
91 BucketList st_lookup_all_scope(char * name);
92 void insertLines(char* name, int lineno);
93
94 #endif
```

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4  #include "symboltable.h"
5  #include "globals.h"
6
7  /* SIZE is the size of the hash table */
8  #define SIZE 211
9
10 /* SHIFT is the power of two used as multiplier
11    in hash function  */
12 #define SHIFT 4
13
14 /* the hash function */
15 static int hash ( char * key )
16 { int temp = 0;
17   int i = 0;
18   while (key[i] != '\0')
```

```
19    { temp = ((temp << SHIFT) + key[i]) % SIZE;
20      ++i;
21    }
22    return temp;
23  }
24
25  /* the hash table */
26  static BucketList hashTable[SIZE];
27
28  /* Procedure st_insert inserts line numbers and
29   * memory locations into the symbol table
30   * loc = memory location is inserted only the
31   * first time, otherwise ignored
32   */
33  void st_insert(char * scopeName, char * name, ExpType type, TreeNode * treeNode, int loc)
34  {
35    int h = hash(name);
36    Scope scope = currScope();
37
38    BucketList l = scope->hashTable[h];
39    // fprintf(listing, "%s %s st_insert\n", scope->name, name);
40    /** try to find bucket */
41    while ((l != NULL) && (strcmp(name, l->name) != 0)) l = l->next;
42
43    /* variable not yet in BucketList */
44    if (l == NULL)
45    {
46      // atualiza a quantidade de elementos do escopo
47      if(treeNode->nodekind != DeclK)
48        scope->qtdeElementos++;
49        else if(treeNode->kind.decl != FunK)
50        {
51          scope->qtdeElementos++;
52          // atualiza a quantidade de posicoes de memoria necessarias
53          scope->posMem++;
54        }
55
56      l = (BucketList) malloc(sizeof(struct BucketListRec));
57      l->name = name;
58      l->treeNode = treeNode;
59      l->lines = (LineList) malloc(sizeof(struct LineListRec));
60      l->lines->lineno = treeNode->lineno;
61      l->type = type;
62      l->memloc = scope->posMem;
63      l->lines->next = NULL;
64      l->next = scope->hashTable[h];
65      scope->hashTable[h] = l;
66    }
67    else
68    {
69      /* already exist in the BucketList */
70      LineList t = l->lines;
71      while (t->next != NULL) t = t->next;
72      t->next = (LineList) malloc(sizeof(struct LineListRec));
73      t->next->lineno = lineno;
74      t->next->next = NULL;
75    }
76  } /* st_insert */
77
78  /* Function st_lookup returns the memory
```

```
79  * location of a variable or NULL if not found
80  */
81
82  BucketList st_lookup(char * name)
83  {
84
85    Scope scope = currScope();
86    int h = hash(name);
87    BucketList bucket = scope->hashTable[h];
88
89    while ((bucket != NULL) && (strcmp(name, bucket->name) != 0)) bucket = bucket->next;
90    return bucket;
91  }
92
93  int st_isGlobal(char* varName, char* scopeName){
94    int i, h;
95    Scope foundScope;
96    BucketList bucket;
97    for(i=0; i< sizeOfList; i++)
98    {
99      if(strcmp(scopeList[i]->name, scopeName)==0)
100     {
101         foundScope=scopeList[i];
102         break;
103     }
104   }
105   h = hash(varName);
106   bucket = foundScope->hashTable[h];
107   while ((bucket != NULL) && (strcmp(varName, bucket->name) != 0)) bucket = bucket->next;
108
109   if(bucket != NULL){
110     return 0;
111   }
112
113   if(bucket == NULL){
114     //PROCURA NO GLOBAL
115     for(i=0; i< sizeOfList; i++)
116     {
117       if(strcmp(scopeList[i]->name, "global")==0)
118       {
119           foundScope=scopeList[i];
120           break;
121       }
122     }
123
124     h = hash(varName);
125     bucket = foundScope->hashTable[h];
126     while ((bucket != NULL) && (strcmp(varName, bucket->name) != 0)) bucket = bucket->
          next;
127
128     if(bucket != NULL){
129       return 1;
130     }
131   }
132 }
133
134 /*Function st_lookup_memPos returns the memory
135  * location of a variable or NULL if not found
136  */
137
```

```
138 int st_lookup_memPos(char* varName, char* scopeName)
139 {
140   int i, h;
141   Scope foundScope;
142   BucketList bucket;
143   for(i=0; i< sizeOfList; i++)
144   {
145     if(strcmp(scopeList[i]->name, scopeName)==0)
146     {
147         foundScope=scopeList[i];
148         break;
149     }
150   }
151   h = hash(varName);
152   bucket = foundScope->hashTable[h];
153   while ((bucket != NULL) && (strcmp(varName, bucket->name) != 0)) bucket = bucket->next;
154
155   if(bucket != NULL){
156     //RETORNA OFFSET LOCAL
157     return bucket->memloc;
158   }
159
160   if(bucket == NULL){
161     //PROCURA NO GLOBAL
162     for(i=0; i< sizeOfList; i++)
163     {
164       if(strcmp(scopeList[i]->name, "global")==0)
165       {
166           foundScope=scopeList[i];
167           break;
168       }
169     }
170
171
172     h = hash(varName);
173     bucket = foundScope->hashTable[h];
174     while ((bucket != NULL) && (strcmp(varName, bucket->name) != 0)) bucket = bucket->
          next;
175
176     if(bucket != NULL){
177       //RETORNA OFFSET DO GLOBAL
178       bucket->memloc;
179     }else{
180       return -999;
181     }
182   }
183 }
184
185 /*
186  * Insert lines
187  */
188
189 void insertLines(char* name, int lineno)
190 {
191   Scope scope = currScope();
192   int h = hash(name);
193   BucketList l = scope->hashTable[h];
194
195   while (scope != NULL)
196   {
```

```
197
198     if (l != NULL)
199     {
200       LineList lines = l->lines;
201
202       while (lines->next != NULL)
203       {
204         lines = lines->next;
205       }
206
207       lines->next =  (LineList) malloc(sizeof(struct LineListRec));
208       lines->next->lineno = lineno;
209       lines->next->next = NULL;
210       return;
211     }
212     scope = scope->parent;
213   }
214 }
215
216 // return the scope case scopeName is already a scope
217 Scope st_lookup_scope(char * scopeName)
218 {
219   Scope scope = NULL;
220   for (int i=0; i<sizeOfList; i++)
221   {
222     if (strcmp(scopeList[i]->name, scopeName) == 0)
223     {
224       scope = scopeList[i];
225       break;
226     }
227   }
228   return scope;
229 }
230
231 // return the bucket related to the name if it already exist
232 BucketList st_lookup_all_scope(char * name)
233 {
234   Scope scope = currScope();
235   int h = hash(name);
236   BucketList bucket;
237
238   while (scope != NULL )
239   {
240     BucketList bucket = scope->hashTable[h];
241     while ((bucket != NULL) && (strcmp(name, bucket->name) != 0)) bucket = bucket->next;
242     if (bucket != NULL) return bucket;
243     scope = scope->parent;
244   }
245   return NULL;
246 }
247
248 /* Procedure printSymTab prints a formatted
249  * listing of the symbol table contents
250  * to the listing file
251  */
252 void printSymTabRows(Scope scope)
253 {
254
255   BucketList * hashTable = scope->hashTable;
256
```

```
257   for (int i=0; i<SIZE; ++i) {
258     if (hashTable[i] != NULL) {
259       BucketList l = hashTable[i];
260       TreeNode *node = l->treeNode;
261       while (l != NULL) {
262
263         LineList lines = l->lines;
264         fprintf(listing,"%-11s", l->name);
265
266         switch (node->nodekind) {
267           case DeclK:
268           switch (node->kind.decl) {
269             case FunK:
270             fprintf(listing, "Funcao           ");
271             break;
272             case VarK:
273             fprintf(listing, "Variavel         ");
274             break;
