Davi Melo Morales

Projeto de compilador para linguagem C-

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

Professora: Ana Carolina Lorena

São José dos Campos - SP
2017

Conteúdo

1 Introdução

1.1 Contextualização e Motivação

Este trabalho dá continuidade ao projeto de um sistema computacional, iniciado através da elaboração de um processador. A fim de que se projetem programas a serem executados nele, torna-se necessário que se desenvolva um compilador para uma linguagem de alto nível.

Tem-se como compilador um software que converte um conjunto de trechos de uma linguagem de alto nível para uma representação em nível mais baixo [4].

Essa função é fundamental para que se possa obter agilidade e para que se elaborem programas mais complexos através das linguagens de nível mais alto. Dentre as principais razões para que se estudem compiladores, enumeram-se[10]:

1. Parsers e interpretadores são extremamente populares: programadores competentes precisam compreender parsers e interpretadores porque, eventualmente, é necessário que pequenas interações destes sejam escritas: cada vez em que se desenvolve um programa extensível ou se lida com um novo tipo de arquivo de entrada, isso é realizado.

2. Melhor competência em escrever e corrigir códigos: supõe-se que um compilador traduza corretamente qualquer programa em sua linguagem de entrada. A fim de se atingir esta meta, desenvolvedores de compiladores precisam entender completamente a linguagem de entrada, inclusive casos excepcionais nunca vistos pela maioria dos programadores. Este entendimento é um passo importante na compreensão de uma linguagem como ela realmente é, e não somente como ela é normalmente escrita.

3. Aprende-se a programar mais rápido: ao se entender um compilador, ha uma ideia mais clara sobre as possíveis otimizações que estão dentro do escopo de um compilador.

Portanto, há motivações tanto acadêmicas quanto profissionais para a realização do presente projeto.

1.2 Objetivos

Projetar e desenvolver uma solução computacional em compiladores que permita a conversão da linguagem didática à linguagem de desenvolvimento do processador objeto elaborado durante a disciplina de Laboratório de Arquitetura e Organização de Computadores.

1.3 Ferramentas Utilizadas

A linguagem base utilizada para o desenvolvimento da etapa de análise do compilador foi C. Juntamente a ela, utilizaram-se ferramentas acessórias, sobre as quais se descorre nas seções a seguir. Para a análise léxica, utilizou-se o Flex 1.3.1, enquanto para a análise sintática, se fez uso do Bison 1.3.2

1.3.1 Flex

Flex consiste em uma ferramenta para se gerar programas de varredura, os quais reconhecem padrões léxicos. Ele lê determinados arquivos de entrada que descrevem

o programa de varredura a ser gerado. A descrição se encontra na forma de pares de expressões regulares e código C, denominados regras. Flex gera, então, um arquivo fonte em C como saída, o *lex.yy.c*, o qual defina uma rotina *yylex()*. Tal arquivo é compilado e e ligado à biblioteca *-lfl* a fim de produzir um executável. Quando o dado executável é executado, ele analisa sua entrada em busca de expressões regulares. Ao encontrar uma, ele executa o código C correspondente [1].

1.3.2 Bison

Bison é um gerador de parser de propósito geral que converte uma gramática livre de contexto para um parser LR determinístico ou LR geral empregando tabelas parser LALR(1). Bison pode ser utilizado para desenvolver uma ampla gama de parsers para linguagens, desde os utilizados em simples calculadoras até linguagens de programação complexas.

Ele tem compatibilidade total com Yacc [2], de maneira que quaisquer gramáticas escritas em Yacc devem funcionar sobre o Bison sem necessidade de mudanças [3].

1.4 Compilador: Fase de Análise

O primeiro estágio de um compilador é a Fase de Análise. Essa fase é chamada de fase de varredura, onde o compilador varre o código fonte, caracter por caracter, e os agrupa em *tokens*. Cada *token* representa uma sequência coerente de caracteres, como variáveis. Essa fase se divide em 3 estágios [11]:

1.4.1 Análise Léxica

Sendo a primeira das três fases, a análise léxica recebe código fonte modificado por pré-processadores de linguagem, que o organiza em forma de sentenças. O analisador léxico quebra os lexemas em *tokens*, removendo comentários e espaços em branco. Caso haja um *token* inválido, um erro é gerado[8].

1.4.2 Análise Sintática

A análise sintática visa a validação da gramática do programa. Nela, verifica-se se as regras sintáticas da linguagem são obedecidas [6].

Varrem-se os *tokens* em sequência recebidos do analisador léxico e se produz uma árvore sintática para representar a hierarquia do programa fonte. Um erro deve ser gerado caso seja reconhecida uma construção indevida.

1.4.3 Análise Semântica

Para que um programa seja semanticamente válido, suas variáveis, funções e demais itens precisam ser definidos apropriadamente; expressões e variáveis devem ser utilizadas de modo a respeitarem o sistema de tipos, controle de acesso, e assim por diante.

Boa parte da análise semântica consiste no rastreamento de declarações de variáveis, funções ou tipos, e checagem de tipos. Em diversas linguagens, por exemplo, identificadores devem ser declarados antes de seu uso. Ao encontrar uma nova declaração, um compilador deve guardar o tipo da informação atribuído ao dado identificador. Então, examinando o restante do programa, ele verifica se o tipo do identificador está sendo respeitado quanto às operações que se realizam [12].

2 Processador

A arquitetura projetada é RISC e se baseia em uma variação monocíclica da arquitetura MIPS, apresentando modificações inspiradas em conceitos presentes no livro *Logic and Computer Design Fundamentals*[5].

2.1 Conjunto de Instruções

Projetou-se um conjunto de instruções específico, o qual serve como fundamento para o desenvolvimento da presente arquitetura. Cada instrução se divide em seções que especificam os recursos a serem utilizados na operação. Tais recursos são explicados a seguir:

- DR: endereço de 5 bits que se refere ao registrador de destino da operação dentre os 32 presentes no banco de registradores;
- SA, SB: endereços de 5 bits que especificam os registradores de origem;
- IM: refere-se a um imediato de tamanho variável que é passado de maneira direta;
- Alavancas: refere-se às chaves de entrada presentes no kit FPGA;
- *Displays*: se refere aos oito *displays* de sete segmentos presentes no kit FPGA;
- PC: refere-se ao *Program Counter*;
- Shamt: quantidade de *bits* a serem deslocados;
- FZ e FN: são dois *flipflops* que guardam sinais checados para a realização de certas operações.

O *opcode* informa à unidade de controle a operação a ser realizada. Para isso, existe a necessidade do mapeamento das instruções em código binário. A Tabela 1 lista o conjunto de instruções executáveis pela arquitetura proposta e apresenta, para cada uma delas, seu mapeamento, mnemônico e a operação que realiza:

Instrução	Opcode	Mnemônico	Operação
Adição	000000	ADD	$R[DR] \leftarrow R[SA] + R[SB]$
Adição Imediato	000001	ADDI	$R[DR] \leftarrow R[SA] + IM$
Subtração	000010	SUB	$R[DR] \leftarrow R[SA] + \overline{R[SB]} + 1$
Subtração Imediato	000011	SUBI	$R[DR] \leftarrow R[SA] + \overline{IM} + 1$
Multiplicação	000100	MUL	$R[DR] \leftarrow R[SA] * R[SB]$
Divisão	000101	DIV	$R[DR] \leftarrow R[SA]/R[SB]$
Incrementa	000110	INC	$R[DR] \leftarrow R[SA] + 1$
Decrementa	000111	DEC	$R[DR] \leftarrow R[SA] - 1$
And	001000	AND	$R[DR] \leftarrow R[SA] \wedge R[SB]$
Or	001001	OR	$R[DR] \leftarrow R[SA] \vee R[SB]$
Resto	001010	MOD	$R[DR] \leftarrow R[SA]\%R[SB]$
Xor	001100	XOR	$R[DR] \leftarrow R[SA] \bigoplus R[SB]$
Not	001101	NOT	$R[DR] \leftarrow \overline{R[SA]}$
Desloca Esquerda	010000	SHL	$R[DR] \leftarrow sl(shamt)R[SA]$
Desloca Direita	010001	SHR	$R[DR] \leftarrow sr(shamt)R[SA]$
Pré-branch	011111	PBC	se $R[SA] = 0, FZ = 1$; $seR[SA] < 0, FN = 1$;
Branch em Zero	010011	BOZ	se $FZ = 1$, então $PC \leftarrow PC + 1 + IM$ se $FZ = 0$, então $PC \leftarrow PC + 1$
Branch em Negativo	010100	BON	se $FN = 1$, então $PC \leftarrow PC + IM$ se $FN = 0$, então $PC \leftarrow PC + 1$
Jump	010101	JMP	$PC \leftarrow IM$
Set on Less Than	010111	SLT	$seR[SA] < R[SB]$, então $R[DR] = 1$
Load	011000	LD	$R[DR] \leftarrow M[IM]$
Store	011001	ST	$M[IM] \leftarrow R[SA]$
Load Imediato	011010	LDI	$R[DR] \leftarrow IM$
Nop	011011	NOP	Sem Operação
HLT	011100	HLT	Parar Operação
Entrada	011101	IN	$R[DR] \leftarrow alavancas$
Saída	011110	OUT	$Displays \leftarrow R[SA]$

Tabela 1: Mapeamento e especificações referentes ao conjunto de instruções

2.1.1 Formatos das Instruções

Projetaram-se quatro formatos distintos para as instruções com o objetivo de transmitir as informações necessárias de maneira simples e eficiente. Por se tratar de uma arquitetura RISC, todas as instruções apresentam o mesmo tamanho, 32 *bits*:

- Formato de três registradores: engloba a maioria das instruções lógicas e aritméticas - tais como ADD, SUB, AND, OR, XOR, SLT -, sendo que dois registradores fonte - SA e SB - fornecem os operandos e o resultado é gravado no registrador destino.

31 - 26	25 - 21	20 - 16	15 - 11	10 - 0
opcode	SA	SB	DR	—

- Formato de dois registradores: apresenta as instruções aritméticas ADDI e SUBI, de modo que os operandos estão contidos no registrador fonte e no imediato. As funções INC e DEC também são descritas por esse formato; nelas o último campo da instrução torna-se desnecessário. As instruções NOT, SL e SR também aparecem nesse formato, de modo que o registrador fonte é negado, deslocado à esquerda ou à direita, respectivamente; o campo final da instrução, no caso das duas últimas,

indica quantos *bits* o operando deve ser deslocado. Todas gravam informações no registrador destino (DR).

31 - 26	25 - 21	20 - 16	15 - 0
opcode	SA	DR	imediato/endereço/shamt

- Formato de um registrador fonte: contém as instruções PBC, LDI, IN e OUT. No caso de PBC, o valor contido no registrador fonte é comparado a uma referência e, a depender desse resultado, FZ ou FN são *setados*; essa instrução apresenta a exceção de que o endereço do registrador é informado das posições [20..16] da instrução. Em LDI, o valor contido no imediato é carregado para registrador destino. Para a instrução IN, os valores informados através dos dispositivos de entrada (alavancas) tomarão o espaço do campo final da instrução e esse valor será carregado para o registrador destino. Para a função de saída, o valor contido no registrador é encaminhado para a saída de dados.

31 - 26	25 - 21	20 - 0
opcode	SA/DR	imediato/entrada/saída

- Formato sem registradores: utilizado pelas instruções JMP, BOZ, BON, NOP e HLT. A instrução JMP confere o valor do imediato ao *PC*. BOZ e BON somam o valor informado no imediato ao PC seguinte. NOP e HLT não necessitam desse valor, de modo que informam apenas a não realização ou parada de operação.

31 - 26	26 - 0
opcode	imediato/–

2.1.2 Modos de Endereçamento

Como pode-se observar nas estruturas das instruções, os seguintes modos de endereçamento foram adotados:

- Imediato: rápido, porém de alcance limitado, esse modo de endereçamento pode ser visto em instruções como SUBI e ADDI, nas quais o valor informado no imediato torna-se um operando.
- Direto: com equilíbrio entre eficiência e alcance, o modo direto está presente, também, nessa arquitetura. As instruções LD e ST apresentam esse modo, uma vez que buscam valores na memória utilizando endereços guardados em instruções.
- Por registrador: utilizado para acessar operandos guardados em registradores, esse modo é utilizado por muitas instruções, tais como: ADD, SUB, OR, XOR entre outras.

Os modos imediato e direto foram adotados por permitirem operações que utilizam os recursos de maneira eficiente em situações onde seja possível um alcance limitado.

Escolheu-se o modo por registrador por ele ser ideal para determinados tipos de operações, como as operações lógicas e aritméticas, que precisam ser realizadas relativamente rápido mas necessitam, também, de operandos de tamanho razoável.

2.2 Esboço da Arquitetura do Processador

À semelhança da arquitetura MIPS, a presente arquitetura contém memória - neste caso subdividida entre memória de dados e memória de instruções -, um banco de registradores, uma unidade lógica e aritmética (ULA) e o PC.

A memória de instruções tem por função receber a instrução do PC e distribuí-la para seus diversos propósitos. O banco de registradores funciona como uma memória de curto prazo e é dotado de 32 registradores de 32 *bits* cada. A ULA tem por finalidade realizar as operações lógicas e aritméticas, enquanto a memória de dados armazena e carrega dados.

Para o caso de haverem operações entre um dado proveniente de um registrador e outro proveniente de um imediato - seja ele de 16 ou 21 *bits* -, há um extensor que emparelha a quantidade de *bits* do imediato em relação ao registrador.

O projeto da arquitetura foi realizado com o objetivo de se adequar às instruções definidas. Portanto, assim como no caso do conjunto de instruções, a arquitetura se baseou no MIPS.

Os dispositivos de entrada e saída também foram levados em consideração na elaboração dessa arquitetura, de maneira que dados tanto deverão poder ser inseridos no sistema através dos periféricos de entrada, quanto haverá retorno para o usuário através do dispositivo de saída.

Estes foram adaptados ao projeto dessa arquitetura, assim como a presença de imediatos de tamanhos distintos e modificações realizadas em virtude de funções como BOZ e BON, onde seu resultado funciona como seletor, de maneira que o valor do imediato é somado ou não ao endereço seguinte para o PC. A Figura 1 o esboço inicial simplificado para a arquitetura elaborada.

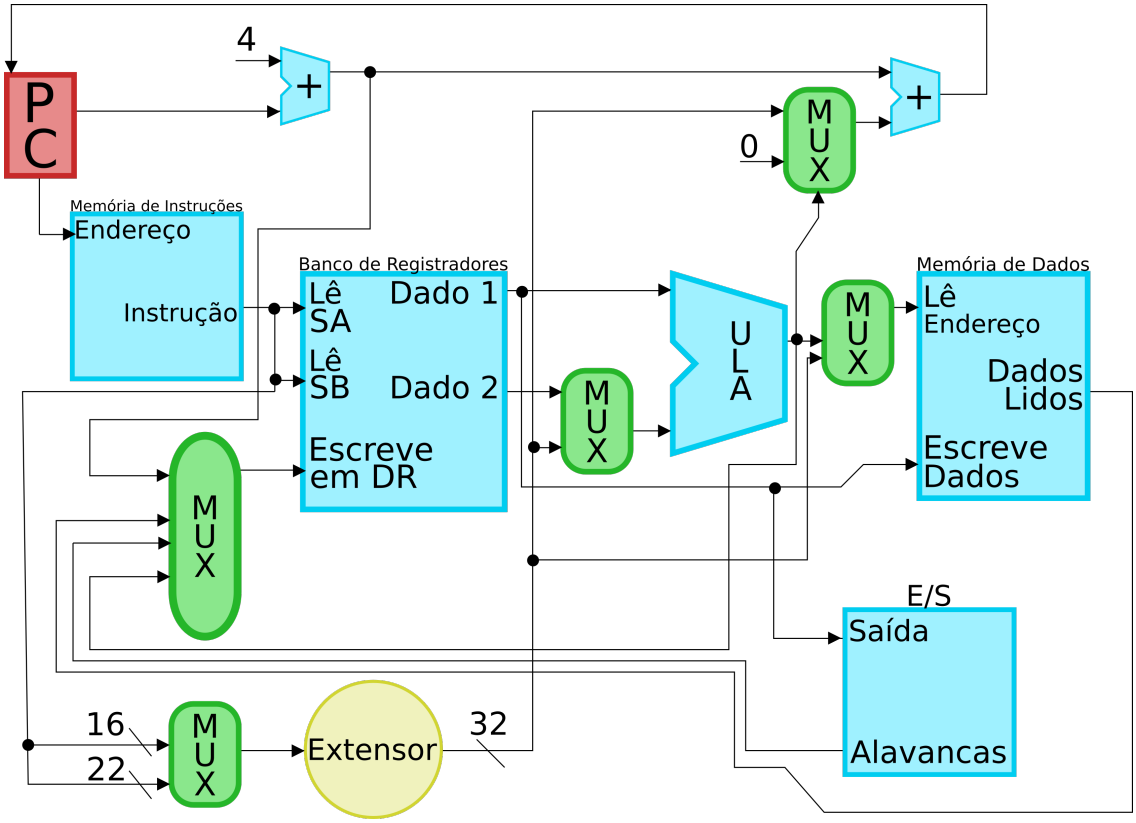

Figura 1: Esquemático do sistema computacional projetado.

2.3 Adendos à Arquitetura

A fim de se utilizar o potencial da linguagem de programação compilada por completo, foi necessário que se desenvolvessem três instruções a mais para o processador. São elas:

1. Load Registrador: seu *opcode* é 100001. Esta permite que se carregue um valor da memória, de maneira que o valor correspondente à posição da memória que se deseja se encontra dentro de um registrador: $R[DR] \leftarrow M[R[SA]]$.

2. Store Registrador: seu *opcode* é 100010. Permite que o valor de um registrador seja guardado em uma posição de memória cujo valor se encontra em outro registrador: $M[R[DR]] \leftarrow R[SA]$.

3. Jump Registrador: seu *opcode* é 100011. Essa instrução salta para uma linha da memória de instruções cujo valor está contido em um registrador: $PC \leftarrow R[SA]$.

A alteração fundamental dessas instruções em relação às suas análogas com imediatos - Load, Store e Jump - se dá na fonte do valor usado em suas operações. Em Load e Store com registradores, o valor selecionado pelo último multiplexador - que acessa a memória - é alterado a partir de uma *flag*, que faz com que este seja a partir do resultado advindo da ULA, a qual neste caso é programada para apenas repassar o valor recebido sem alterações.

Para que se realize o *Jump* com registradores, foi necessário que o endereço recebido para o salto fosse alterado na entrada do PC. Antes, simplesmente se pegavam os 10 últimos *bits* da instrução *jump*. Agora, PC recebe 10 *bits* provenientes da Unidade de Controle, a qual seleciona os 10 últimos *bits* da instrução - para o *jump* convencional, ou os 10 últimos *bits* do valor contido no registrador fonte $A =$ para o *jump* com registrador.

Abaixo se apresentam os códigos correspondentes a essas instruções na Unidade de Controle:

```
6'b100001: begin//load register
writeDataSelection = 1'b1;
writeRegister = 1'b1;
aluSelection = 4'b0000;
extenderSelection = 2'bxx;
immediateSelection = 1'b0;
tripleMuxSelection = 2'b01;
lastMuxSel = 1'b0;
writeEnable = 1'b0;
IO_RAMwrite = 1'b0;
enable = 1'b0;
mainAddress = 10'b0;
jump = 1'b0;
bzero = 1'b0;
bnegative = 1'b0;
HLT = 0;
end

6'b100010:  begin//store register
writeDataSelection = 1'bx;
writeRegister = 1'b0;
aluSelection = 4'b0000;
extenderSelection = 2'bxx;
immediateSelection = 1'b0;
tripleMuxSelection = 2'bxx;
lastMuxSel = 1'b0;
writeEnable = 1'b1;
```

```
IO_RAMwrite = 1'b0;
enable = 1'b0;
mainAddress = 10'b0;
jump = 1'b0;
bzero = 1'b0;
bnegative = 1'b0;
HLT = 0;
end

6'b100011: begin//jmp register
writeDataSelection = 1'bx;
writeRegister = 1'b0;
aluSelection = 4'bxxxx;
extenderSelection = 2'bxx;//xx
immediateSelection = 1'bx;
tripleMuxSelection = 2'bxx;
lastMuxSel = 1'bx;
writeEnable = 1'b0;
IO_RAMwrite = 1'b0;
enable = 1'b0;//vai que
mainAddress = srcRegister[9:0];
jump = 1'b1;
bzero = 1'b0;
bnegative = 1'b0;
HLT = 0;
end
```

3 O compilador: fase de análise

O compilador desenvolvido analisa códigos em C menos, gera uma tabela de literais na análise léxica, uma tabela de símbolos e uma árvore sintática na análise sintática. Efetua, também, a checagem de erros de natureza léxica, sintática ou semântica no código.

3.1 Análise Léxica

A análise léxica foi implementada com o uso da ferramenta *flex* 1.3.1. Nela, procura-se gerar uma tabela de literais a partir do código analisado. Os possíveis literais definidos para esta implementação se apresentam na Tabela 2.

Entrada	Token
int	INT
float	FLOAT
if	IF
else	ELSE
return	RETURN
void	VOID
while	WHILE
+	PLUS
−	MINUS
*	TIMES
<	LT
/	OVER
<=	LET
>	HT
>=	HET
==	EQ
! =	NEQ
=	ASSIGN
;	SEMI
,	COMMA
(	LPAREN
)	RPAREN
[	LBRACK
]	RBRACK
{	LCAPSULE
}	RCAPSULE

Tabela 2: Mapeamento e especificações referentes ao conjunto de instruções

Nessa etapa, os espaços em branco e comentários são removidos, os quais começam com '/*' e terminam com '*/'. A linguagem regular adotada se apresenta no código abaixo.

```
digit        [0-9]
numberi      {digit}+
numberf      {digit}+?\.{digit}+|-?{digit}+\.{digit}+?
letter       [a-zA-Z]
identifier   {letter}+{letter}*
newline      \n
whitespace   [ \t\r]+
other    [^0-9a-zA-Z;/=\-"+""*""(")""\n"\[\]\,\{\}\<\>|!=\==\<=\>=]
```

Nela, consideram-se dígitos de 1 a 9. Números são considerados, tanto inteiros quanto números no formato *float*. As letras são consideradas, assim como identificadores. Identificam-se novas linhas e espaços, além de um conjunto de símbolos diversos.

Nessa varredura, os identificadores foram guardados em uma tabela *hash* de listas dinâmicas. A estrutura guarda o escopo, linhas em que o identificador aparece, nome, tipo e se é função ou não.

O código gerado pelo Flex referente a esta etapa se encontra na Seção 2 do apêndice.

3.2 Análise Sintática

O parser elaborado nesta fase foi realizado através da ferramenta *Bison* 1.3.2. Geram-se nela a Tabela de símbolos (Figura 2) e a árvore de análise sintática (Figura 3.

Name(ID)	Type(ID)	Type(Data)	Scope	Appears in lines
output	func	void	none	36
input	func	int	none	34,35
i	var	int	main	30,44
main	func	void	none	22
u	var	int	gdc	4,7,8
u	var	int	main	28,39,43,44
v	var	int	gdc	4,7,8
v	vet	int	main	29,44
x	var	int	output	17
x	var	int	main	26,34,36,40,41,45
y	var	int	main	27,35,36
gdc	func	int	none	4,8,36

Figura 2: Exemplo de tabela de símbolos.

```
programa
  declaration-list
    declaration-list
      declaration-list
        declaration-list
          declaration
            fun-declaration
              type-specifier
                INT
              ID
              LPAREN
              params
                param-list
                  param-list
                    param
                      type-specifier
                        INT
                      ID
```

Figura 3: Exemplo de saída de árvore sintática - níveis representado através da identação.

Conforme a linguagem C menos, as seguintes construções são permitidas para este trabalho:

1. Variáveis e funções devem ser declaradas antes de serem usadas;
2. Tipos possíveis de variáveis: int e float;
3. Tipos possíveis de funções: void, int e float;
4. Variáveis podem ser declaradas como vetores da forma nome[tamanho];
5. O laço de repetição permitido é o *while*;
6. As estruturas condicionais permitidas são *if* e *else*;
7. Exceto na declaração de funções, laços de repetição ou estruturas condicionais, todas as linhas devem terminar com ';';
8. Laços de repetição, condicionais e funções devem apresentar chaves no começo e em seu fim da seguinte forma: est(){ ... }.

A gramática BNF considerada se apresenta no Código 4.

```
programa → declaração-lista
declaração-lista → declaração-lista declaração | declaração
declaração → var-declaração | fun-declaração
var-declaração → tipo-especificador ID ; | tipo-especificador ID [ NUM ] ;
tipo-especificador → int | void
fun-declaração → tipo-especificador ID ( params ) composto-decl
params → param-lista | void
param-lista → param-lista, param | param
param → tipo-especificador ID | tipo-especificador ID [ ]
composto-decl → { local-declarações statement-lista } | {local-declarações} | {statement-lista} | { }
local-declarações → local-declarações var-declaração | var-declaração
statement-lista → statement-lista statement | statement
statement → expressão-decl | composto-decl | seleção-decl | iteração-decl | retorno-decl
expressão-decl → expressão ; | ;
seleção-decl → if ( expressão ) statement | if ( expressão ) statement else statement
iteração-decl → while ( expressão ) statement
retorno-decl → return ; | return expressão;
expressão → var = expressão | simples-expressão
var → ID | ID [ expressão ]
simples-expressão → soma-expressão relacional soma-expressão | soma-expressão
relacional → <= | < | > | >= | == | !=
soma-expressão → soma-expressão soma termo | termo
soma → + | -
termo → termo mult fator | fator
mult → * | /
fator → ( expressão ) | var | ativação | NUM
ativação → ID ( arg-lista ) | ID ( )
arg-lista → arg-lista , expressão | expressão
```

Figura 4: Gramática BNF para a linguagem C-

A estrutura adotada nessa etapa foi a Árvore Genérica, onde se guarda a linha de aparição e item da gramática, assim como um nó filho e um nó irmão.

Hierarquicamente, portanto, as informações são guardadas dentro dessa árvore sintática.

O código referente aos desenvolvimentos dessa fase se encontram na Seção 1 do Apêndice.

3.3 Análise Semântica

A análise semântica elaborada verifica as seguintes condições:

1. Se as variáveis foram declaradas antes do uso;
2. Se há variáveis recebendo retorno de funções void;
3. Se há variáveis declaradas como void;
4. Se há declarações duplas num mesmo escopo;
5. Se as funções foram definidas antes do uso;
6. Se existe a função 'main';
7. Possíveis duplicações entre nomes de funções e variáveis.

Essas verificações são feitas fazendo-se uso da tabela de símbolos, ao longo da própria varredura de análise léxica e, também, em uma pós varredura. O código se encontra na Seção 2 do Apêndice.

As seguintes verificações são feitas através da tabela de símbolos:

- Existência de variáveis void:

```
int checkVoid(TipoLista *list, int index){
TipoID *p = list[index].start;
//Laco que percorre todas as posicoes da tabela hash
while(p!=NULL){
    if(p->linhas[0] != 0) {
      //Aqui se comparam os valores. Caso haja uma variavel void, o
          erro e apresentado.
      if (!strcmp(p->tipoID, "var")&&!strcmp(p->tipoData,"void")) {
        return p->linhas[0];
      }
    }
      p = p->prox;
    }
    return 0;
}
```

- Existência da função main:

```
int checkMain(TipoLista *list, int index){
TipoID *p = list[index].start;
//Laco que percorre todas as posicoes da tabela hash
while(p!=NULL){
    if(p->linhas[0] != 0) {
      //Aqui se comparam os valores. Caso nao haja uma funcao void de
          nome main, retornara erro.
      if (!strcmp(p->tipoID, "func")&&
          !strcmp(p->tipoData,"void")&&
          !strcmp(p->nomeID, "main")&&
          p->linhas[1]==0)
        return 0;
    }
      p = p->prox;
    }
    return 1;
}
```

- Duas variáveis declaradas em um mesmo escopo:

```
int checkDecScope(TipoLista *list, int index){
  TipoID *p = list[index].start;
  TipoID *w;
//Laco que percorre todas as posicoes da tabela hash. Procuram-se
    apenas variaveis ou vetores
  while(p!=NULL&&(!strcmp(p->tipoID,"var")||!strcmp(p->tipoID,"vet"))
     ){
    if(p->linhas[0] != 0) {
      w = p->prox;
      //Segundo laco que percorre tabela para que haja comparacao.
      while (w!=NULL&&(!strcmp(p->tipoID,"var")||!strcmp(p->tipoID,"
         vet"))){
      //Aqui ocorrem as comparacoes entre as variaveis para que nao
          haja dupicadas.
        if(!strcmp(w->nomeID,p->nomeID)&&!strcmp(p->escopo,w->escopo)
           ){
          return w->linhas[0];
        }
        w = w->prox;
      }
    }
      p = p->prox;
    }
    return 0;
  }
```

De forma análoga às apresentadas anteriormente, se verifica também se há variáveis e funções com nomes iguais.

As verificações relacionadas a chamadas de funções ou variáveis não declaradas são feitas ao longo da primeira varredura e atribuição de *tokens*, onde quando uma variável ou função é chamada, checa-se se já houve declaração.

Por fim, a verificação quanto a atribuições de variáveis *void* a identificadores inteiros é feita em uma última varredura, onde se separam os dois lados da declaração e se verifica se a atribuição está sendo feita de forma válida.

4 O Compilador: Fase de Síntese

A fase de síntese tem por propósito gerar o programa alvo para o sistema ao qual ele foi projetado. Para tal, ela faz uso do código intermediário gerado e da tabela de símbolos.

4.1 Gerenciamento da Memória

No presente projeto, a memória é gerenciada de forma estática. Cada variável é mapeada a um item dentro dessa lista, o qual tem correspondente nas posições de memória. A função *declaration_variables*, presente no código *object.c*, realiza essa correspondência entre a tabela de símbolos e uma lista que permita o uso das variáveis no código objeto.

Quando uma variável é chamada, o valor correspondente é procurado nessa lista e passado para a instrução através da instrução *search_variable*, a qual retorna a posição na memória com base no nome da variável e em seu escopo, seja ela uma variável simples ou um vetor. Esse mesmo valor pode ser usado para que se guardem valores na memória.

Mais detalhes a respeito de como se dá a interação com a memória, inclusive a respeito de registradores temporários, se encontram na Seção 4.3, onde as explicações são apresentadas acompanhando seu contexto na geração de código objeto.

4.2 Geração de Código Intermediário

O código intermediário é tido como uma estrutura de dados que representa o programa fonte ao longo do processo de tradução.

A representação em árvore sintáxica pouco se assemelha à de um código fonte; portanto, recomenda-se que essa estrutura seja substituída por outra - ou outras - que se adequem melhor à representação do código fonte. Este é o espaço ocupado pelo código intermediário.

A representação utilizada a fim de se elaborar o código fonte foi a de quatro endereços, onde um é reservado para identificar o tipo de operação, dois para a realização de possíveis subatividades e, por fim, o último recebe o resultado da operação realizada.

As operações definidas e suas funções se localizam em *intermediate.h*, conforme a tabela abaixo.

Operação	Descrição
AddK	Operações aritméticas de adição
SubK	Operações aritméticas de subtração
TimK	Operações aritméticas de multiplicação
OvrK	Operações aritméticas de divisão
EqlK	Operações comparativas de igualdade
Neqk	Operações comparativas de desigualdade
GtrK	Operações comparativas de superioridade
GeqK	Operações comparativas de superioridade ou igualdade
LsrK	Operações comparativas de inferioridade
Leqk	Operações comparativas de inferioridade ou igualdade
AsvK	Operações aritméticas de adição
AsaK	Operações aritméticas de adição
InnK, OutK	Entrada e saída
PrmK, CalK, RetK	parâmetros, chamada e retorno de funções
IffK	Desvio condicional: se falso, desvio
GtoK	Ir para
HltK	Parar operação
LblK	Etiqueta

Tabela 3: Instruções de controle de programa

Conforme a árvore é percorrida, os dados são guardados nas quádruplas.

Como se apresenta no código 4, todas as funções realizadas pela linguagem C- se compões desses termos.

As quádruplas se guardam conforme em uma estrutura de lista a ser utilizada posteriormente na geração de código objeto. Abaixo, se observa a estrutura de uma quádrupla.

```
typedef struct quadruple{
  OpKind op;
  Address address_1, address_2, address_3;
  struct quadruple *next;
}quadruple;
```

A função base para a geração do código intermediário se encontra no arquivo 4. É a função "generate_intermediate_code", a qual se apresenta a seguir:

```
void generate_intermediate_code(list_quadruple *quad_list, TreeNode *
    tree){
  if(tree!=NULL){
```

```
    switch (tree->nodekind) {
      case StmtK:
        // printf("statement\n");
        generate_statement(quad_list, tree);
        break;
      case ExpK:
        // printf("expression\n");
        generate_expression(quad_list, tree);
        break;
      default:
        break;
    }
    generate_intermediate_code(quad_list, tree->sibling);
  }
}
```

Esta percorre a árvore de maneira transversal, verificando entre vizinhos a presença de *statements* e expressões. Conforme estas são encontradas, seus filhos são explorados para a criação das quádruplas.

4.2.1 *Statements*

Dentro da categoria dos *statements*, temos as seguintes possíveis classificações:

- IfK: relativo às operações de *if* e *if else*;
- WhileK: relativo aos laços *while*;
- AssignK: corresponde à assinalação de itens a variáveis;
- ReturnK: diz respeito a retornos de funções;
- FuncK: apresenta as declarações de funções.

Nas operações *IfK*, o resultado da operação relacional que o condiciona é armazenado em uma variável temporária. Caso seu valor seja positivo, a leitura continua a partir do *if*. Caso contrário, salta para a posição especificada na instrução.

Em casos em que ocore um *else*, há um salto desse *else* caso o *if* seja afirmativo. Caso contrário, o *else* é executado diretamente.

Nas operações *WhileK*, uma etiqueta é atribuída a uma operação relacional. Caso seja positiva, a operação dentro do laço é executada e um *goto* direciona de volta a essa etiqueta para nova verificação. Caso negativo, o caminho se direciona a uma etiqueta colocada após o *goto* previamente mencionado.