275             case ArrVarK:
276             fprintf(listing, "Vetor            ");
277             break;
278             case ParamK:
279             fprintf(listing, "Parametro        ");
280             break;
281             case ArrParamK:
282             fprintf(listing, "Parametro Vetor  ");
283             break;
284             default:
285             break;
286           }
287           break;
288           default:
289           break;
290         }
291
292         switch (l->type) {
293           case Void:
294           fprintf(listing, "Void             ");
295           break;
296           case Integer:
297           fprintf(listing, "Integer          ");
298           break;
299           case IntegerArray:
300           fprintf(listing, "Vetor de Integers ");
301           break;
302           default:
303           break;
304         }
305
306         // print memory location
307           fprintf(listing, "%d     ", l->memloc);
308
309         // print line numbers
310         while (lines->next != NULL) {
311           fprintf(listing, "%d, ", lines->lineno);
312           lines = lines->next;
313         }
314         fprintf(listing, "%d\n", lines->lineno);
315         l = l->next;
316       }
```

```
317     }
318   }
319 }
320 void printSymTab(FILE * listing) {
321
322   fprintf(listing, "\n---------------------\n");
323   fprintf(listing, "|  Tabela de simbolos  |");
324   fprintf(listing, "\n---------------------\n\n");
325
326   for (int i = 0; i<sizeOfList; ++i) {
327     Scope scope = scopeList[i];
328     if (scope->nestedLevel > 0) continue;
329     fprintf(listing, "Escopo : %s\n", scope->name);
330     fprintf(listing, "
          ---------------------------------------------------------------------------\n");
331     fprintf(listing, "Nome       Tipo             Tipo de Dado     loc    Numero das
          linhas \n");
332     fprintf(listing, "---------  ---------------  ------------     ---
          ------------------\n");
333     printSymTabRows(scope);
334     fprintf(listing, "Qdade elementos: %d, qtdade memoria: %d\n",scope->qtdeElementos,
          scope->posMem);
335     fprintf(listing, "
          ---------------------------------------------------------------------------\n");
336     fputc('\n', listing);
337   }
338 } /* printSymTab */
339
340
341 /* Scope functions */
342
343 Scope newScope(char * scopeName, ExpType type)
344 {
345   Scope newScope = (Scope) malloc(sizeof(struct ScopeListRec));
346   newScope->name = scopeName;
347   newScope->type = type;
348   newScope->qtdeElementos = 0;
349   newScope->posMem = 0;
350   return newScope;
351 }
352
353 void popScope(void)
354 {
355   scopeStack[topScope--] = NULL;
356 }
357
358 void pushScope(Scope scope)
359 {
360   for (int i=0; i<sizeOfList; i++)
361   {
362     if (strcmp(scopeList[i]->name, scope->name) == 0) {
363       scope->nestedLevel++;
364     }
365   }
366   scopeStack[topScope++] = scope;
367   insertScopeToList(scope);
368 }
369
370 void insertScopeToList(Scope scope)
371 {
```

```
372   scopeList[sizeOfList++] = scope;
373 }
374
375 Scope currScope()
376 {
377   return scopeStack[topScope-1];
378 }
379 /*End of scope functions*/
```

APÊNDICE H – Código-fonte codegenerate.c

```h
#ifndef _CODEGENERATE_H_
#define _CODEGENERATE_H_

#include "globals.h"

typedef enum ausente_LEX {GOTO, LAB, ASSIGN, STORE, LOADI, LOAD, FUN, VAR,
             VAR_VET, SOMi, PAR_VET, PAR, CALL, FUN_END, SUBi,
             AND, OR, MOD, XOR, NOT, SLL, SRL, CLKADJ, BEQ,
             BNE, LUI, SET} op;

void printNode(TreeNode *tree);

char * printType(TreeNode *tree);

void printCode(TreeNode *tree);

#endif
```

```c
#include <stdio.h>
#include <string.h>
#include "globals.h"
#include "symboltable.h"
#include "util.h"
#include "codegenerate.h"
#include "assemblygenerate.h"

FILE *intermed;

int isFirst = 0;

int label=0;
int parametro_on=0;
int parametros=0;
int reg_status[MAX_REG-1];


int proxRegLivre(){
    int i;
    for(i=1;i<=MAX_REG;i++){
        if(reg_status[i]==0){
            reg_status[i]=1;
            return i;
        }
    }
}

void insereMain(){

    Scope globalScope = st_lookup_scope("global");

 //   fprintf(intermed,"(SOMi, 0, sp, %d)\n", globalScope->qtdeElementos);  //inicia sp
```

```
34      releaseQuadList(SOMi, 0, sp, globalScope->qtdeElementos, NULL);
35
36    //  fprintf(intermed,"(SOMi, 0, hp, %d)\n", MAX_MEM);  //inicia hp
37      releaseQuadList(SOMi, 0, hp, MAX_MEM, NULL);
38
39    //  fprintf(intermed,"(SOMi, sp, fp, 0)\n"); // atualiza fp com o valor de st
40      releaseQuadList(SOMi, sp, fp, 0, NULL);
41
42    //  fprintf(intermed,"(SOMi, 0, sp, 1)\n");  //incrementa sp
43      releaseQuadList(SOMi, 0, sp, 1, NULL);
44
45      Scope mainScope = st_lookup_scope("main");
46
47    //  fprintf(intermed,"(SOMi, 0, sp, %d)\n", mainScope->qtdeElementos); //incrementa sp
        com parametros e variaveis do registro de ativac o
48      releaseQuadList(SOMi, 0, sp, mainScope->qtdeElementos, NULL);
49
50  }
51
52  char * printType(TreeNode* tree) {
53    if (tree->child[0] != NULL) {
54      switch (tree->child[0]->type) {
55        case Integer:
56          return "int";
57          break;
58        case Void:
59          return "void";
60          break;
61        case IntegerArray:
62          return "int array";
63          break;
64        default:
65          break;
66      }
67    } else {
68      switch (tree->type) {
69        case Integer:
70          return "int";
71          break;
72        case Void:
73          return "void";
74          break;
75        case IntegerArray:
76          return "int array";
77          break;
78        default:
79          break;
80      }
81    }
82  }
83
84  void printNode(TreeNode * t){
85      int loc_label;
86      TreeNode *aux;
87
88      switch(t->nodekind){
89          case StmtK:
90              switch(t->kind.stmt)
91              {
92                  case IfK:
```

```
93                      label=label+2;
94                      loc_label=label;
95                      if(t->child[0]!=NULL)
96                          printCode(t->child[0]);
97                      char *token = printToken(2, t->attr.type,0);
98
99                      fprintf(intermed,"(IF, _t%d, 0, L%d)\n", t->child[0]->reg,loc_label
                            -2);
100                     releaseQuadList(IF,t->child[0]->reg, 0, loc_label-2, NULL);
101
102                     if(t->child[2]!=NULL)
103                         printCode(t->child[2]);
104
105                     fprintf(intermed,"(GOTO, -, -, L%d)\n",loc_label-1);
106                     releaseQuadList(GOTO,0,0,loc_label-1, "lab");
107
108                     fprintf(intermed,"(LAB, -, -, %d)\n",loc_label-2);
109                     releaseQuadList(LAB,0,0,loc_label-2, "lab");
110
111                     if(t->child[1]!=NULL)
112                         printCode(t->child[1]);
113
114                     fprintf(intermed,"(GOTO, -, -, L%d)\n",loc_label-1);
115                     releaseQuadList(GOTO,0,0,loc_label-1, "lab");
116
117                     fprintf(intermed,"(LAB, -, -, %d)\n",loc_label-1);
118                     releaseQuadList(LAB,0,0,loc_label-1, "lab");
119
120                     if(t->child[0]!=NULL)
121                         reg_status[t->child[0]->reg]=0;
122
123
124                 break;
125                 case WhileK:
126                     label=label+3;
127                     loc_label=label;
128
129                     fprintf(intermed,"(LAB, -, -, L%d)\n",loc_label-3);
130                     releaseQuadList(LAB,0,0,loc_label-3, "lab");
131
132                     if(t->child[0]!=NULL)
133                         printCode(t->child[0]);
134
135                     fprintf(intermed,"(WHILE, _t%d, zero -, L%d)\n",t->child[0]->reg,
                            loc_label-2);
136                     releaseQuadList(WHILE,t->child[0]->reg, 0, loc_label-2, NULL);
137
138                     fprintf(intermed,"(GOTO, -, -, L%d)\n",loc_label-1);
139                     releaseQuadList(GOTO,0,0,loc_label-1, "lab");
140
141                     fprintf(intermed,"(LAB, -, -, %d)\n",loc_label-2);
142                     releaseQuadList(LAB,0,0,loc_label-2, "lab");
143
144                     if(t->child[1]!=NULL)
145                         printCode(t->child[1]);
146
147                     fprintf(intermed,"(LAB, -, -, %d)\n",loc_label-1);
148                     releaseQuadList(LAB,0,0,loc_label-1, "lab");
149
150                     if(t->child[0]!=NULL)
```

```
151                         reg_status[t->child[0]->reg]=0;
152
153
154                 break;
155                 case AssignK:
156                     if(t->child[0]!=NULL)
157                         printCode(t->child[0]);
158                     if(t->child[1]!=NULL)
159                         printCode(t->child[1]);
160
161                     if((t->child[1]->nodekind) == ExpK)
162                         if(t->child[1]->kind.exp == CallK){
163                             fprintf(intermed,"(ASSIGN, rv, -, _t%d)\n", t->child[0]->reg)
                                    ;
164                             releaseQuadList(ASSIGN, rv, 0, t->child[0]->reg, NULL);
165
166                         }else{
167                             fprintf(intermed,"(ASSIGN, _t%d, -, _t%d)\n", t->child[1]->
                                    reg, t->child[0]->reg);
168                             releaseQuadList(ASSIGN, t->child[1]->reg, 0, t->child[0]->reg
                                    , NULL);
169
170                         }
171                         // verificar se e global
172                         if(st_isGlobal(t->child[0]->attr.name,t->scopeName)==1){
173                         //    fprintf(intermed,"(STORE, _t%d, gp, %d)\n", t->child[0]->
                                reg, st_lookup_memPos(t->child[0]->attr.name,t->child[0]->
                                scopeName));
174                             releaseQuadList(STORE, t->child[0]->reg, fp, st_lookup_memPos
                                    (t->child[0]->attr.name,t->child[0]->scopeName), NULL);
175                         }else{
176                         //    fprintf(intermed,"(STORE, _t%d, fp, %d)\n", t->child[0]->
                                reg, st_lookup_memPos(t->child[0]->attr.name,t->child[0]->
                                scopeName));
177                             releaseQuadList(STORE, t->child[0]->reg, fp, st_lookup_memPos
                                    (t->child[0]->attr.name,t->child[0]->scopeName), NULL);
178                         }
179
180                     if(t->child[0]!=NULL)
181                         reg_status[t->child[0]->reg]=0;
182                     if(t->child[1]!=NULL)
183                         reg_status[t->child[1]->reg]=0;
184
185
186                 break;
187                 case CompoundK:
188                     if(t->child[0]!=NULL)
189                         printCode(t->child[0]);
190                     if(t->child[1]!=NULL)
191                         printCode(t->child[1]);
192
193
194                 break;
195                 case ReturnK:
196                     if(t->child[0]==NULL)
197                     {
198                     }else
199                     {
200                         printCode(t->child[0]);
201
```

```
202                         fprintf(intermed,"(RETURN, _t%d, -, rv)\n", t->child[0]->reg);
203                         releaseQuadList(RETURN, t->child[0]->reg,0, rv, NULL);
204 
205                     //    fprintf(intermed,"(GOTO, -,- , %s)\n", t->scopeName);
206                         releaseQuadList(GOTO,0,0,-1,t->scopeName);
207 
208                         reg_status[t->child[0]->reg]=0;
209                     }
210 
211 
212                 break;
213                 default:
214                 break;
215             }
216         break;
217         case ExpK:
218             switch(t->kind.exp)
219             {
220                 case OpK:
221                 break;
222                 case ConstK:
223                     t->reg=proxRegLivre();
224 
225                     fprintf(intermed,"(LOADI, - , _t%d, %d)\n",t->reg, t->attr.val);
226                     releaseQuadList(LOADI,0,t->reg,t->attr.val, NULL);
227 
228                 break;
229                 case IdK:
230                     t->reg=proxRegLivre();
231 
232                     if(st_isGlobal(t->attr.name,t->scopeName) == 1){
233                         fprintf(intermed,"(LOAD, %s, _t%d, %d)\n",t->attr.name, t->reg, (
                                st_lookup_memPos(t->attr.name,"global")-1));
234                         releaseQuadList(LOAD, 0, t->reg, st_lookup_memPos(t->attr.name,t
                                ->scopeName), NULL);
235                     }
236                     else{
237                         fprintf(intermed,"(LOAD, %s, _t%d, %d)\n",t->attr.name, t->reg,
                                st_lookup_memPos(t->attr.name,t->scopeName));
238                         releaseQuadList(LOAD, fp, t->reg, st_lookup_memPos(t->attr.name,t
                                ->scopeName), NULL);
239                     }
240 
241 
242 
243                 break;
244                 case TypeK:
245 
246                 break;
247                 case ArrIdK:
248                     if(t->child[0]!=NULL)
249                         printCode(t->child[0]);
250 
251                     // refazer o calculo de indice considerando heap
252                     int heapExactPos = proxRegLivre();
253                     int heapBeginPos = proxRegLivre();
254 
255                     if(st_isGlobal(t->attr.name,t->scopeName)==1){
256 
257                         fprintf(intermed,"(LOAD, 0, _t%d, %d)\n", heapBeginPos, (
```

```
                        st_lookup_memPos(t->attr.name, "global")-1));
258                     releaseQuadList(LOAD,0,heapBeginPos,(st_lookup_memPos(t->attr.
                        name, "global")-1),NULL);
259
260                     fprintf(intermed,"(SOM, _t%d, _t%d, _t%d)\n", heapBeginPos, t->
                        child[0]->reg, heapExactPos);
261                     releaseQuadList(SOM,heapBeginPos, t->child[0]->reg, heapExactPos,
                         NULL);
262
263                     fprintf(intermed,"(LOAD, %s[_t%d], _t%d, 0)\n", t->attr.name, t->
                        child[0]->reg, t->reg);
264                     releaseQuadList(LOAD,heapExactPos,t->reg,0,NULL);
265
266
267                 }else{
268                     fprintf(intermed,"(LOAD, fp, _t%d, %d)\n", heapBeginPos,
                        st_lookup_memPos(t->attr.name, t->scopeName));
269                     releaseQuadList(LOAD,fp,heapBeginPos,st_lookup_memPos(t->attr.