Operações *AssignK* atribuem a ora posições em vetores, ora variáveis, operações, chamadas de funções, variáveis ou temporários.

ReturnK's retornam valores nulos, variáveis, funções ou temporários de operações anteriores.

Operações *FuncK* conferem uma etiqueta à linha em que estão, anunciando o início de uma dada função.

4.2.2 *Expressions*

Dentro desta categoria, há as seguintes classificações:

- TypeK: corresponde a parâmetros e declarações;
- RelOpK: relativo a operações comparativas;

- ArithOpK: se refere a operações aritméticas;
- CallK: corresponde a chamadas de funções.

As operações relacionais e aritméticas atribuem a uma dada variável temporária o resultado da operação realizada.

As do tipo *CallK* apresentam primeiro os parâmetros, os quais são vizinhos uns dos outros. Apresentam, então, a função chamada e o número de parâmetros correspondentes.

4.3 Geração de Código Objeto

A geração de código objeto tem sua implementação representada nos arquivos *object.c* - onde se realiza o processo estrutural de tradução - e no arquivo *target.c*, onde as informações são representadas de forma inteligível ao processador a ser utilizado. Utiliza, porém, informações provenientes de outras etapas do projeto para estruturar e implementar o código alvo a partir do código intermediário.

A função primária de geração de código objeto se identifica como *generate_code_launcher*, e está presente no arquivo *object.c*.

Nela, as variáveis identificadas para cada função são procuradas na tabela de símbolos e adicionadas a uma lista de variáveis - *variables_list*. A estrutura correspondente às variáveis - *type_variable* - carrega um índice correspondente à sua posição individual na memória, um índice correspondente à sua posição no vetor (em casos de vetores), seu tipo de variável - variável simples ou vetor - e a vizinha na lista.

É chamada, então, a função *generate_code*. Esta percorre a lista de quádruplas, tratando individualmente as instruções advindas e gerando, então, a lista de instruções através das funções: *format_zero*, *format_one*, *format_two* e *format_three*. Essas funções realizam a inserção das instruções em sua lista específica - *instructions_list* - de forma correspondente aos formatos de instrução do processador alvo [7].

A estrutura que comporta a instrução guarda, além das informações a serem impressas no código alvo, outras informações utilizadas em tratamentos dentro da implementação. Abaixo se observa a estrutura do tipo instrução e, posteriormente, a explicação de seus itens:

```
typedef struct type_instruction{
  int line;
  int register_a;
  int register_b;
  int register_c;
  int immediate;
  int target_label;
  kind_jump jump;
  char label_name[50];
  galetype type;
  struct type_instruction *next;
}type_instruction;
```

- line: corresponde à linha no código objeto à qual a instrução pertence;
- register_a, register_b, register_c: números de registradores a serem usados nas instruções. Para instruções sem registradores, todos recebem 0; Para instruções com 1 registrador, apenas register_a recebe o valor; para instruções de 2 registradores, register_a é o registrador fonte, register_b é o registrador destino. Em instruções de 3 registradores, register_a e register_b são registradores fontes e register_c é o registrador destino.

- immediate: guarda o imediato e instruções que podem contê-lo. Pode conter tamanhos variados a depender da instrução, conforme 2.1.1.
- target_label: assume o valor que identifica a etiqueta que se representa nessa instrução; útil para a resolução de saltos.
- jump: classifica o tipo de etiqueta - se de função ou apenas uma etiqueta numérica. Utilizada na resolução posterior dos saltos.
- label_name: identifica o nome da etiqueta, representado pela função chamada que corresponde à existência da etiqueta.
- type: classifica que tipo de instrução do processador será acionada a partir desse item.

4.3.1 Registradores

Para fins de compreensão, nesta seção, as operações representadas por quádruplas serão referidas como *operações*, enquanto as instruções a fazerem parte do código alvo serão referidas como *instruções*.

Para todas as operações realizadas, pode haver até dois registradores operandos - 3 e 4, no caso - e 1 registrador resultado - registrador 1. Vinte registradores - do 7 ao 27 - foram reservados para variáveis temporárias. O registrador de retorno (30) armazena o valor retornado por funções e o registrador topo armazena o topo da pilha utilizada para controlar a posição dos retornos das funções.

4.3.2 Operações de Soma e Subtração

Para que ocorra uma operação de soma ou subtração, é necessário que se conheçam os operandos, que são recuperados na primeira parte da operação.

As instruções correspondentes a essas operações são: ADD e SUB - instruções de 3 registradores, dois fonte e um destino - e ADDI e SUBI - instruções de 2 registradores e 1 imediato, sendo 1 registrador fonte, o imediato como outra fonte, e o 1 registrador como destino.

Um operando pode vir a ser de três tipos: *String*, correspondente a uma variável específica; *Temp*, correspondente a um temporário resultante de uma operação anterior; *IntConst*, um inteiro constante.

Para variáveis, a posição de memória correspondente deve ser procurada pela função *search_variables*, a qual acessa a lista de variáveis e busca a correspondente por nome e escopo. O valor encontrado vai como imediato, então, da instrução LD (*load*), carregando-a ao registrador correspondente ao operando desejado.

Para temporárias, a função *search_temporary* busca o valor do registrador que contém o valor a ser carregado. Ela faz a busca a partir do índice da temporária procurada, o qual fica guardado no vetor global *temporaries*. Carrega-se, então, o valor contido no registrador temporário ao registrador operando desejado através da instrução ADDi (*add imediato*) somada a 0. A *flag* de temporário é levantada.

Para constantes inteiras, caso o valor seja inferior a 65000 - tamanho máximo que um imediato pode assumir em instruções de dois registradores, próximo a $2\hat{1}6$ - guarda-se esse valor no compilador para uso posterior, e a *flag* de operador imediato é levantada. Caso contrário, a instrução LDI (*load imediato*) carrega o valor do imediato para o registrador operando desejado.

A segunda parte da operação consiste na realização da operação aritmética e armazenamento do resultado em um temporário.

A estrutura desse armazenamento ocorre de maneira que considera a presença de imediatos a serem adicionados diretamente à instrução - como mencionado anteriormente, menores que 65000. Portanto, caso ambos os operandos sejam imediatos dessa natureza, sua operação aritmética ocorrerá dentro do próprio compilador, sendo apenas o resultado carregado ao registrador resultado através da operação LDI (*load imediato*). Havendo um operando - seja esquerdo ou direito - como imediato desse tipo, a instrução ADDI (ou SUBI) operará sobre o valor contido no registrador operando ao valor do imediato operando, escrevendo o resultado no registrador resultado. As *flags* de imediato levantadas outrora, então, se abaixam.

Em casos onde nenhum dos operandos é imediato, a operação ADD ou SUB é realizada sobre os operandos e registrada no registrador resultado.

Caso algum temporário tenha sido usado, então, ele é liberado pela função *release_temporary* - que atribui 0 à posição correspondente à temporária - e a *flag* de temporário é abaixada.

O valor do registrador resultado, então, alimenta a função *map_temporary*. Esta função percorre o vetor de temporárias e, ao encontrar uma posição zerada, atribui a ela o valor da temporária t correspondente. Através de ADDI, então, o valor do registrador resultado é carregado ao registrador temporário identificado. O código abaixo apresenta o que ocorre:

```
void map_temporary(list_instructions *instructions_list, int
    register_temporary, int register_source){
        int i, temp = -1;
        for(i = 0; i < MAXREGISTER; ++i){
                if(temporaries[i] == 0){
                        temporaries[i] = register_temporary;
                        temp = i+OFFSET;
                        break;
                }
        }
        if(temp < OFFSET){
                printf("ERROR: register file overflow: no temporaries
                    available at requested instruction\n");
                exit(0);
        }
        format_two(instructions_list, G_ADDI, register_source, temp,
            0, "none");
}
```

4.3.3 Operações de Multiplicação e Divisão

Ocorrem de maneira semelhante às de adição e subtração (Seção 4.3.2). Como não há instruções que multipliquem ou dividam imediatos, porém, todas as operações são realizadas entre três registradores através das instruções MUL e DIV. Em casos onde ambos os operandos são imediatos, a operação ocorre no compilador e é carregada diretamente ao registrador destino, de onde se armazena em um registrador temporário.

4.3.4 Operações Lógicas

Nelas, o carregamento de operandos é feito de forma semelhante à das operações de adição e subtração (Seção 4.3.2). Para cada operação, temos:

- EqlK (igual): caso ambos operandos imediatos, o resultado é obtido em nível de compilador. Caso contrário, através da instrução SLT (*set on less than*) 1: se o

operando a é menor que o operando b; 2: se o operando b é menor que o operando a. A instrução OR, então, é realizada entre os resultados obtidos, e seu valor negado é gravado no registrador resultado através da instrução NOT.

- NeqK (diferente): Semelhante à operação EqlK, porém o valor obtido pelo OR não é negado no resultado.
- GtrK (maior que): Caso ambos operandos imediatos, o resultado é obtido em nível de compilador. Caso contrário, o valor de SLT é atribuído ao registrador resultado (utilizam-se os operandos em ordem trocada nas instruções com registradores).
- GeqK (maior ou igual): aso ambos operandos imediatos, o resultado é obtido em nível de compilador. Caso contrário: SLT sobre os operandos e, em seguida, o valor é negado.
- LsrK (menor que): análogo a GtrK, porém sem trocar os operandos de posição.
- LeqK (menor ou igual que): análogo a GeqK, porém trocando os operandos de posição para instruções com registradores operandos.

As temporárias são, então, liberadas e o resultado é mapeado.

4.3.5 Operações de Atribuição de Variáveis ou Vetores

Em AsvK e AsaK - atribuição de variáveis e de parâmetros, respectivamente - um valor contido em um registrador temporário é atribuído a uma posição de memória. Para variáveis, o registrador temporário é buscado com *search_temporary*, a posição de memória com *search_variable*, e o valor é guardado na posição de memória com a instrução ST (*store*).

Quando se trata de vetores, além da posição de memória correspondente à variável, é necessário buscar a subposição correspondente ao índice do vetor considerado. Este pode estar como variável, temporária ou constante. Para todos os casos, os valores recuperados são somados à posição de início do vetor e, então, o valor contido na temporária é atribuído à posição de memória encontrada.

4.3.6 Operações de Listagens de Parâmetros, Chamadas de Função e Retornos

A operação correspondente a listagens de parâmetros é PrmK. Quando uma operação dessas é acionada, um parâmetro é adicionado a uma lista especial que os armazena: *parameters_list* para uso posterior. Eles podem ser constantes inteiras, temporárias ou variáveis. A estrutura que lhes armazena contém apenas seu tipo - igual ao tipo da quádrupla que lhe alimenta -, seu valor ou nome e o escopo onde é listada.

Operações CalK correspondem a chamadas de função. Quado uma função é chamada, a primeira ação é o consumo dos parâmetros, de maneira que os valores dos parâmetros, pertencentes à função que chama, são atribuídos aos seus correspondentes da função chamada através da função *consume_parameters*.

Em *consume_parameters*, a lista de parâmetros é percorrida. Primeiro, o valor do parâmetro é carregado ao registrador resultado, seja ele advindo de uma temporária, variável, ou apenas uma constante.

O parâmetro é guardado na memória, então, da seguinte forma:

A tabela de símbolos é percorrida; caso se encontre um item na tabela do escopo da função chamada e de mesma posição na ordem de parâmetros. Caso o destino seja uma

variável, o valor do parâmetro é guardado na posição de memória da função chamada. Caso se trate de um vetor, para cada item do vetor, o parâmetro é procurado na memória, esse valor é guardado no registrador resultado, e então guardado na posição de memória do destino.

Caso seja a primeira chamada de função da execução, a *flag* primeira chamada será igual a zero, e então o topo da pilha de chamadas será inicializado a partir da primeira posição de memória ainda não utilizada, sendo o valor do topo guardado no registrador topo. Levanta-se, então, a *flag* de primeira chamada.

O topo da pilha se acresce, então, com ADDI. O número da linha de retorno é adicionado ao registrador retorno. Através de STR (*store register*), então, guarda-se na posição de memória contida no registrador topo o valor contido no registrador resultado, correspondente à linha para a qual, posteriormente, a operação de retorno terá de voltar. Salta-se, então, para o início da função chamada através de um JMP (*jump*).

A operação de retorno, RetK, precisa retornar o resultado da função chamada para uma temporária presente na função que chama - caso seja int. Primeiramente, o valor a ser retornado é carregado para o registrador resultado, de maneira análoga a como operandos são carregados na Seção 4.3.2. Esse valor, então, é guardado no registrador de retorno. Para o retorno à função que chamou, então, é feito um salto para a linha de valor contido no registrador topo. Uma instrução de salto semelhante a essa é posicionada no final de cada função, para o caso dela não apresentar um *return*.

Voltando à chamada de função, o topo da pilha de chamadas é decrementado.

Chama-se, então, a função *restore_arrays*, a qual é bastante semelhante à *consume_parameters*, porém busca os vetores da função chamada e armazena seus valores alterados na função que chama, retornando seus valores. Nessa função, a lista de parâmetros reinicializada.

4.3.7 Operações de Entrada e Saída

Para operações de entrada (InnK), a instrução IN é acionada (*input*) é acionada para o registrador resultado, o qual é mapeado a um registrador temporário.

Em operações de saída, o parâmetro é buscado e carregado ao registrador resultado, a lista de parâmetros é reinicializada e as instruções POUT e OUT (*pré-output e output* são acionadas, levando como saída o registrador resultado.

4.3.8 Operações de *No Operation* e *Halt*

Nesses casos, as operações correspondentes do código alvo são simplesmente acionadas. Não há quaisquer instruções adicionais.

4.3.9 Operações *If False* e *Go To*

Nessas operações, o valor a ser testado se encontra em uma temporária. O registrador temporário, então, é buscado, e a instrução PBC (*pré-branch*) é acionada para testar o valor. Quando essa instrução é acionada, um *flipflop* guarda se o seu resultado foi igual a zero. A instrução BOZ (*branch on zero)*, então, realiza a ramificação do código em caso afirmativo. A instrução de branch é adicionada, mas sem o valor do salto da ramificação, o qual é tratado posteriormente na função *treat_jumps_n_branches*.

Para *go to's*, uma operação de *jump* é adicionada com, com o valor da etiqueta ou com o nome da função, para que seja tratada posteriormente em *treat_jumps_n_branches*.

4.3.10 Operações de Etiqueta, Constantes, Variáveis ou Vetores

Operações de etiqueta - LblK -, são usadas para a marcação de posições dentro do código para onde pode vir a ser necessário saltar. No compilador, elas são armazenadas em uma lista específica a ser usada para fins de tratamento. Abaixo se apresenta o conteúdo das informações guardadas nos itens da lista de etiquetas:

```
typedef struct type_label{
  AddrKind type;
  char name[50];
  int index;
  int line;
  struct type_label *next;
}type_label;
```

Onde *type* indica se corresponde à etiqueta de uma função ou numérica, *name* carrega o possível nome da função, *index* carrega seu índice dentre as etiquetas do código e *line* armazena sua linha.

Há operações que carregam valores de constantes, variáveis ou de posições específicas de vetores a variáveis temporárias, as quais são: CstK, VstK e AstK, respectivamente.

Quando se carregam constantes, um valor de constante é carregado ao registrador resultado de forma análoga ao que acontece em Seção 4.3.2. Da mesma forma, o carregamento de variáveis é feito a partir da memória semelhantemente ao carregamento de operandos em Seção 4.3.2.

O carregamento de posições de vetores precisa considerar a posição do vetor na memória, e a posição que se quer do próprio vetor. Para se obter a primeira, basta usar a busca de variável para o início do vetor. O deslocamento, porém, é buscado de formas diferentes a depender da natureza do item correspondente a ele.

Se ele for uma constante inteira, a própria função de busca de variável retornará seu valor. Caso seja uma variável, a posição dessa variável precisa ser buscada. O valor contido nela deve ser carregado a um registrador operando. Somado ao valor da posição do vetor, esse valor representa a posição de memória da qual se deseja carregar o valor a ser inserido no registrador temporário. Carrega-se, então, ao registrador resultado, através da instrução LDR (*load register*), o valor contido na posição de memória indicada pelo valor do registrador operando.

Se o item for uma temporária, o valor contido no registrador temporário deve ser atribuído a um registrador operando; esse valor, somado à posição do vetor, deve ser imputado sobre um outro registrador operando, para que enfim, semelhantemente ao que ocorre para variáveis, se carregue o valor desejado ao registrador resultado.

O valor contido no registrador resultado dessas operações de carregamento se mapeia, finalmente, a um registrador temporário correspondente à variável temporária.

4.4 Tratamento de Saltos e Ramificações

As instruções de saltos e ramificações são adicionadas sem os valores imediatos correspondentes a para onde se devem saltar ou ramificar em um primeiro momento, mas guardam referências a respeito da etiqueta para onde se destinam.

Após a inserção de todas as instruções, a função *treat_jumps_n_branches* percorre a lista de etiquetas. Percorrendo a lista de etiquetas, se deve verificar se se trata de uma etiqueta numérica ou de texto. Etiquetas numéricas identificam posições no código, enquanto as etiquetas de texto são usadas para indicar a posição de uma função, sendo úteis para o salto para *main* contido no início do código.

Dentro da identificação de tipo de etiqueta, se percorre a lista de instruções, e se verifica se há correspondência entre a etiqueta e a etiqueta alvo da instrução. Caso afirmativo, se for uma instrução de salto, seu imediato receberá o valor da linha da etiqueta; caso seja de ramificação, receberá o valor da linha da etiqueta menos o valor da linha da instrução menos um, pois em ramificações, o valor do imediato é somado ao da linha atual.

4.5 Código Alvo

Após gerado o código objeto, este precisa ser traduzido de forma a ser inteligível para o processador. Para tal, há a conversão para código alvo, a qual se realiza em *target.c* através da função *print_target_code*.

Nessa função, os valores das instruções, registradores e imediatos são mapeados às posições das instruções e convertidos para binário. Isso é feito instrução pós instrução, percorrendo a lista de instruções. Os valores convertidos são impressos, então, no arquivo *simpleInstructionsRam.v*, o qual é carregado ao processador *Galetron* [7] para a realização do programa produzido.

5 Exemplos

Os códigos de exemplo utilizados para o teste do compilador são: *fibonacci*, *sort* e *foo*; *fibonacci* recupera o valor da sequência de *fibonacci* correspondente ao termo inserido, sort recupera o valor na posição de memória ordenada requisitada através do termo inserido, e *foo* retorna o valor de uma operação didática.

As subseções seguintes discorrem a respeito dos testes realizados com cada uma dessas funções.

5.1 Fibonacci

O programa *fibonacci* é apresentado a seguir:

```
int fibonacci(int n){
    int c;
    int next;
    int first;
    int second;
    first = 0;
    second = 1;
    c = 0;
    while(c <= n){
        if(c <= 1)
            next = c;
        else{
            next = first + second;
            first = second;
            second = next; /* Estava second = first */
        }
        c = c + 1;
    }
    return next;
}
void main(void){
  int n;
  n = input();
  output(fibonacci(n));
}
```

Como operações mais complexas, esse programa testa as seguintes:

- Entrada;
- Saída;
- Chamada de função;
- Laço condicional;
- Condição *if*/*else*.

Seu correspondente em código alvo é o seguinte:

```
instructionsRAM[0] = 32'
    b01101100000000000000000000000000;//Nop
instructionsRAM[1] = 32'
    b01010100000000000000000000110011;//Jump to #51
instructionsRAM[2] = 32'
    b01101000001000000000000000000000;//Loadi #0 to r
    [1]
instructionsRAM[3] = 32'
    b00000100111000010000000000000000;//ADDi r[1], #0
     to r[7]
instructionsRAM[4] = 32'
    b01100100111000000000000000000101;//Store r[7] in
     m[#5]
instructionsRAM[5] = 32'
    b01101000001000000000000000000001;//Loadi #1 to r
    [1]
instructionsRAM[6] = 32'
    b00000100111000010000000000000000;//ADDi r[1], #0
     to r[7]
instructionsRAM[7] = 32'
    b01100100111000000000000000000100;//Store r[7] in
     m[#4]
instructionsRAM[8] = 32'
    b01101000001000000000000000000000;//Loadi #0 to r
    [1]
instructionsRAM[9] = 32'
    b00000100111000010000000000000000;//ADDi r[1], #0
     to r[7]
instructionsRAM[10] = 32'
    b01100100111000000000000000000110;//Store r[7] in
     m[#6]
instructionsRAM[11] = 32'
    b01100000011000000000000000000110;//Load m[#6] to
     r[3]
instructionsRAM[12] = 32'
    b01100000100000000000000000000111;//Load m[#7] to
     r[4]
instructionsRAM[13] = 32'
    b01011100011001000001100000000000;//SLT if r[4] <
     r[3], r[3] = 1 else r[3] = 0
instructionsRAM[14] = 32'
    b00110100001000110000000000000000;//NOT r[3] to r
    [1]
instructionsRAM[15] = 32'
    b00000100111000010000000000000000;//ADDi r[1], #0
     to r[7]
```

```
instructionsRAM[16] = 32'
    b01111100000001110000000000000000;//Pre Branch r
    [7]
instructionsRAM[17] = 32'
    b01001100000000000000000000011011;//Branch on
    Zero #27
instructionsRAM[18] = 32'
    b01100000011000000000000000000110;//Load m[#6] to
     r[3]
instructionsRAM[19] = 32'
    b01101000100000000000000000000001;//Loadi #1 to r
    [4]
instructionsRAM[20] = 32'
    b01011100011001000001100000000000;//SLT if r[4] <
     r[3], r[3] = 1 else r[3] = 0
instructionsRAM[21] = 32'
    b00110100001000110000000000000000;//NOT r[3] to r
    [1]
instructionsRAM[22] = 32'
    b00000100111000010000000000000000;//ADDi r[1], #0
     to r[7]
instructionsRAM[23] = 32'
    b01111100000001110000000000000000;//Pre Branch r
    [7]
instructionsRAM[24] = 32'
    b01001100000000000000000000000100;//Branch on
    Zero #4
instructionsRAM[25] = 32'
    b01100000011000000000000000000110;//Load m[#6] to
     r[3]
instructionsRAM[26] = 32'
    b00000100111000110000000000000000;//ADDi r[3], #0
     to r[7]
instructionsRAM[27] = 32'
    b01100100111000000000000000001000;//Store r[7] in
     m[#8]
instructionsRAM[28] = 32'
    b01010100000000000000000000101000;//Jump to #40
instructionsRAM[29] = 32'
    b01100000011000000000000000000101;//Load m[#5] to
     r[3]
instructionsRAM[30] = 32'
    b01100000100000000000000000000100;//Load m[#4] to
     r[4]
instructionsRAM[31] = 32'
    b00000000001000110010000000000000;//ADD r[3],r[4]
     to r[1]
instructionsRAM[32] = 32'
    b00000100111000010000000000000000;//ADDi r[1], #0
     to r[7]
instructionsRAM[33] = 32'
    b01100100111000000000000000001000;//Store r[7] in
     m[#8]
instructionsRAM[34] = 32'
    b01100000011000000000000000000100;//Load m[#4] to
     r[3]
instructionsRAM[35] = 32'
    b00000100111000110000000000000000;//ADDi r[3], #0
     to r[7]
instructionsRAM[36] = 32'
```

```
    b01100100111000000000000000000101;//Store r[7] in
     m[#5]
instructionsRAM[37] = 32'
    b01100000011000000000000000001000;//Load m[#8] to
     r[3]
instructionsRAM[38] = 32'
    b00000100111000110000000000000000;//ADDi r[3], #0
     to r[7]
instructionsRAM[39] = 32'
    b01100100111000000000000000000100;//Store r[7] in
     m[#4]
instructionsRAM[40] = 32'
    b01100000011000000000000000000110;//Load m[#6] to
     r[3]
instructionsRAM[41] = 32'
    b00000100001000110000000000000001;//ADDi r[3], #1
     to r[1]
instructionsRAM[42] = 32'
    b00000100111000010000000000000000;//ADDi r[1], #0
     to r[7]
instructionsRAM[43] = 32'
    b01100100111000000000000000000110;//Store r[7] in
     m[#6]
instructionsRAM[44] = 32'
    b01010100000000000000000000001011;//Jump to #11
instructionsRAM[45] = 32'
    b01100000001000000000000000001000;//Load m[#8] to
     r[1]
instructionsRAM[46] = 32'
    b00000111110000010000000000000000;//ADDi r[1], #0
     to r[30]
instructionsRAM[47] = 32'
    b10000100001111110000000000000000;//Loadr m[r
    [31]] to r[1]
instructionsRAM[48] = 32'
    b10001100000000001000000000000000;//Jump to r[1]
instructionsRAM[49] = 32'
    b10000100001111110000000000000000;//Loadr m[r
    [31]] to r[1]
instructionsRAM[50] = 32'
    b10001100000000001000000000000000;//Jump to r[1]
instructionsRAM[51] = 32'
    b01110100001000000000000000000000;//Input to r[1]
instructionsRAM[52] = 32'
    b00000100111000010000000000000000;//ADDi r[1], #0
     to r[7]
instructionsRAM[53] = 32'
    b01100100111000000000000000000010;//Store r[7] in
     m[#2]
instructionsRAM[54] = 32'
    b01100000001000000000000000000010;//Load m[#2] to
     r[1]
instructionsRAM[55] = 32'
    b01100100001000000000000000000111;//Store r[1] in
     m[#7]
instructionsRAM[56] = 32'
    b01101011111000000000000000001010;//Loadi #10 to
    r[31]
instructionsRAM[57] = 32'
    b00000111111111110000000000000001;//ADDi r[31],
```

```
    #1 to r[31]
instructionsRAM[58] = 32'
    b01101000001000000000000000111101;//Loadi #61 to
    r[1]
instructionsRAM[59] = 32'
    b10001000001111110000000000000000;//rStore to r
    [1] in m[r[31]]
instructionsRAM[60] = 32'
    b01010100000000000000000000000010;//Jump to #2
instructionsRAM[61] = 32'
    b00001111111111110000000000000001;//SUBi r[31],
    #1 to r[31]
instructionsRAM[62] = 32'
    b00000100111111100000000000000000;//ADDi r[30],
    #0 to r[7]
instructionsRAM[63] = 32'
    b00000100001001110000000000000000;//ADDi r[7], #0
     to r[1]
instructionsRAM[64] = 32'
    b01111000001000000000000000000000;//Pre Output r
    [1]
instructionsRAM[65] = 32'
    b10000000001000000000000000000000;//Output r[1]
instructionsRAM[66] = 32'
    b10000000001000000000000000000000;//Output r[1]
instructionsRAM[67] = 32'
    b01110000000000000000000000000000;//Hlt
```

Nesse código, na linha 1, se salta para a linha 51, correspondente ao início da função *main*. De 51 até 53, a entrada é guardada em M[7], posição correspondente à variável *n*. A variável *n* é, então, carregada como parâmetro da função *fibonacci* em 54 e 55. Em 56, o topo da pilha de chamadas é adicionado ao registrador 31 e incrementado em 57. Em 58 e 59, o valor da linha para onde deve-se saltar no retorno de função é carregado ao topo da pilha de chamadas. Salta-se para a linha 2, início de *fibonacci*.

Nas linhas 2 até 10 se atribuem os valores às variáveis, *first*, *second* e *c*.

De 11 a 13 se testa a validade para o laço; em 15 e 16, a condição testada condiciona a ramificação para 27 linhas posteriores - com o incremento do PC, linha 45. Nota-se que na linha anterior a essa, há um salto para a linha 11, onde a condição do laço se verifica novamente.

A condição do *if* é testada das linhas 18 a 24, onde a ramificação pode levar ao *else* - linha 29. Caso a ramificação não seja satisfeita, as instruções seguintes à linha 24 são executadas. Nota-se que há um salto justamente na linha 28; este permite que o *else* seja pulado caso o *if* se realize.

As linhas 29 a 33 ilustram uma operação aritmética, onde:

- O valor de M[5] é carregado a R[3];
- O valor de M[4] é carregado a R[4];
- R[1] = R[3] + R[4];
- R[7] = R[1];
- M[8] = R[7].

O valor a ser retornado, presente em M[8], é carregado ao registrador destino da função (R[30]) em 45 e 46. O valor que contém a linha para onde o fluxo deve voltar é carregado

do registrador topo para o registrador 1, e o salto é realizado para 61, que fora carregado antes da chamada da função para esse o registrador topo.

Em 61, o topo da pilha de chamadas é decrementado. Em 62 e 63, o resultado da função é carregado ao registrador 1 e, enfim, de 64 a 67, o valor do registrador 1 é apresentado por *output*.

5.2 Foo

O código de entrada para *Foo* é apresentado a seguir:

```
int vetor[2];
int foo(int x, int y, int z, int w){
    int result;
    result = 0;
    while(result >= 0){
        if(x < 2)
            result = x + y;
        if(y > 2)
            result = result + y + z;
        else
            result = result * 3;
        if(w <= 4){
            if(z == 0)
                return 0;
            if(result != 0)
                return result/(w - 1);
        }
    }
}
void main(void){
    vetor[0] = input();
    vetor[1] = foo(1, vetor[0], 3, 4);
    output(vetor[1]);
}
```