                        name, t->scopeName),NULL);
270
271                     fprintf(intermed,"(SOM, _t%d, _t%d, _t%d)\n", heapBeginPos, t->
                        child[0]->reg, heapExactPos);
272                     releaseQuadList(SOM,heapBeginPos, t->child[0]->reg, heapExactPos,
                         NULL);
273
274                     fprintf(intermed,"(LOAD, %s[_t%d], _t%d, 0)\n", t->attr.name, t->
                        child[0]->reg, t->reg);
275                     releaseQuadList(LOAD,heapExactPos,t->reg,0,NULL);
276                 }
277
278                 if(t->child[0]!=NULL)
279                     reg_status[t->child[0]->reg]=0;
280
281                 reg_status[heapExactPos]=0;
282                 reg_status[heapBeginPos]=0;
283
284             break;
285             case CallK:
286                 if(t->child[0]!=NULL)
287                 {
288                     parametro_on=1;
289                     printCode(t->child[0]);
290                     parametros++;
291                 }
292
293                 aux=t->child[0];
294
295
296              //   fprintf(intermed,"(STORE, fp, sp, 0)\n");  //armazena posic o fp
                     em fp_velho
297                 releaseQuadList(STORE,fp,sp,0, NULL);
298
299              //   fprintf(intermed,"(SOMi, 0, sp, 1)\n"); //incrementa sp
300                 releaseQuadList(SOMi, 0,sp, 1, NULL);
301
302     //          fprintf(intermed,"(STORE, 0, sp, 0)\n");  //armazena a linha da
            instruc o de retorno
303
304              //   fprintf(intermed,"(SOMi, sp, fp, 0)\n"); // atualiza fp com o valor
                    de st
```

```
305                     releaseQuadList(SOMi, sp, fp, 0, NULL);
306
307                 //   fprintf(intermed,"(SOMi, 0, sp, 1)\n");  //incrementa sp
308                     releaseQuadList(SOMi, 0, sp, 1, NULL);
309
310                     Scope newScope = st_lookup_scope(t->scopeName);
311
312                 //   fprintf(intermed,"(SOMi, 0, sp, %d)\n", newScope->qtdeElementos);
                         //incrementa sp com parametros e variaveis do registro de ativac o
313                     releaseQuadList(SOMi, 0, sp, newScope->qtdeElementos, NULL);
314
315                     int i = 1;
316                     while(aux!=NULL){
317                         fprintf(intermed,"(PAR, -, -, _t%d)\n", aux->reg);
318
319                 //        fprintf(intermed,"(STORE, _t%d, fp, %d)\n", aux->reg, i);
320                         releaseQuadList(STORE, aux->reg,fp,i, NULL);
321
322                         aux=aux->sibling;
323                         i++;
324                     }
325
326                     t->reg=proxRegLivre();
327                     fprintf(intermed,"(CALL, %s, %d, _t%d)\n",t->attr.name, parametros, t
                         ->reg);
328                     releaseQuadList(CALL,t->reg,t->reg,t->reg,t->attr.name);
329
330                     parametros = 0;
331
332                     aux=t->child[0];
333                     while(aux!=NULL){
334                         reg_status[aux->reg]=0;
335                         aux=aux->sibling;
336                     }
337
338
339                 break;
340                 case CalcK:
341                     if(t->child[0]!=NULL)
342                         printCode(t->child[0]);
343                     if(t->child[2]!=NULL)
344                         printCode(t->child[2]);
345
346                     t->reg=proxRegLivre();
347                     char *token = printToken(2,t->child[1]->attr.op, NULL);
348                     if(t->child[1]->attr.op == IME){
349                         int a = proxRegLivre();
350                         int b = proxRegLivre();
351
352                         fprintf(intermed,"(%s, _t%d, _t%d, _t%d)\n", "MEN" ,t->child[0]->
                             reg,t->child[2]->reg, a);
353                         releaseQuadList(MEN, t->child[0]->reg,t->child[2]->reg, a, NULL);
354
355                         fprintf(intermed,"(%s, _t%d, _t%d, _t%d)\n", "IGL" ,t->child[0]->
                             reg,t->child[2]->reg, b);
356                         releaseQuadList(IGL, t->child[0]->reg,t->child[2]->reg, b, NULL);
357
358                         fprintf(intermed,"(%s, _t%d, _t%d, _t%d)\n", "OR" ,a,b, t->reg);
359                         releaseQuadList(OR, t->child[0]->reg,t->child[2]->reg, t->reg,
                             NULL);
```

```
360
361                     reg_status[a]=0;
362                     reg_status[b]=0;
363
364                 }else if(t->child[1]->attr.op == IMA){
365                     int a = proxRegLivre();
366                     int b = proxRegLivre();
367
368                     fprintf(intermed,"(%s, _t%d, _t%d, _t%d)\n", "MAI" ,t->child[0]->
                            reg,t->child[2]->reg, a);
369                     releaseQuadList(MAI, t->child[0]->reg,t->child[2]->reg, a, NULL);
370
371                     fprintf(intermed,"(%s, _t%d, _t%d, _t%d)\n", "IGL" ,t->child[0]->
                            reg,t->child[2]->reg, b);
372                     releaseQuadList(IGL, t->child[0]->reg,t->child[2]->reg, b, NULL);
373
374                     fprintf(intermed,"(%s, _t%d, _t%d, _t%d)\n", "OR" ,a,b, t->reg);
375                     releaseQuadList(OR, t->child[0]->reg,t->child[2]->reg, t->reg,
                            NULL);
376
377                     reg_status[a]=0;
378                     reg_status[b]=0;
379
380                 }else if(t->child[1]->attr.op == DIF){
381                     int a = proxRegLivre();
382
383                     fprintf(intermed,"(%s, _t%d, _t%d, _t%d)\n", "SET" , t->child
                            [0]->reg, t->child[2]->reg, a);
384                     releaseQuadList(SET, t->child[0]->reg,t->child[2]->reg, a, NULL);
385
386                     fprintf(intermed,"(%s, _t%d, 0, _t%d)\n", "NOT" , a, t->reg);
387                     releaseQuadList(NOT, a, 0, t->reg, NULL);
388
389                     reg_status[a]=0;
390                 }else{
391                 fprintf(intermed,"(%s, _t%d, _t%d, _t%d)\n", token ,t->child[0]->reg,
                        t->child[2]->reg, t->reg);
392                 releaseQuadList(t->child[1]->attr.op, t->child[0]->reg,t->child[2]->
                        reg, t->reg, NULL);
393                 }
394
395                 if(t->child[0]!=NULL)
396                     reg_status[t->child[0]->reg]=0;
397                 if(t->child[2]!=NULL)
398                     reg_status[t->child[2]->reg]=0;
399
400
401             break;
402             default:
403             break;
404         }
405     break;
406     case DeclK:
407         switch(t->kind.stmt)
408         {
409             case VarK:
410                 fprintf(intermed,"(VAR, %s, -, %s)\n",t->attr.name, printType(t));
411
412
413             break;
```

```
414                 case FunK:
415                     fprintf(intermed,"(FUN, %s, -, %s)\n",t->attr.name, printType(t));
416                     releaseQuadList(FUN, 0, 0, 0, t->attr.name);
417 
418                 //    fprintf(intermed,"(STORE, ra, fp, 0)\n"); //armazena o endereco de
                        retorno em fp
419                     releaseQuadList(STORE, 31, fp, 0, NULL);
420 
421                     if(t->child[1]!=NULL)
422                         printCode(t->child[1]);
423                     if(t->child[2]!=NULL)
424                         printCode(t->child[2]);
425 
426 
427                 /* //    fprintf(intermed, "(LOAD, fp, %d, 0)\n", t->reg); //carrega
                        posic o da instruc o de retorno
428                     releaseQuadList(LOAD,fp, ra,0, NULL);
429 
430                 //    fprintf(intermed, "(LOAD, fp, fp, -1)\n"); //fp recebe valor fp
                        velho
431                     releaseQuadList(LOAD, fp, fp, -1, NULL);
432 
433                 //    fprintf(intermed,"(SUBi, sp, sp, %d)\n", (st_lookup_scope(t->
                        scopeName)->qtdeElementos+2));
434                     releaseQuadList(SUBi, sp, sp, (st_lookup_scope(t->scopeName)->
                            qtdeElementos+2), NULL); */
435 
436                 //    fprintf(intermed,"(GOTO,0,0,%d)\n",t->reg); //jump register para o
                        endereco de retorno (antigo ra)
437                     releaseQuadList(FUN_END, 0,(st_lookup_scope(t->scopeName)->
                            qtdeElementos+2),ra, t->attr.name);
438 
439 
440                 break;
441                 case ArrVarK: // alocar espaco no heap e salvar a posic o do heap no
                        registro de ativac o
442                     fprintf(intermed,"(VAR_VET, *%s, %d, %s) \n", t->attr.arr.name,t->
                            attr.arr.size, printType(t));
443 
444                     if(st_isGlobal(t->attr.arr.name,t->scopeName)==1){
445                  //       fprintf(intermed,"(SUBi, hp, hp, %d)\n", t->attr.arr.size);
446                         releaseQuadList(SUBi, hp,hp,t->attr.arr.size, NULL);
447 
448                  //       fprintf(intermed,"(LOAD, hp, 0, %d)\n", st_lookup_memPos(t->
                             attr.arr.name, "global")-1);
449                         releaseQuadList(LOAD, hp, 0, (st_lookup_memPos(t->attr.arr.name,
                                "global")-1),NULL);
450 
451                     }else{
452                 //        fprintf(intermed,"(SUBi, hp, hp, %d)\n", t->attr.arr.size);
453                         releaseQuadList(SUBi, hp,hp,t->attr.arr.size, NULL);
454 
455                 //        fprintf(intermed,"(LOAD, hp, fp, %d)\n", st_lookup_memPos(t->
                        attr.arr.name, t->scopeName));
456                         releaseQuadList(LOAD, hp, fp, st_lookup_memPos(t->attr.arr.name,
                                "global"),NULL);
457                     }
458 
459                 break;
460                 case ArrParamK:
```

```
461                     fprintf(intermed,"(PAR_VET, * %s, -, %s)\n", t->attr.name, printType(
                            t));
462
463
464                 break;
465                 case ParamK:
466                     if(t->attr.name!=NULL)
467                     fprintf(intermed,"(PAR, %s, -, %s)\n", t->attr.name, printType(t));
468
469
470                 break;
471                 default:
472                 break;
473             }
474         break;
475         default:
476         break;
477     }
478 }
479
480 void printCode(TreeNode *t){
481     TreeNode *irmaos;
482     switch(t->nodekind){
483         case StmtK:
484             switch(t->kind.stmt)
485             {
486                 case IfK:
487                     printNode(t);
488
489                     irmaos=t->sibling;
490                     while(irmaos!=NULL){
491                         if(parametro_on)
492                             parametros++;
493                         printNode(irmaos);
494                         irmaos=irmaos->sibling;
495                     }
496                     parametro_on=0;
497
498                 break;
499                 case WhileK:
500                     printNode(t);
501
502                     irmaos=t->sibling;
503                     while(irmaos!=NULL){
504                         if(parametro_on)
505                             parametros++;
506                         printNode(irmaos);
507                         irmaos=irmaos->sibling;
508                     }
509                     parametro_on=0;
510
511                 break;
512                 case AssignK:
513                     printNode(t);
514
515                     irmaos=t->sibling;
516                     while(irmaos!=NULL){
517                         if(parametro_on)
518                             parametros++;
519                         printNode(irmaos);
```

```
520                         irmaos=irmaos->sibling;
521                     }
522                     parametro_on=0;
523 
524                 break;
525                 case CompoundK:
526                     printNode(t);
527 
528                     irmaos=t->sibling;
529                     while(irmaos!=NULL){
530                         if(parametro_on)
531                             parametros++;
532                         printNode(irmaos);
533                         irmaos=irmaos->sibling;
534                     }
535                     parametro_on=0;
536 
537                 break;
538                 case ReturnK:
539                     printNode(t);
540 
541                     irmaos=t->sibling;
542                     while(irmaos!=NULL){
543                         if(parametro_on)
544                             parametros++;
545                         printNode(irmaos);
546                         irmaos=irmaos->sibling;
547                     }
548                     parametro_on=0;
549 
550                 break;
551                 default:
552                 break;
553             }
554         break;
555         case ExpK:
556             switch(t->kind.exp)
557             {
558                 case OpK:
559                     //printNode(t);
560 
561                 break;
562                 case ConstK:
563                     printNode(t);
564 
565                     irmaos=t->sibling;
566                     while(irmaos!=NULL){
567                         if(parametro_on)
568                             parametros++;
569                         printNode(irmaos);
570                         irmaos=irmaos->sibling;
571                     }
572                     parametro_on=0;
573 
574                 break;
575                 case IdK:
576                     printNode(t);
577 
578                     irmaos=t->sibling;
579                     while(irmaos!=NULL){
```

```
580                         if(parametro_on)
581                             parametros++;
582                         printNode(irmaos);
583                         irmaos=irmaos->sibling;
584                     }
585                     parametro_on=0;
586 
587                 break;
588                 case TypeK:
589                     //printNode(t);
590 
591                 break;
592                 case ArrIdK:
593                     printNode(t);
594 
595                     irmaos=t->sibling;
596                     while(irmaos!=NULL){
597                         if(parametro_on)
598                             parametros++;
599                         printNode(irmaos);
600                         irmaos=irmaos->sibling;
601                     }
602                     parametro_on=0;
603 
604                 break;
605                 case CallK:
606                     printNode(t);
607 
608                     irmaos=t->sibling;
609                     while(irmaos!=NULL){
610                         if(parametro_on)
611                             parametros++;
612                         printNode(irmaos);