O código gerado a partir de Foo se apresenta a seguir:

```
instructionsRAM[0] = 32'
    b01101100000000000000000000000000;//Nop
instructionsRAM[1] = 32'
    b01010100000000000000000001010101;//Jump to #85
instructionsRAM[2] = 32'
    b01101000001000000000000000000000;//Loadi #0 to r
    [1]
instructionsRAM[3] = 32'
    b00000100111000010000000000000000;//ADDi r[1], #0
     to r[7]
instructionsRAM[4] = 32'
    b01100100111000000000000000000111;//Store r[7] in
     m[#7]
instructionsRAM[5] = 32'
    b01100000011000000000000000000111;//Load m[#7] to
     r[3]
instructionsRAM[6] = 32'
    b01101000100000000000000000000000;//Loadi #0 to r
    [4]
instructionsRAM[7] = 32'
    b01011100001000110010000000000000;//SLT if r[3] <
     r[4], r[1] = 1 else r[1] = 0
```

```
instructionsRAM[8] = 32'
    b00110100001000010000000000000000;//NOT r[1] to r
    [1]
instructionsRAM[9] = 32'
    b00000100111000010000000000000000;//ADDi r[1], #0
     to r[7]
instructionsRAM[10] = 32'
    b01111100000001110000000000000000;//Pre Branch r
    [7]
instructionsRAM[11] = 32'
    b01001100000000000000000001000111;//Branch on
    Zero #71
instructionsRAM[12] = 32'
    b01100000011000000000000000000100;//Load m[#4] to
     r[3]
instructionsRAM[13] = 32'
    b01101000100000000000000000000010;//Loadi #2 to r
    [4]
instructionsRAM[14] = 32'
    b01011100001000110010000000000000;//SLT if r[3] <
     r[4], r[1] = 1 else r[1] = 0
instructionsRAM[15] = 32'
    b00000100111000010000000000000000;//ADDi r[1], #0
     to r[7]
instructionsRAM[16] = 32'
    b01111100000001110000000000000000;//Pre Branch r
    [7]
instructionsRAM[17] = 32'
    b01001100000000000000000000000101;//Branch on
    Zero #5
instructionsRAM[18] = 32'
    b01100000011000000000000000000100;//Load m[#4] to
     r[3]
instructionsRAM[19] = 32'
    b01100000100000000000000000000101;//Load m[#5] to
     r[4]
instructionsRAM[20] = 32'
    b00000000001000110010000000000000;//ADD r[3],r[4]
     to r[1]
instructionsRAM[21] = 32'
    b00000100111000010000000000000000;//ADDi r[1], #0
     to r[7]
instructionsRAM[22] = 32'
    b01100100111000000000000000000111;//Store r[7] in
     m[#7]
instructionsRAM[23] = 32'
    b01100000011000000000000000000101;//Load m[#5] to
     r[3]
instructionsRAM[24] = 32'
    b01101000100000000000000000000010;//Loadi #2 to r
    [4]
instructionsRAM[25] = 32'
    b01011100001001000001100000000000;//SLT if r[4] <
     r[3], r[1] = 1 else r[1] = 0
instructionsRAM[26] = 32'
    b00000100111000010000000000000000;//ADDi r[1], #0
     to r[7]
instructionsRAM[27] = 32'
    b01111100000001110000000000000000;//Pre Branch r
    [7]
```

```
instructionsRAM[28] = 32'
    b01001100000000000000000000001010;//Branch on
    Zero #10
instructionsRAM[29] = 32'
    b01100000011000000000000000000111;//Load m[#7] to
     r[3]
instructionsRAM[30] = 32'
    b01100000100000000000000000000101;//Load m[#5] to
     r[4]
instructionsRAM[31] = 32'
    b00000000001000110010000000000000;//ADD r[3],r[4]
     to r[1]
instructionsRAM[32] = 32'
    b00000100111000010000000000000000;//ADDi r[1], #0
     to r[7]
instructionsRAM[33] = 32'
    b00000100011001110000000000000000;//ADDi r[7], #0
     to r[3]
instructionsRAM[34] = 32'
    b01100000100000000000000000000110;//Load m[#6] to
     r[4]
instructionsRAM[35] = 32'
    b00000000001000110010000000000000;//ADD r[3],r[4]
     to r[1]
instructionsRAM[36] = 32'
    b00000100111000010000000000000000;//ADDi r[1], #0
     to r[7]
instructionsRAM[37] = 32'
    b01100100111000000000000000000111;//Store r[7] in
     m[#7]
instructionsRAM[38] = 32'
    b01010100000000000000000000101100;//Jump to #44
instructionsRAM[39] = 32'
    b01100000011000000000000000000111;//Load m[#7] to
     r[3]
instructionsRAM[40] = 32'
    b01101000100000000000000000000011;//Loadi #3 to r
    [4]
instructionsRAM[41] = 32'
    b00010000001000110010000000000000;//TIMES r[3],r
    [4] to r[1]
instructionsRAM[42] = 32'
    b00000100111000010000000000000000;//ADDi r[1], #0
     to r[7]
instructionsRAM[43] = 32'
    b01100100111000000000000000000111;//Store r[7] in
     m[#7]
instructionsRAM[44] = 32'
    b01100000011000000000000000000011;//Load m[#3] to
     r[3]
instructionsRAM[45] = 32'
    b01101000100000000000000000000100;//Loadi #4 to r
    [4]
instructionsRAM[46] = 32'
    b01011100011001000001100000000000;//SLT if r[4] <
     r[3], r[3] = 1 else r[3] = 0
instructionsRAM[47] = 32'
    b00110100001000110000000000000000;//NOT r[3] to r
    [1]
instructionsRAM[48] = 32'
```

```
    b00000100111000010000000000000000;//ADDi r[1], #0
     to r[7]
instructionsRAM[49] = 32'
    b01111100000001110000000000000000;//Pre Branch r
    [7]
instructionsRAM[50] = 32'
    b01001100000000000000000000011111;//Branch on
    Zero #31
instructionsRAM[51] = 32'
    b01100000011000000000000000000110;//Load m[#6] to
     r[3]
instructionsRAM[52] = 32'
    b01101000100000000000000000000000;//Loadi #0 to r
    [4]
instructionsRAM[53] = 32'
    b01011100001000110010000000000000;//SLT if r[3] <
     r[4], r[1] = 1 else r[1] = 0
instructionsRAM[54] = 32'
    b01011100011001000001100000000000;//SLT if r[4] <
     r[3], r[3] = 1 else r[3] = 0
instructionsRAM[55] = 32'
    b00100100001000010001100000000000;//OR r[1],r[3]
    to r[1]
instructionsRAM[56] = 32'
    b00110100001000010000000000000000;//NOT r[1] to r
    [1]
instructionsRAM[57] = 32'
    b00000100111000010000000000000000;//ADDi r[1], #0
     to r[7]
instructionsRAM[58] = 32'
    b01111100000001110000000000000000;//Pre Branch r
    [7]
instructionsRAM[59] = 32'
    b01001100000000000000000000000011;//Branch on
    Zero #3
instructionsRAM[60] = 32'
    b00000111110000010000000000000000;//ADDi r[1], #0
     to r[30]
instructionsRAM[61] = 32'
    b10000100001111110000000000000000;//Loadr m[r
    [31]] to r[1]
instructionsRAM[62] = 32'
    b10001100000000001000000000000000;//Jump to r[1]
instructionsRAM[63] = 32'
    b01100000011000000000000000000111;//Load m[#7] to
     r[3]
instructionsRAM[64] = 32'
    b01101000100000000000000000000000;//Loadi #0 to r
    [4]
instructionsRAM[65] = 32'
    b01011100001000110010000000000000;//SLT if r[3] <
     r[4], r[1] = 1 else r[1] = 0
instructionsRAM[66] = 32'
    b01011100011001000001100000000000;//SLT if r[4] <
     r[3], r[3] = 1 else r[3] = 0
instructionsRAM[67] = 32'
    b00100100001000010001100000000000;//OR r[1],r[3]
    to r[1]
instructionsRAM[68] = 32'
    b00000100111000010000000000000000;//ADDi r[1], #0
```

```
    to r[7]
instructionsRAM[69] = 32'
    b01111100000001110000000000000000;//Pre Branch r
    [7]
instructionsRAM[70] = 32'
    b01001100000000000000000000001011;//Branch on
    Zero #11
instructionsRAM[71] = 32'
    b01100000011000000000000000000011;//Load m[#3] to
     r[3]
instructionsRAM[72] = 32'
    b00001100001000110000000000000001;//SUBi r[3], #1
     to r[1]
instructionsRAM[73] = 32'
    b00000100111000010000000000000000;//ADDi r[1], #0
     to r[7]
instructionsRAM[74] = 32'
    b01100000011000000000000000000111;//Load m[#7] to
     r[3]
instructionsRAM[75] = 32'
    b00000100100001110000000000000000;//ADDi r[7], #0
     to r[4]
instructionsRAM[76] = 32'
    b00010100001000110010000000000000;//DIVIDE r[3],r
    [4] to r[1]
instructionsRAM[77] = 32'
    b00000100111000010000000000000000;//ADDi r[1], #0
     to r[7]
instructionsRAM[78] = 32'
    b00000100001001110000000000000000;//ADDi r[7], #0
     to r[1]
instructionsRAM[79] = 32'
    b00000111110000010000000000000000;//ADDi r[1], #0
     to r[30]
instructionsRAM[80] = 32'
    b10000100001111110000000000000000;//Loadr m[r
    [31]] to r[1]
instructionsRAM[81] = 32'
    b10001100000000010000000000000000;//Jump to r[1]
instructionsRAM[82] = 32'
    b01010100000000000000000000000101;//Jump to #5
instructionsRAM[83] = 32'
    b10000100001111110000000000000000;//Loadr m[r
    [31]] to r[1]
instructionsRAM[84] = 32'
    b10001100000000010000000000000000;//Jump to r[1]
instructionsRAM[85] = 32'
    b01110100001000000000000000000000;//Input to r[1]
instructionsRAM[86] = 32'
    b00000101000000010000000000000000;//ADDi r[1], #0
     to r[8]
instructionsRAM[87] = 32'
    b01100101000000000000000000001001;//Store r[8] in
     m[#9]
instructionsRAM[88] = 32'
    b01100000001000000000000000001001;//Load m[#9] to
     r[1]
instructionsRAM[89] = 32'
    b00000101000000010000000000000000;//ADDi r[1], #0
     to r[8]
```

```
instructionsRAM[90] = 32'
    b01101000001000000000000000000001;//Loadi #1 to r
    [1]
instructionsRAM[91] = 32'
    b01100100001000000000000000000100;//Store r[1] in
     m[#4]
instructionsRAM[92] = 32'
    b00000100001010000000000000000000;//ADDi r[8], #0
     to r[1]
instructionsRAM[93] = 32'
    b01100100001000000000000000000101;//Store r[1] in
     m[#5]
instructionsRAM[94] = 32'
    b01101000001000000000000000000011;//Loadi #3 to r
    [1]
instructionsRAM[95] = 32'
    b01100100001000000000000000000110;//Store r[1] in
     m[#6]
instructionsRAM[96] = 32'
    b01101000001000000000000000000100;//Loadi #4 to r
    [1]
instructionsRAM[97] = 32'
    b01100100001000000000000000000011;//Store r[1] in
     m[#3]
instructionsRAM[98] = 32'
    b01101011111000000000000000001100;//Loadi #12 to
    r[31]
instructionsRAM[99] = 32'
    b00000111111111110000000000000001;//ADDi r[31],
    #1 to r[31]
instructionsRAM[100] = 32'
    b01101000001000000000000001100111;//Loadi #103 to
     r[1]
instructionsRAM[101] = 32'
    b10001000001111110000000000000000;//rStore to r
    [1] in m[r[31]]
instructionsRAM[102] = 32'
    b01010100000000000000000000000010;//Jump to #2
instructionsRAM[103] = 32'
    b00000111111111110000000000000001;//SUBi r[31],
    #1 to r[31]
instructionsRAM[104] = 32'
    b00000101000111100000000000000000;//ADDi r[30],
    #0 to r[8]
instructionsRAM[105] = 32'
    b01100101000000000000000000001010;//Store r[8] in
     m[#10]
instructionsRAM[106] = 32'
    b01100000001000000000000000001010;//Load m[#10]
    to r[1]
instructionsRAM[107] = 32'
    b00000101000000010000000000000000;//ADDi r[1], #0
     to r[8]
instructionsRAM[108] = 32'
    b00000100001010000000000000000000;//ADDi r[8], #0
     to r[1]
instructionsRAM[109] = 32'
    b01111000001000000000000000000000;//Pre Output r
    [1]
instructionsRAM[110] = 32'
```

```
    b10000000001000000000000000000000;//Output r[1]
instructionsRAM[111] = 32'
    b10000000001000000000000000000000;//Output r[1]
instructionsRAM[112] = 32'
    b01110000000000000000000000000000;//Hlt
```

Esse programa utiliza vetores, testando as duas seguintes condições:

- Atribuição a vetor;
- Passagem de posição de vetor como parâmetro de função.

A atribuição ao vetor ocorre das linhas 85 a 87, onde se realiza a entrada para o registrador 1, seu valor é guardado no registrador temporário 8 e, então, guardado na posição de memória 9.

A passagem do vetor como parâmetro exige atenção para ser observada. Nota-se que nas linhas 88 e 89, o valor de M[9] é carregado ao registrador temporário R[8]. Após, em 90 e 91, o valor 1 ser carregado à posição de memória correspondente à variável na função sendo chamada, o valor de R[8], que contém a posição desejada, é carregado à segunda posição dos parâmetros, enviando corretamente o valor do vetor à função que está sendo chamada.

5.3 Sort

O código de entrada para *Sort* é apresentado a seguir:

```
void sort(int num[], int tam){
    int i;
    int j;
    int min;
    int aux;
    i = 0;
    while (i < tam-1){
        min = i;
        j = i + 1;
        while (j < tam){
            if(num[j] < num[min])
                min = j;
            j = j + 1;
        }
        if (i != min){
            aux = num[i];
            num[i] = num[min];
            num[min] = aux;
        }
        i = i + 1;
    }
}
void main(void){
    int vetor[4];
    int i;
    vetor[0] = 9;
    vetor[1] = 6;
    vetor[2] = 8;
    vetor[3] = 7;
    sort(vetor, 4);
    i = input();
    output(vetor[i]);
}
```

O código gerado pelo compilador para esse programa se apresenta a seguir:

```
instructionsRAM[0] = 32'
    b01101100000000000000000000000000;//Nop
instructionsRAM[1] = 32'
    b01010100000000000000000001010001;//Jump to #81
instructionsRAM[2] = 32'
    b01101000001000000000000000000000;//Loadi #0 to r
    [1]
instructionsRAM[3] = 32'
    b00000100111000010000000000000000;//ADDi r[1], #0
     to r[7]
instructionsRAM[4] = 32'
    b01100100111000000000000000001001;//Store r[7] in
     m[#9]
instructionsRAM[5] = 32'
    b01100000011000000000000000001100;//Load m[#12]
    to r[3]
instructionsRAM[6] = 32'
    b00001100001000110000000000000001;//SUBi r[3], #1
     to r[1]
instructionsRAM[7] = 32'
    b00000100111000010000000000000000;//ADDi r[1], #0
     to r[7]
instructionsRAM[8] = 32'
    b01100000011000000000000000001001;//Load m[#9] to
     r[3]
instructionsRAM[9] = 32'
    b00000100100001110000000000000000;//ADDi r[7], #0
     to r[4]
instructionsRAM[10] = 32'
    b01011100001000110010000000000000;//SLT if r[3] <
     r[4], r[1] = 1 else r[1] = 0
instructionsRAM[11] = 32'
    b00000100111000010000000000000000;//ADDi r[1], #0
     to r[7]
instructionsRAM[12] = 32'
    b01111100000001110000000000000000;//Pre Branch r
    [7]
instructionsRAM[13] = 32'
    b01001100000000000000000001000001;//Branch on
    Zero #65
instructionsRAM[14] = 32'
    b01100000011000000000000000001001;//Load m[#9] to
     r[3]
instructionsRAM[15] = 32'
    b00000100111000110000000000000000;//ADDi r[3], #0
     to r[7]
instructionsRAM[16] = 32'
    b01100100111000000000000000001101;//Store r[7] in
     m[#13]
instructionsRAM[17] = 32'
    b01100000011000000000000000001001;//Load m[#9] to
     r[3]
instructionsRAM[18] = 32'
    b00000100001000110000000000000001;//ADDi r[3], #1
     to r[1]
instructionsRAM[19] = 32'
    b00000100111000010000000000000000;//ADDi r[1], #0
     to r[7]
instructionsRAM[20] = 32'
```

```
    b01100100111000000000000000001010;//Store r[7] in
     m[#10]
instructionsRAM[21] = 32'
    b01100000011000000000000000001010;//Load m[#10]
    to r[3]
instructionsRAM[22] = 32'
    b01100000100000000000000000001100;//Load m[#12]
    to r[4]
instructionsRAM[23] = 32'
    b01011100001000110010000000000000;//SLT if r[3] <
     r[4], r[1] = 1 else r[1] = 0
instructionsRAM[24] = 32'
    b00000100111000010000000000000000;//ADDi r[1], #0
     to r[7]
instructionsRAM[25] = 32'
    b01111100000001110000000000000000;//Pre Branch r
    [7]
instructionsRAM[26] = 32'
    b01001100000000000000000000010110;//Branch on
    Zero #22
instructionsRAM[27] = 32'
    b01100000011000000000000000001010;//Load m[#10]
    to r[3]
instructionsRAM[28] = 32'
    b00000100100000110000000000001110;//ADDi r[3],
    #14 to r[4]
instructionsRAM[29] = 32'
    b10000100001001000000000000000000;//Loadr m[r[4]]
     to r[1]
instructionsRAM[30] = 32'
    b00000100111000010000000000000000;//ADDi r[1], #0
     to r[7]
instructionsRAM[31] = 32'
    b01100000011000000000000000001101;//Load m[#13]
    to r[3]
instructionsRAM[32] = 32'
    b00000100100000110000000000001110;//ADDi r[3],
    #14 to r[4]
instructionsRAM[33] = 32'
    b10000100001001000000000000000000;//Loadr m[r[4]]
     to r[1]
instructionsRAM[34] = 32'
    b00000101000000010000000000000000;//ADDi r[1], #0
     to r[8]
instructionsRAM[35] = 32'
    b00000100011001110000000000000000;//ADDi r[7], #0
     to r[3]
instructionsRAM[36] = 32'
    b00000100100010000000000000000000;//ADDi r[8], #0
     to r[4]
instructionsRAM[37] = 32'
    b01011100001000110010000000000000;//SLT if r[3] <
     r[4], r[1] = 1 else r[1] = 0
instructionsRAM[38] = 32'
    b00000100111000010000000000000000;//ADDi r[1], #0
     to r[7]
instructionsRAM[39] = 32'
    b01111100000001110000000000000000;//Pre Branch r
    [7]
instructionsRAM[40] = 32'
```

```
    b01001100000000000000000000000011;//Branch on
    Zero #3
instructionsRAM[41] = 32'
    b01100000011000000000000000001010;//Load m[#10]
    to r[3]
instructionsRAM[42] = 32'
    b00000100111000110000000000000000;//ADDi r[3], #0
     to r[7]
instructionsRAM[43] = 32'
    b01100100111000000000000000001101;//Store r[7] in
     m[#13]
instructionsRAM[44] = 32'
    b01100000011000000000000000001010;//Load m[#10]
    to r[3]
instructionsRAM[45] = 32'
    b00000100001000110000000000000001;//ADDi r[3], #1
     to r[1]
instructionsRAM[46] = 32'
    b00000100111000010000000000000000;//ADDi r[1], #0
     to r[7]
instructionsRAM[47] = 32'
    b01100100111000000000000000001010;//Store r[7] in
     m[#10]
instructionsRAM[48] = 32'
    b01010100000000000000000000010101;//Jump to #21
instructionsRAM[49] = 32'
    b01100000011000000000000000001001;//Load m[#9] to
     r[3]
instructionsRAM[50] = 32'
    b01100000100000000000000000001101;//Load m[#13]
    to r[4]
instructionsRAM[51] = 32'
    b01011100001000110010000000000000;//SLT if r[3] <
     r[4], r[1] = 1 else r[1] = 0
instructionsRAM[52] = 32'
    b01011100011001000001100000000000;//SLT if r[4] <
     r[3], r[3] = 1 else r[3] = 0
instructionsRAM[53] = 32'
    b00100100001000010001100000000000;//OR r[1],r[3]
    to r[1]
instructionsRAM[54] = 32'
    b00000100111000010000000000000000;//ADDi r[1], #0
     to r[7]
instructionsRAM[55] = 32'
    b01111100000001110000000000000000;//Pre Branch r
    [7]
instructionsRAM[56] = 32'
    b01001100000000000000000000010001;//Branch on
    Zero #17
instructionsRAM[57] = 32'
    b01100000011000000000000000001001;//Load m[#9] to
     r[3]
instructionsRAM[58] = 32'
    b00000100100000110000000000001110;//ADDi r[3],
    #14 to r[4]
instructionsRAM[59] = 32'
    b10000100001001000000000000000000;//Loadr m[r[4]]
     to r[1]
instructionsRAM[60] = 32'
    b00000100111000010000000000000000;//ADDi r[1], #0
```

```
    to r[7]
instructionsRAM[61] = 32'
    b01100100111000000000000000001011;//Store r[7] in
     m[#11]
instructionsRAM[62] = 32'
    b01100000011000000000000000001101;//Load m[#13]
    to r[3]
instructionsRAM[63] = 32'
    b00000100100000110000000000001110;//ADDi r[3],
    #14 to r[4]
instructionsRAM[64] = 32'
    b10000100001001000000000000000000;//Loadr m[r[4]]
     to r[1]
instructionsRAM[65] = 32'
    b00000100111000010000000000000000;//ADDi r[1], #0
     to r[7]
instructionsRAM[66] = 32'
    b01100000011000000000000000001001;//Load m[#9] to
     r[3]
instructionsRAM[67] = 32'
    b00000100100000110000000000001110;//ADDi r[3],
    #14 to r[4]
instructionsRAM[68] = 32'
    b10001000111001000000000000000000;//rStore to r
    [7] in m[r[4]]
instructionsRAM[69] = 32'
    b01100000011000000000000000001011;//Load m[#11]
    to r[3]
instructionsRAM[70] = 32'
    b00000100111000110000000000000000;//ADDi r[3], #0
     to r[7]
instructionsRAM[71] = 32'
    b01100000011000000000000000001101;//Load m[#13]
    to r[3]
instructionsRAM[72] = 32'
    b00000100100000110000000000001110;//ADDi r[3],
    #14 to r[4]
instructionsRAM[73] = 32'
    b10001000111001000000000000000000;//rStore to r
    [7] in m[r[4]]
instructionsRAM[74] = 32'
    b01100000011000000000000000001001;//Load m[#9] to
     r[3]
instructionsRAM[75] = 32'
    b00000100001000110000000000000001;//ADDi r[3], #1
     to r[1]
instructionsRAM[76] = 32'
    b00000100111000010000000000000000;//ADDi r[1], #0
     to r[7]
instructionsRAM[77] = 32'
    b01100100111000000000000000001001;//Store r[7] in
     m[#9]
instructionsRAM[78] = 32'
    b01010100000000000000000000000101;//Jump to #5
instructionsRAM[79] = 32'
    b10000100001111110000000000000000;//Loadr m[r
    [31]] to r[1]
instructionsRAM[80] = 32'
    b10001100000000001000000000000000;//Jump to r[1]
instructionsRAM[81] = 32'
```

```
    b01101000001000000000000000001001;//Loadi #9 to r
    [1]
instructionsRAM[82] = 32'
    b00000100111000010000000000000000;//ADDi r[1], #0
     to r[7]
instructionsRAM[83] = 32'
    b01100100111000000000000000000010;//Store r[7] in
     m[#2]
instructionsRAM[84] = 32'
    b01101000001000000000000000000110;//Loadi #6 to r
    [1]
instructionsRAM[85] = 32'
    b00000100111000010000000000000000;//ADDi r[1], #0
     to r[7]
instructionsRAM[86] = 32'
    b01100100111000000000000000000011;//Store r[7] in
     m[#3]
instructionsRAM[87] = 32'
    b01101000001000000000000000001000;//Loadi #8 to r
    [1]
instructionsRAM[88] = 32'
    b00000100111000010000000000000000;//ADDi r[1], #0
     to r[7]
instructionsRAM[89] = 32'
    b01100100111000000000000000000100;//Store r[7] in
     m[#4]
instructionsRAM[90] = 32'
    b01101000001000000000000000000111;//Loadi #7 to r
    [1]
instructionsRAM[91] = 32'
    b00000100111000010000000000000000;//ADDi r[1], #0
     to r[7]
instructionsRAM[92] = 32'
    b01100100111000000000000000000101;//Store r[7] in
     m[#5]
instructionsRAM[93] = 32'
    b01100000001000000000000000000010;//Load m[#2] to
     r[1]
instructionsRAM[94] = 32'
    b01100000001000000000000000000010;//Load m[#2] to
     r[1]
instructionsRAM[95] = 32'
    b01100100001000000000000000001110;//Store r[1] in
     m[#14]
instructionsRAM[96] = 32'
    b01100000001000000000000000000011;//Load m[#3] to
     r[1]
instructionsRAM[97] = 32'
    b01100100001000000000000000001111;//Store r[1] in
     m[#15]
instructionsRAM[98] = 32'
    b01100000001000000000000000000100;//Load m[#4] to
     r[1]
instructionsRAM[99] = 32'
    b01100100001000000000000000010000;//Store r[1] in
     m[#16]
instructionsRAM[100] = 32'
    b01100000001000000000000000000101;//Load m[#5] to
     r[1]
instructionsRAM[101] = 32'
```

```
    b01100100001000000000000000010001;//Store r[1] in
     m[#17]
instructionsRAM[102] = 32'
    b01100000001000000000000000000110;//Load m[#6] to
     r[1]
instructionsRAM[103] = 32'
    b01100100001000000000000000010010;//Store r[1] in
     m[#18]
instructionsRAM[104] = 32'
    b01101000001000000000000000000100;//Loadi #4 to r
    [1]
instructionsRAM[105] = 32'
    b01100100001000000000000000001100;//Store r[1] in
     m[#12]
instructionsRAM[106] = 32'
    b01101011111000000000000000010100;//Loadi #20 to
    r[31]
instructionsRAM[107] = 32'
    b00000111111111111000000000000001;//ADDi r[31],
    #1 to r[31]
instructionsRAM[108] = 32'
    b01101000001000000000000001101111;//Loadi #111 to
     r[1]
instructionsRAM[109] = 32'
    b10001000001111111000000000000000;//rStore to r
    [1] in m[r[31]]
instructionsRAM[110] = 32'
    b01010100000000000000000000000010;//Jump to #2
instructionsRAM[111] = 32'
    b00000111111111111000000000000001;//SUBi r[31],
    #1 to r[31]
instructionsRAM[112] = 32'
    b01100000001000000000000000001110;//Load m[#14]
    to r[1]
instructionsRAM[113] = 32'
    b01100100001000000000000000000010;//Store r[1] in
     m[#2]
instructionsRAM[114] = 32'
    b01100000001000000000000000001111;//Load m[#15]
    to r[1]
instructionsRAM[115] = 32'
    b01100100001000000000000000000011;//Store r[1] in
     m[#3]
instructionsRAM[116] = 32'
    b01100000001000000000000000010000;//Load m[#16]
    to r[1]
instructionsRAM[117] = 32'
    b01100100001000000000000000000100;//Store r[1] in
     m[#4]
instructionsRAM[118] = 32'
    b01100000001000000000000000010001;//Load m[#17]
    to r[1]
instructionsRAM[119] = 32'
    b01100100001000000000000000000101;//Store r[1] in
     m[#5]
instructionsRAM[120] = 32'
    b01100000001000000000000000010010;//Load m[#18]
    to r[1]
instructionsRAM[121] = 32'
    b01100100001000000000000000000110;//Store r[1] in
```

```
    m[#6]
instructionsRAM[122] = 32'
    b01110100001000000000000000000000;//Input to r[1]
instructionsRAM[123] = 32'
    b00000100111000010000000000000000;//ADDi r[1], #0
     to r[7]
instructionsRAM[124] = 32'
    b01100100111000000000000000000111;//Store r[7] in
     m[#7]
instructionsRAM[125] = 32'
    b01100000011000000000000000000111;//Load m[#7] to
     r[3]
instructionsRAM[126] = 32'
    b00000100100000110000000000000010;//ADDi r[3], #2
     to r[4]
instructionsRAM[127] = 32'
    b10000100001001000000000000000000;//Loadr m[r[4]]
     to r[1]
instructionsRAM[128] = 32'
    b00000100111000010000000000000000;//ADDi r[1], #0
     to r[7]
instructionsRAM[129] = 32'
    b00000100001001110000000000000000;//ADDi r[7], #0
     to r[1]
instructionsRAM[130] = 32'
    b01111000001000000000000000000000;//Pre Output r
    [1]
instructionsRAM[131] = 32'
    b10000000001000000000000000000000;//Output r[1]
instructionsRAM[132] = 32'
    b10000000001000000000000000000000;//Output r[1]
instructionsRAM[133] = 32'
    b01110000000000000000000000000000;//Hlt
```

Esse programa acrescenta a passagem de um vetores como parâmetros e o carregamento de posições de vetores especificadas em variáveis.

Se observa o carregamento dos parâmetros do vetor para a função chamada nas linhas de 94 até 103, onde os valores do vetor em *main* são passados aos valores do vetor em *sort*. Eles são carregados de volta ao vetor presente na *main* nas linhas de 112 até 121.

O carregamento de posição de vetor especificada em variável é feito para o *output*, e é obtido nas linhas de 125 a 128. Nelas, o valor de i é guardado em R[3], adicionado ao deslocamento correspondente à posição do vetor e guardado em R[4], o valor na posição de memória correspondente ao que está em R[4] é guardado em R[1] e, então, atribuído ao registrador temporário R[7], de onde se carrega o valor posteriormente apresentado como saída.

5.4 Testes

Todos os exemplos apresentados foram testados no processador alvo [7]. Os testes se realizaram tanto através de forma de onda quanto no próprio *kit FPGA*. Os valores de saída obtidos a partir dos valores de entrada corresponderam todos aos valores esperados para cada uma das funções, os quais foram conferidos através da adaptação para linguagem C, compilação com o compilador gcc [9] e teste.

Observa-se que o programa é executado corretamente a partir de ser adicionado à placa. Caso seja *resetado*, o sistema não realiza as operações corretamente. Portanto, deve-se verificar a função *Hlt* e como o reinício do processador está sendo efetuado para

que os códigos possam ser executados por mais de uma vez após a passagem para o FPGA.

6 Conclusões

A realização deste projeto permitiu a compreensão de início a fim do processo de compilação de um código em alto nível. Contribuiu para o desenvolvimento do aluno na resolução de problemas complexos, práticas de desenvolvimento e integração entre projetos, de maneira que a arquitetura do processador precisou de ajustes para que o compilador pudesse funcionar. Compreendeu-se, também, quais instruções em um processador são essenciais para o desenvolvimento de um código de alto nível, e quais são desnecessárias. Assim, do processador ao compilador, se vê um projeto só. Foi aprendida, também, a importância da flexibilidade de implementação do planejamento, uma vez que as etapas posteriores sempre dependeram das etapas anteriores.

As principais dificuldades encontradas na elaboração do projeto foram:

- Para o código intermediário: as estruturas necessárias para a implementação dessa etapa haviam sido implementadas de forma incompleta em etapas anteriores. Portanto, foi necessário reestruturar a árvore de análise sintática, o que demandou bastante tempo de trabalho.

- Para o código objeto: foram necessárias informações advindas de todas as etapas anteriores do projeto. Portanto, alterações foram realizadas em diversas partes dos códigos anteriores - e até mesmo do processador - para que este pudesse ser gerado.

- A falta de experiência no desenvolvimento de projetos dessa complexidade - sobretudo compiladores - gerou diversas dificuldades e, ao mesmo tempo, muito acrescentou em aprendizado.

Oportunidades de melhoria ainda existem na parte de otimização de código, assim como na correção da função de reinicio do processador, por fins de praticidade.

Referências

[1] D. Compiletools. Flex. http://dinosaur.compilertools.net/flex/manpage.html. Acesso em 24-03-2017.

[2] D. Compiletools. Yacc. http://dinosaur.compilertools.net/yacc/. Acesso em 24-03-2017.

[3] GNU. Bison. https://www.gnu.org/software/bison/. Acesso em 24-03-2017.

[4] P. Magazine. Compiler definition. http://www.pcmag.com/encyclopedia/term/40105/compiler. Acesso em 24-03-2017.

[5] K. C. Mano Morris. *Arquitetura e organização de computadores: projeto para o desempenho*. PEARSON, 4 edition, 2008.

[6] J. D. Marangon. Compiladores para humanos - a estrutura de um compilador. https://johnidm.gitbooks.io/compiladores-para-humanos/content/part1/structure-of-a-compiler.html. Acesso em 24-03-2017.

[7] D. Morales. Galetron - a functional cpu designed in verilog. https://github.com/davimmorales/processor-galetron. Acesso em 24-06-2017.

[8] T. Point. Compiler design - lexical analysis. https://www.tutorialspoint.com/compiler_design/compiler_design_lexical_analysis.htm. Acesso em 24-03-2017.

[9] O. S. Project. Gcc, the gnu compiler collection. https://gcc.gnu.org/. Acesso em 24-06-2017.

[10] J. Regehr. Why take a compilers course? http://blog.regehr.org/archives/169. Acesso em 24-03-2017.

[11] M. Tekwani. Phases of the compiler systems. https://pt.slideshare.net/mukeshnt/phases-of-the-compilersystems. Acesso em 24-03-2017.

[12] S. University. Semantic analysis. https://web.stanford.edu/class/archive/cs/cs143/cs143.1128/handouts/180%20Semantic%20Analysis.pdf. Acesso em 24-03-2017.

7 Apêndice

A seguir estão os códigos fontes escritos para este projeto.

Algoritmo 1: cminus.y - Análise Sintática

```
%{
  #define YYPARSER    /* distinguishes Yacc output from other code
     files */

//GLC para gerar parser para C-

#include "globals.h"
#include "util.h"
#include "scan.h"
#include "parse.h"
/*#include "cminus.tab.h"*/



/*#define YYDEBUG 0*/

#define YYSTYPE TreeNode *
static TreeNode * savedTree;    /* stores syntax tree for later
   return */
static int yylex(void);

extern char* yytext;
extern int yylineno;
FILE *arq;

void yyerror(char*);
%}

%start programa
%token INT
%token FLOAT
%token IF
%token ELSE
%token RETURN   304
%token VOID     305
%token WHILE    306
%token PLUS     307
%token MINUS    308
%token TIMES    309
%token OVER     310
%token LT                311
%token LET          312
%token HT                313
%token HET          314
%token EQ                315
%token NEQ               316
%token ASSIGN   317
%token SEMI       318
%token COMMA      319
%token LPAREN   320
%nonassoc RPAREN         321
%token LBRACK   322
%token RBRACK   323
```

```
%token LCAPSULE 324
%token RCAPSULE 325
%token NUM          326
%token ID                   328
%token NEWLINE  329
%token ERROR    331

%%

programa        :       /* Entrada Vazia */
                                        | declaration-list
          {
            /*$$ = allocateNode("programa");
            addChild($$,$1);*/
            savedTree = $1;
          }
                                        ;

declaration-list        : declaration-list declaration
                  { YYSTYPE t = $1;
                    if(t != NULL){
                      while(t->sibling != NULL)
                        t = t->sibling;
                      t->sibling = $2;
                      $$ = $1;
                    }
                    else
                      $$ = $2;
                    }
                    /*$$ = allocateNode("declaration-list");
                   addChild($$,$1);
                   addChild($$,$2);}*/
                                                                        |

                                                                            dec


                   {
                     $$ = $1;
                     /*$$ = allocateNode("declaration-list");*/
                    /*addChild($$,$1);*/
                  }
                                                                        ;

declaration             :   variable_declaration
                                {
                                    $$ = $1;
                                }
                        |   function_declaration
                                {
                                    $$ = $1;
                                }
                        ;
variable_declaration    :   INT id SEMI
                                {
                                    $$ = newExpNode(TypeK);
                                    $$->type = Integer;
                                    $$->child[0] = $2;
                                    $2->nodekind = StmtK;
                                    $2->kind.stmt = VarK;
                                    $2->type = Integer;
```

```
                        }
                    |   INT id LBRACK num RBRACK SEMI
                        {
                            $$ = newExpNode(TypeK);
                            $$->type = Integer;
                            $$->child[0] = $2;
                            $2->nodekind = StmtK;
                            $2->kind.stmt = VecK;
                            $2->attr.value = $4->attr.value;
                            $2->type = Integer;
                        }
                    ;
function_declaration    :   INT id LPAREN parameters RPAREN
   compound_declaration
                        {
                            $$ = newExpNode(TypeK);
                            $$->type = Integer;
                            $$->child[0] = $2;
                            $2->nodekind = StmtK;
                            $2->kind.stmt = FuncK;
                            $2->child[0] = $4;
                            $2->child[1] = $6;
                            $2->type = Integer;
                            setScope($2->child[0], $2->attr.
                               name);
                            setScope($2->child[1], $2->attr.
                               name);
                        }
                    |   VOID id LPAREN parameters RPAREN
                       compound_declaration
                        {
                            $$ = newExpNode(TypeK);
                            $$->type = Void;
                            $$->child[0] = $2;
                            $2->nodekind = StmtK;
                            $2->kind.stmt = FuncK;
                            $2->child[0] = $4;
                            $2->child[1] = $6;
                            $2->type = Void;
                            setScope($2->child[0], $2->attr.
                               name);
                            setScope($2->child[1], $2->attr.
                               name);
                        }
                    ;
parameters              :   list_parameters
                        {
                            $$ = $1;
                        }
                    |   VOID
                        {
                            $$ = NULL;
                        }
                    ;
list_parameters         :   list_parameters COMMA parameter
                        {
                            YYSTYPE t = $1;
                            if(t != NULL){
                                while(t->sibling != NULL)
                                    t = t->sibling;
```

```
                                t->sibling = $3;
                                $$ = $1;
                            }
                            else
                                $$ = $3;
                        }
                    |   parameter
                        {
                            $$ = $1;
                        }
                    ;
parameter           :   INT id
                        {
                            $$ = newExpNode(TypeK);
                            $$->type = Integer;
                            $$->child[0] = $2;
                            $2->nodekind = StmtK;
                            $2->kind.stmt = FuncVarK;
                            $2->type = Integer;
                        }
                    |   INT id LBRACK RBRACK
                        {
                            $$ = newExpNode(TypeK);
                            $$->type = Integer;
                            $$->child[0] = $2;
                            $2->nodekind = StmtK;
                            $2->kind.stmt = FuncVecK;
                            $2->type = Integer;
                        }
                    ;
compound_declaration    :   LCAPSULE local_declarations
   list_statement RCAPSULE
                        {
                            YYSTYPE t = $2;
                            if(t != NULL){
                                while(t->sibling != NULL)
                                    t = t->sibling;
                                t->sibling = $3;
                                $$ = $2;
                            }
                            else
                                $$ = $3;
                        }
                    |   LCAPSULE local_declarations RCAPSULE
                        {
                            $$ = $2;
                        }
                    |   LCAPSULE list_statement RCAPSULE
                        {
                            $$ = $2;
                        }
                    |   LCAPSULE RCAPSULE
                        {}
                    ;
local_declarations      :   local_declarations variable_declaration
                        {
                            YYSTYPE t = $1;
                            if(t != NULL){
                                while(t->sibling != NULL)
                                    t = t->sibling;
```

```
                                t->sibling = $2;
                                $$ = $1;
                            }
                            else
                                $$ = $2;
                        }
                    |   variable_declaration
                        {
                            $$ = $1;
                        }
                    ;
list_statement      :   list_statement statement
                        {
                            YYSTYPE t = $1;
                            if(t != NULL){
                                while(t->sibling != NULL)
                                    t = t->sibling;
                                t->sibling = $2;
                                $$ = $1;
                            }
                            else
                                $$ = $2;
                        }
                    |   statement
                        {
                            $$ = $1;
                        }
                    ;
statement           :   expression_declaration
                        {
                            $$ = $1;
                        }
                    |   compound_declaration
                        {
                            $$ = $1;
                        }
                    |   selection_declaration
                        {
                            $$ = $1;
                        }
                    |   iteration_declaration
                        {
                            $$ = $1;
                        }
                    |   return_declaration
                        {
                            $$ = $1;
                        }
                    ;
expression_declaration  :   expression SEMI
                        {
                            $$ = $1;
                        }
                    |   SEMI
                        {}
                    ;
selection_declaration   :   IF LPAREN expression RPAREN statement
                        {
                            $$ = newStmtNode(IfK);
                            $$->child[0] = $3;
```

```
                                $$->child[1] = $5;
                            }
                        |   IF LPAREN expression RPAREN statement
                            ELSE statement
                            {
                                $$ = newStmtNode(IfK);
                                $$->child[0] = $3;
                                $$->child[1] = $5;
                                $$->child[2] = $7;
                            }
;
iteration_declaration   :   WHILE LPAREN expression RPAREN statement
                            {
                                $$ = newStmtNode(WhileK);
                                $$->child[0] = $3;
                                $$->child[1] = $5;
                            }
                        ;
return_declaration      :   RETURN SEMI
                            {
                                $$ = newStmtNode(ReturnK);
                                $$->type = Void;
                            }
                        |   RETURN expression SEMI
                            {
                                $$ = newStmtNode(ReturnK);
                                $$->child[0] = $2;
                                $$->type = $2->type;
                            }
                        ;
expression              :   variable ASSIGN expression
                            {
                                $$ = newStmtNode(AssignK);
                                $$->child[0] = $1;
                                $$->child[1] = $3;
                            }
                        |   simple_expression
                            {
                                $$ = $1;
                            }
                        ;
variable                :   id
                            {
                                $$ = $1;
                                $$->type = Integer;
                            }
                        |   id LBRACK expression RBRACK
                            {
                                $$ = $1;
                                $$->kind.exp = VecIndexK;
                                $$->child[0] = $3;
                                $$->type = Integer;
                            }
                        ;
simple_expression       :   plus_minus_expression relational_operator
    plus_minus_expression
                            {
                                $$ = $2;
                                $$->child[0] = $1;
                                $$->child[1] = $3;
```

```
                        }
                    |   plus_minus_expression
                        {
                            $$ = $1;
                        }
                    ;
relational_operator :   EQ
                        {
                            $$ = newExpNode(RelOpK);
                            $$->attr.oprtr = EQ;
                            $$->attr.name = "==";
                            $$->type = Boolean;
                        }
                    |   NEQ
                        {
                            $$ = newExpNode(RelOpK);
                            $$->attr.oprtr = NEQ;
                            $$->attr.name = "!=";
                            $$->type = Boolean;
                        }
                    |   LT
                        {
                            $$ = newExpNode(RelOpK);
                            $$->attr.oprtr = LT;
                            $$->attr.name = "<";
                            $$->type = Boolean;
                        }
                    |   LET
                        {
                            $$ = newExpNode(RelOpK);
                            $$->attr.oprtr = LET;
                            $$->attr.name = "<=";
                            $$->type = Boolean;
                        }
                    |   HT
                        {
                            $$ = newExpNode(RelOpK);
                            $$->attr.oprtr = HT;
                            $$->attr.name = ">";
                            $$->type = Boolean;
                        }
                    |   HET
                        {
                            $$ = newExpNode(RelOpK);
                            $$->attr.oprtr = HET;
                            $$->attr.name = ">=";
                            $$->type = Boolean;
                        }
                    ;
plus_minus_expression   :   plus_minus_expression plus_minus term
                        {
                            $$ = $2;
                            $$->child[0] = $1;
                            $$->child[1] = $3;
                        }
                    |   term
                        {
                            $$ = $1;
                        }
                    ;
```

```
plus_minus              :   PLUS
                            {
                                $$ = newExpNode(ArithOpK);
                                $$->attr.oprtr = PLUS;
                                $$->attr.name = "+";
                                $$->type = Integer;
                            }
                        |   MINUS
                            {
                                $$ = newExpNode(ArithOpK);
                                $$->attr.oprtr = MINUS;
                                $$->attr.name = "-";
                                $$->type = Integer;
                            }
                        ;
term                    :   term times_over factor
                            {
                                $$ = $2;
                                $$->child[0] = $1;
                                $$->child[1] = $3;
                            }
                        |   factor
                            {
                                $$ = $1;
                            }
                        ;
times_over              :   TIMES
                            {
                                $$ = newExpNode(ArithOpK);
                                $$->attr.oprtr = TIMES;
                                $$->attr.name = "*";
                                $$->type = Integer;
                            }
                        |   OVER
                            {
                                $$ = newExpNode(ArithOpK);
                                $$->attr.oprtr = OVER;
                                $$->attr.name = "/";
                                $$->type = Integer;
                            }
                        ;
factor                  :   LPAREN expression RPAREN
                            {
                                $$ = $2;
                            }
                        |   variable
                            {
                                $$ = $1;
                            }
                        |   function_call
                            {
                                $$ = $1;
                            }
                        |   num
                            {
                                $$ = $1;
                            }
                        ;
function_call           :   id LPAREN list_arguments RPAREN
                            {
```

```
                                $$ = $1;
                                $$->child[0] = $3;
                                $$->kind.exp = CallK;
                            }
                        |   id LPAREN RPAREN
                            {
                                $$ = $1;
                                $$->kind.exp = CallK;
                            }
                        ;
list_arguments          :   list_arguments COMMA expression
                            {
                                YYSTYPE t = $1;
                                if(t != NULL){
                                    while(t->sibling != NULL)
                                        t = t->sibling;
                                    t->sibling = $3;
                                    $$ = $1;
                                }
                                else
                                    $$ = $3;
                            }
                        |   expression
                            {
                                $$ = $1;
                            }
                        ;
id                      :   ID
                            {
                                $$ = newExpNode(IdK);
                                $$->attr.name = copyString(
                                    tokenString);
                            }
                        ;
num                     :   NUM
                            {
                                $$ = newExpNode(ConstK);
                                $$->attr.value = atoi(tokenString
                                    );
                                $$->type = Integer;
                            }
                        ;

%%

/*TreeNode * allocateToken(char const *token)
{
        //reset(strExp);
        //strncpy(strExp, yytext, sizeof(strExp));
  strExp.erase();
  strExp = yytext;
        TreeNode *branch = allocateNode(token);
        //puts( yytext );
//      sprintf(strExp, "%s", yytext);
        TreeNode *leaf = allocateNode(strExp.c_str());//because it
            must be a char const*
  // TreeNode *leaf = allocateNode("galeto");
//      addChild(branch, leaf);
        return branch;
}*/
```

```
/*TreeNode * allocateNode(char const *node)
{

  if(line_flag==0){
    reference_line = yylineno;
    line_flag = 1;
  }

        TreeNode *newNode = (TreeNode*)malloc(sizeof(TreeNode));
        newNode->lineno = yylineno - reference_line;
  /*newNode->name = strcpy(newNode->name, yylex());*/

/*      newNode->str = (char*) calloc(sizeof(char),20);
        strcpy(newNode->str, node);

        newNode->child = NULL;
        newNode->sibling = NULL;
  /*printf("alocou no str->%s\n", newNode->str);*/
        /*return newNode;
}*/

/*TreeNode* addSibling(TreeNode* first, TreeNode* newSibling){
        if(first->sibling == NULL){first->sibling = newSibling;}
        else{first->sibling = addSibling(first->sibling, newSibling)
           ;}
        return first;
}*/

/*TreeNode* addChild(TreeNode* node, TreeNode* childNode){
        if(node->child!=NULL){node->child = addSibling(node->child,
           childNode);}
        else{node->child = childNode;}
        return node;
}*/

/*TreeNode* freeTree(TreeNode * tree){
        if(tree != NULL)
        {
                if(tree->sibling != NULL){tree->sibling = freeTree(
                   tree->sibling);}

                if(tree->child != NULL){tree->child = freeTree(tree->
                   child);}

                if(tree->child == NULL && tree->sibling == NULL)
                {
                        free(tree);
                        return NULL;
                }
        }
}*/

/* printSpaces indents by printing spaces */
/*static void printSpaces(void)
{
        int i;
        for (i=0;i<indentNo;i++)
        {
                //fprintf(arq, "        ");
```

```
                printf("_");
        }
}*/


/* procedure printTree prints a syntax tree to the
 * listing file using indentation to indicate subtrees
 */
/*void effPrintTree(TreeNode * tree)
{
        INDENT;
        while (tree != NULL)
        {
                printSpaces();
                //fprintf(arq, "%s\n",tree->str);
    printf("%s %d\n",tree->str, tree->lineno);

                tree = tree->child;

                while(tree != NULL)
                {
                        effPrintTree(tree);
                        tree = tree->sibling;
                }
                UNINDENT;
        }
}*/

/*void printTree()
{
  printf("Imprimindo arvore sintatica...\n");
        arq = fopen("syntaticTree.xls", "w");
        effPrintTree(tree);
        fclose(arq);
}*/



void yyerror (char* s)  /* Called by yyparse on error */
{
        extern char* yytext;
        /*cout << s << ": " << yytext << endl << "At line: " <<
           yylineno <<  endl;*/
  printf("Syntax error at line %d\n", yylineno);
  printToken(yychar, tokenString);

  /*strExp = (char*) calloc(sizeof(char),40);*/
}

static int yylex(void){
    return getToken();
}

TreeNode * parse(void){
    yyparse();
    return savedTree;
}
```

Algoritmo 2: cminus.l - Análise Léxica

```
/****************************************************/
/* File: cminus.l                                   */
/* Lex specification for C- and Table Generator     */
/* Trabalho Pratico de Compiladores                 */
/* Davi Morales and Mateus Franco                   */
/****************************************************/
%{

  #include "globals.h"
  #include "util.h"
  #include "scan.h"
  #include "parse.h"
  /*#include "intermediate.h"*/
  #include "object.h"


char tokenString[MAXTOKENLEN+1];


void abrirArq();

/*void printTree();*/

int lineno = 1;
int linenumber = 0;


%}

digit       [0-9]
number      {digit}+
letter      [a-zA-Z]
identifier  {letter}+{letter}*
newline     \n
whitespace  [ \t\r]+
other    [^0-9a-zA-Z;/=\-"+""*""(")""\n"\[\]\,\{\}\<\>|!=\==\<=\>=]
%option yylineno
%%

"int"                 {/*printf("INT ");*/ return INT;}
"float"             {/*printf("FLOAT ");*/ return FLOAT;}
"if"          {/*printf("IF ");*/ return IF;}
"else"              {/*printf("ELSE ");*/ return ELSE;}
"return"            {/*printf("RETURN ");*/ return RETURN;}
"void"              {/*printf("VOID ");*/return VOID;}
"while"             {/*printf("WHILE ");*/ return WHILE;}
"+"                           {/*printf("PLUS ");*/ return PLUS;}
"-"                           {/*printf("MINUS ");*/ return MINUS;}
"*"                           {/*printf("TIMES ");*/ return TIMES;}
"<"                           {/*printf("LT ");*/ return LT;}
"/"                           {/*printf("OVER ");*/ return OVER;}
"<="          {/*printf("LET ");*/ return LET;}
">"                           {/*printf("HT ");*/ return HT;}
">="          {/*printf("HET ");*/ return HET;}
"=="          {/*printf("EQ ");*/ return EQ;}
"!="          {/*printf("NEQ ");*/ return NEQ;}
"="                           {/*printf("ASSIGN ");*/ return ASSIGN;}
";"                           {/*printf("SEMI ");*/ return SEMI;}
","                           {/*printf("COMMA ");*/ return COMMA;}
")"           {/*printf("RPAREN ");*/ return RPAREN;}
```

```
"("                             {/*printf("LPAREN ");*/ return LPAREN;}
"]"                             {/*printf("RBRACK ");*/ return RBRACK;}
"["                             {/*printf("LBRACK ");*/ return LBRACK;}
"{"                             {/*printf("LCAPSULE ");*/ return
    LCAPSULE;}
"}"                             {/*printf("RCAPSULE ");*/ return
    RCAPSULE;}
"/*"            {       char c, d;
                        c = input();
                        do
                        {       if (c == EOF) break;
                                d = input();
                                if (c == '\n')
                                        lineno++;
                                if (c == '*' && d == '/') break;
                                c = d;
                    } while (1);
                  }
                {newline}       { lineno++; /*printf("\t%d\n", lineno);
                    */}
                {whitespace}    ;
                {number}        { /*printf("NUM ");* strcpy(id,yytext);
                    */ return NUM;}
                {identifier}    {/*printf("ID "); strcpy(id,yytext);*/
                    return ID;}
<<EOF>>           return(0);
{other}           { printf("Lexical Error at line %d\n", lineno);}

%%

int yywrap() {
  return 1;
}

TokenType getToken(void){
  static int firstTime = TRUE;
  TokenType current;
  if(firstTime){
    firstTime = FALSE;
    linenumber++;
    FILE * file;
    yyin = fopen("entrada.txt", "r");
  }
  current = yylex();
  strncpy(tokenString, yytext, MAXTOKENLEN);

  /*printf("\t%d", linenumber);*/
  /*printf("%d %s %d\n", current, tokenString, linenumber);*/
  /*printToken(current, tokenString); this is the one*/
  return current;
}

/*may be useful someday*/
/*static void traverse(TreeNode * t, void (* preProc) (TreeNode *),
    void (* postProc) (TreeNode *)){
     if(t != NULL){
         preProc(t);
         {
             int i;
             for(i = 0; i < MAXCHILDREN; ++i)
```

```
                traverse(t->child[i], preProc, postProc);
            }
            postProc(t);
            traverse(t->sibling, preProc, postProc);
        }
}*/

char L_side[20];
char R_side[20];
int verState = 0;
int voidFlag = 0;
int arrayFlag = 0;



/*typedef struct TipoID{
     char nomeID[20];
     char tipoID[20];
     char tipoData[10];
     char escopo[30];
     int linhas[50];
     int top;
     struct TipoID *prox;
}TipoID;*/

typedef struct TypeSync{
  char name[45];
  char scope[45];
  int type;//where 1 stands for var, 2 for array and 3 for function
  int value;
  int top;
  struct TypeSync *next;
}TypeSync;

typedef struct{
  TypeSync *start;
}SyncList;

void inicializaLista(TipoLista *lista){
    lista->start = NULL;
}

int contaChar(const char *str)
{
    int i = 0;
    for(;str[i] != 0; ++i);
    return i;
}

int string2int(const char *num)
{
  int  i, len, a;
  int result=0;
  len = contaChar(num);
  for(i=0; i<len; i++){
    result = result * 10 + ( num[i] - '0' );
  }
  return result;
}
```

```
void insert(SyncList *list, char scope[], char nameID[], char typeID
    []){
  TypeSync *new_node = malloc(sizeof(TypeSync));
  if(!strcmp(typeID,"func")){
    new_node->type = 3;
    strcpy(new_node->scope, "none");
  }
  else if(!strcmp(typeID,"var")){
    new_node->type = 1;
    strcpy(new_node->scope, scope);
  }
  else{
    new_node->type = 2;
    strcpy(new_node->scope, scope);

  }
  strcpy(new_node->name,nameID);
  TypeSync *p = list->start;
  if(p==NULL)
    list->start = new_node;
  else{
    while(p->next!=NULL)
      p = p->next;
    p->next = new_node;
    /*printf("%s\n", ->name);*/
  }
}


void insere(TipoLista *lista, char scope[], char nameID[], char
    typeID[], char typeData[], int nline, int index, int size)
{
    // Alocaaao do na que sera indexado
    TipoID *novoNo = malloc(sizeof(TipoID));
    // Inicializaaao do vetor de linhas
    int i;
    for(i=0;i<50;i++) {
      novoNo->linhas[i] = 0;
    }
    // Atribuiaao da linha na posiaao inicial

    if(strcmp(nameID,"input") == 0 || strcmp(nameID,"output") == 0) {
      novoNo->linhas[0] = 0;
      novoNo->top = 0;
    } else {
      novoNo->linhas[0] = nline;
      novoNo->top = 1; // proxima posiaao de inserir numero da linha
    }

    novoNo->intermediate_start = 0;
    novoNo->intermediate_finish = 0;
    novoNo->indice_parametro = 0;

    // Inicializaaao dos demais campos do na com os parametros de
        entrada
    if(!strcmp(typeID,"func")) {
      strcpy(novoNo->escopo, "none");
      /*strcpy(novoNo->tipoData, "none");*/
    } else {
      strcpy(novoNo->escopo, scope);
```

```
        /*strcpy(novoNo->tipoData, typeData);*/
    }
    strcpy(novoNo->tipoData, typeData);
    strcpy(novoNo->nomeID, nameID);
    strcpy(novoNo->tipoID, typeID);
    if(!strcmp(typeID,"vet"))//&&!strcmp(scope,"global"))
        novoNo->array_size = size;
    /*printf("%s\n", novoNo->escopo);*/
    TipoID *p = lista[index].start;

    if(p == NULL) { // Lista vazia
        lista[index].start = novoNo;
    } else {  // Lista nao vazia. Insere no final
        while(p->prox!=NULL){
            p = p->prox;
        }
        p->prox = novoNo;
    }
}

int checkExistance(TipoLista *Lista, char s[], int noline, int index,
    char scope[], int flag){
    int i;
    TipoID *c = Lista[index].start;
    if(flag)
      return 0;
    while(c!=NULL){
        if(!strcmp(s,c->nomeID)){
          if(!strcmp(scope,c->escopo)||!strcmp(c->tipoID, "func")||!
              strcmp(c->escopo, "global")){
          for(i=0;i<c->top;i++){
            if(c->linhas[i]==noline)
              return 1;
          }
          c->linhas[c->top] = noline;
          c->top++;
          return 1;
        }
      }
        c = c->prox;
    }
    return 0;
}

void printWTable(TipoLista *lista, int index) {
  int i;
  TipoID *p = lista[index].start;
  while(p!=NULL){
      i = 0;
      if(p->linhas[0] != 0) {
        printf("%6s    %6s    %6s    %6s        ", p->nomeID, p->
            tipoID, p->tipoData, p->escopo);
        if(!strcmp(p->tipoID, "func"))
          printf("     %d, %d \t \t ", p->intermediate_start, p->
              intermediate_finish);
        else
          printf("\t \t \t ");
        while(p->linhas[i]!=0){
          printf("%d", p->linhas[i]);
          if(i<p->top-1)
```

```
            printf(",");
          i++;
        }
        if(!strcmp(p->tipoID,"vet"))//&&!strcmp(p->escopo,"global"))
          printf("\t array size: %d", p->array_size);

          printf("\t \t parameter: %d\n", p->indice_parametro);

        printf("\n");
      }
        p = p->prox;
  }
}

void printIDList(SyncList *list) {
  TypeSync *p = list->start;
  while(p!=NULL){
      printf("%s %s %d\n", p->name, p->scope, p->type);
      p = p->next;
  }
}

void abrirArq()
{
  yyin = fopen("entrada.txt", "r");

}

/*Semantic Analysis functions*/

/*Checks existance of a given variable*/
int buscaVariavel(TipoLista *list, char nomeID[], char escopo[]) {
    int hash = string2int(nomeID)%211;
    TipoID *c = list[hash].start;
    while(c != NULL) {
      if((!strcmp(nomeID, c->nomeID)) && (!strcmp(escopo, c->escopo))
          || (!strcmp("global", c->escopo))) {
        return 1;
      }
      c = c->prox;
    }
    if(c == NULL) {
      /*printf("\nVariavel %s nao encontrada no escopo %s\n", nomeID,
          escopo);*/
      return 0;
    }
}

int functionType(TipoLista *list, char nomeID[]){
  int hash = string2int(nomeID)%211;
  TipoID *c = list[hash].start;
  while(c != NULL){
    if(!strcmp(nomeID, c->nomeID)&&!strcmp(c->tipoData, "void"))
      return 0;
    else
      return 1;
  }
}

/*Checks existance of a given function*/
```

```
int functionLookup(TipoLista *list, char nomeID[]) {
    int hash = string2int(nomeID)%211;
    TipoID *c = list[hash].start;
    char function[] = "func";
    while(c != NULL) {
      if((!strcmp(nomeID, c->nomeID)) && (!strcmp(function, c->tipoID
          ))) {
        return 1;
      }
      c = c->prox;
    }
    /*if(c == NULL) {*/
      return 0;
    /*}*/
}

/*Role: Checks Existance of void variables*/
int checkVoid(TipoLista *list, int index){
TipoID *p = list[index].start;
while(p!=NULL){
    if(p->linhas[0] != 0) {
      //p->nomeID, p->tipoID, p->tipoData, p->escopo);
      if (!strcmp(p->tipoID, "var")&&!strcmp(p->tipoData,"void")) {
        return p->linhas[0];
      }
    }
      p = p->prox;
    }
    return 0;
}


/*Role: Checks Existance of Main function*/
int checkMain(TipoLista *list, int index){
TipoID *p = list[index].start;
while(p!=NULL){
    if(p->linhas[0] != 0) {
      //p->nomeID, p->tipoID, p->tipoData, p->escopo);
      if (!strcmp(p->tipoID, "func")&&
          !strcmp(p->tipoData,"void")&&
          !strcmp(p->nomeID, "main")&&
          p->linhas[1]==0)
        return 0;
    }
      p = p->prox;
    }
    return 1;
}

/*Role: Checks Existance of equal declarations in a same scope*/
int checkDecScope(TipoLista *list, int index){
  TipoID *p = list[index].start;
  TipoID *w;

  while(p!=NULL&&(!strcmp(p->tipoID,"var")||!strcmp(p->tipoID,"vet"))
      ){
    if(p->linhas[0] != 0) {
      w = p->prox;
      while (w!=NULL&&(!strcmp(p->tipoID,"var")||!strcmp(p->tipoID,"
          vet"))){
```

```
        if(!strcmp(w->nomeID,p->nomeID)&&!strcmp(p->escopo,w->escopo)
           ){
          return w->linhas[0];
        }
        w = w->prox;
      }
    }
      p = p->prox;
    }
    return 0;
  }

/*Role: Checks Existance of variables and functions with similar
    names*/
int checkSameVarFunc(TipoLista *list, int index){
  TipoID *p = list[index].start;
  TipoID *w;
  while (p!=NULL) {
    if (p->linhas[0] != 0) {
      w = p->prox;
      while (w!=NULL) {
        if (!strcmp(w->nomeID,p->nomeID)&&(((!strcmp(w->tipoID,"var")
           ||!strcmp(w->tipoID,"vet"))
            &&!strcmp(p->tipoID,"func"))||((!strcmp(p->tipoID,"var")
               ||!strcmp(w->tipoID,"vet"))
            &&!strcmp(w->tipoID,"func"))
          )){
            if (w->linhas[0])
              w->linhas[0];
            else
              return p->linhas[0];
        }

            w = w->prox;
      }
    }
    p = p->prox;
  }
  return 0;
}



int semanticAnalysis(TipoLista *hashList){
  int i;
  int j;
  int checkMainFlag = 1;
  int checkDecScopeFlag;
  int checkSameVarFuncFlag;

  for(i = 0;i<211;i++){
      if(&hashList[i]!=NULL){
        // Check Existance of void variables
        int checkVoidFlag;
        checkVoidFlag = checkVoid(hashList, i);
        if (checkVoidFlag)
          printf("Semantic error at line %d: Variable declared as
              void\n ", checkVoidFlag);
        //Check Existance of main function
       if (!checkMain(hashList, i))
```

```
          checkMainFlag = 0;
         //Check doubl  e var/vet declarations in a same scope
        checkDecScopeFlag = checkDecScope(hashList,i);
        if(checkDecScopeFlag)
             printf("Semantic error at line %d: double declaration at
                a same scope\n ", checkDecScopeFlag);
  //Check variables and functions with similar names
        checkSameVarFuncFlag = checkSameVarFunc(hashList,i);
        if(checkSameVarFuncFlag)
             printf("Semantic error at line %d: variable and function
                with same name\n ", checkSameVarFuncFlag);
      }
    }
    if (checkMainFlag)
      printf("Semantic error: main function not present or different
         from 'void main(void)'\n ");
  return 0;
}

int main() {
/*
extern int yydebug;
yydebug = 1;
*/
TreeNode * syntaxTree;
FILE *f_in;
FILE *f_out;
int array_size;
int i;
int w;
int flag;
int buf[100000];
char escopo[30];  //  escopo da funaao
char nomeID[20];  //  nome do ID
char tipoID[3];   //  tipo nenhum <var, fun, vet>
char tipoData[10]; //  tipo de dados <int, float, void>
char nomeIDAnt[20];
char array_size_char[30];
int line = 1;
int hash = 0;
// Alocando o vetor estatico e inicializando ponteiros com NULL
TipoLista vetor[211]; //lista de listas
SyncList id_list;
list_quadruple quad_list;

for(i = 0; i < 211; i++) {
  vetor[i].start = NULL;
}

id_list.start = NULL;

for (i=0;i<=100000;i++) buf[i] = 0;

abrirArq();

int token;
int cont = 0;             //  contador de chaves inicia com zero no
   IDS.escopo global
strcpy(nomeID, "nome");
strcpy(escopo, "global"); //  inicia laao com escopo global
```

```
strcpy(tipoData, "void"); //  tipo void por default
strcpy(tipoID, "non");    //  tipo nenhum <var, fun, vet>
flag = 0;
w = 0;


// Inserindo funaaes predefinidas int input() e void output()
insere(vetor, escopo, "input", "func", "int", -1, 39, 0);
insere(vetor, escopo, "output", "func", "void", -1, 34, 0);

while ((token=yylex()) != '\0') {
  buf[w] = token;
  /*printf("%d\t", token);*/
  /*printf("%s\n", yytext);*/
  w++;
    switch(token) {

      case VOID:
        strcpy(tipoData, "void");
        flag = 2;
    //    puts(tipoData);
      break;

      case FLOAT:
        strcpy(tipoData, "float");
        flag = 1;
  //      puts(tipoData);
      break;

      case INT:
        strcpy(tipoData, "int");
        flag = 1;
  //      puts(tipoData);
      break;

      case LCAPSULE:
        cont++;
        flag = 0;
      break;

      case RCAPSULE:
        cont--;
        if(cont == 0) strcpy(escopo,"global");
      break;

      case ID:
        hash = string2int(nomeID);
        strcpy(nomeID, yytext);
        token = yylex();
        buf[w] = token;
        /*printf("%d\t", token);*/
        w++;
        // IDS.linhas[0] = noline;

        if(token == LPAREN) {
          strcpy(tipoID, "func");
          if(strcmp(escopo,"global") == 0) {
            strcpy(escopo, nomeID);
          }
          if(flag==0&&!(functionLookup(vetor, nomeID)))
```

```
            printf("\nSemantic Error: Non declared function called,
                '%s()'. Line %d\n", nomeID, lineno);
        } else {
          if(token == LBRACK) {

            token = yylex();
            buf[w] = token;
            w++;

            if(token == NUM){
              if(flag==1 && !(buscaVariavel(vetor, nomeID, escopo)))
                strcpy(array_size_char, yytext);
              array_size = string2int(array_size_char);
            }

            strcpy(tipoID, "vet");
            if(flag == 0 && !(buscaVariavel(vetor, nomeID, escopo)))
              printf("\nSemantic error: bad declaration for Variable
                  '%s'. Line %d\n", nomeID, lineno);


          } else {
            // variavel
            strcpy(tipoID, "var");
            if(flag == 0 && !(buscaVariavel(vetor, nomeID, escopo)))
              printf("\nSemantic error: bad declaration for Variable
                  '%s'. Line %d\n", nomeID, lineno);

          }
        }

        hash = string2int(nomeID)%211;
        int newID = checkExistance(vetor, nomeID, lineno, hash,
            escopo, flag);
        if(!newID){
          insere(vetor, escopo, nomeID, tipoID, tipoData, lineno,
              hash, array_size);
        }
        insert(&id_list, escopo, nomeID,tipoID);
        if(token==SEMI) flag = 0;
      break;

      case SEMI:
        flag = 0;
        break;
    }
}


f_out = fopen("out.txt", "w");
i = 0;
  while (buf[i] != 0) {
    switch(buf[i]) {
    case INT: fprintf(f_out, "INT\n"); break;
    case FLOAT: fprintf(f_out, "FLOAT\n"); break;
    case IF:    fprintf(f_out, "IF\n"); break;
    case ELSE:  fprintf(f_out, "ELSE\n"); break;
    case RETURN:fprintf(f_out, "RETURN\n"); break;
    case VOID:  fprintf(f_out, "VOID\n"); break;
    case WHILE: fprintf(f_out, "WHILE\n"); break;
```

```
  case PLUS:  fprintf(f_out, "PLUS\n"); break;
  case MINUS: fprintf(f_out, "MINUS\n"); break;
  case TIMES: fprintf(f_out, "TIMES\n"); break;
  case OVER:  fprintf(f_out, "OVER\n"); break;
  case LT:    fprintf(f_out, "LT\n"); break;
  case LET:   fprintf(f_out, "LET\n"); break;
  case HT:    fprintf(f_out, "HT\n"); break;
  case HET:   fprintf(f_out, "HET\n"); break;
  case EQ:    fprintf(f_out, "EQ\n"); break;
  case NEQ:   fprintf(f_out, "NEQ\n"); break;
  case ASSIGN:fprintf(f_out, "ASSIGN\n"); break;
  case SEMI:  fprintf(f_out, "SEMI\n"); break;
  case COMMA: fprintf(f_out, "COMMA\n"); break;
  case LPAREN:fprintf(f_out, "LPAREN\n"); break;
  case RPAREN:fprintf(f_out, "RPAREN\n"); break;
  case LBRACK:fprintf(f_out, "LBRACK\n"); break;
  case RBRACK:fprintf(f_out, "RBRACK\n"); break;
  case LCAPSULE:      fprintf(f_out, "LCAPSULE\n"); break;
  case RCAPSULE:      fprintf(f_out, "RCAPSULE\n"); break;
  case NUM:   fprintf(f_out, "NUM\n"); break;
  case ID:    fprintf(f_out, "ID\n"); break;
  }
i++;
}

// Verificaaao de atribuiaaes
abrirArq();
lineno = 1;
strcpy(escopo, "global"); //  inicia laao com escopo global
while ((token=yylex()) != '\0') {

    switch(token) {

      case ASSIGN:
          if(verState == 0) {
            verState = 1; // passa a procurar lado direito
          }
      break;

      case SEMI:
          if(verState == 1) {
            verState = 0; // volta a procurar lado esquerdo
          }

      break;

      case ID:
          if(verState == 0) {
            strcpy(L_side,yytext);
            if(functionLookup(vetor, L_side)) {
              strcpy(escopo, yytext);
            }
          } else {
            if(verState == 1) {
              while(token != SEMI && token != LPAREN) {
                strcpy(R_side,yytext);
                if(functionLookup(vetor, R_side)){
                  if(!functionType(vetor, R_side))
                    voidFlag = 1;
                }
```

```
                    token = yylex();
                    if((token == MINUS) || (token == PLUS) || (token ==
                        TIMES) || (token == OVER)) token == yylex();
                  }
                  verState = 0;
                }
              }
            break;
        }
    }

    printf("\nParser running...\n");
    abrirArq();
    /*if (yyparse()==0) printf("\nSyntax Analysis OK\n");
    else printf("\nERROR in Syntax Analysis\n");*/

    /*printTree();*/

    if(voidFlag)
      printf("Semantic error: Void attribution to variable\n");

    printf("Running Semantic Analysis...\n");
    semanticAnalysis(vetor);
    printf("Semantic Analysis Finished\n");


    printf("Finished.\n");

    /*for(i = 0; i < 211; i++) {
      hash_table[i].start = NULL;
    }*/

    /*printIDList(&id_list);*/

    /*while(getToken() != ENDFILE);*/
    syntaxTree = parse();

    // printTree(syntaxTree);
    generate_icode_launcher(&quad_list, syntaxTree, &vetor);

    generate_code_launcher(&quad_list, &vetor);

    printf("\n");
    /*print_quadruple_list(&quad_list);*/


    printf("Name(ID)  Type(ID)  Type(Data)   Scope   Intermediate Index
            Appears in lines       Array Size\n");
    for(i = 0;i<211;i++){
      if(&vetor[i]!=NULL)
      printWTable(vetor, i);
    }

    return 0;
}
```