613                         irmaos=irmaos->sibling;
614                     }
615                     parametro_on=0;
616 
617                 break;
618                 case CalcK:
619                     printNode(t);
620 
621                     irmaos=t->sibling;
622                     while(irmaos!=NULL){
623                         if(parametro_on)
624                             parametros++;
625                         printNode(irmaos);
626                         irmaos=irmaos->sibling;
627                     }
628                     parametro_on=0;
629 
630                 break;
631                 default:
632                 break;
633             }
634         break;
635         case DeclK:
636             switch(t->kind.stmt)
637             {
638                 case VarK:
639                     if(isFirst == 0)
```

```
640                         insereMain();
641                     printNode(t);
642
643                     if(isFirst == 0){
644                     TreeNode *new;
645
646                     for(new = t; new->sibling != NULL; new = new->sibling){
647                          while((new->sibling!=NULL)&&((new->sibling->kind.stmt == VarK)
                                 ||(new->sibling->kind.stmt == ArrVarK))){
648                             printNode(new->sibling);
649                             new->sibling = new->sibling->sibling;
650                         }
651                     }
652                       //  fprintf(intermed,"(GOTO, -, -, main)\n");
653                         releaseQuadList(GOTO, 0, 0, 0, "main");
654                         isFirst = 1;
655                     }
656
657                     irmaos=t->sibling;
658                     while(irmaos!=NULL){
659                         if(parametro_on)
660                             parametros++;
661                         printNode(irmaos);
662                         irmaos=irmaos->sibling;
663                     }
664                     parametro_on=0;
665
666                 break;
667                 case FunK:
668
669                     if(isFirst == 0){
670                     TreeNode *new;
671
672                     for(new = t; new->sibling != NULL; new = new->sibling){
673                          while((new->sibling!=NULL)&&((new->sibling->kind.stmt == VarK)
                                 ||(new->sibling->kind.stmt == ArrVarK))){
674                             printNode(new->sibling);
675                             new->sibling = new->sibling->sibling;
676                         }
677                     }
678                        if(isFirst == 0)
679                             insereMain();
680                     //    fprintf(intermed,"(GOTO, -, -, main)\n");
681                         releaseQuadList(GOTO, 0, 0, 0, "main");
682                         isFirst = 1;
683                     }
684
685                     printNode(t);
686
687                     irmaos=t->sibling;
688                     while(irmaos!=NULL){
689                         if(parametro_on)
690                             parametros++;
691                         printNode(irmaos);
692                         irmaos=irmaos->sibling;
693                     }
694                     parametro_on=0;
695
696                 break;
697                 case ArrVarK:
```

```
698                     if(isFirst == 0)
699                         insereMain();
700                     printNode(t);
701 
702                     if(isFirst == 0){
703                     TreeNode *new;
704 
705                     for(new = t; new->sibling != NULL; new = new->sibling){
706                         while((new->sibling!=NULL)&&((new->sibling->kind.stmt == VarK)||(
                                new->sibling->kind.stmt == ArrVarK))){
707                             printNode(new->sibling);
708                             new->sibling = new->sibling->sibling;
709                         }
710                     }
711 
712                      //   fprintf(intermed,"(GOTO, -, -, main)\n");
713                         releaseQuadList(GOTO, 0, 0, 0, "main");
714                         isFirst = 1;
715                     }
716 
717                     irmaos=t->sibling;
718                     while(irmaos!=NULL){
719                         if(parametro_on)
720                             parametros++;
721                         printNode(irmaos);
722                         irmaos=irmaos->sibling;
723                     }
724                     parametro_on=0;
725 
726                 break;
727                 case ArrParamK:
728                     printNode(t);
729 
730                     irmaos=t->sibling;
731                     while(irmaos!=NULL){
732                         if(parametro_on)
733                             parametros++;
734                         printNode(irmaos);
735                         irmaos=irmaos->sibling;
736                     }
737                     parametro_on=0;
738 
739                 break;
740                 case ParamK:
741                     printNode(t);
742 
743                     irmaos=t->sibling;
744                     while(irmaos!=NULL){
745                         if(parametro_on)
746                             parametros++;
747                         printNode(irmaos);
748                         irmaos=irmaos->sibling;
749                     }
750                     parametro_on=0;
751 
752                 break;
753                 default:
754                 break;
755             }
756         break;
```

```
757             default:
758             break;
759         }
760 }
```

APÊNDICE I – código-fonte assembly.c

```
#ifndef _ASSEMBLYGENERATE_H_
#define _ASSEMBLYGENERATE_H_

//define os nomes das operacoes em assembly
typedef enum tipoR {in, out, add, sub, mul, divi, slt, sgt, set,
            jr, and, or, mod, xor, not, move, sll, srl}operation;

typedef enum tipoIeJ{r, nop, j, jal, load, store, addi, subi, beq,
                    bne, loadi, lui, lab, fun_b, fun_f}operation2;

#define ra 31
#define fp 30
#define sp 29
#define hp 28
#define rv 27
#define zero 0

//define os tipos de instrucoes
typedef enum{R, I, J}instructType;

//define estrutura da instruc o tipo R
typedef struct
{
    operation op;
    int rs;
    int rt;
    int rd;
    int shamt;
    int func;
}rType;

//define estrutura da instruc o tipo I
typedef struct
{
    operation2 op;
    int rs;
    int rt;
    int immediate;
}iType;

//define estrutura da instruc o tipo J
typedef struct
{
    operation2 op;
    int target;
    char *name;
}jType;

//define estrutura da quadrupla
typedef struct quadModel
{
    instructType instrType;
    union
    {
```

```
55          rType quadR;
56          iType quadI;
57          jType quadJ;
58      }quadType;
59      struct quadModel *proxQuad;
60      int lineNo;
61  }quadModel;
62
63  //define estrutura do no da lista de labels
64  typedef struct labelNode
65  {
66      int labelName;
67      int lineNo;
68      struct labelNode *proxLabel;
69  }labelNode;
70
71  //define estrutura do no da lista de funcoes
72  typedef struct funNode
73  {
74      char* funName;
75      int lineNo;
76      int endlineNo;
77      struct funNode *proxFun;
78  }funNode;
79
80  //armazena inicio e fim da lista de quadruplas
81  quadModel *beginQuadList;
82  quadModel *endQuadList;
83
84  //armazena inicio e fim da lista de linhas de labels