Algoritmo 3: globals.h - cabeçalhos de funções suplementares

```
/*************************************************/
```

```c
/* File: globals.h                                          */
/* Global types and vars for C- compiler                    */
/* must come before other include files                     */
/************************************************************/

#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 "cminus.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 6

/* 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 */
extern FILE* code; /* code text file for TM simulator */

extern int linenumber; /* source line number for listing */

/************************************************/
/***********   Syntax tree for parsing ************/
/************************************************/
```

```
typedef enum {StmtK, ExpK} NodeKind;
typedef enum {IfK, WhileK, AssignK, ReturnK, VarK, VecK, FuncK,
   FuncVarK, FuncVecK} StmtKind;
typedef enum {TypeK, RelOpK, ArithOpK, ConstK, IdK, VecIndexK, CallK}
    ExpKind;

/* ExpType is used for type checking */
typedef enum {Void, Integer, Boolean} ExpType;

#define MAXCHILDREN 3

union treeKind{
    StmtKind stmt;
    ExpKind exp;
};

struct treeAttr{
    TokenType oprtr;
    int value;
    char * name;
};

typedef struct treeNode{
    struct treeNode * child[MAXCHILDREN];
    struct treeNode * sibling;
    int linenumber;
    char * scope;
    NodeKind nodekind;
    union treeKind kind;
    struct treeAttr attr;
    ExpType type; /* for type checking of exps */
} TreeNode;

typedef struct TipoID{
    char nomeID[20];
    char tipoID[20];
    char tipoData[10];
    char escopo[30];
    int linhas[50];
    int indice_parametro;
    int intermediate_start;
    int intermediate_finish;
    int declaration_flag;
    int array_size;
    int top;
    struct TipoID *prox;
}TipoID;

typedef struct{
    TipoID *start;
}TipoLista;

// extern TipoLista hash_table[211];


/**************************************************/
/***********   Flags for tracing       ************/
/**************************************************/

/* EchoSource = TRUE causes the source program to
```

```
 * be echoed to the listing file with line numbers
 * during parsing
 */
extern int EchoSource;

/* TraceScan = TRUE causes token information to be
 * printed to the listing file as each token is
 * recognized by the scanner
 */
extern int TraceScan;

/* TraceParse = TRUE causes the syntax tree to be
 * printed to the listing file in linearized form
 * (using indents for children)
 */
extern int TraceParse;

/* TraceAnalyze = TRUE causes symbol table inserts
 * and lookups to be reported to the listing file
 */
extern int TraceAnalyze;

/* TraceCode = TRUE causes comments to be written
 * to the TM code file as code is generated
 */
extern int TraceCode;

/* Error = TRUE prevents further passes if an error occurs */
extern int Error;
#endif
```

Algoritmo 4: intermediate.c - geração de código intermediário

```
#include "intermediate.h"

static int temporary = 0;
static int label = 0;
static int flag_param = 0;
int test_count=0;
int number_of_quadruples = 0;
FILE *file_quadruples;
FILE *file_read_quadruples;

int counter = 0;
int flag_later = 0;
int indx = 0;
int index_parameters = 0;

int check_if_int(TipoLista *table, char name[]){
  TipoID *item;
  int hash;
  hash = string2int(name)%211;
  item = table[hash].start;
  while (item!=NULL) {
    if(!strcmp(item->nomeID, name)){
      if (!strcmp(item->tipoID, "func")) {
        if (!strcmp(item->tipoData, "int")) {
          return 1;
        }
```

```
      }
    }
    item = item->prox;
  }
  return 0;
}

void add_parameters_to_table(TipoLista *table, list_quadruple *
    quad_list) {

  int hash;
  quadruple *p = quad_list->start;
  TipoID *q;
  while (p!=NULL) {
    switch (p->op) {
      case PrmVarK:
      case PrmArrK:
        hash = string2int(p->address_1.name)%211;
        q = table[hash].start;
        while (q!=NULL) {
          if (!strcmp(p->address_1.name, q->nomeID)&&!strcmp(p->
              address_2.name, q->escopo)) {
            q->indice_parametro = p->address_3.value;
            break;
          }
          q = q->prox;
        }
        // printf("%s %s %d\n", p->address_1.name, p->address_2.name,
            p->address_3.value);
        break;
      default:
        break;
    }
    p = p->next;
  }
}

void add_indexes_to_table(TipoLista *table, list_quadruple *quad_list
    ){

  int hash;
  quadruple *p = quad_list->start;
  quadruple *main_id;
  TipoID *q;
  TipoID *previous_function;


  previous_function = NULL;

  while (p!=NULL) {
    if (p->op==LblK) {
        if (p->address_3.kind==String) {
          hash = string2int(p->address_3.name)%211;
          q = table[hash].start;
          while (q!=NULL) {
            if (!strcmp(q->tipoID,"func")&&!strcmp(q->nomeID,p->
                address_3.name)) {
              if(previous_function!=NULL)
                previous_function->intermediate_finish = p->index-1;
              q->intermediate_start = p->index;
```

```
                previous_function = q;
                break;
              }
              q = q->prox;
            }
          }
      }
      main_id = p;
      p = p->next;
    }
    hash = string2int("main")%211;
    q = table[hash].start;
    q->intermediate_finish = main_id->index;

}

void store_quadruple(OpKind o, AddrKind k1, AddrKind k2, AddrKind k3,
                     int v1, int v2, int v3,
                     char n1[], char n2[], char n3[]){
                      // if(k1!=IntConst)
                      //   v1 = 0;
                      // if(k2!=IntConst)
                      //   v2 = 0;
                      // if(k3!=IntConst)
                      //   v3 = 0;
                      if(k1!=String)
                        strcpy(n1, "empty");
                      if(k2!=String)
                        strcpy(n2, "empty");
                      if(k3!=String)
                        strcpy(n3, "empty");
                      number_of_quadruples++;
                        fprintf(file_quadruples, "%d %d %d %d %d %d %
                           d %s %s %s ",
                        o, k1, k2, k3, v1, v2, v3, n1, n2, n3);
                     }

void insert_quadruple(list_quadruple *quad_list, quadruple *quad){
   quadruple *p = quad_list->start;
   quadruple *alloc_quad = malloc(sizeof(quadruple));
   *alloc_quad = *quad;
  //  printf("                                %d\n", quad->op);
   int breaker = 0;
   alloc_quad->index = indx;
   indx++;
   if(p==NULL){
     quad_list->start = alloc_quad;
     quad_list->start->next = NULL;
   }
   else{
     while(p->next!=NULL){
       p = p->next;
     }
      p->next = alloc_quad;
      p->next->next = NULL;
   }
}

void insert_quadruples(list_quadruple *quad_list){
  flag_later = 1;
```

```
char word[1024];
int item;
item = 0;
// quad_list->start = NULL;
// quad = NULL;
quadruple *quad = malloc(sizeof(quadruple));


while (fscanf(file_read_quadruples, "%1023s", word) == 1) {
  // quad->next = NULL;
  // printf("%d %s\n", item, word);
  switch (item) {
    case 0:
      quad->op = (OpKind)atoi(word);
      // printf("%d\n", quad->op);
      item++;
      break;
    case 1:
      quad->address_1.kind = (AddrKind)atoi(word);
      item++;
      break;
    case 2:
      quad->address_2.kind = (AddrKind)atoi(word);
      item++;
      break;
    case 3:
      quad->address_3.kind = (AddrKind)atoi(word);
      item++;
      break;
    case 4:
      quad->address_1.value = atoi(word);
      item++;
      break;
    case 5:
      quad->address_2.value = atoi(word);
      item++;
      break;
    case 6:
      quad->address_3.value = atoi(word);
      item++;
      break;
    case 7:
      strcpy(quad->address_1.name, word);
      item++;
      break;
    case 8:
      strcpy(quad->address_2.name, word);
      item++;
      break;
    case 9:
      strcpy(quad->address_3.name, word);
      // printf("%d\n", item);
      item = 0;
      insert_quadruple(quad_list, quad);
      quadruple *quad = malloc(sizeof(quadruple));
      break;
    default:
      break;
  }
}
```

```
}

void print_quadruple_list(list_quadruple *quad_list){
  quadruple *p = quad_list->start;
  while (p!=NULL) {
    printf("index: %d \toperation: %d \tscope: %s\n \t1: Kind: %d \
       tName:%s \tValue: %d\n \t2: Kind: %d \tName:%s \tValue: %d\n \
       t3: Kind: %d \tName:%s \tValue: %d\n",
    p->index, p->op, p->scope,
    p->address_1.kind, p->address_1.name, p->address_1.value,
    p->address_2.kind, p->address_2.name, p->address_2.value,
    p->address_3.kind, p->address_3.name, p->address_3.value);
    p = p->next;
  }
}

static void generate_relop(list_quadruple *quad_list, TreeNode *tree,
    TipoLista *table) {
  TreeNode *c0, *c1, *c2;
  c0 = tree->child[0];
  c1 = tree->child[1];
  c2 = tree->child[2];
  int aux;

  quadruple *quad = malloc(sizeof(quadruple));

  switch (tree->attr.oprtr) {
    case EQ:
    if(c0){
      if(c0->kind.stmt == VarK){
        strcpy(quad->address_1.name, c0->attr.name);
        quad->address_1.kind = String;
        strcpy(quad->scope, tree->scope);

        // printf("%s\n", quad->address_1.name);
      }else if (c0->kind.exp == ConstK) {
        quad->address_1.value = c0->attr.value;
        quad->address_1.kind = IntConst;
        // printf("%d\n", quad->address_1.value);
      }else{
        generate_intermediate_code(quad_list, c0, table);
        // printf("%d\n", temporary);
        quad->address_1.value = temporary;
        quad->address_1.kind = Temp;
        // temporary++;
      }
    }

    if(c1){
      if(c1->kind.stmt == VarK){
        strcpy(quad->address_2.name, c1->attr.name);
        quad->address_2.kind = String;
        strcpy(quad->scope, tree->scope);

        // printf("%s\n", quad->address_2.name);
      }else if (c1->kind.exp == ConstK) {
        quad->address_2.value = c1->attr.value;
        quad->address_2.kind = IntConst;
        // printf("%d\n", quad->address_2.value);
      }else{
```

```
    generate_intermediate_code(quad_list, c1, table);
    // printf("%d\n", temporary);
    quad->address_2.value = temporary;
    quad->address_2.kind = Temp;
    // temporary++;
  }
}
temporary++;
quad->address_3.value = temporary;
quad->address_3.kind = Temp;

quad->op = EqlK;
switch (quad->address_3.kind) {
  case String: printf("%s ", quad->address_3.name);
    break;
    case IntConst: printf("%d ", quad->address_3.value);
    break;
    case Temp: printf("t%d ", quad->address_3.value);
    default: break;
  }
  printf("= ");

switch (quad->address_1.kind) {
  case String: printf("%s ", quad->address_1.name);
    break;
  case IntConst: printf("%d ", quad->address_1.value);
    break;
  case Temp: printf("t%d ", quad->address_1.value);
  default: break;
}

printf("== ");
switch (quad->address_2.kind) {
  case String: printf("%s ", quad->address_2.name);
    break;
  case IntConst: printf("%d ", quad->address_2.value);
    break;
  case Temp: printf("t%d ", quad->address_2.value);
  default: break;
}
printf("\n");

insert_quadruple(quad_list, quad);
store_quadruple(quad->op, quad->address_1.kind, quad->address_2.
   kind, quad->address_3.kind,
                quad->address_1.value, quad->address_2.value,
                   quad->address_3.value,
                quad->address_1.name, quad->address_2.name, quad
                   ->address_3.name);
  break;
case NEQ:
if(c0){
  if(c0->kind.stmt == VarK){
    strcpy(quad->address_1.name, c0->attr.name);
    quad->address_1.kind = String;
    strcpy(quad->scope, tree->scope);

    // printf("%s\n", quad->address_1.name);
  }else if (c0->kind.exp == ConstK) {
    quad->address_1.value = c0->attr.value;
```

```
    quad->address_1.kind = IntConst;
    // printf("%d\n", quad->address_1.value);
  }else{
    generate_intermediate_code(quad_list, c0, table);
    // printf("%d\n", temporary);
    quad->address_1.value = temporary;
    quad->address_1.kind = Temp;
    // temporary++;
  }
}

if(c1){
  if(c1->kind.stmt == VarK){
    strcpy(quad->address_2.name, c1->attr.name);
    quad->address_2.kind = String;
    strcpy(quad->scope, tree->scope);

    // printf("%s\n", quad->address_2.name);
  }else if (c1->kind.exp == ConstK) {
    quad->address_2.value = c1->attr.value;
    quad->address_2.kind = IntConst;
    // printf("%d\n", quad->address_2.value);
  }else{
    generate_intermediate_code(quad_list, c1, table);
    // printf("%d\n", temporary);
    quad->address_2.value = temporary;
    quad->address_2.kind = Temp;
    // temporary++;
  }
}
temporary++;
quad->address_3.value = temporary;
quad->address_3.kind = Temp;

quad->op = NeqK;
insert_quadruple(quad_list, quad);
store_quadruple(quad->op, quad->address_1.kind, quad->address_2.
   kind, quad->address_3.kind,
                quad->address_1.value, quad->address_2.value,
                   quad->address_3.value,
                quad->address_1.name, quad->address_2.name, quad
                   ->address_3.name);


                switch (quad->address_3.kind) {
                  case String: printf("%s ", quad->address_3.name
                     );
                    break;
                    case IntConst: printf("%d ", quad->address_3.
                       value);
                    break;
                    case Temp: printf("t%d ", quad->address_3.
                       value);
                    default: break;
                  }
                  printf("= ");

                switch (quad->address_1.kind) {
                  case String: printf("%s ", quad->address_1.name
                     );
```

```
                    break;
                  case IntConst: printf("%d ", quad->address_1.
                      value);
                    break;
                  case Temp: printf("t%d ", quad->address_1.value
                      );
                  default: break;
                }

                printf("!= ");
                switch (quad->address_2.kind) {
                  case String: printf("%s ", quad->address_2.name
                      );
                    break;
                  case IntConst: printf("%d ", quad->address_2.
                      value);
                    break;
                  case Temp: printf("t%d ", quad->address_2.value
                      );
                  default: break;
                }
                printf("\n");
break;
case LT:
  if(c0){
    if(c0->kind.stmt == VarK){
      strcpy(quad->address_1.name, c0->attr.name);
      quad->address_1.kind = String;
      strcpy(quad->scope, tree->scope);

      // printf("%s\n", quad->address_1.name);
    }else if (c0->kind.exp == ConstK) {
      quad->address_1.value = c0->attr.value;
      quad->address_1.kind = IntConst;
      // printf("%d\n", quad->address_1.value);
    }else{
      generate_intermediate_code(quad_list, c0, table);
      // printf("%d\n", temporary);
      quad->address_1.value = temporary;
      quad->address_1.kind = Temp;
      // temporary++;
    }
  }

  if(c1){
    if(c1->kind.stmt == VarK){
      strcpy(quad->address_2.name, c1->attr.name);
      quad->address_2.kind = String;
      strcpy(quad->scope, tree->scope);

      // printf("%s\n", quad->address_2.name);
    }else if (c1->kind.exp == ConstK) {
      quad->address_2.value = c1->attr.value;
      quad->address_2.kind = IntConst;
      // printf("%d\n", quad->address_2.value);
    }else{
      generate_intermediate_code(quad_list, c1, table);
      // printf("%d\n", temporary);
      quad->address_2.value = temporary;
      quad->address_2.kind = Temp;
```

```
        // temporary++;
      }
    }
    temporary++;
    quad->address_3.value = temporary;
    quad->address_3.kind = Temp;

    quad->op = LsrK;
    insert_quadruple(quad_list, quad);
    store_quadruple(quad->op, quad->address_1.kind, quad->address_2
        .kind, quad->address_3.kind,
                    quad->address_1.value, quad->address_2.value,
                       quad->address_3.value,
                    quad->address_1.name, quad->address_2.name,
                       quad->address_3.name);
                    switch (quad->address_3.kind) {
                      case String: printf("%s ", quad->address_3.
                         name);
                        break;
                        case IntConst: printf("%d ", quad->
                           address_3.value);
                        break;
                        case Temp: printf("t%d ", quad->address_3.
                           value);
                        default: break;
                      }
                      printf("= ");

                    switch (quad->address_1.kind) {
                      case String: printf("%s ", quad->address_1.
                         name);
                        break;
                      case IntConst: printf("%d ", quad->address_1.
                         value);
                        break;
                      case Temp: printf("t%d ", quad->address_1.
                         value);
                      default: break;
                    }

                    printf("< ");
                    switch (quad->address_2.kind) {
                      case String: printf("%s ", quad->address_2.
                         name);
                        break;
                      case IntConst: printf("%d ", quad->address_2.
                         value);
                        break;
                      case Temp: printf("t%d ", quad->address_2.
                         value);
                      default: break;
                    }
                    printf("\n");
  break;
case LET:
  if(c0){
    if(c0->kind.stmt == VarK){
      strcpy(quad->address_1.name, c0->attr.name);
      quad->address_1.kind = String;
      strcpy(quad->scope, tree->scope);
```

```
    // printf("%s\n", quad->address_1.name);
  }else if (c0->kind.exp == ConstK) {
    quad->address_1.value = c0->attr.value;
    quad->address_1.kind = IntConst;
    // printf("%d\n", quad->address_1.value);
  }else{
    generate_intermediate_code(quad_list, c0, table);
    // printf("%d\n", temporary);
    quad->address_1.value = temporary;
    quad->address_1.kind = Temp;
    // temporary++;
  }
}

if(c1){
  if(c1->kind.stmt == VarK){
    strcpy(quad->address_2.name, c1->attr.name);
    quad->address_2.kind = String;
    strcpy(quad->scope, tree->scope);

    // printf("%s\n", quad->address_2.name);
  }else if (c1->kind.exp == ConstK) {
    quad->address_2.value = c1->attr.value;
    quad->address_2.kind = IntConst;
    // printf("%d\n", quad->address_2.value);
  }else{
    generate_intermediate_code(quad_list, c1, table);
    // printf("%d\n", temporary);
    quad->address_2.value = temporary;
    quad->address_2.kind = Temp;
    // temporary++;
  }
}
temporary++;
quad->address_3.value = temporary;
quad->address_3.kind = Temp;

quad->op = LeqK;
insert_quadruple(quad_list, quad);
store_quadruple(quad->op, quad->address_1.kind, quad->address_2
    .kind, quad->address_3.kind,
                quad->address_1.value, quad->address_2.value,
                   quad->address_3.value,
                quad->address_1.name, quad->address_2.name,
                   quad->address_3.name);
                switch (quad->address_3.kind) {
                  case String: printf("%s ", quad->address_3.
                     name);
                    break;
                    case IntConst: printf("%d ", quad->
                       address_3.value);
                    break;
                    case Temp: printf("t%d ", quad->address_3.
                       value);
                    default: break;
                  }
                  printf("= ");

                switch (quad->address_1.kind) {
```

```
                        case String: printf("%s ", quad->address_1.
                            name);
                          break;
                        case IntConst: printf("%d ", quad->address_1.
                            value);
                          break;
                        case Temp: printf("t%d ", quad->address_1.
                            value);
                        default: break;
                      }

                      printf("<= ");
                      switch (quad->address_2.kind) {
                        case String: printf("%s ", quad->address_2.
                            name);
                          break;
                        case IntConst: printf("%d ", quad->address_2.
                            value);
                          break;
                        case Temp: printf("t%d ", quad->address_2.
                            value);
                        default: break;
                      }
                      printf("\n");
    break;
  case HT:
    if(c0){
      if(c0->kind.stmt == VarK){
        strcpy(quad->address_1.name, c0->attr.name);
        quad->address_1.kind = String;
        strcpy(quad->scope, tree->scope);

        // printf("%s\n", quad->address_1.name);
      }else if (c0->kind.exp == ConstK) {
        quad->address_1.value = c0->attr.value;
        quad->address_1.kind = IntConst;
        // printf("%d\n", quad->address_1.value);
      }else{
        generate_intermediate_code(quad_list, c0, table);
        // printf("%d\n", temporary);
        quad->address_1.value = temporary;
        quad->address_1.kind = Temp;
        // temporary++;
      }
    }

    if(c1){
      if(c1->kind.stmt == VarK){
        strcpy(quad->address_2.name, c1->attr.name);
        quad->address_2.kind = String;
        strcpy(quad->scope, tree->scope);

        // printf("%s\n", quad->address_2.name);
      }else if (c1->kind.exp == ConstK) {
        quad->address_2.value = c1->attr.value;
        quad->address_2.kind = IntConst;
        // printf("%d\n", quad->address_2.value);
      }else{
        generate_intermediate_code(quad_list, c1, table);
        // printf("%d\n", temporary);
```

```
        quad->address_2.value = temporary;
        quad->address_2.kind = Temp;
        // temporary++;
      }
    }
    temporary++;
    quad->address_3.value = temporary;
    quad->address_3.kind = Temp;

    quad->op = GtrK;
    insert_quadruple(quad_list, quad);
    store_quadruple(quad->op, quad->address_1.kind, quad->address_2
        .kind, quad->address_3.kind,
                    quad->address_1.value, quad->address_2.value,
                        quad->address_3.value,
                    quad->address_1.name, quad->address_2.name,
                        quad->address_3.name);
                    switch (quad->address_3.kind) {
                      case String: printf("%s ", quad->address_3.
                          name);
                        break;
                        case IntConst: printf("%d ", quad->
                            address_3.value);
                        break;
                        case Temp: printf("t%d ", quad->address_3.
                            value);
                        default: break;
                      }
                      printf("= ");

                    switch (quad->address_1.kind) {
                      case String: printf("%s ", quad->address_1.
                          name);
                        break;
                      case IntConst: printf("%d ", quad->address_1.
                          value);
                        break;
                      case Temp: printf("t%d ", quad->address_1.
                          value);
                      default: break;
                    }

                    printf("> ");
                    switch (quad->address_2.kind) {
                      case String: printf("%s ", quad->address_2.
                          name);
                        break;
                      case IntConst: printf("%d ", quad->address_2.
                          value);
                        break;
                      case Temp: printf("t%d ", quad->address_2.
                          value);
                      default: break;
                    }
                    printf("\n");
  break;
case HET:
  if(c0){
    if(c0->kind.stmt == VarK){
      strcpy(quad->address_1.name, c0->attr.name);
```

```
      quad->address_1.kind = String;
      strcpy(quad->scope, tree->scope);

      // printf("%s\n", quad->address_1.name);
    }else if (c0->kind.exp == ConstK) {
      quad->address_1.value = c0->attr.value;
      quad->address_1.kind = IntConst;
      // printf("%d\n", quad->address_1.value);
    }else{
      generate_intermediate_code(quad_list, c0, table);
      // printf("%d\n", temporary);
      quad->address_1.value = temporary;
      quad->address_1.kind = Temp;
      // temporary++;
    }
  }

  if(c1){
    if(c1->kind.stmt == VarK){
      strcpy(quad->address_2.name, c1->attr.name);
      quad->address_2.kind = String;
      strcpy(quad->scope, tree->scope);

      // printf("%s\n", quad->address_2.name);
    }else if (c1->kind.exp == ConstK) {
      quad->address_2.value = c1->attr.value;
      quad->address_2.kind = IntConst;
      // printf("%d\n", quad->address_2.value);
    }else{
      generate_intermediate_code(quad_list, c1, table);
      // printf("%d\n", temporary);
      quad->address_2.value = temporary;
      quad->address_2.kind = Temp;
      // temporary++;
    }
  }
  temporary++;
  quad->address_3.value = temporary;
  quad->address_3.kind = Temp;

  quad->op = GeqK;
  insert_quadruple(quad_list, quad);
  store_quadruple(quad->op, quad->address_1.kind, quad->address_2
     .kind, quad->address_3.kind,
                  quad->address_1.value, quad->address_2.value,
                     quad->address_3.value,
                  quad->address_1.name, quad->address_2.name,
                     quad->address_3.name);
                  switch (quad->address_3.kind) {
                    case String: printf("%s ", quad->address_3.
                       name);
                      break;
                      case IntConst: printf("%d ", quad->
                         address_3.value);
                      break;
                      case Temp: printf("t%d ", quad->address_3.
                         value);
                      default: break;
                    }
                    printf("= ");
```

```
                        switch (quad->address_1.kind) {
                          case String: printf("%s ", quad->address_1.
                              name);
                            break;
                          case IntConst: printf("%d ", quad->address_1.
                              value);
                            break;
                          case Temp: printf("t%d ", quad->address_1.
                              value);
                          default: break;
                        }

                        printf(">= ");
                        switch (quad->address_2.kind) {
                          case String: printf("%s ", quad->address_2.
                              name);
                            break;
                          case IntConst: printf("%d ", quad->address_2.
                              value);
                            break;
                          case Temp: printf("t%d ", quad->address_2.
                              value);
                          default: break;
                        }
                        printf("\n");
      break;
    default:
      break;
  }
}

static void generate_arithop(list_quadruple *quad_list, TreeNode *
    tree, TipoLista *table){
  TreeNode *c0, *c1, *c2;
  c0 = tree->child[0];
  c1 = tree->child[1];
  c2 = tree->child[2];

  quadruple *quad = malloc(sizeof(quadruple));

  quad->address_1.kind = Empty;
  quad->address_2.kind = Empty;
  quad->address_3.kind = Empty;

  switch (tree->attr.oprtr) {
    case PLUS:
      if(c0){
        if(c0->kind.stmt == VarK){
          strcpy(quad->address_1.name, c0->attr.name);
          quad->address_1.kind = String;
          strcpy(quad->scope, tree->scope);

          // printf("%s\n", quad->address_1.name);
        }else if (c0->kind.exp == ConstK) {
          quad->address_1.value = c0->attr.value;
          quad->address_1.kind = IntConst;
          // printf("%d\n", quad->address_1.value);
        }else{
          generate_intermediate_code(quad_list, c0, table);
```

```
    // printf("%d\n", temporary);
    quad->address_1.value = temporary;
    quad->address_1.kind = Temp;
    // temporary++;
  }
}

if(c1){
  if(c1->kind.stmt == VarK){
    strcpy(quad->address_2.name, c1->attr.name);
    quad->address_2.kind = String;
    strcpy(quad->scope, tree->scope);

    // printf("%s\n", quad->address_2.name);
  }else if (c1->kind.exp == ConstK) {
    quad->address_2.value = c1->attr.value;
    quad->address_2.kind = IntConst;
    // printf("%d\n", quad->address_2.value);
  }else{
    generate_intermediate_code(quad_list, c1, table);
    // printf("%d\n", temporary);
    quad->address_2.value = temporary;
    quad->address_2.kind = Temp;
    // temporary++;
  }
}


temporary++;
quad->address_3.value = temporary;
quad->address_3.kind = Temp;

quad->op = AddK;
insert_quadruple(quad_list, quad);
store_quadruple(quad->op, quad->address_1.kind, quad->address_2
    .kind, quad->address_3.kind,
                quad->address_1.value, quad->address_2.value,
                   quad->address_3.value,
                quad->address_1.name, quad->address_2.name,
                   quad->address_3.name);
                switch (quad->address_3.kind) {
                  case String: printf("%s ", quad->address_3.
                     name);
                    break;
                    case IntConst: printf("%d ", quad->
                       address_3.value);
                    break;
                    case Temp: printf("t%d ", quad->address_3.
                       value);
                    default: break;
                  }
                  printf("= ");

                switch (quad->address_1.kind) {
                  case String: printf("%s ", quad->address_1.
                     name);
                    break;
                  case IntConst: printf("%d ", quad->address_1.
                     value);
                    break;
```

```
                    case Temp: printf("t%d ", quad->address_1.
                        value);
                    default: break;
                  }

                  printf("+ ");
                  switch (quad->address_2.kind) {
                    case String: printf("%s ", quad->address_2.
                        name);
                      break;
                    case IntConst: printf("%d ", quad->address_2.
                        value);
                      break;
                    case Temp: printf("t%d ", quad->address_2.
                        value);
                    default: break;
                  }
                  printf("\n");
    break;
  case MINUS:
    if(c0){
      if(c0->kind.stmt == VarK){
        strcpy(quad->address_1.name, c0->attr.name);
        quad->address_1.kind = String;
        strcpy(quad->scope, tree->scope);

        // printf("%s\n", quad->address_1.name);
      }else if (c0->kind.exp == ConstK) {
        quad->address_1.value = c0->attr.value;
        quad->address_1.kind = IntConst;
        // printf("%d\n", quad->address_1.value);
      }else{
        generate_intermediate_code(quad_list, c0, table);
        // printf("t%d\n", temporary);
        quad->address_1.value = temporary;
        quad->address_1.kind = Temp;
        // temporary++;
      }
    }

    if(c1){
      if(c1->kind.stmt == VarK){
        strcpy(quad->address_2.name, c1->attr.name);
        quad->address_2.kind = String;
        strcpy(quad->scope, tree->scope);

        // printf("%s\n", quad->address_2.name);
      }else if (c1->kind.exp == ConstK) {
        quad->address_2.value = c1->attr.value;
        quad->address_2.kind = IntConst;
        // printf("%d\n", quad->address_2.value);
      }else{
        generate_intermediate_code(quad_list, c1, table);
        // printf("t%d\n", temporary);
        quad->address_2.value = temporary;
        quad->address_2.kind = Temp;
        // temporary++;
      }
    }
    temporary++;
```

```
    quad->address_3.value = temporary;
    quad->address_3.kind = Temp;

    quad->op = SubK;
    insert_quadruple(quad_list, quad);
    store_quadruple(quad->op, quad->address_1.kind, quad->address_2
        .kind, quad->address_3.kind,
                    quad->address_1.value, quad->address_2.value,
                       quad->address_3.value,
                    quad->address_1.name, quad->address_2.name,
                       quad->address_3.name);
                    switch (quad->address_3.kind) {
                      case String: printf("%s ", quad->address_3.
                         name);
                        break;
                        case IntConst: printf("%d ", quad->
                           address_3.value);
                        break;
                        case Temp: printf("t%d ", quad->address_3.
                           value);
                        default: break;
                      }
                      printf("= ");

                    switch (quad->address_1.kind) {
                      case String: printf("%s ", quad->address_1.
                         name);
                        break;
                      case IntConst: printf("%d ", quad->address_1.
                         value);
                        break;
                      case Temp: printf("t%d ", quad->address_1.
                         value);
                      default: break;
                    }

                    printf("- ");
                    switch (quad->address_2.kind) {
                      case String: printf("%s ", quad->address_2.
                         name);
                        break;
                      case IntConst: printf("%d ", quad->address_2.
                         value);
                        break;
                      case Temp: printf("t%d ", quad->address_2.
                         value);
                      default: break;
                    }
                    printf("\n");
      break;
    case TIMES:
      if(c0){
        if(c0->kind.stmt == VarK){
          strcpy(quad->address_1.name, c0->attr.name);
          quad->address_1.kind = String;
          strcpy(quad->scope, tree->scope);

          // printf("%s\n", quad->address_1.name);
        }else if (c0->kind.exp == ConstK) {
          quad->address_1.value = c0->attr.value;
```

```
      quad->address_1.kind = IntConst;
      // printf("%d\n", quad->address_1.value);
    }else{
      generate_intermediate_code(quad_list, c0, table);
      // printf("t%d\n", temporary);
      quad->address_1.value = temporary;
      quad->address_1.kind = Temp;
      // temporary++;
    }
  }

  if(c1){
    if(c1->kind.stmt == VarK){
      strcpy(quad->address_2.name, c1->attr.name);
      quad->address_2.kind = String;
      strcpy(quad->scope, tree->scope);

      // printf("%s\n", quad->address_2.name);
    }else if (c1->kind.exp == ConstK) {
      quad->address_2.value = c1->attr.value;
      quad->address_2.kind = IntConst;
      // printf("%d\n", quad->address_2.value);
    }else{
      generate_intermediate_code(quad_list, c1, table);
      // printf("t%d\n", temporary);
      quad->address_2.value = temporary;
      quad->address_2.kind = Temp;
      // temporary++;
    }
  }
  temporary++;
  quad->address_3.value = temporary;
  quad->address_3.kind = Temp;

  quad->op = TimK;
  insert_quadruple(quad_list, quad);
  store_quadruple(quad->op, quad->address_1.kind, quad->address_2
      .kind, quad->address_3.kind,
                  quad->address_1.value, quad->address_2.value,
                      quad->address_3.value,
                  quad->address_1.name, quad->address_2.name,
                      quad->address_3.name);
                  switch (quad->address_3.kind) {
                    case String: printf("%s ", quad->address_3.
                        name);
                      break;
                      case IntConst: printf("%d ", quad->
                          address_3.value);
                      break;
                      case Temp: printf("t%d ", quad->address_3.
                          value);
                      default: break;
                    }
                    printf("= ");

                  switch (quad->address_1.kind) {
                    case String: printf("%s ", quad->address_1.
                        name);
                      break;
                    case IntConst: printf("%d ", quad->address_1.
```

```
                        value);
                      break;
                    case Temp: printf("t%d ", quad->address_1.
                        value);
                    default: break;
                  }

                  printf("* ");
                  switch (quad->address_2.kind) {
                    case String: printf("%s ", quad->address_2.
                        name);
                      break;
                    case IntConst: printf("%d ", quad->address_2.
                        value);
                      break;
                    case Temp: printf("t%d ", quad->address_2.
                        value);
                    default: break;
                  }
                  printf("\n");
    break;
  case OVER:
    if(c0){
      if(c0->kind.stmt == VarK){
        strcpy(quad->address_1.name, c0->attr.name);
        quad->address_1.kind = String;
        strcpy(quad->scope, tree->scope);
        // printf("%s\n", quad->address_1.name);
      }else if (c0->kind.exp == ConstK) {
        quad->address_1.value = c0->attr.value;
        quad->address_1.kind = IntConst;
        // printf("%d\n", quad->address_1.value);
      }else{
        generate_intermediate_code(quad_list, c0, table);
        // printf("t%d\n", temporary);
        quad->address_1.value = temporary;
        quad->address_1.kind = Temp;
        // temporary++;
      }
    }

    if(c1){
      if(c1->kind.stmt == VarK){
        strcpy(quad->address_2.name, c1->attr.name);
        quad->address_2.kind = String;
        strcpy(quad->scope, tree->scope);

        // printf("%s\n", quad->address_2.name);
      }else if (c1->kind.exp == ConstK) {
        quad->address_2.value = c1->attr.value;
        quad->address_2.kind = IntConst;
        // printf("%d\n", quad->address_2.value);
      }else{
        generate_intermediate_code(quad_list, c1, table);
        // printf("t%d\n", temporary);
        quad->address_2.value = temporary;
        quad->address_2.kind = Temp;
        // temporary++;
      }
    }
```

```
        temporary++;
        quad->address_3.value = temporary;
        quad->address_3.kind = Temp;

        quad->op = OvrK;
        insert_quadruple(quad_list, quad);
        store_quadruple(quad->op, quad->address_1.kind, quad->address_2
            .kind, quad->address_3.kind,
                        quad->address_1.value, quad->address_2.value,
                            quad->address_3.value,
                        quad->address_1.name, quad->address_2.name,
                            quad->address_3.name);
                        switch (quad->address_3.kind) {
                          case String: printf("%s ", quad->address_3.
                              name);
                            break;
                            case IntConst: printf("%d ", quad->
                                address_3.value);
                            break;
                            case Temp: printf("t%d ", quad->address_3.
                                value);
                            default: break;
                          }
                          printf("= ");

                        switch (quad->address_1.kind) {
                          case String: printf("%s ", quad->address_1.
                              name);
                            break;
                          case IntConst: printf("%d ", quad->address_1.
                              value);
                            break;
                          case Temp: printf("t%d ", quad->address_1.
                              value);
                          default: break;
                        }

                        printf("/ ");
                        switch (quad->address_2.kind) {
                          case String: printf("%s ", quad->address_2.
                              name);
                            break;
                          case IntConst: printf("%d ", quad->address_2.
                              value);
                            break;
                          case Temp: printf("t%d ", quad->address_2.
                              value);
                          default: break;
                        }
                        printf("\n");
        break;
      default:
        break;
    }
  }


  static void generate_statement(list_quadruple *quad_list, TreeNode *
      tree, TipoLista *table) {
    TreeNode *c0, *c1, *c2;
```

```
c0 = tree->child[0];
c1 = tree->child[1];
c2 = tree->child[2];
int aux;
int aux2;
int aux3;

quadruple *quad0 = malloc(sizeof(quadruple));
quadruple *quad1 = malloc(sizeof(quadruple));
quadruple *quad2 = malloc(sizeof(quadruple));
quadruple *quad3 = malloc(sizeof(quadruple));

quad0->address_1.kind = Empty;
quad0->address_2.kind = Empty;
quad0->address_3.kind = Empty;

quad1->address_1.kind = Empty;
quad1->address_2.kind = Empty;
quad1->address_3.kind = Empty;

quad2->address_1.kind = Empty;
quad2->address_2.kind = Empty;
quad2->address_3.kind = Empty;

quad3->address_1.kind = Empty;
quad3->address_2.kind = Empty;
quad3->address_3.kind = Empty;

switch (tree->kind.stmt) {
  case IfK:
    // if
    if(c0){
      generate_intermediate_code(quad_list, c0, table);
      aux = temporary;
      aux2 = label;
      label++;
      printf("If_false t%d goto L%d\n", aux, aux2);
      quad0->address_1.kind = Temp;
      quad0->address_1.value = aux;
      quad0->address_2.kind = Empty;
      quad0->address_3.kind = LabAddr;
      quad0->address_3.value = aux2;
      quad0->op = IffK;
      insert_quadruple(quad_list, quad0);
      store_quadruple(quad0->op, quad0->address_1.kind, quad0->
         address_2.kind, quad0->address_3.kind,
                      quad0->address_1.value, quad0->address_2.
                         value, quad0->address_3.value,
                      quad0->address_1.name, quad0->address_2.name,
                          quad0->address_3.name);

      //add quad0
    }
    // then
    if(c1)
      generate_intermediate_code(quad_list, c1, table);
    if (c2) {
      aux3 = label;
      label++;
      printf("goto L%d\n", aux3);
```

```
      quad1->address_1.kind = Empty;
      quad1->address_2.kind = Empty;
      quad1->address_3.kind = LabAddr;
      quad1->address_3.value = aux3;
      quad1->op = GtoK;
      insert_quadruple(quad_list, quad1);
      store_quadruple(quad1->op, quad1->address_1.kind, quad1->
         address_2.kind, quad1->address_3.kind,
                      quad1->address_1.value, quad1->address_2.
                         value, quad1->address_3.value,
                      quad1->address_1.name, quad1->address_2.name,
                          quad1->address_3.name);

      //add quad1
    }
    printf("L%d: ", aux2);
    quad2->address_1.kind = Empty;
    quad2->address_2.kind = Empty;
    quad2->address_3.kind = LabAddr;
    quad2->address_3.value = aux2;
    quad2->op = LblK;
    insert_quadruple(quad_list, quad2);
    store_quadruple(quad2->op, quad2->address_1.kind, quad2->
       address_2.kind, quad2->address_3.kind,
                    quad2->address_1.value, quad2->address_2.value,
                        quad2->address_3.value,
                    quad2->address_1.name, quad2->address_2.name,
                       quad2->address_3.name);
    // add quad2
    // else
    if(c2){
      generate_intermediate_code(quad_list, c2, table);
      quad3->address_1.kind = Empty;
      quad3->address_2.kind = Empty;
      quad3->address_3.kind = LabAddr;
      quad3->address_3.value = aux3;
      quad3->op = LblK;
      insert_quadruple(quad_list, quad3);
      store_quadruple(quad3->op, quad3->address_1.kind, quad3->
         address_2.kind, quad3->address_3.kind,
                      quad3->address_1.value, quad3->address_2.
                         value, quad3->address_3.value,
                      quad3->address_1.name, quad3->address_2.name,
                          quad3->address_3.name);
      // add quad3
      printf("L%d: ", aux3);
    }
    break;
  case WhileK:
    if (c0) {
      aux2 = label;
      label++;
      aux3 = label;
      label++;
      printf("L%d: ", aux2);
      quad0->address_1.kind = Empty;
      quad0->address_2.kind = Empty;
      quad0->address_3.kind = LabAddr;
      quad0->address_3.value = aux2;
      quad0->op = LblK;
```

```
    insert_quadruple(quad_list, quad0);
    store_quadruple(quad0->op, quad0->address_1.kind, quad0->
       address_2.kind, quad0->address_3.kind,
                    quad0->address_1.value, quad0->address_2.
                       value, quad0->address_3.value,
                    quad0->address_1.name, quad0->address_2.name,
                        quad0->address_3.name);

    // add quad0
    generate_intermediate_code(quad_list, c0, table);
    aux = temporary;
    printf("If_false t%d goto L%d\n", aux, aux3);
    quad1->address_1.kind = Temp;
    quad1->address_1.value = aux;
    quad1->address_2.kind = Empty;
    quad1->address_3.kind = LabAddr;
    quad1->address_3.value = aux3;
    quad1->op = IffK;
    insert_quadruple(quad_list, quad1);
    store_quadruple(quad1->op, quad1->address_1.kind, quad1->
       address_2.kind, quad1->address_3.kind,
                    quad1->address_1.value, quad1->address_2.
                       value, quad1->address_3.value,
                    quad1->address_1.name, quad1->address_2.name,
                        quad1->address_3.name);

    // add quad1
}
    if(c1)
      generate_intermediate_code(quad_list, c1, table);
    printf("goto L%d\n", aux2);
    quad2->address_1.kind = Empty;
    quad2->address_2.kind = Empty;
    quad2->address_3.kind = LabAddr;
    quad2->address_3.value = aux2;
    quad2->op = GtoK;
    insert_quadruple(quad_list, quad2);
    store_quadruple(quad2->op, quad2->address_1.kind, quad2->
       address_2.kind, quad2->address_3.kind,
                    quad2->address_1.value, quad2->address_2.
                       value, quad2->address_3.value,
                    quad2->address_1.name, quad2->address_2.name,
                        quad2->address_3.name);
    // add quad2

    printf("L%d: ", aux3);
    quad3->address_1.kind = Empty;
    quad3->address_2.kind = Empty;
    quad3->address_3.kind = LabAddr;
    quad3->address_3.value = aux3;
    quad3->op = LblK;
    insert_quadruple(quad_list, quad3);
    store_quadruple(quad3->op, quad3->address_1.kind, quad3->
       address_2.kind, quad3->address_3.kind,
                    quad3->address_1.value, quad3->address_2.
                       value, quad3->address_3.value,
                    quad3->address_1.name, quad3->address_2.name,
                        quad3->address_3.name);
    // add quad3
  break;
```

```
case AssignK://PROBLEMS
  // printf("AssignK\n");
  generate_intermediate_code(quad_list, c1, table);

  printf("%s", c0->attr.name);
  quad0->op = AsvK;
  quad0->address_1.kind = Temp;
  quad0->address_1.value = temporary;
  quad0->address_2.kind = Empty;
  quad0->address_3.kind = String;
  strcpy(quad0->address_3.name, c0->attr.name);
  if (c0->child[0]) {
    quad0->op = AsaK;
    TreeNode *g0 = c0->child[0];
    if (g0->kind.exp==ConstK) {
      quad0->address_2.kind = IntConst;
      quad0->address_2.value = g0->attr.value;
      printf("[%d] =", quad0->address_2.value);
    }else if (g0->kind.exp==VarK) {
      quad0->address_2.kind = String;
      strcpy(quad0->address_2.name, g0->attr.name);
      printf("[%s] =", quad0->address_2.name);
      strcpy(quad0->scope, g0->scope);

    }else{
      generate_intermediate_code(quad_list, c0, table);
      aux = temporary;
      quad0->address_2.kind = Temp;
      quad0->address_2.value = aux;
      printf("[t%d] =", quad0->address_2.value);
    }
  }else
    printf(" =");

  // TreeNode *g1 = c1->child[0];
  // if(g1)
  // while (g1->child[0]) {
  //   g1 = g1->child[0];
  // }
  // if(c1->kind.exp == CallK){
  //   generate_intermediate_code(quad_list, c1);
  //   quad0->address_1.kind = Temp;
  //   quad0->address_1.value = temporary;
  //   temporary++;
  //   printf(" t%d\n", quad0->address_1.value);
  // }else if(g1){
  // if(g1->kind.exp = ConstK){
  //     quad0->address_1.kind = IntConst;
  //     quad0->address_1.value = g1->attr.value;
  //     printf(" %d\n", quad0->address_1.value);
  // }else if(g1->kind.exp = VarK){
  //     quad0->address_1.kind = String;
  //     strcpy(quad0->address_1.name, g1->attr.name);
  //     printf(" %s\n", quad0->address_1.name);
  // }
  // }
  // else {
  //   generate_intermediate_code(quad_list, c1);

    printf(" t%d\n", quad0->address_1.value);
```

```
  // }
    // quad0->next = NULL;
    insert_quadruple(quad_list, quad0);
    store_quadruple(quad0->op, quad0->address_1.kind, quad0->
       address_2.kind, quad0->address_3.kind,
                    quad0->address_1.value, quad0->address_2.
                       value, quad0->address_3.value,
                    quad0->address_1.name, quad0->address_2.name,
                        quad0->address_3.name);
  break;
case ReturnK:
  // printf("ReturnK\n");
  quad0->address_1.kind = Empty;
  quad0->address_2.kind = Empty;
  if (!c0) {
    printf("Return void\n");
    quad0->address_3.kind = Empty;
  }else{
    TreeNode *g1 = c0->child[0];
    if(g1){
      generate_intermediate_code(quad_list, c0, table);
      aux = temporary;
      printf("Return t%d\n", aux);
      quad0->address_3.kind = Temp;
      quad0->address_3.value = aux;
    }else{
      if (c0->kind.stmt == ConstK) {
        printf("Return %d\n", c0->attr.value);
        quad0->address_3.kind = IntConst;
        quad0->address_3.value = c0->attr.value;
      }else{
        printf("Return %s\n", c0->attr.name);
        quad0->address_3.kind = String;
        strcpy(quad0->address_3.name, c0->attr.name);
      }
    }
  }
  quad0->op = RetK;
  insert_quadruple(quad_list, quad0);
  store_quadruple(quad0->op, quad0->address_1.kind, quad0->
     address_2.kind, quad0->address_3.kind,
                  quad0->address_1.value, quad0->address_2.value,
                      quad0->address_3.value,
                  quad0->address_1.name, quad0->address_2.name,
                     quad0->address_3.name);
  // add quad0
  break;
case VarK:
  // printf("Vark\n");
  // not used in intermediate code generation
  break;
case VecK:
  // printf("Veck\n");
  // not used in intermediate code generation
  break;
case FuncK:
  // printf("FuncK\n");
  printf("%s: ", tree->attr.name);
  quad0->address_1.kind = Empty;
  quad0->address_2.kind = Empty;
```

```
      quad0->address_3.kind = String;
      strcpy(quad0->address_3.name, tree->attr.name);
      quad0->op = LblK;
      insert_quadruple(quad_list, quad0);
      store_quadruple(quad0->op, quad0->address_1.kind, quad0->
         address_2.kind, quad0->address_3.kind,
                      quad0->address_1.value, quad0->address_2.value,
                          quad0->address_3.value,
                      quad0->address_1.name, quad0->address_2.name,
                         quad0->address_3.name);

      // add quad0
      if (c0) {
        index_parameters = 1;
        generate_intermediate_code(quad_list, c0, table);
      }
      if (c1) {
        generate_intermediate_code(quad_list, c1, table);
      }
      if(!strcmp(tree->attr.name, "main")){
        break;
      }
      quad1->address_1.kind = Empty;
      quad1->address_2.kind = Empty;
      quad1->address_3.kind = String;
      strcpy(quad1->address_3.name, tree->attr.name);
      quad1->op = EofK;
      insert_quadruple(quad_list, quad1);
      break;
    case FuncVecK:
      strcpy(quad0->address_1.name, tree->attr.name);
      quad0->address_1.kind = String;
      strcpy(quad0->address_2.name, tree->scope);
      quad0->address_2.kind = String;
      quad0->address_3.value = index_parameters;
      quad0->address_3.kind = IntConst;
      quad0->op = PrmArrK;
      insert_quadruple(quad_list, quad0);
      index_parameters++;
      break;
    case FuncVarK:
      strcpy(quad0->address_1.name, tree->attr.name);
      quad0->address_1.kind = String;
      strcpy(quad0->address_2.name, tree->scope);
      quad0->address_2.kind = String;
      quad0->address_3.value = index_parameters;
      quad0->address_3.kind = IntConst;
      quad0->op = PrmVarK;
      insert_quadruple(quad_list, quad0);
      index_parameters++;
      break;
    default:
    printf("Oops: uncomment something!\n");
      break;
  }
}

static void generate_expression(list_quadruple *quad_list, TreeNode *
   tree, TipoLista *table) {
  TreeNode *c0, *c1;
```

```
c0 = tree->child[0];
c1 = tree->child[1];
int count;
int i;
int aux = 0;

quadruple *quad0 = malloc(sizeof(quadruple));
quadruple *quad1 = malloc(sizeof(quadruple));
quadruple *quad2 = malloc(sizeof(quadruple));

quadruple *quad_aux[9];

for (i = 0; i < 9; i++) {
  quad_aux[i] = malloc(sizeof(quadruple));
}

quad0->address_1.kind = Empty;
quad0->address_2.kind = Empty;
quad0->address_3.kind = Empty;

quad1->address_1.kind = Empty;
quad1->address_2.kind = Empty;
quad1->address_3.kind = Empty;

quad2->address_1.kind = Empty;
quad2->address_2.kind = Empty;
quad2->address_3.kind = Empty;

switch (tree->kind.exp) {
  case TypeK:
    generate_intermediate_code(quad_list, c0, table);
    break;
  case RelOpK:
    generate_relop(quad_list, tree, table);//print nothing
    break;
  case ArithOpK:
    generate_arithop(quad_list, tree, table);
    break;
  case ConstK:
    quad0->address_1.value = tree->attr.value;
    quad0->address_1.kind = IntConst;
    quad0->address_3.kind = Temp;
    quad0->address_3.value = temporary+1;
    quad0->op = CstK;
    insert_quadruple(quad_list, quad0);
    store_quadruple(quad0->op, quad0->address_1.kind, quad0->
       address_2.kind, quad0->address_3.kind,
                    quad0->address_1.value, quad0->address_2.value,
                        quad0->address_3.value,
                    quad0->address_1.name, quad0->address_2.name,
                       quad0->address_3.name);
    temporary++;
    printf("t%d = %d\n", quad0->address_3.value, quad0->address_1.
       value);
    break;
  case IdK:
  strcpy(quad0->address_1.name, tree->attr.name);
  quad0->address_1.kind = String;
  quad0->address_3.kind = Temp;
  quad0->address_3.value = temporary+1;
```

```
quad0->op = VstK;
insert_quadruple(quad_list, quad0);
store_quadruple(quad0->op, quad0->address_1.kind, quad0->
    address_2.kind, quad0->address_3.kind,
                quad0->address_1.value, quad0->address_2.value,
                    quad0->address_3.value,
                quad0->address_1.name, quad0->address_2.name,
                    quad0->address_3.name);
temporary++;
printf("t%d = %s\n", quad0->address_3.value, quad0->address_1.
    name);
  break;
case VecIndexK:
strcpy(quad0->address_1.name, tree->attr.name);
quad0->address_1.kind = String;

if (c0->kind.exp==ConstK) {
  quad0->address_2.kind = IntConst;
  quad0->address_2.value = c0->attr.value;
}else if (c0->kind.exp==VarK) {
  quad0->address_2.kind = String;
  strcpy(quad0->address_2.name, c0->attr.name);
  strcpy(quad0->scope, tree->scope);

}else{
  generate_intermediate_code(quad_list, c0, table);
  aux = temporary;
  quad0->address_2.kind = Temp;
  quad0->address_2.value = aux;
}

quad0->address_3.kind = Temp;
quad0->address_3.value = temporary+1;
quad0->op = AstK;
insert_quadruple(quad_list, quad0);
store_quadruple(quad0->op, quad0->address_1.kind, quad0->
    address_2.kind, quad0->address_3.kind,
                quad0->address_1.value, quad0->address_2.value,
                    quad0->address_3.value,
                quad0->address_1.name, quad0->address_2.name,
                    quad0->address_3.name);
temporary++;
switch (c0->kind.exp) {
  case ConstK:
    printf("t%d = %s[%d]\n", quad0->address_3.value, quad0->
        address_1.name, quad0->address_2.value);
    break;
  case VarK:
    printf("t%d = %s[%s]\n", quad0->address_3.value, quad0->
        address_1.name, quad0->address_2.name);
    break;
  default:
    printf("t%d = %s[t%d]\n", quad0->address_3.value, quad0->
        address_1.name, quad0->address_2.value);
    break;
  }


  break;
case CallK:
```

```
i=0;

count = 0;
quad0->address_3.kind = String;
strcpy(quad0->address_3.name, tree->attr.name);
if (!strcmp(quad0->address_3.name, "input")) {
  quad0->op = InnK;
}else{
  quad0->op = CalK;
}

if (!c0) {
  quad0->address_1.kind = Empty;
  if (check_if_int(table, quad0->address_3.name)) {
    temporary++;
    quad0->address_2.kind = Temp;
    quad0->address_2.value = temporary;
  }else{
    quad0->address_2.kind = Empty;
  }

  insert_quadruple(quad_list, quad0);

  printf("t%d = call %s, 0\n", temporary, tree->attr.name);
  // store_quadruple(quad0->op, quad0->address_1.kind, quad0->
     address_2.kind, quad0->address_3.kind,
                  // quad0->address_1.value, quad0->address_2.
                     value, quad0->address_3.value,
                  // quad0->address_1.name, quad0->address_2.
                     name, quad0->address_3.name);
  break;
}

do{

        if (c0->child[0]) {
          int aux;

          generate_children_code(quad_list, c0, table);

          quad0 = malloc(sizeof(quadruple));
          quad0->address_1.kind = Empty;
          quad0->address_2.kind = Empty;
          quad0->address_3.kind = Empty;

          aux = temporary;
          printf("param t%d\n", aux);
          quad0->address_1.kind = Empty;
          quad0->address_2.kind = Empty;
          quad0->address_3.kind = Temp;
          quad0->address_3.value = aux;
        }else if(!c0->child[0]){
          switch (c0->kind.exp) {
            case ConstK:
              printf("param %d\n", c0->attr.value);
              quad0->address_1.kind = Empty;
              quad0->address_2.kind = Empty;
              quad0->address_3.kind = IntConst;
              quad0->address_3.value = c0->attr.value;
```

```
                break;
              case VarK:
                printf("param %s\n", c0->attr.name);
                quad0->address_1.kind = Empty;
                quad0->address_2.kind = Empty;
                quad0->address_3.kind = String;
                strcpy(quad0->address_3.name, c0->attr.name);
                strcpy(quad0->scope, tree->scope);
                break;
            }
          }
          count++;
            c0 = c0->sibling;

            quad0->op = PrmK;
            insert_quadruple(quad_list, quad0);
            store_quadruple(quad0->op, quad0->address_1.kind,
               quad0->address_2.kind, quad0->address_3.kind,
                           quad0->address_1.value, quad0->
                              address_2.value, quad0->address_3.
                              value,
                           quad0->address_1.name, quad0->
                              address_2.name, quad0->address_3.
                              name);
          // add quad0
        }while (c0);

        quad1->address_3.kind = String;
        strcpy(quad1->address_3.name, tree->attr.name);
        if (check_if_int(table, quad1->address_3.name)) {
          temporary++;
          quad1->address_2.kind = Temp;
          quad1->address_2.value = temporary;
        }else{
          quad1->address_2.kind = Empty;
        }
          quad1->address_1.kind = IntConst;
          quad1->address_1.value = count;
          // quad1->address_2.kind = Temp;
          // quad1->address_2.value = temporary;

          if (!strcmp(quad1->address_3.name,"output")) {
            quad1->op = OutK;
          }else{
            quad1->op = CalK;
          }

          insert_quadruple(quad_list, quad1);
          store_quadruple(quad1->op, quad1->address_1.kind, quad1
             ->address_2.kind, quad1->address_3.kind,
                          quad1->address_1.value, quad1->
                             address_2.value, quad1->address_3.
                             value,
                          quad1->address_1.name, quad1->address_2
                             .name, quad1->address_3.name);


    printf("t%d = call %s, %d\n", temporary, tree->attr.name,
       count);
//   }
```

```
    //

    //
    //   // add quad1
      break;
    // }
    default:
      break;
  }
}

void generate_intermediate_code(list_quadruple *quad_list, TreeNode *
    tree, TipoLista *table){
  if(tree!=NULL){
    switch (tree->nodekind) {
      case StmtK:
        // printf("statement\n");
        generate_statement(quad_list, tree, table);
        break;
      case ExpK:
        // printf("expression\n");
        generate_expression(quad_list, tree, table);
        break;
      default:
        break;
    }
    generate_intermediate_code(quad_list, tree->sibling, table);
  }
}

void generate_children_code(list_quadruple *quad_list, TreeNode *tree
    , TipoLista *table){
  if(tree!=NULL){
    switch (tree->nodekind) {
      case StmtK:
        // printf("statement\n");
        generate_statement(quad_list, tree, table);
        break;
      case ExpK:
        // printf("expression\n");
        generate_expression(quad_list, tree, table);
        break;
      default:
        break;
    }
  }
}

void generate_icode_launcher(list_quadruple *quad_list, TreeNode *
    tree, TipoLista *vetor){
    int i;
    file_quadruples = fopen("file_quadruples.txt", "w");
    quad_list->start = NULL;

    quadruple *quad = malloc(sizeof(quadruple));
    quad->address_1.kind = Empty;
    quad->address_2.kind = Empty;
    quad->address_3.kind = Empty;
    quad->op = NopK;
    insert_quadruple(quad_list, quad);
```

```
    quad->address_3.kind = String;
    strcpy(quad->address_3.name, "main");
    quad->op = GtoK;
    insert_quadruple(quad_list, quad);

    generate_intermediate_code(quad_list, tree, vetor);
    quad->address_3.kind = Empty;
    quad->op = HltK;
    insert_quadruple(quad_list, quad);

    fclose( file_quadruples );



    add_indexes_to_table(vetor, quad_list);
    add_parameters_to_table(vetor, quad_list);
    printf("\n\n\n\n\n\n\n\n\n");
    //
    print_quadruple_list(quad_list);

    // file_read_quadruples = fopen("file_quadruples.txt", "r");
    // quad_list->start = NULL;
    // printf("\n\n\n\n\n\n\n\n\n");
    // insert_quadruples(quad_list);
    // print_quadruple_list(quad_list);
    // fclose( file_read_quadruples );
}
```