85  labelNode *beginLabelList;
86  labelNode *endLabelList;
87
88  //armazena inicio e fim da lista de linhas de funcoes
89  funNode *beginFunList;
90  funNode *endFunList;
91
92  void createQuadR(int op, int rs, int rt, int rd, int shamt, int funct);
93  void createQuadI(int op, int rs, int rt, int immediate);
94  void createQuadJ(int op, int value, char*name);
95
96  void releaseLabel(int label);
97  int searchLabel(int label);
98
99  void releaseFun(char* fun);
100 void releaseFunEnd(char* fun);
101 int searchFun(char* fun);
102 int searchFunEnd(char* fun);
103
104 void releaseQuadList(int op, int t1, int t2, int t3, char *name);
105
106 void printQuadR(char* op, int rs, int rt, int rd, int shamt, int funct);
107 void printQuadI(char* op, int rs, int rt, int immediate);
108 void printQuadJ(char* op, char* target);
109
110 #endif // ASSEMBLYGENERATE_H
```

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
```

```
4  #include "globals.h"
5  #include "util.h"
6  #include "codegenerate.h"
7  #include "assemblygenerate.h"
8
9  FILE *assembly;
10 int currLine = 0;
11
12 //func o de criac o de quadrupla do tipo R
13 void createQuadR(int op, int rs, int rt, int rd, int shamt, int funct){
14     quadModel *newQuad = (quadModel*)calloc(1, sizeof(quadModel));
15     newQuad->instrType = R;
16     newQuad->lineNo = currLine;
17     newQuad->quadType.quadR.op = op;
18     newQuad->quadType.quadR.rs = rs;
19     newQuad->quadType.quadR.rs = rt;
20     newQuad->quadType.quadR.rs = rd;
21     newQuad->quadType.quadR.shamt = shamt;
22     newQuad->quadType.quadR.func = funct;
23
24     if(beginQuadList == NULL)
25     {
26         beginQuadList = newQuad;
27         endQuadList = newQuad;
28         endQuadList->proxQuad = NULL;
29     }
30     else
31     {
32         endQuadList->proxQuad = newQuad;
33         endQuadList = newQuad;
34     }
35     currLine++;
36 }
37
38 //func o de criac o de quadrupla do tipo I
39 void createQuadI(int op, int rs, int rt, int immediate){
40     quadModel *newQuad = (quadModel*)calloc(1, sizeof(quadModel));
41     newQuad->instrType = I;
42     newQuad->lineNo = currLine;
43     newQuad->quadType.quadI.op = op;
44     newQuad->quadType.quadI.rs = rs;
45     newQuad->quadType.quadI.rt = rt;
46     newQuad->quadType.quadI.immediate = immediate;
47
48     if(beginQuadList == NULL)
49     {
50         beginQuadList = newQuad;
51         endQuadList = newQuad;
52         endQuadList->proxQuad = NULL;
53     }
54     else
55     {
56         endQuadList->proxQuad = newQuad;
57         endQuadList = newQuad;
58     }
59     currLine++;
60 }
61
62 //func o de criac o de quadrupla do tipo J
63 void createQuadJ(int op, int value, char* name){
```

```
64      quadModel *newQuad = (quadModel*)calloc(1, sizeof(quadModel));
65      newQuad->instrType = J;
66      newQuad->lineNo = currLine;
67      newQuad->quadType.quadJ.op = op;
68      newQuad->quadType.quadJ.target = value;
69      if(name!=NULL)
70          newQuad->quadType.quadJ.name = name;
71
72      if(beginQuadList == NULL)
73      {
74          beginQuadList = newQuad;
75          endQuadList = newQuad;
76          endQuadList->proxQuad = NULL;
77      }
78      else
79      {
80          endQuadList->proxQuad = newQuad;
81          endQuadList = newQuad;
82      }
83      currLine++;
84  }
85
86  //func o que insere novo label
87  void releaseLabel(int label){
88      labelNode *newLabel = (labelNode*)calloc(1,sizeof(labelNode));
89
90      newLabel->labelName = label;
91      newLabel->lineNo = currLine;
92
93      if(beginLabelList == NULL)
94      {
95          beginLabelList = newLabel;
96          endLabelList = newLabel;
97          endLabelList->proxLabel = NULL;
98      }else
99      {
100         endLabelList->proxLabel = newLabel;
101         endLabelList = newLabel;
102     }
103 }
104
105 //func o que retorna a linha da label
106 int searchLabel(int label){
107     labelNode *searchLabel = beginLabelList;
108     while ((searchLabel!=NULL)&&(searchLabel->labelName!=label))
109     {
110         searchLabel = searchLabel->proxLabel;
111     }
112     if(searchLabel!=NULL)
113         return searchLabel->lineNo;
114     else
115         return (-1);
116 }
117
118 //func o que insere um novo no de func o
119 void releaseFun(char* fun){
120     funNode *newFun = (funNode*)calloc(1,sizeof(funNode));
121
122     newFun->funName = fun;
123     newFun->lineNo = currLine;
```

```
124     newFun->endlineNo = -1;
125 
126     if(beginFunList == NULL)
127     {
128         beginFunList = newFun;
129         endFunList = newFun;
130         endFunList->proxFun = NULL;
131     }else
132     {
133         endFunList->proxFun = newFun;
134         endFunList = newFun;
135     }
136 
137 }
138 
139 void releaseFunEnd(char* fun){
140     funNode *searchFun = beginFunList;
141     while ((searchFun!=NULL)&&(strcmp(fun, searchFun->funName)!=0))
142     {
143         searchFun = searchFun->proxFun;
144     }
145     if(searchFun !=NULL)
146         searchFun->endlineNo = currLine;
147 }
148 
149 //func o que retorna a linha da func o
150 int searchFun(char* fun){
151     funNode *searchFun = beginFunList;
152     while ((searchFun!=NULL)&&(strcmp(fun, searchFun->funName)!=0))
153     {
154         searchFun = searchFun->proxFun;
155     }
156     if(searchFun!=NULL)
157         return searchFun->lineNo;
158     else
159         return (-1);
160 }
161 
162 //func o que retorna a linha final da func o
163 int searchFunEnd(char* fun){
164     funNode *searchFun = beginFunList;
165     while ((searchFun!=NULL)&&(strcmp(fun, searchFun->funName)!=0))
166     {
167         searchFun = searchFun->proxFun;
168     }
169     if(searchFun!=NULL)
170         return searchFun->endlineNo;
171     else
172         return (-1);
173 }
174 
175 void printQuadR(char* op, int rs, int rt, int rd, int shamt, int funct)
176 {
177     fprintf(assembly,"(%s, %d, %d, %d, %d, %d)\n", op, rs, rt, rd, shamt, funct);
178 }
179 
180 void printQuadI(char* op, int rs, int rt, int immediate)
181 {
182     fprintf(assembly,"(%s, %d, %d, %d)\n", op, rs, rt, immediate);
183 }
```

```
184
185 void printQuadJ(char* op, char* target)
186 {
187     fprintf(assembly,"(%s, %s)\n", op, target);
188 }
189
190 void releaseQuadList(int op, int t1, int t2, int t3, char *name)
191 {
192     switch (op)
193     {
194     case IF:
195         createQuadR(bne,t1,t2,t3,0,0);
196         printQuadR("bne", t1,t2,t3,0,0);
197
198         break;
199
200     case GOTO:
201         if(name != NULL)
202         {
203             createQuadJ(j, t3, name);
204             printQuadJ("j",name);
205         }else
206         {
207             createQuadJ(j, t3, NULL);
208             printQuadJ("j",name);
209         }
210
211         break;
212
213     case LAB:
214         releaseLabel(t3);
215
216         break;
217
218     case WHILE:
219         createQuadR(bne,t1,t2,t3,0,0);
220         printQuadR("bne", t1,t2,t3,0,0);
221
222         break;
223
224     case ASSIGN:
225         createQuadR(move,t1,t2,t3,0,0);
226         printQuadR("move",t1,t2,t3,0,0);
227
228         break;
229
230     case STORE:
231         createQuadI(store, t1,t2,t3);
232         printQuadI("store", t1,t2,t3);
233
234         break;
235
236     case RETURN:
237         createQuadR(move,t1,t2,t3,0,0);
238         printQuadR("move",t1,t2,t3,0,0);
239
240         break;
241     case LOADI:
242         createQuadI(loadi,0,t2,t3);
243         printQuadI("loadi",0,t2,t3);