Algoritmo 5: intermediate.h - cabeçalhos da geração de código intermediário

```
/****************************************************/
/* File: parse.h                                    */
/* The parser interface for the C- compiler         */
/****************************************************/

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

#ifndef _INTERMEDIATE_H
#define _INTERMEDIATE_H_


/* Function parse returns the newly
 * constructed syntax tree
 */
typedef enum {
  // arithmetic [0~3]
  AddK, SubK, TimK, OvrK,
  // relational [4~9]
  EqlK, NeqK, GtrK, GeqK, LsrK, LeqK,
  // data transfer [10~14]
  AsvK, AsaK, CstK, VstK, AstK,
  // i/o [15,16]
  InnK, OutK,
  // function/procedure [17~19]
  PrmK, CalK, RetK,
  // flow [20~23]
  IffK, GtoK, HltK, LblK,
  //control
```

```
  PrmArrK, PrmVarK, EofK, NopK
 } OpKind;

typedef enum {Empty, IntConst, String, Temp, LabAddr} AddrKind;

typedef struct{
    AddrKind kind;
    int value;
    char name[20];
} Address;

typedef struct quadruple{
  int index;
  char scope[50];
  OpKind op;
  Address address_1, address_2, address_3;
  struct quadruple *next;
}quadruple;

typedef struct{
    quadruple *start;
}list_quadruple;

void insert_quadruple(list_quadruple *quad_list, quadruple *quad);

int string2int(const char *num);

void insert_quadruples(list_quadruple *quad_list);

void printWTable(TipoLista *lista, int index);

void generate_icode_launcher(list_quadruple *quad_list, TreeNode *
    tree, TipoLista *vetor);

static void generate_statement(list_quadruple *quad_list, TreeNode *
    tree, TipoLista *table);

void generate_intermediate_code(list_quadruple *quad_list, TreeNode *
    tree, TipoLista *table);

void add_parameters_to_table(TipoLista *table, list_quadruple *
    quad_list);

void print_quadruple_list(list_quadruple *quad_list);

void store_quadruple(OpKind o, AddrKind k1, AddrKind k2, AddrKind k3,
                     int v1, int v2, int v3,
                     char n1[], char n2[], char n3[]);

void add_indexes_to_table(TipoLista *table, list_quadruple *quad_list
    );

void generate_children_code(list_quadruple *quad_list, TreeNode *tree
    , TipoLista *table);



#endif
```

Algoritmo 6: util.c - funções suplementares

```c
/****************************************************/
/* File: util.c                                     */
/* Utility function implementation                  */
/* for the C- compiler                              */
/****************************************************/

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

/* Procedure customPrint customizes print
 * of text in bold to the listing file
 */
void customPrint(const char* string, int bold){
    if(bold == TRUE)
        printf( BOLD "%s" RESET, string);
    else
        printf( "%s", string);
}

/* Procedure printToken prints a token
 * and its lexeme to the listing file
 */
void printToken(TokenType token, const char* tokenString){
    switch(token){
        case IF:
            printf( "Keyword: if\n");
            break;
        case ELSE:
            printf( "Keyword: else\n");
            break;
        case WHILE:
            printf( "Keyword: while\n");
            break;
        case INT:
            printf( "Keyword: int\n");
            break;
        case VOID:
            printf( "Keyword: void\n");
            break;
        case RETURN:
            printf( "Keyword: return\n");
            break;
        case ASSIGN:
            printf( "Special symbol: =\n");
            break;
        case LT:
            printf( "Special symbol: <\n");
            break;
        case LET:
            printf( "Special symbol: <=\n");
            break;
        case HT:
            printf( "Special symbol: >\n");
            break;
        case HET:
            printf( "Special symbol: >=\n");
            break;
        case EQ:
            printf( "Special symbol: ==\n");
            break;
```

```
            case NEQ:
                printf( "Special symbol: !=\n");
                break;
            case LPAREN:
                printf( "Special symbol: (\n");
                break;
            case RPAREN:
                printf( "Special symbol: )\n");
                break;
            case LBRACK:
                printf( "Special symbol: [\n");
                break;
            case RBRACK:
                printf( "Special symbol: ]\n");
                break;
            case LCAPSULE:
                printf( "Special symbol: {\n");
                break;
            case RCAPSULE:
                printf( "Special symbol: }\n");
                break;
            case SEMI:
                printf( "Special symbol: ;\n");
                break;
            case COMMA:
                printf( "Special symbol: ,\n");
                break;
            case PLUS:
                printf( "Special symbol: +\n");
                break;
            case MINUS:
                printf( "Special symbol: -\n");
                break;
            case TIMES:
                printf( "Special symbol: *\n");
                break;
            case OVER:
                printf( "Special symbol: /\n");
                break;
            case ENDFILE:
                printf("End of file (EOF)\n");
                break;
            case NUM:
                printf( "Number: %s\n", tokenString);
                break;
            case ID:
                printf( "Identifier: %s\n", tokenString);
                break;
            case ERROR:
                printf( "Error: %s\n", tokenString);
                break;
            default: /* should never happen */
                printf("Unknown token: %d\n", token);
        }
    }

    /* Function newStmtNode creates a new statement
     * node for syntax tree construction
     */
    TreeNode * newStmtNode(StmtKind kind){
```

```
    TreeNode * t = (TreeNode *) malloc(sizeof(TreeNode));
    int i;
    if(t == NULL)
        printf( "Out of memory error at line %d\n", linenumber);
    else{
        for(i = 0; i < MAXCHILDREN; ++i)
            t->child[i] = NULL;
        t->sibling = NULL;
        t->nodekind = StmtK;
        t->kind.stmt = kind;
        t->linenumber = linenumber;
        t->scope = "global";
    }
    return t;
}

/* Function newExpNode creates a new expression
 * node for syntax tree construction
 */
TreeNode * newExpNode(ExpKind kind){
    TreeNode * t = (TreeNode *) malloc(sizeof(TreeNode));
    int i;
    if(t == NULL)
        printf( "Out of memory error at line %d\n", linenumber);
    else{
        for(i = 0; i < MAXCHILDREN; ++i)
            t->child[i] = NULL;
        t->sibling = NULL;
        t->nodekind = ExpK;
        t->kind.exp = kind;
        t->linenumber = linenumber;
        t->type = Void;
        t->scope = "global";
    }
    return t;
}

/* Function countParamArg counts how many parameters/arguments
 * exist in the function declaration/call
 */
int countParamArg(TreeNode * t){
    int count = 0;
    while(t != NULL)
        t = t->sibling, ++count;
    return count;
}

/* Function getParamArgName gets parameter's/argument's name
 * of a function declaration/call
 */
char * getParamArgName(TreeNode * t, int index){
    int i = 0;
    while(t != NULL && i < index)
        t = t->sibling, ++i;
    return copyString(t->child[0]->attr.name);
}

/* Function copyString allocates and makes a new
 * copy of an existing string
 */
```

```
char * copyString(char * s){
    int n;
    char * t;
    if(s == NULL)
        return NULL;
    n = strlen(s)+1;
    t = (char *) malloc(n*sizeof(char));
    if(t == NULL)
        printf( "Out of memory error at line %d\n", linenumber);
    else
        strcpy(t, s);
    return t;
}

/* Function joinNameScope joins name and scope to
 * produce new scope for all children nodes of a function
 */
char * joinNameScope(char * name, char * sep, char * scope){
    char * newScope;
    int length = strlen(name)+strlen(sep)+strlen(scope)+1;
    newScope = (char *) malloc(length*sizeof(char));
    if(newScope == NULL)
        printf( "Out of memory error at line %d\n", linenumber);
    else{
        strcpy(newScope, name);
        strcat(newScope, sep);
        strcat(newScope, scope);
    }
    return newScope;
}

/* Function setScope defines scope for all children
 * nodes of a function
 */
void setScope(TreeNode * t, char * scope){
    while(t != NULL){
        t->scope = copyString(scope);
        int i;
        if(t->nodekind == StmtK && t->kind.stmt == FuncK){
            char * newScope = joinNameScope(t->attr.name, "/", scope)
                ;
            for(i = 0; i < MAXCHILDREN; ++i)
                setScope(t->child[i], newScope);
        }
        else{
            for(i = 0; i < MAXCHILDREN; ++i)
                setScope(t->child[i], scope);
        }
        t = t->sibling;
    }
}

/* Variable indentno is used by printTree to
 * store current number of spaces to indent
 */
static int indentno = 0;

/* macros to increase/decrease indentation */
#define INDENT indentno+=4
#define UNINDENT indentno-=4
```

```
/* printSpaces indents by printing spaces */
static void printSpaces(){
    int i;
    for(i = 0; i < indentno; ++i)
        printf( " ");
}

/* procedure printTree prints a syntax tree to the
 * listing file using indentation to indicate subtrees
 */
void printTree(TreeNode * tree){
    int i;
    INDENT;
    while(tree != NULL){
        printSpaces();
        if(tree->nodekind == StmtK){
            switch(tree->kind.stmt){
                case IfK:
                    if(tree->child[2] != NULL)
                        printf( "If-Else:\n");
                    else
                        printf( "If:\n");
                    break;
                case WhileK:
                    printf( "While:\n");
                    break;
                case AssignK:
                    printf( "Assign to:\n");
                    break;
                case ReturnK:
                    if(tree->child[0] != NULL)
                        printf( "Return:\n");
                    else
                        printf( "Return\n");
                    break;
                case VarK:
                    printf( "Variable declaration: %s \n", tree->attr
                        .name);
                    break;
                case VecK:
                    printf( "Vector declaration: %s[%d]\n", tree->
                        attr.name, tree->attr.value);
                    break;
                case FuncK:
                    if(tree->child[0] != NULL){
                        int k = countParamArg(tree->child[0]);
                        printf( "Function declaration: %s(%d
                            parameter%s)\n",
                                tree->attr.name, k, ((k>1) ? "s" : ""
                                    ));
                    }
                    else
                        printf( "Function declaration: %s(void)\n",
                            tree->attr.name);
                    break;
                case FuncVarK:
                    printf( "Variable: %s\n", tree->attr.name);
                    break;
                case FuncVecK:
```

```
                    printf( "Vector: %s[]\n", tree->attr.name);
                    break;
                default:
                    printf( "Unknown StmtNode kind\n");
                    break;
            }
        }
        else if(tree->nodekind == ExpK){
            switch(tree->kind.exp){
                case TypeK:
                    if(tree->type == Integer)
                        printf( "Type: Integer\n");
                    else if(tree->type == Void)
                        printf( "Type: Void\n");
                    break;
                case RelOpK:
                    printf( "Relational operator: %s\n", tree->attr.
                       name);
                    break;
                case ArithOpK:
                    printf( "Arithmetic operator: %s\n", tree->attr.
                       name);
                    break;
                case ConstK:
                    printf( "Constant: %d\n", tree->attr.value);
                    break;
                case IdK:
                    printf( "Identifier: %s Scope: %s\n", tree->attr.
                       name, tree->scope);
                    break;
                case VecIndexK:
                    printf( "Vector index: %s[]\n", tree->attr.name);
                    break;
                case CallK:
                    if(tree->child[0] != NULL){
                        int k = countParamArg(tree->child[0]);
                        printf( "Function call: %s(%d argument%s)\n",
                                tree->attr.name, k, ((k>1) ? "s" : ""
                                   ));
                    }
                    else
                        printf( "Function call: %s()\n", tree->attr.
                           name);
                    break;
                default:
                    printf( "Unknown ExpNode kind\n");
                    break;
            }
        }
        else
            printf( "Unknown node kind\n");
        for(i = 0; i < MAXCHILDREN; ++i)
            printTree(tree->child[i]);
        tree = tree->sibling;
    }
    UNINDENT;
}
```

Algoritmo 7: object.c - Geração de Código Objeto

```
//RISK: READING AND WRITING IN THE SAME REGISTER

#include "object.h"



int register_zero = 0;
int register_result = 1;
int register_operator_loader = 2;
int register_operator_left = 3;
int register_operator_right = 4;
int register_context_offset = 5;
int register_operator_offset = 6;
int register_return = 30;
int register_top = 31;
int line_counter = 0;
int memory_index = 0;
int flag_first_call = 0;
int absolute_size = 0;


/* macro to control processor register file overflow */
#define MAXREGISTER 20
#define OFFSET 7


/* array to control available temporaries
* at current intermediate code instruction
*/
static int temporaries[MAXREGISTER];

/* Function get_temporary returns
* next available temporary,
* terminating execution of the program
* if there is no available temporary
*/
void map_temporary(list_instructions *instructions_list, int
   register_temporary, int register_source){
        int i, temp = -1;
        for(i = 0; i < MAXREGISTER; ++i){
                if(temporaries[i] == 0){
                        temporaries[i] = register_temporary;
                        temp = i+OFFSET;
                        break;
                }
        }
        if(temp < OFFSET){
                printf("ERROR: register file overflow: no temporaries
                    available at requested instruction\n");
                exit(0);
        }
        format_two(instructions_list, G_ADDI, register_source, temp,
           0, "none");
}

/* Function releaseTemp releases
* specified temporary
*/
void release_temporary(int temp){
        int t = temp-OFFSET;
```

```
        if(t >= 0 && t < MAXREGISTER)
        temporaries[t] = 0;
}

/*Function search_temporary searches
a specific temporary*/
int search_temporary(int index_temporary){
        int i;
        for (i = 0; i < MAXREGISTER; i++) {
                if (temporaries[i]==index_temporary) {
                        return i+OFFSET;
                }
        }
}

// void copy_list_parameters(list_parameters *source_list,
   list_parameters *target_list){
//      type_parameter *source_parameter = source_list->start;
//      type_parameter *target_parameter;
//
//      while (source_parameter!=NULL) {
//
//              target_parameter = target_list->start;
//
//              type_parameter *new_parameter = malloc(sizeof(
   type_parameter));
//
//              new_parameter->kind = source_parameter->kind;
//              new_parameter->value = source_parameter->value;
//              strcpy(new_parameter->name, source_parameter->name);
//              strcpy(new_parameter->scope, source_parameter->scope)
   ;
//
//              if(target_parameter==NULL){
//                      target_list->start = new_parameter;
//                      target_list->start->next = NULL;
//              }
//              else{
//                      while(target_parameter->next!=NULL){
//                              target_parameter = target_parameter->
   next;
//                      }
//                      target_parameter->next = new_parameter;
//                      target_parameter->next->next = NULL;
//              }
//
//              source_parameter = source_parameter->next;
//      }
// }

void treat_jumps_n_branches(list_instructions *instructions_list,
   list_labels *labels_list, list_labels *calls_list){
        type_label *label = labels_list->start;
        type_label *call = calls_list->start;
        type_instruction *instruction;

        while (label!=NULL) {
                switch (label->type) {
                        case LabAddr:
                                instruction = instructions_list->
```

```
    start;
while (instruction!=NULL) {
        if (label->index==instruction
            ->target_label) {
                if (instruction->jump
                    ==label_kind) {
                        if (!strcmp(
                            instruction
                            ->
                            label_name
                            , "none"))
                             {
                        if (
                            instruction
                            ->type==
                            G_BOZ) {
                                instruction
                                    ->
                                    immediate
                                     =

                                    label
                                    ->
                                    line
                                    -
                                    instruction
                                    ->
                                    line
                                    -1;

                        }else if(
                            instruction
                            ->type==
                            G_JMP){
                                instruction
                                    ->
                                    immediate
                                     =

                                    label
                                    ->
                                    line
                                    ;
                                //
                                    printf
                                    ("
                                    LL
                                    :
                                    %d
                                    ,
                                    LI
                                    :
                                    %d

                                    LK
                                    :
                                    %d

                                    LN
                                    :
```

```
                                                %s
                                                \n
                                                ",

                                                label
                                                ->
                                                line
                                                ,
                                                label
                                                ->
                                                index
                                                ,
                                                label
                                                ->
                                                type
                                                ,
                                                instruction
                                                ->
                                                label_name
                                                );
                                        }
                                        }
                                }
                        }
                        instruction = instruction->
                            next;
                }
                break;
                case String:
                        instruction =
                            instructions_list->start;
                        while (instruction!=NULL) {
                                if (instruction->type
                                    ==G_JMP) {
                                        if(!strcmp(
                                            instruction
                                            ->
                                            label_name
                                            , label->
                                            name)){
                                                if(
                                                    instruction
                                                    ->
                                                    jump
                                                    ==
                                                    label_kind
                                                    ){
                                                        instruc
                                                            ->
                                                            imm

                                                            =

                                                            labe
                                                            ->
                                                            lin
                                                            ;

                                                }
                                        }
```

```
                                                }
                                                instruction =
                                                    instruction->next;
                                        }
                                        break;
                                default:
                                        printf("ERROR: unknown type label: %d
                                            \n", label->type);
                                break;
                        }
                        label = label->next;
                }

                while (call!=NULL) {
                        instruction = instructions_list->start;
                        while (instruction!=NULL) {
                                if (instruction->type==G_JMP){
                                                if(!strcmp(call->name
                                                    , instruction->
                                                    label_name)){
                                                        if (
                                                            instruction
                                                            ->jump==
                                                            call_kind)
                                                             {
                                                        instruction->
                                                            immediate
                                                            = call->
                                                            line;
                                                }
                                                }
                                        }
                                        instruction = instruction->
                                            next;
                                }
                        call = call->next;
                }
        }


        void consume_parameters(TipoLista *table, list_instructions *
            instructions_list, list_parameters *parameters_list,
            list_variables *variables_list, char function[]){
                        type_variable *variable = variables_list->start;
                        type_instruction *instruction = instructions_list->
                            start;
                        type_parameter *parameter = parameters_list->start;
                        TipoID *table_item;

                        int memory_from;
                        int memory_to;
                        int temp_from;
                        int parameter_index = 1;
                        int i;
                        int array_index;
                        int arr_size;

                        while(parameter!=NULL){
                                arr_size = 0;
                                //loading parameter to register result
```

```
switch (parameter->kind){
        case String:
                //search for parameter in
                    memory
                memory_from = search_variable
                    (variables_list, parameter
                    ->name, 0, parameter->
                    scope);
                //load from memory to
                    register result
                format_one(instructions_list,
                     G_LD, register_result,
                    memory_from);
                break;
        case IntConst:
          //load parameter as immediate to
              register result
                format_one(instructions_list,
                     G_LDI, register_result,
                    parameter->value);
                break;
        case Temp:
                //search for register that
                    stores parameter
                temp_from = search_temporary(
                    parameter->value);
                //load parameter from
                    temporary to register
                    result
                format_two(instructions_list,
                     G_ADDI, temp_from,
                    register_result, 0, "none"
                    );
                break;
}

//storing parameter in memory
for ( i = 0; i < 211; i++) {
        if(&table[i]!=NULL){
                table_item = table[i].start;
                while (table_item!=NULL) {
                        if (!strcmp(
                            table_item->escopo
                            , function)) {
                                if (
                                    table_item
                                    ->
                                    indice_parametro
                                    ==
                                    parameter_index
                                    ) {
                                        if (!
                                            strcmp
                                            (
                                            table_item
                                            ->
                                            tipoID
                                            ,"
                                            var
                                            ")
```

```
)
{
    //
        sea
        for
        pos
        in
        mem
        to
        sto
        par
    memory_
        =
        sea
        (
        var
        ,
        tab
        ->
        nom
        ,
        0,
        tab
        ->
        esc
        )
        ;
    //
        sto
        val
        fro
        reg
        res
        int
        mem
    format_
        (
        ins
        ,
```

```
                G_ST
                ,
                regi
                ,
                memc
                )
                ;
        }else
            {
                //
                goes
                thro
                ,
                from
                '
                arra
                and
                stor
                valu
                inte
                ,
                to
                '
                arra
            for
                (
                arra
                =
                0;
                arra
                <=
                tabl
                ->
                arra
                ;
                arra
                ++)
                {
```

```
                                                                    }
                                                                        }
                                                                    }
                                                                    table_item =
                                                                        table_item->prox;
                                                                }
                                                            }
                                                        }

                                                        parameter_index++;
                                                        parameter = parameter->next;
                                                }
                                                // parameters_list->start = NULL;
                                                release_temporary(temp_from);
}

void array_size_placer(TipoLista *table){
        //find array size
        int i;
        int array_memory_index;
        TipoID *table_item;
        for(i = 0;i<211;i++){
                if(&table[i]!=NULL){
                        table_item = table[i].start;
                        while (table_item!=NULL) {
                                        if(!strcmp(table_item->tipoID
                                            , "vet")){
                                                if (table_item->
                                                    array_size>0) {
                                                        absolute_size
                                                            =
                                                        table_item
                                                        ->
                                                        array_size
                                                        ;
                                                }
                                        }
                                table_item = table_item->prox;
                        }
                }
        }

        //place array size
        for(i = 0;i<211;i++){
                if(&table[i]!=NULL){
```

```
                        table_item = table[i].start;
                        while (table_item!=NULL) {
                                        if(!strcmp(table_item->tipoID
                                            , "vet")){
                                                if (table_item->
                                                    array_size==0) {
                                                        table_item->
                                                            array_size
                                                             =
                                                            absolute_size
                                                            ;
                                                }
                                        }
                                table_item = table_item->prox;
                        }
                }
        }
}

void restore_arrays(TipoLista *table, list_instructions *
    instructions_list, list_parameters *parameters_list,
    list_variables *variables_list, char function[]){
                type_variable *variable = variables_list->start;
                type_instruction *instruction = instructions_list->
                    start;
                type_parameter *parameter = parameters_list->start;
                TipoID *table_item;

                int memory_from;
                int memory_to;
                int temp_from;
                int parameter_index = 1;
                int i;
                int array_index;

                while(parameter!=NULL){

                        //load variable from memory to register
                            result
                        for ( i = 0; i < 211; i++) {
                                if(&table[i]!=NULL){
                                        table_item = table[i].start;
                                        while (table_item!=NULL) {
                                                if (!strcmp(
                                                    table_item->escopo
                                                    , function)) {
                                                        if (
                                                            table_item
                                                            ->
                                                            indice_parametro
                                                            ==
                                                            parameter_index
                                                            ) {
                                                                if (!
                                                                    strcmp
                                                                    (
                                                                    table_item
                                                                    ->
                                                                    tipoID
                                                                    ,"
```

```
vet
")
)
{
    //
        goes

        thro

        ,
        from
        '

        arra

        and

        sto

        valu

        into

        ,
        to
        '

        arra

    for

        (
        arra

        =

        0;

        arra

        <=

        tabl
        ->
        arra
        ;

        arra
        ++)

        {
```

```
                                                                    }
                                                                }
                                                            }
                                                            table_item =
                                                                table_item->prox;
                                                        }
                                                    }
                                                }

                                                //loading parameter to register result
                                                // switch (parameter->kind){
                                                //      case String:
                                                //              //search for parameter in
                                                    memory
                                                //              memory_from = search_variable
                                                    (variables_list, parameter->name, 0,
                                                    parameter->scope);
                                                //              //load from memory to
                                                    register result
                                                //              format_one(instructions_list,
                                                     G_LD, register_result, memory_from);
                                                //              break;
                                                //      case IntConst:
                                                //        //load parameter as immediate to
                                                    register result
                                                //              format_one(instructions_list,
                                                     G_LDI, register_result, parameter->value)
                                                    ;
                                                //              break;
                                                //      case Temp:
                                                //              //search for register that
                                                    stores parameter
                                                //              temp_from = search_temporary(
                                                    parameter->value);
                                                //              //load parameter from
                                                    temporary to register result
                                                //              format_two(instructions_list,
                                                     G_ADDI, temp_from, register_result, 0, "
                                                    none");
                                                //              break;
                                                // }


                                                parameter_index++;
                                                parameter = parameter->next;
                                            }
                                            parameters_list->start = NULL;
}

//insertion function for variables
void insert_variable(list_variables *variables_list, int index, int
    index_array, kind_variable kind, char id[], char scope[]){
        type_variable *p = variables_list->start;
        type_variable *new_variable = malloc(sizeof(type_variable));
```

```
        new_variable->index = index;
        new_variable->index_array = index_array;
        new_variable->kind = kind;
        strcpy(new_variable->scope, scope);
        strcpy(new_variable->id, id);

        if(p==NULL){
                variables_list->start = new_variable;
                variables_list->start->next = NULL;
        }
        else{
                while(p->next!=NULL){
                        p = p->next;
                }
                p->next = new_variable;
                p->next->next = NULL;
        }
}

void insert_label(list_labels *labels_list, AddrKind type, char name
    [], int index, int line){
                                type_label *l = labels_list->start;
                                type_label *new_label = malloc(sizeof
                                    (type_label));

                                if (type==LabAddr) {
                                        new_label->index = index;
                                        strcpy(new_label->name, "none
                                            ");
                                }else{
                                        new_label->index = 0;
                                        strcpy(new_label->name, name)
                                            ;
                                }
                                new_label->type = type;
                                new_label->line = line;

                                if(l==NULL){
                                        labels_list->start =
                                            new_label;
                                        labels_list->start->next =
                                            NULL;
                                }
                                else{
                                        while(l->next!=NULL){
                                                l = l->next;
                                        }
                                        l->next = new_label;
                                        l->next->next = NULL;
                                }
                        }

// void insert_call(list_parameters *parameters_list, list_calls *
    returner_calls_list, char name[], int index, int line){
//      type_call *call = returner_calls_list->start;
//      type_call *new_call = malloc(sizeof(type_call));//malloc(
    sizeof(type_call));
//      type_parameter *source_parameter = parameters_list->start;
//      type_parameter *target_parameter;
```

```
//      type_parameter *aux;
//
//      strcpy(new_call->name, name);
//      new_call->index = index;
//      new_call->line = line;
//      new_call->parameters = malloc(sizeof(type_parameter));
//      new_call->parameters = NULL;
//      // printf("here\n");
//
//      while (source_parameter!=NULL) {
//
//
//              target_parameter = new_call->parameters;
//
//              type_parameter *new_parameter = malloc(sizeof(
    type_parameter));
//
//              new_parameter->kind = source_parameter->kind;
//              new_parameter->value = source_parameter->value;
//              strcpy(new_parameter->name, source_parameter->name);
//              strcpy(new_parameter->scope, source_parameter->scope)
    ;
//
//              // printf("AUX: %s %d\n", source_parameter->name,
    source_parameter->value);
//
//              if(target_parameter==NULL){
//                      new_call->parameters = new_parameter;
//                      new_call->parameters->next = NULL;
//              }
//              else{
//                      while(target_parameter->next!=NULL){
//                              target_parameter = target_parameter->
    next;
//                      }
//                      target_parameter->next = new_parameter;
//                      target_parameter->next->next = NULL;
//              }
//              source_parameter = source_parameter->next;
//      }
//
//      // printf("%d\n", new_call->parameters->start->value);
//      // aux = parameters_list->start;
//      // while (aux!=NULL) {
//      //      printf("AUX: %s %d\n", aux->name, aux->value);
//      //      aux = aux->next;
//      // }
//
// }


void print_parameters(list_parameters *parameters_list) {
        type_parameter *p = parameters_list->start;
        while (p!=NULL) {
                printf("kind: %d \t value: %d \t name: %s scope: %s\n
                    ", p->kind, p->value, p->name, p->scope);
                p = p->next;
        }
}
```

```
void print_variables(list_variables *variables_list) {
        type_variable *p = variables_list->start;
        while (p!=NULL) {
                printf("index: %d \t array: %d \t id: %s \t scope: %s
                    \n", p->index, p->index_array, p->id, p->scope);
                p = p->next;
        }
}

void print_instructions(list_instructions *instructions_list){
        type_instruction *p = instructions_list->start;
        while (p!=NULL) {
                printf("%4d: \ttype: %d \tsource_a: %d \tsource_b: %d
                    \n \ttarget: %d \timmediate: %d\n",
                p->line, p->type, p->register_a, p->register_b, p->
                    register_c,p->immediate);
                p = p->next;
        }
}

void print_labels(list_labels *labels_list) {
        type_label *p = labels_list->start;
        while (p!=NULL) {
                printf("NAME:%s INDEX:%d LINE:%d TYPE:%d \n", p->name
                    , p->index, p->line, p->type);
                p = p->next;
        }
}

//searches the position in memory of a variable given its name, scope
    and array index
int search_variable(list_variables *variables_list, char name[], int
   array_position, char scope[]){
        type_variable *p = variables_list->start;
        char local_scope[50];
        strcpy(local_scope, scope);

        while (p!=NULL) {
                // printf("index: %d \t array: %d \t id: %s \t scope:
                    %s\n", p->index, p->index_array, p->id, p->scope)
                    ;
                if(!strcmp(p->scope, "global")){
                        if(!strcmp(name, p->id)){
                                strcpy(local_scope, "global");
                                break;
                        }
                }
                p = p->next;
        }

        p = variables_list->start;
        while (p!=NULL) {
                if(!strcmp(p->scope, local_scope)){
                        if(!strcmp(p->id, name)){
                                // printf("SEARCH VARIABLE\n");
                                if(p->kind==variable_kind){
                                        // printf("VARIABLE INDEX: %d
                                            \n", p->index);
                                        // printf("%s %d %s\n", name,
                                            array_position, scope);
```

```
                                        return p->index;

                                    }
                                    else{  return p->index+array_position
                                        ; }
                                }
                        }
                        p = p->next;
                }

                printf("ERROR: variable not found!\n");
                printf("%s %d %s\n", name, array_position, local_scope);
                return 0;
}

//reserves spaces in memory for variables of a given function
void declaration_variables(list_variables *variables_list, TipoLista
    *table, char scope[]){
                int i;
                int array_memory_index;
                TipoID *table_item;
                for(i = 0;i<211;i++){
                        if(&table[i]!=NULL){
                                table_item = table[i].start;
                                while (table_item!=NULL) {
                                        if (!strcmp(table_item->escopo, scope
                                            )) {
                                                if (!strcmp(table_item->
                                                    tipoID, "var")) {
                                                        insert_variable(
                                                            variables_list,
                                                            memory_index, 0,
                                                            variable_kind,
                                                            table_item->nomeID
                                                            , scope);
                                                        memory_index++;
                                                }
                                                else if(!strcmp(table_item->
                                                    tipoID, "vet")){
                                                        array_memory_index =
                                                            memory_index;
                                                        for (size_t j = 0; j
                                                            <= table_item->
                                                            array_size; j++) {
                                                                insert_variable
                                                                    (
                                                                    variables_list
                                                                    ,
                                                                    array_memory_index
                                                                    , j,
                                                                    array_kind
                                                                    ,
                                                                    table_item
                                                                    ->nomeID,
                                                                    scope);
                                                                memory_index
                                                                    ++;
                                                        }
                                                }
```

```
                                }
                                table_item = table_item->prox;
                        }
                }
        }
}


//numbers relate to amount of registers in the operation
void format_zero(list_instructions *instructions_list,  galetype type
    , int immediate, AddrKind kind, char label_string[], kind_jump
    jump){

        type_instruction *p =instructions_list->start;
        type_instruction *new_instruction = malloc(sizeof(
            type_instruction));

        new_instruction->line = line_counter;
        new_instruction->register_a = 0;
        new_instruction->register_b = 0;
        new_instruction->register_c = 0;
        new_instruction->immediate = immediate;
        new_instruction->type = type;
        if(type==G_BOZ||type==G_JMP){
                // if(kind==String){
                        strcpy(new_instruction->label_name,
                            label_string);
                        // new_instruction->jump = jump;
                // }else if(kind==LabAddr){
                        new_instruction->target_label = immediate;
                        new_instruction->jump = jump;
                // }
        } else
        new_instruction->target_label = 0;


        if (p==NULL) {
                instructions_list->start = new_instruction;
                instructions_list->start->next = NULL;
        }
        else{
                while (p->next!=NULL) {
                        p = p->next;
                }
                p->next = new_instruction;
                p->next->next = NULL;
        }
        line_counter++;
}

void format_one(list_instructions *instructions_list, galetype type,
    int register_a, int immediate){
        // printf("%4d: \ttype: %d \tregister: %d \timmediate: %d\n",
             line_counter, type, register_a, immediate);

        type_instruction *p =instructions_list->start;
        type_instruction *new_instruction = malloc(sizeof(
            type_instruction));
```

```
        new_instruction->line = line_counter;
        new_instruction->register_a = register_a;
        new_instruction->register_b = 0;
        new_instruction->register_c = 0;
        new_instruction->immediate = immediate;
        new_instruction->type = type;
        new_instruction->target_label = 0;



        if (p==NULL) {
                instructions_list->start = new_instruction;
                instructions_list->start->next = NULL;
        }
        else{
                while (p->next!=NULL) {
                        p = p->next;
                }
                p->next = new_instruction;
                p->next->next = NULL;
        }
        line_counter++;
}

void format_two(list_instructions *instructions_list, galetype type,
   int register_source, int register_target, int immediate, char
   label_string[]){
        // printf("%4d: \ttype: %d \tsource: %d \ttarget: %d \
           timmediate: %d\n",
        // line_counter, type, register_source,register_target,
           immediate);

        type_instruction *p =instructions_list->start;
        type_instruction *new_instruction = malloc(sizeof(
           type_instruction));

        new_instruction->line = line_counter;
        new_instruction->register_a = register_source;
        new_instruction->register_b = 0;
        new_instruction->register_c = register_target;
        new_instruction->immediate = immediate;
        strcpy(new_instruction->label_name, label_string);
        new_instruction->type = type;
        new_instruction->target_label = 0;



        if (p==NULL) {
                instructions_list->start = new_instruction;
                instructions_list->start->next = NULL;
        }
        else{
                while (p->next!=NULL) {
                        p = p->next;
                }
                p->next = new_instruction;
                p->next->next = NULL;
        }
        line_counter++;
}
```

```
void format_three(list_instructions *instructions_list, galetype type
    , int register_source_a, int register_source_b, int
    register_target){
        // printf("%4d: \ttype: %d \tsource_a: %d \tsource_b: %d \
            ttarget: %d\n",
        // line_counter, type, register_source_a, register_source_b,
            register_target);

        type_instruction *p =instructions_list->start;
        type_instruction *new_instruction = malloc(sizeof(
            type_instruction));

        new_instruction->line = line_counter;
        new_instruction->register_a = register_source_a;
        new_instruction->register_b = register_source_b;
        new_instruction->register_c = register_target;
        new_instruction->immediate = 0;
        new_instruction->type = type;
        new_instruction->target_label = 0;



        if (p==NULL) {
                instructions_list->start = new_instruction;
                instructions_list->start->next = NULL;
        }
        else{
                while (p->next!=NULL) {
                        p = p->next;
                }
                p->next = new_instruction;
                p->next->next = NULL;
        }
        line_counter++;
}

void generate_code(list_instructions *instructions_list,
    list_quadruple *quad_list, TipoLista *table, list_variables *
    variables_list, list_parameters *parameters_list, list_labels *
    labels_list, list_labels *calls_list){
        quadruple *p = quad_list->start;
        TipoID *table_item;
        type_instruction *instruction;
        type_variable *v = variables_list->start;
        type_parameter *par;
        list_calls *returner_calls_list;
        // list_parameters *parameters_list_aux;

        returner_calls_list = (list_calls*) malloc(sizeof(list_calls)
            );
        // parameters_list_aux = (list_parameters*) malloc(sizeof(
            list_parameters));

        returner_calls_list->start = NULL;
        // parameters_list_aux->start = NULL;

        char current_scope[50];
```

```
int i;
int flag_immediate_left = 0;
int flag_immediate_right = 0;
int flag_temp_left = 0;
int flag_temp_right = 0;
int immediate_left = 0;
int immediate_right = 0;
int register_temporary_left;
int register_temporary_right;
int register_temporary;
int memory_position;
int memory_offset;
int counter;

while (p!=NULL) {
        flag_temp_left = 0;
        flag_temp_right = 0;
        flag_immediate_left = 0;
        flag_immediate_right = 0;

        //the following procedures retrieve the scope of the
           possible variable/array
        for(i = 0;i<211;i++){
                if(&table[i]!=NULL){
                        table_item = table[i].start;
                        while (table_item!=NULL) {
                                if(!strcmp(table_item->tipoID
                                    , "func")){
                                        if (p->index>
                                           table_item->
                                           intermediate_start
                                                &&p->index<
                                                   table_item
                                                   ->
                                                   intermediate_finish
                                                   ) {
                                                        strcpy
                                                           (
                                                           current_scop
                                                           ,
                                                           table_item
                                                           ->
                                                           nomeID
                                                           );
                                                        break
                                                           ;
                                                }
                                        }
                                        table_item =
                                           table_item->prox;
                                }
                        }
                }


                switch (p->op) {
                        case AddK:
                        case SubK:
```

```
//left operand
if (p->address_1.kind==Temp) {
        register_temporary_left =
           search_temporary(p->
           address_1.value);
        format_two(instructions_list,
            G_ADDI,
           register_temporary_left,
           register_operator_left, 0,
            "none");
        flag_temp_left = 1;
}else if(p->address_1.kind==String){
        int memory_position_left = 0;
        memory_position_left =
           search_variable(
           variables_list, p->
           address_1.name, 0,
           current_scope);

        format_one(instructions_list,
            G_LD,
           register_operator_left,
           memory_position_left);
}else if(p->address_1.kind==IntConst)
   {
        if(p->address_1.value<65000){
                flag_immediate_left =
                    1;
                immediate_left = p->
                   address_1.value;
        }
        else{
                format_one(
                   instructions_list,
                    G_LDI,
                   register_operator_left
                   , p->address_1.
                   value);
        }
}else{
        printf("ERROR: intermediate
           variable kind unknown: %d
           !\n", p->address_1.kind);
}

//right operand
if (p->address_2.kind==Temp) {
        register_temporary_right =
           search_temporary(p->
           address_2.value);
        format_two(instructions_list,
            G_ADDI,
           register_temporary_right,
           register_operator_right,
           0, "none");
        flag_temp_right = 1;
}else if(p->address_2.kind==String){
        int memory_position_right =
           0;
        memory_position_right =
```

```
                search_variable(
                variables_list, p->
                address_2.name, 0,
                current_scope);


            format_one(instructions_list,
                 G_LD,
                register_operator_right,
                memory_position_right);
    }else if(p->address_2.kind==IntConst)
        {
            if(p->address_2.value<65000){
                    flag_immediate_right
                        = 1;
                    immediate_right = p->
                        address_2.value;
            }
            else{
                    format_one(
                        instructions_list,
                         G_LDI,
                        register_operator_right
                        , p->address_2.
                        value);
            }
    }else{
            printf("ERROR: intermediate
                variable kind unknown: %d
                !\n", p->address_2.kind);
    }

    switch (p->op) {
            case AddK:
            if(flag_immediate_left&&
                flag_immediate_right){
                    format_one(
                        instructions_list,
                         G_LDI,
                        register_result,
                        immediate_left+
                        immediate_right);
                    flag_immediate_left =
                         0;
                    flag_immediate_right
                        = 0;
            }else if (flag_immediate_left
                ) {
                    format_two(
                        instructions_list,
                         G_ADDI,
                        register_operator_right
                        , register_result,
                         immediate_left, "
                        none");
                    flag_immediate_left =
                         0;
            }else if (
                flag_immediate_right) {
                    format_two(
```

```
                instructions_list,
                 G_ADDI,
                register_operator_left
                , register_result,
                 immediate_right,
                "none");
            flag_immediate_right
                = 0;
        }else{
            format_three(
                instructions_list,
                 G_ADD,
                register_operator_left
                ,
                register_operator_right
                , register_result)
                ;
        }
        break;
        case SubK:
        if(flag_immediate_left&&
            flag_immediate_right){
            format_one(
                instructions_list,
                 G_LDI,
                register_result,
                immediate_left-
                immediate_right);
            flag_immediate_left =
                 0;
            flag_immediate_right
                = 0;
        }else if (flag_immediate_left
            ) {
            format_two(
                instructions_list,
                 G_SUBI,
                register_operator_right
                , register_result,
                 immediate_left, "
                none");
            flag_immediate_left =
                 0;
        }else if (
            flag_immediate_right) {
            format_two(
                instructions_list,
                 G_SUBI,
                register_operator_left
                , register_result,
                 immediate_right,
                "none");
            flag_immediate_right
                = 0;
        }else{
            format_three(
                instructions_list,
                 G_SUB,
                register_operator_left
                ,
```

```
                        register_operator_right
                        , register_result)
                        ;
                }
        }

        if (flag_temp_left) {
                release_temporary(
                   register_temporary_left);
                flag_temp_left = 0;
        }
        if (flag_temp_right) {
                release_temporary(
                   register_temporary_right);
                flag_temp_right = 0;
        }
        map_temporary(instructions_list, p->
           address_3.value, register_result);


        break;
        case TimK:
        case OvrK:
        case EqlK:
        case NeqK:
        case GtrK:
        case GeqK:
        case LsrK:
        case LeqK:
        //left operand
        if (p->address_1.kind==Temp) {
                register_temporary_left =
                   search_temporary(p->
                   address_1.value);
                format_two(instructions_list,
                    G_ADDI,
                   register_temporary_left,
                   register_operator_left, 0,
                    "none");
                flag_temp_left = 1;

        }else if(p->address_1.kind==String){
                int memory_position_left = 0;
                memory_position_left =
                   search_variable(
                   variables_list, p->
                   address_1.name, 0,
                   current_scope);


                format_one(instructions_list,
                    G_LD,
                   register_operator_left,
                   memory_position_left);
        }else if(p->address_1.kind==IntConst)
           {
                if(p->address_1.value<65000){
                        flag_immediate_left =
                            1;
                        immediate_left = p->
```

```
                address_1.value;
        }
        format_one(instructions_list,
            G_LDI,
            register_operator_left, p
            ->address_1.value);
}else{
        printf("ERROR: intermediate
            variable kind unknown!\n")
            ;
}

//right operand
if (p->address_2.kind==Temp) {
        register_temporary_right =
            search_temporary(p->
            address_2.value);
        format_two(instructions_list,
            G_ADDI,
            register_temporary_right,
            register_operator_right,
            0, "none");
        flag_temp_right = 1;

}else if(p->address_2.kind==String){
        int memory_position_right =
            0;
        memory_position_right =
            search_variable(
            variables_list, p->
            address_2.name, 0,
            current_scope);


        format_one(instructions_list,
            G_LD,
            register_operator_right,
            memory_position_right);
}else if(p->address_2.kind==IntConst)
    {
        if(p->address_2.value<65000){
                flag_immediate_right
                    = 1;
                immediate_right = p->
                    address_2.value;
        }
        format_one(instructions_list,
            G_LDI,
            register_operator_right, p
            ->address_2.value);
}else{
        printf("ERROR: intermediate
            variable kind unknown!\n")
            ;
}

switch (p->op) {
        case TimK:
        if(flag_immediate_left&&
            flag_immediate_right){
```

```
                format_one(
                    instructions_list,
                     G_LDI,
                    register_result,
                    immediate_left*
                    immediate_right);
                flag_immediate_left =
                     0;
                flag_immediate_right
                    = 0;
            }else{
                format_three(
                    instructions_list,
                     G_MUL,
                    register_operator_left
                    ,
                    register_operator_right
                    , register_result)
                    ;
            }
            break;
            case OvrK:
            if(flag_immediate_left&&
                flag_immediate_right){
                format_one(
                    instructions_list,
                     G_LDI,
                    register_result,
                    immediate_left/
                    immediate_right);
                flag_immediate_left =
                     0;
                flag_immediate_right
                    = 0;
            }else{
                format_three(
                    instructions_list,
                     G_DIV,
                    register_operator_left
                    ,
                    register_operator_right
                    , register_result)
                    ;
            }
            break;
            case EqlK:
            if(flag_immediate_left&&
                flag_immediate_right){
                format_one(
                    instructions_list,
                     G_LDI,
                    register_result,
                    immediate_left==
                    immediate_right);
                flag_immediate_left =
                     0;
                flag_immediate_right
                    = 0;
            }else{
                format_three(
```

```
                    instructions_list,
                     G_SLT,
                    register_operator_left
                    ,
                    register_operator_right
                    , register_result)
                    ;
                format_three(
                    instructions_list,
                     G_SLT,
                    register_operator_right
                    ,
                    register_operator_left
                    ,
                    register_operator_left
                    );
                format_three(
                    instructions_list,
                     G_OR,
                    register_result,
                    register_operator_left
                    , register_result)
                    ;
                format_two(
                    instructions_list,
                     G_NOT,
                    register_result,
                    register_result,
                    0, "none");
        }
        break;
        case NeqK:
        if(flag_immediate_left&&
            flag_immediate_right){
                format_one(
                    instructions_list,
                     G_LDI,
                    register_result,
                    immediate_left!=
                    immediate_right);
                flag_immediate_left =
                     0;
                flag_immediate_right
                    = 0;
        }else{
                format_three(
                    instructions_list,
                     G_SLT,
                    register_operator_left
                    ,
                    register_operator_right
                    , register_result)
                    ;
                format_three(
                    instructions_list,
                     G_SLT,
                    register_operator_right
                    ,
                    register_operator_left
                    ,
```

```
                register_operator_left
                );
            format_three(
                instructions_list,
                 G_OR,
                register_result,
                register_operator_left
                , register_result)
                ;
    }
    break;
    case GtrK:
    if(flag_immediate_left&&
        flag_immediate_right){
            format_one(
                instructions_list,
                 G_LDI,
                register_result,
                immediate_left>
                immediate_right);
            flag_immediate_left =
                 0;
            flag_immediate_right
                = 0;
    }else{
            format_three(
                instructions_list,
                 G_SLT,
                register_operator_right
                ,
                register_operator_left
                , register_result)
                ;
    }
    break;
    case GeqK:
    if(flag_immediate_left&&
        flag_immediate_right){
            format_one(
                instructions_list,
                 G_LDI,
                register_result,
                immediate_left>=
                immediate_right);
            flag_immediate_left =
                 0;
            flag_immediate_right
                = 0;
    }else{
            format_three(
                instructions_list,
                 G_SLT,
                register_operator_left
                ,
                register_operator_right
                , register_result)
                ;
            format_two(
                instructions_list,
                 G_NOT,
```

```
                register_result,
                register_result,
                0, "none");
        }
        break;
        case LsrK:
        if(flag_immediate_left&&
            flag_immediate_right){
                format_one(
                    instructions_list,
                     G_LDI,
                    register_result,
                    immediate_left<
                    immediate_right);
                flag_immediate_left =
                     0;
                flag_immediate_right
                    = 0;
        }else{
                format_three(
                    instructions_list,
                     G_SLT,
                    register_operator_left
                    ,
                    register_operator_right
                    , register_result)
                    ;
        }
        break;
        case LeqK:
        if(flag_immediate_left&&
            flag_immediate_right){
                format_one(
                    instructions_list,
                     G_LDI,
                    register_result,
                    immediate_left<=
                    immediate_right);
                flag_immediate_left =
                     0;
                flag_immediate_right
                    = 0;
        }else{
                format_three(
                    instructions_list,
                     G_SLT,
                    register_operator_right
                    ,
                    register_operator_left
                    ,
                    register_operator_left
                    );
                format_two(
                    instructions_list,
                     G_NOT,
                    register_operator_left
                    , register_result,
                     0, "none");
        }
        break;
```

```
}

if (flag_temp_left) {
        release_temporary(
           register_temporary_left);
        flag_temp_left = 0;
}
if (flag_temp_right) {
        release_temporary(
           register_temporary_right);
        flag_temp_right = 0;
}
map_temporary(instructions_list, p->
   address_3.value, register_result);
break;
case AsvK:

register_temporary = search_temporary
   (p->address_1.value);
memory_position = search_variable(
   variables_list, p->address_3.name,
    0, current_scope);
format_one(instructions_list, G_ST,
   register_temporary,
   memory_position);

release_temporary(register_temporary)
   ;

break;
case AsaK:
register_temporary = search_temporary
   (p->address_1.value);
switch (p->address_2.kind) {
        case IntConst:
        memory_offset = p->address_2.
           value;
        memory_position =
           search_variable(
           variables_list, p->
           address_3.name,
           memory_offset,
           current_scope);
        format_one(instructions_list,
            G_ST, register_temporary,
            memory_position);
        break;
        case String:

        memory_position =
           search_variable(
           variables_list, p->
           address_2.name, 0,
           current_scope);
        memory_offset =
           search_variable(
           variables_list, p->
           address_3.name, 0,
           current_scope);
        format_one(instructions_list,
```

```
                G_LD,
                register_operator_left,
                memory_position);
            format_two(instructions_list,
                G_ADDI,
                register_operator_left,
                register_operator_right,
                memory_offset, "none");
            format_two(instructions_list,
                G_STR,
                register_operator_right,
                register_temporary, 0, "
                none");

            break;
            case Temp:

            memory_offset =
                search_variable(
                variables_list, p->
                address_3.name, 0,
                current_scope);
            register_temporary_left =
                search_temporary(p->
                address_2.value);
            format_two(instructions_list,
                G_ADDI,
                register_temporary_left,
                register_operator_right,
                memory_offset, "none");
            format_two(instructions_list,
                G_STR,
                register_operator_right,
                register_temporary, 0, "
                none");
            release_temporary(
                register_temporary_left);

            break;
            default:
            break;
}

release_temporary(register_temporary)
    ;

break;
case PrmK:
            par = parameters_list->start;
            type_parameter *new_parameter
                = malloc(sizeof(
                type_parameter));
            new_parameter->kind = p->
                address_3.kind;
            strcpy(new_parameter->scope,
                current_scope);
            switch (p->address_3.kind) {
                    case IntConst:
                    case Temp:
                            new_parameter
```

```
                                        ->value =
                                        p->
                                        address_3.
                                        value;
                                    break;
                            case String:
                                    strcpy(
                                        new_parameter
                                        ->name, p
                                        ->
                                        address_3.
                                        name);
                                    break;
                            default:
                                    printf("
                                        Unknown
                                        parameter
                                        kind!\n");
                                    break;
                    }
                    if(par==NULL){
                            parameters_list->
                                start =
                                new_parameter;
                            parameters_list->
                                start->next = NULL
                                ;
                    }
                    else{
                            while (par->next!=
                                NULL) {
                                    par = par->
                                        next;
                            }
                            par->next =
                                new_parameter;
                            par->next->next =
                                NULL;
                    }
                    break;
            case InnK:
                    format_one(instructions_list,
                         G_IN, register_result, 0)
                        ;
                    map_temporary(
                        instructions_list, p->
                        address_2.value,
                        register_result);
                    break;
            case OutK:
            //find parameter
            par = parameters_list->start;
            // while (par!=NULL) {
                    //load parameter to register
                        result
                    switch (par->kind){
                            case String:
                                    //search for
                                        parameter
                                        in memory
```

```
                memory_position
                    =
                    search_variable
                    (
                    variables_list
                    , par->
                    name, 0,
                    par->scope
                    );
                //load from
                    memory to
                    register
                    result
                format_one(
                    instructions_list
                    , G_LD,
                    register_result
                    ,
                    memory_position
                    );
                break;
            case IntConst:
                //load
                    parameter
                    as
                    immediate
                    to
                    register
                    result
                format_one(
                    instructions_list
                    , G_LDI,
                    register_result
                    , par->
                    value);
                break;
            case Temp:
                //search for
                    register
                    that
                    stores
                    parameter
                register_temporary
                    =
                    search_temporary
                    (par->
                    value);
                //load
                    parameter
                    from
                    temporary
                    to
                    register
                    result
                format_two(
                    instructions_list
                    , G_ADDI,
                    register_temporary
                    ,
                    register_result
```

```
                                        , 0, "none
                                        ");
                                    break;
                            }

                    // par = par->next;
// }

                    parameters_list->start = NULL
                        ;

                    format_one(instructions_list,
                         G_POUT, register_result,
                        0);
                    format_one(instructions_list,
                         G_OUT, register_result,
                        0);
                    format_one(instructions_list,
                         G_OUT, register_result,
                        0);


  break;
case CalK:

                        //
                            copy_list_parameters
                            (parameters_list,
                            parameters_list_aux
                            );

                        // print_parameters(
                            parameters_list_aux
                            );


                        //takes parameters
                            and stores them in
                             memory for the
                            function to use
                        consume_parameters(
                            table,
                            instructions_list,
                             parameters_list,
                            variables_list, p
                            ->address_3.name);


                        //cheks if it's the
                            first call and
                            initializes calls
                            stack
                        if(!flag_first_call){
                                format_one(
                                    instructions_list
                                    , G_LDI,
                                    register_top
                                    ,
                                    memory_index
                                    );
```

```
            flag_first_call
                = 1;
    }
    //increases stack top
    format_two(
        instructions_list,
         G_ADDI,
        register_top,
        register_top, 1, "
        none");
    //loads line number
        to register
    format_one(
        instructions_list,
         G_LDI,
        register_result,
        line_counter+3);
    //adds line to memory
    format_two(
        instructions_list,
         G_STR,
        register_top,
        register_result,
        0, "none");
    //jumps to function
    format_zero(
        instructions_list,
         G_JMP, 0, String,
         p->address_3.name
        , label_kind);
    //decreases top
    format_two(
        instructions_list,
         G_SUBI,
        register_top,
        register_top, 1, "
        none");

    //counter is intended
         to work with
        various calls for
        a same function
    counter = 0;
    insert_label(
        calls_list, String
        , p->address_3.
        name, counter,
        line_counter);

    //picks altered
        arrays from
        function and
        stores their value
         back to
        parameters'
        positions
    restore_arrays(table,
         instructions_list
        , parameters_list,
         variables_list, p
```

```
                    ->address_3.name);
                // insert_call(
                    parameters_list_aux
                    ,
                    returner_calls_list
                    , p->address_3.
                    name, counter,
                    line_counter);
                //
                    parameters_list_aux
                    ->start = NULL;

                //map temporary from
                    return
                if (p->address_2.kind
                    ==Temp) {
                        map_temporary
                            (
                            instructions_list
                            , p->
                            address_2.
                            value,
                            register_return
                            );
                        // printf("%s
                            mapped\n
                            ", p->
                            address_3.
                            name);
                }


        break;

case EofK:
        // format_zero(
            instructions_list, G_JMP,
            0, String, current_scope,
            call_kind);
        //loads to register result
            the value in top position
        format_two(instructions_list,
             G_LDR, register_top,
            register_result, 0, "none"
            );
        //jump to position in
            register result
        format_one(instructions_list,
             G_JMPR, register_result,
            0);
        break;
case NopK:
        format_zero(instructions_list
            , G_NOP, 0, IntConst, "
            none", label_kind);
        break;
case RetK:

if(p->address_3.kind!=Empty){
        if (p->address_3.kind==Temp)
```

```
                {
                    register_temporary =
                        search_temporary(p
                        ->address_3.value)
                        ;
                    format_two(
                        instructions_list,
                         G_ADDI,
                        register_temporary
                        , register_result,
                         0, "none");
                }else if(p->address_3.kind==
                    String){
                    int memory_position =
                         0;
                    memory_position =
                        search_variable(
                        variables_list, p
                        ->address_3.name,
                        0, current_scope);
                    format_one(
                        instructions_list,
                         G_LD,
                        register_result,
                        memory_position);
                }else if(p->address_3.kind==
                    IntConst){
                    if(p->address_3.value
                        <65000){
                            immediate_left
                                = p->
                                address_3.
                                value;
                    }
                    else{
                            format_one(
                                instructions_list
                                , G_LDI,
                                register_result
                                , p->
                                address_3.
                                value);
                    }
                }else{
                    printf("ERROR:
                        intermediate
                        variable kind
                        unknown: %d!\n", p
                        ->address_1.kind);
                }
                format_two(instructions_list,
                     G_ADDI, register_result,
                    register_return, 0,
                    current_scope);
            }

            // format_zero(
                instructions_list, G_JMP,
                0, String, current_scope,
                call_kind);
```

```
            //loads to register result
                the value in top position
            format_two(instructions_list,
                 G_LDR, register_top,
                register_result, 0, "none"
                );
            //jump to position in
                register result
            format_one(instructions_list,
                 G_JMPR, register_result,
                0);

    break;
    case IffK:
    register_temporary_left =
        search_temporary(p->address_1.
        value);
    // format_one(instructions_list,
        G_PBC, register_operator_left, 0);
    format_one(instructions_list, G_PBC,
        register_temporary_left, 0);

    format_zero(instructions_list, G_BOZ,
         p->address_3.value, LabAddr, "
        none", label_kind);

    release_temporary(
        register_temporary_left);
    break;
    case GtoK:
    if (p->address_3.kind==LabAddr) {
            format_zero(instructions_list
                , G_JMP, p->address_3.
                value, LabAddr, "none",
                label_kind);
    }else if(p->address_3.kind==String){
            format_zero(instructions_list
                , G_JMP, p->address_3.
                value, String, p->
                address_3.name, label_kind
                );
    }else{
            printf("EXCEPTION\n");
    }
    break;
    case HltK:
             format_zero(
                instructions_list, G_HLT,
                 0, IntConst, "none",
                label_kind);
    break;
    case LblK:

            insert_label(labels_list, p->
                address_3.kind, p->
                address_3.name, p->
                address_3.value,
                line_counter);

                    break;
```

```
case CstK:
if(p->address_1.value
   <65000){
        immediate_left
            = p->
           address_1.
           value;
        flag_immediate_left
            = 1;
}else{
        format_one(
           instructions_list
           , G_LDI,
           register_operator_le
           , p->
           address_1.
           value);
}

if (
   flag_immediate_left
   ) {
        format_one(
           instructions_list
           , G_LDI,
           register_result
           ,
           immediate_left
           );
        flag_immediate_left
            = 0;
}
map_temporary(
   instructions_list,
    p->address_3.
   value,
   register_result);

        break;
case VstK:
        memory_position
            =
           search_variable
           (
           variables_list
           , p->
           address_1.
           name, 0,
           current_scope
           );
        format_one(
           instructions_list
           , G_LD,
           register_operator_le
           ,
           memory_position
           );
        map_temporary
           (
           instructions_list
```

```
                , p->
                address_3.
                value,
                register_operator_le
                );
            break;
        case AstK:
            switch (p->
                address_2.
                kind) {
                    case
                        IntConst
                        :
                            memory_

                                =

                                sea
                                (
                                var
                                ,

                                p
                                ->
                                addr
                                .
                                name
                                ,

                                p
                                ->
                                addr
                                .
                                valu
                                ,

                                curr
                                )
                                ;

                            format_
                                (
                                inst
                                ,

                                G_LL
                                ,

                                regi
                                ,

                                memo
                                )
                                ;

                            break
                                ;

                    case
                        String
```

```
 :
memory_positi

   =

   search_va
   (
   variables_
   ,

   p
   ->
   address_1
   .
   name
   ,

   0,

   current_s
   )
   ;

      memory_

         =

         sea
         (
         var
         ,

         p
         ->
         addr
         .
         name
         ,

         0,

         curr
         )
         ;

      format_
         (
         inst
         ,

         G_LL
         ,

         regi
         ,

         memo
         )
         ;
```

```
                format_
                    (
                    inst
                    ,
                    G_AD
                    ,
                    regi
                    ,
                    regi
                    ,
                    memo
                    ,
                    "
                    none
                    "
                    )
                    ;
                format_
                    (
                    inst
                    ,
                    G_LD
                    ,
                    regi
                    ,
                    regi
                    ,
                    0,
                    "
                    none
                    "
                    )
                    ;
                break
                    ;
            case
                Temp
                :
                memory_
                    =
                    sear
                    (
                    vari
                    ,
```

```
    p
    ->
    addr
    .
    name
    ,

    0,

    curr
    )
    ;

registe

    =

    sear
    (
    p
    ->
    addr
    .
    valu
    )
    ;

format_
    (
    inst
    ,

    G_AD
    ,

    regi
    ,

    regi
    ,

    0,

    "
    none
    "
    )
    ;

format_
    (
    inst
    ,

    G_AD
    ,

    regi
    ,
```

```
                reg
                ,
                mem
                ,
                "
                none
                "
                )
                ;
            format_
                (
                ins
                ,
                G_LL
                ,
                reg
                ,
                reg
                ,
                0,
                "
                none
                "
                )
                ;
            release
                (
                reg
                )
                ;
            break
                ;
        default
            :
            printf
                (
                "
                ASTR
                Type
                Prob
                :
                %
                d
                \
                n
                "
```

```
                                                                ,
                                                                p
                                                                ->
                                                                addr
                                                                .
                                                                kind
                                                                )
                                                                ;

                                                            break
                                                                ;

                                                }

                                                map_temporary
                                                    (
                                                    instructions_list
                                                    , p->
                                                    address_3.
                                                    value,
                                                    register_result
                                                    );
                                                break;
                                        default:
                                        break;
                                }
                                // printf("%d\n", p->op);
                                // printf("SCOPE: %s\n", p->
                                    scope);
                                p = p->next;
                        }
                }


                void generate_code_launcher(list_quadruple *
                    quad_list, TipoLista *table){
                        int i;
                        //stores all variables positions
                        list_variables *variables_list;
                        list_instructions *instructions_list;
                        list_parameters *parameters_list;
                  list_labels *labels_list;
                        list_labels *calls_list;

                        variables_list = (list_variables*)
                            malloc(sizeof(list_variables));
                        instructions_list = (
                            list_instructions*) malloc(sizeof(
                            list_instructions));
                        parameters_list = (list_parameters*)
                            malloc(sizeof(list_parameters));
                        labels_list = (list_labels*) malloc(
                            sizeof(list_labels));
                        calls_list = (list_labels*) malloc(
                            sizeof(list_labels));

                        variables_list->start = NULL;
                        instructions_list->start = NULL;
                        parameters_list->start = NULL;
```

```
labels_list->start = NULL;
calls_list->start = NULL;

array_size_placer(table);


//global and main variables' memory
    allocation
TipoID *table_item;
for(i = 0;i<211;i++){
        if(&table[i]!=NULL){
                table_item = table[i
                    ].start;
                while (table_item!=
                    NULL) {
                        if (!strcmp(
                            table_item
                            ->tipoID,
                            "func")) {
                                declaration_var
                                    (
                                    variables_li
                                    ,
                                    table
                                    ,
                                    table_item
                                    ->
                                    nomeID
                                    );
                                memory_index
                                    ++;

                        }
                        table_item =
                            table_item
                            ->prox;
                }
        }
}

declaration_variables(variables_list,
     table, "global");
// declaration_variables(
    variables_list, table, "main");


printf("\n");
generate_code(instructions_list,
    quad_list, table, variables_list,
    parameters_list, labels_list,
    calls_list);
treat_jumps_n_branches(
    instructions_list, labels_list,
    calls_list);


print_variables(variables_list);
// print_instructions(
    instructions_list);
// print_labels(labels_list);
```

```
                                print_target_code(instructions_list);

                                // print_parameters(parameters_list);
                        }
```