```

```
244
245         break;
246
247     case LOAD:
248         createQuadI(load,t1,t2,t3);
249         printQuadI("load",t1,t2,t3);
250
251         break;
252
253     case SOM:
254         createQuadR(add,t1,t2,t3,0,0);
255         printQuadR("add",t1,t2,t3,0,0);
256
257         break;
258
259     case CALL:
260         if(strcmp(name, "input")==0){
261             createQuadR(in,t1,t2,t3,0,0);
262             printQuadR("in",t1,t2,t3,0,0);
263         }else if(strcmp(name, "output")==0){
264             createQuadR(out,t1,t2,t3,0,0);
265             printQuadR("out",t1,t2,t3,0,0);
266         }else{
267             createQuadJ(jal,t3, name);
268             printQuadJ("jal",name);
269         }
270
271         break;
272
273     case MUL:
274         createQuadR(mul,t1,t2,t3,0,0);
275         printQuadR("mul",t1,t2,t3,0,0);
276
277         break;
278
279     case DIV:
280         createQuadR(divi,t1,t2,t3,0,0);
281         printQuadR("div",t1,t2,t3,0,0);
282
283         break;
284
285     case MEN:
286         createQuadR(slt,t1,t2,t3,0,0);
287         printQuadR("slt",t1,t2,t3,0,0);
288
289         break;
290
291     case MAI:
292         createQuadR(sgt,t1,t2,t3,0,0);
293         printQuadR("sgt",t1,t2,t3,0,0);
294
295         break;
296
297     case IGL:
298         createQuadR(set,t1,t2,t3,0,0);
299         printQuadR("set",t1,t2,t3,0,0);
300
301         break;
302
303     case FUN:
```

```
304         releaseFun(name);
305 
306         break;
307 
308     case FUN_END:
309         releaseFunEnd(name);
310         releaseQuadList(LOAD,fp, ra,0, NULL);
311         releaseQuadList(LOAD, fp, fp, -1, NULL);
312         releaseQuadList(SUBi, sp, sp, t2, NULL);
313         createQuadJ(jr,t3, name);
314         printQuadJ("jr", name);
315 
316         break;
317 
318     case SUBi:
319         createQuadI(subi,t1,t2,t3);
320         printQuadI("subi",t1,t2,t3);
321 
322         break;
323 
324     case AND:
325         createQuadR(and,t1,t2,t3,0,0);
326         printQuadR("and",t1,t2,t3,0,0);
327 
328         break;
329 
330     case OR:
331         createQuadR(or,t1,t2,t3,0,0);
332         printQuadR("or",t1,t2,t3,0,0);
333 
334         break;
335 
336     case MOD:
337         createQuadR(mod,t1,t2,t3,0,0);
338         printQuadR("mod",t1,t2,t3,0,0);
339 
340         break;
341 
342     case XOR:
343         createQuadR(xor,t1,t2,t3,0,0);
344         printQuadR("xor",t1,t2,t3,0,0);
345 
346         break;
347 
348     case NOT:
349         createQuadR(not,t1,t2,t3,0,0);
350         printQuadR("not",t1,t2,t3,0,0);
351 
352         break;
353 
354     case SLL:
355         createQuadR(sll,t1,t2,t3,0,0);
356         printQuadR("sll",t1,t2,t3,0,0);
357 
358         break;
359 
360     case SRL:
361         createQuadR(srl,t1,t2,t3,0,0);
362         printQuadR("srl",t1,t2,t3,0,0);
363 
```

```
364         break;
365
366     case CLKADJ:
367         createQuadR(nop,t1,t2,t3,0,0);
368         printQuadR("nop",t1,t2,t3,0,0);
369
370         break;
371
372     case BEQ:
373         createQuadI(beq,t1,t2,t3);
374         printQuadI("beq", t1,t2,t3);
375
376
377         break;
378     case BNE:
379         createQuadI(bne,t1,t2,t3);
380         printQuadI("bne", t1,t2,t3);
381         break;
382
383     case LUI:
384         createQuadI(lui,t1,t2,t3);
385         printQuadI("lui", t1,t2,t3);
386         break;
387
388     case SUB:
389         createQuadR(sub,t1,t2,t3,0,0);
390         printQuadR("sub",t1,t2,t3,0,0);
391         break;
392
393     case SOMi:
394         createQuadI(addi,t1,t2,t3);
395         printQuadI("addi",t1,t2,t3);
396         break;
397
398     default:
399         break;
400     }
401 }
```

APÊNDICE J – Código-fonte bincode.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "globals.h"
#include "util.h"
#include "codegenerate.h"
#include "assemblygenerate.h"
#include "bincode.h"


FILE *bincode;

int convertBin(int n, int bits){
    int c, k;

    for (c = bits-1; c >= 0; c--)
  {
    k = n >> c;

    if (k & 1)
      fprintf(bincode,"1");
    else
      fprintf(bincode,"0");
  }
  fprintf(bincode," ");
}

void getBin(quadModel *quad){

    quadModel *ptrQuad = quad;

    while (ptrQuad!=NULL)
    {
        if(ptrQuad->instrType == J)
        {

            if((ptrQuad->quadType.quadJ.op == j)||(ptrQuad->quadType.quadJ.op == jal))
                {
                    if(strcmp(ptrQuad->quadType.quadJ.name,"lab")==0)
                         ptrQuad->quadType.quadJ.target = searchLabel(ptrQuad->quadType.
                             quadJ.target);
                    else if(ptrQuad->quadType.quadJ.target == -1)
                        ptrQuad->quadType.quadJ.target = searchFunEnd(ptrQuad->quadType.
                            quadJ.name);
                    else
                        ptrQuad->quadType.quadJ.target = searchFun(ptrQuad->quadType.
                            quadJ.name);

                  //  fprintf(bincode,"op: %d name:%s target: %d \n", ptrQuad->quadType.
                      quadJ.op, ptrQuad->quadType.quadJ.name, ptrQuad->quadType.quadJ.
                      target);

                }
        }
```

```
50          ptrQuad = ptrQuad->proxQuad;
51      }
52
53      ptrQuad = quad;
54
55      while(ptrQuad != NULL){
56
57          switch (ptrQuad->instrType)
58          {
59          case R:
60              convertBin(0,6);
61              convertBin(ptrQuad->quadType.quadR.rs,5);
62              convertBin(ptrQuad->quadType.quadR.rt,5);
63              convertBin(ptrQuad->quadType.quadR.rd,5);
64              convertBin(ptrQuad->quadType.quadR.shamt,5);
65              convertBin(ptrQuad->quadType.quadR.op,6);
66              fprintf(bincode, "    ");
67              fprintf(bincode,"R    %d %d %d %d %d %d\n", 0, ptrQuad->quadType.quadR.rs,
                    ptrQuad->quadType.quadR.rt, ptrQuad->quadType.quadR.rd, 0, ptrQuad->
                    quadType.quadR.op);
68              break;
69          case J:
70              if((ptrQuad->quadType.quadJ.op != fun_b)
71                  &&(ptrQuad->quadType.quadJ.op!=lab)
72                  &&(ptrQuad->quadType.quadJ.op!=fun_f)){
73                      convertBin(ptrQuad->quadType.quadJ.op,6);
74                      convertBin(ptrQuad->quadType.quadJ.target,26);
75                      fprintf(bincode, "        ");
76                  }
77                   if((ptrQuad->quadType.quadJ.op != fun_b)
78                      &&(ptrQuad->quadType.quadJ.op!=lab)
79                      &&(ptrQuad->quadType.quadJ.op!=fun_f))
80                          fprintf(bincode, "J    %d %d %s\n",ptrQuad->quadType.quadJ.op,
                                ptrQuad->quadType.quadJ.target, ptrQuad->quadType.quadJ.name)
                                ;
81              break;
82
83          case I:
84              convertBin(ptrQuad->quadType.quadI.op,6);
85              convertBin(ptrQuad->quadType.quadI.rs,5);
86              convertBin(ptrQuad->quadType.quadI.rt,5);
87              convertBin(ptrQuad->quadType.quadI.immediate,16);
88              fprintf(bincode, "      ");
89              fprintf(bincode, "I    %d %d %d %d\n", ptrQuad->quadType.quadI.op,ptrQuad->
                    quadType.quadI.rs,ptrQuad->quadType.quadI.rt, ptrQuad->quadType.quadI.
                    immediate);
90              break;
91          default:
92              break;
93          }
94          ptrQuad = ptrQuad->proxQuad;
95      }
96  }
```