Algoritmo 8: object.h - Estruturas e Cabeçalhos para Geração de Código Objeto

```
#ifndef _object_H_
#define _object_H_

#include "intermediate.h"
#include <stdbool.h>

#define STACK_SIZE 500

// globals
int line_return[STACK_SIZE];
int temporary_return[STACK_SIZE];
int top;


typedef enum{
    /*0 Adiaao  000000 @*/ G_ADD, // R[DR] <- R[SA] + R[SB]
  /*1 Adiaao Imediato  000001 @*/ G_ADDI, // R[DR] <- R[SA] + IM
  /*2 Subtraaao  000010 @*/ G_SUB, // R[DR] <- R[SA] + \overline{R[SB
     ]} + 1
  /*3 Subtraaao Imediato  000011 @*/ G_SUBI, // R[DR] <- R[SA] + \
     overline{IM} + 1
  /*4 Multiplicaaao  000100 @*/ G_MUL, // R[DR] <- R[SA] * R[SB]
  /*5 Divisao  000101 @*/ G_DIV, // R[DR] <- R[SA] / R[SB]
  /*6 Incrementa  000110 @*/ G_INC, // R[DR] <- R[SA] + 1
  /*7 Decrementa  000111 @*/ G_DEC, // R[DR] <- R[SA] - 1
  /*8 And  001000 @*/ G_AND, // R[DR] <- R[SA] \wedge R[SB]
  /*9 Or  001001 @*/ G_OR, // R[DR] <- R[SA] \vee R[SB]
  /*10 Resto  001010 @*/ G_MOD, // R[DR] <- R[SA] \% R[SB]
  /*11 Xor  001100 @*/ G_XOR, // R[DR] <- R[SA] \bigoplus R[SB]
  /*12 Not  001101 @*/ G_NOT, // R[DR] <- \overline{R[SA]}
  /*13 Desloca Esquerda  010000 @*/ G_SHL, // R[DR] <- sl (shamt)R[SA
     ]
  /*14 Desloca Direita  010001 @*/ G_SHR, // R[DR] <- sr (shamt)R[SA]
  /*15 Pra-branch  011111 @*/ G_PBC, // se R[SA] = 0, FZ = 1;  se R[
     SA] < 0, FN = 1;
  /*16 Branch em Zero  010011 @*/ G_BOZ, // se FZ = 1, entao PC <- PC
      + 1 + IM e FZ = 0, entao PC <- PC + 1
  /*17 Branch em Negativo  010100 @*/ G_BON, // se FN = 1, entao PC
     <- PC + IM e FN = 0, entao PC <- PC + 1
  /*18 Jump  010101 @*/ G_JMP, // PC <- IM
  /*19 Set on Less Than  010111 @*/ G_SLT, // se R[SA] < R[SB], entao
      R[DR] <- 1
  /*20 Load  011000 @*/ G_LD, // R[DR] <- M[IM]
  /*21 Store  011001 @*/ G_ST, // M[IM] <- R[SA]
  /*22 Load Imediato  011010 @*/ G_LDI, // R[DR] <- IM
  /*23 Nop  011011 @*/ G_NOP, // Sem Operaaao
  /*24 Entrada  011101 @*/ G_IN, // R[DR] <- alavancas
  /*25 HLT  011100 @*/ G_HLT, // Parar Operaaao
  /*26 Pra Saada  011110 @*/ G_POUT, // Displays <- R[SA]
  /*27 Saada  100000 @*/ G_OUT, // Displays <- R[SA]
```

```
  /*28 Load Registrador 1000001*/ G_LDR, // R[DR] <- M[R[SA]]
  /*29 Store Registrador 100010*/ G_STR, // M[R[SA]] -> R[DR]
  /*30 Jump Registrador  100011 &*/ G_JMPR // PC <- R[SA]

}galetype;

typedef enum{
  label_kind,
  call_kind
}kind_jump;

typedef enum{
  array_kind,
  variable_kind
}kind_variable;

FILE *file_target_code;


typedef struct type_variable{
    int index;
    int index_array;
    char id[50];
    char scope[50];
    kind_variable kind;
    struct type_variable *next;
}type_variable;

typedef struct type_instruction{
  int line;
  int register_a;
  int register_b;
  int register_c;
  int immediate;
  int target_label;
  kind_jump jump;
  char label_name[50];
  galetype type;
  struct type_instruction *next;
}type_instruction;

typedef struct type_parameter{
  AddrKind kind;
  int value;
  char name[50];
  char scope[50];
  struct type_parameter *next;
}type_parameter;

typedef struct type_label{
  AddrKind type;
  char name[50];
  int index;
  int line;
  struct type_label *next;
}type_label;



typedef struct{
```

```
  int index;
  int line;
  char name[50];
  struct type_call *next;
  type_parameter *parameters;
}type_call;

typedef struct{
  type_call *start;
}list_calls;

typedef struct{
  type_parameter *start;
}list_parameters;

typedef struct{
    type_instruction *start;
}list_instructions;

typedef struct{
    type_variable *start;
}list_variables;

typedef struct{
    type_label *start;
}list_labels;



//reserves spaces in memory for variables of a given function
void declaration_variables(list_variables *variables_list, TipoLista
   *table, char scope[]);

//numbers relate to amount of registers in the operation
void insert_variable(list_variables *variables_list, int index, int
   index_array, kind_variable kind, char id[], char scope[]);

void insert_label(list_labels *labels_list, AddrKind type, char name
   [], int index, int line);

void insert_call(list_parameters *parameters_list, list_calls *
   returner_calls_list, char name[], int index, int line);



//searches the position in memory of a variable given its name, scope
    and array index
int search_variable(list_variables *variables_list, char name[], int
   array_position, char scope[]);

void format_zero(list_instructions *instructions_list, galetype type,
    int immediate, AddrKind kind, char label_string[], kind_jump jump
   );

void format_one(list_instructions *instructions_list, galetype type,
   int register_a, int immediate);

void format_two(list_instructions *instructions_list, galetype type,
   int register_source, int register_target, int immediate, char
```

```
    label_string[]);

void format_three(list_instructions *instructions_list, galetype type
    , int register_source_a, int register_source_b, int
    register_target);

void generate_code(list_instructions *instructions_list,
    list_quadruple *quad_list, TipoLista *table, list_variables *
    variables_list, list_parameters *parameters_list, list_labels *
    labels_list, list_labels *calls_list);

void generate_code_launcher(list_quadruple *quad_list, TipoLista *
    table);

void consume_parameters(TipoLista *table, list_instructions *
    instructions_list, list_parameters *parameters_list,list_variables
     *variables_list, char function[]);

void restore_arrays(TipoLista *table, list_instructions *
    instructions_list, list_parameters *parameters_list,
    list_variables *variables_list, char function[]);

void treat_jumps_n_branches(list_instructions *instructions_list,
    list_labels *labels_list, list_labels *calls_list);

void print_parameters(list_parameters *parameters_list);









#endif
```

Algoritmo 9: target.c - geração de código alvo

```
#include "object.h"

/* Function to convert a decinal number to binary number */
char *decimal_to_binary(int decimal, int lenght){
  int c, d, count;
  char *pointer;

  count = 0;
  pointer = (char*) malloc(32+1);

  if (pointer==NULL) {
    exit(EXIT_FAILURE);
  }

  for ( c = lenght-1; c >= 0; c--) {
    d = decimal >> c;

    if(d & 1){
      *(pointer + count) = 1 + '0';
    }else{
      *(pointer + count) = 0 + '0';
```

```
    }
    count++;
  }
  *(pointer + count) = '\0';

  return pointer;
}

int count_instructions(list_instructions *instructions_list){
  type_instruction *instruction = instructions_list->start;
  int count;
  count = 0;

  while (instruction!=NULL) {
    count++;
    instruction = instruction->next;
  }

  return count;
}

void print_target_code(list_instructions *instructions_list){
  file_target_code = fopen("simpleInstructionsRam.v", "w");

  type_instruction *instruction = instructions_list->start;

  fprintf(file_target_code, "module simpleInstructionsRam(clock,
     address, iRAMOutput);\n");
  fprintf(file_target_code, "\t input [9:0] address;\n");
  fprintf(file_target_code, "\t input clock;\n");
  fprintf(file_target_code, "\t output [31:0] iRAMOutput;\n");
  fprintf(file_target_code, "\t integer firstClock = 0;\n");
  fprintf(file_target_code, "\t reg [31:0] instructionsRAM[%d:0];\n",
      count_instructions(instructions_list));
  fprintf(file_target_code, "\n");
  fprintf(file_target_code, "\t always at ( posedge clock ) begin\n")
     ;
  fprintf(file_target_code, "\t \t if (firstClock==0) begin\n \n");

  while (instruction!=NULL) {
    fprintf(file_target_code, "\t \t instructionsRAM[%d] = 32'b",
       instruction->line);
    switch (instruction->type) {
      case G_ADD:
        fprintf(file_target_code, "000000%s%s%s00000000000;//ADD r[%d
           ],r[%d] to r[%d]\n",
        decimal_to_binary(instruction->register_c, 5),
        decimal_to_binary(instruction->register_a, 5),
        decimal_to_binary(instruction->register_b, 5),
        instruction->register_a, instruction->register_b, instruction
           ->register_c);
      break;
      case G_ADDI:
        fprintf(file_target_code, "000001%s%s%s;//ADDi r[%d], #%d to
           r[%d]\n",
        decimal_to_binary(instruction->register_c, 5),
        decimal_to_binary(instruction->register_a, 5),
        decimal_to_binary(instruction->immediate, 16),
        instruction->register_a, instruction->immediate, instruction
           ->register_c);
```

```
break;
case G_SUB:
  fprintf(file_target_code, "000010%s%s%s00000000000;//SUB r[%d
     ],r[%d] to r[%d]\n",
  decimal_to_binary(instruction->register_c, 5),
  decimal_to_binary(instruction->register_a, 5),
  decimal_to_binary(instruction->register_b, 5),
  instruction->register_a, instruction->register_b, instruction
     ->register_c);
break;
case G_SUBI:
  fprintf(file_target_code, "000011%s%s%s;//SUBi r[%d], #%d to
     r[%d]\n",
  decimal_to_binary(instruction->register_c, 5),
  decimal_to_binary(instruction->register_a, 5),
  decimal_to_binary(instruction->immediate, 16),
  instruction->register_a, instruction->immediate, instruction
     ->register_c);
break;
case G_MUL:
  fprintf(file_target_code, "000100%s%s%s00000000000;//TIMES r
     [%d],r[%d] to r[%d]\n",
  decimal_to_binary(instruction->register_c, 5),
  decimal_to_binary(instruction->register_a, 5),
  decimal_to_binary(instruction->register_b, 5),
  instruction->register_a, instruction->register_b, instruction
     ->register_c);
break;
case G_DIV:
  fprintf(file_target_code, "000101%s%s%s00000000000;//DIVIDE r
     [%d],r[%d] to r[%d]\n",
  decimal_to_binary(instruction->register_c, 5),
  decimal_to_binary(instruction->register_a, 5),
  decimal_to_binary(instruction->register_b, 5),
  instruction->register_a, instruction->register_b, instruction
     ->register_c);
break;
case G_INC:
  fprintf(file_target_code, "forgot %d\n", instruction->type);
break;
case G_DEC:
  fprintf(file_target_code, "forgot %d\n", instruction->type);
break;
case G_AND:
  fprintf(file_target_code, "001000%s%s%s00000000000;//AND r[%d
     ],r[%d] to r[%d]\n",
  decimal_to_binary(instruction->register_c, 5),
  decimal_to_binary(instruction->register_a, 5),
  decimal_to_binary(instruction->register_b, 5),
  instruction->register_a, instruction->register_b, instruction
     ->register_c);
break;
case G_OR:
  fprintf(file_target_code, "001001%s%s%s00000000000;//OR r[%d
     ],r[%d] to r[%d]\n",
  decimal_to_binary(instruction->register_c, 5),
  decimal_to_binary(instruction->register_a, 5),
  decimal_to_binary(instruction->register_b, 5),
  instruction->register_a, instruction->register_b, instruction
     ->register_c);
```

```
break;
case G_MOD:
  fprintf(file_target_code, "forgot %d\n", instruction->type);
break;
case G_XOR:
  fprintf(file_target_code, "forgot %d\n", instruction->type);
break;
case G_NOT:
  fprintf(file_target_code, "001101%s%s0000000000000000;//NOT r
     [%d] to r[%d]\n",
  decimal_to_binary(instruction->register_c, 5),
  decimal_to_binary(instruction->register_a, 5),
  instruction->register_a, instruction->register_c);
break;
case G_SHL:
  fprintf(file_target_code, "forgot %d\n", instruction->type);
break;
case G_SHR:
  fprintf(file_target_code, "forgot %d\n", instruction->type);
break;
case G_PBC:
  fprintf(file_target_code, "01111100000%s0000000000000000;//
     Pre Branch r[%d]\n",
  decimal_to_binary(instruction->register_a, 5),
  instruction->register_a);
break;
case G_BOZ:
  fprintf(file_target_code, "010011%s;//Branch on Zero #%d\n",
  decimal_to_binary(instruction->immediate, 26),
  instruction->immediate);
break;
case G_BON:
  fprintf(file_target_code, "forgot %d\n", instruction->type);
break;
case G_JMP:
  fprintf(file_target_code, "010101%s;//Jump to #%d\n",
  decimal_to_binary(instruction->immediate, 26),
  instruction->immediate);
break;
case G_JMPR:
  fprintf(file_target_code, "10001100000%s%s;//Jump to r[%d]\n"
     ,
  decimal_to_binary(instruction->register_a, 5),
  decimal_to_binary(instruction->immediate, 16),
  instruction->register_a);
break;
case G_NOP:
  fprintf(file_target_code, "
     01101100000000000000000000000000;//Nop\n");
break;
case G_SLT:
  fprintf(file_target_code, "010111%s%s%s00000000000;//SLT if r
     [%d] < r[%d], r[%d] = 1 else r[%d] = 0\n",
  decimal_to_binary(instruction->register_c, 5),
  decimal_to_binary(instruction->register_a, 5),
  decimal_to_binary(instruction->register_b, 5),
  instruction->register_a, instruction->register_b, instruction
     ->register_c, instruction->register_c);
break;
case G_LD:
```

```
    fprintf(file_target_code, "011000%s%s;//Load m[#%d] to r[%d]\
        n",
    decimal_to_binary(instruction->register_a, 5),
    decimal_to_binary(instruction->immediate, 21),
    instruction->immediate, instruction->register_a);
  break;
  case G_ST:
    fprintf(file_target_code, "011001%s%s;//Store r[%d] in m[#%d
        ]\n",
    decimal_to_binary(instruction->register_a, 5),
    decimal_to_binary(instruction->immediate, 21),
    instruction->register_a, instruction->immediate);
  break;
  case G_LDI:
    fprintf(file_target_code, "011010%s%s;//Loadi #%d to r[%d]\n"
        ,
    decimal_to_binary(instruction->register_a, 5),
    decimal_to_binary(instruction->immediate, 21),
    instruction->immediate, instruction->register_a);
  break;
  case G_IN:
    fprintf(file_target_code, "011101%s%s;//Input to r[%d]\n",
    decimal_to_binary(instruction->register_a, 5),
    decimal_to_binary(instruction->immediate, 21),
    instruction->register_a);
  break;
  case G_HLT:
    fprintf(file_target_code, "
        01110000000000000000000000000000;//Hlt\n");
  break;
  case G_POUT:
    fprintf(file_target_code, "011110%s%s;//Pre Output r[%d]\n",
    decimal_to_binary(instruction->register_a, 5),
    decimal_to_binary(instruction->immediate, 21),
    instruction->register_a);
  break;
  case G_OUT:
    fprintf(file_target_code, "100000%s%s;//Output r[%d]\n",
    decimal_to_binary(instruction->register_a, 5),
    decimal_to_binary(instruction->immediate, 21),
    instruction->register_a);
  break;
  case G_LDR:
    fprintf(file_target_code, "100001%s%s%s;//Loadr m[r[%d]] to r
        [%d]\n",
    decimal_to_binary(instruction->register_c, 5),
    decimal_to_binary(instruction->register_a, 5),
    decimal_to_binary(instruction->immediate, 16),
    instruction->register_a, instruction->register_c);
  break;
  case G_STR:
    fprintf(file_target_code, "100010%s%s%s;//rStore to r[%d] in
        m[r[%d]] \n",
    decimal_to_binary(instruction->register_c, 5),
    decimal_to_binary(instruction->register_a, 5),
    decimal_to_binary(instruction->immediate, 16),
    instruction->register_c, instruction->register_a);
  break;
  default:
    printf("Galetype Unknown: %d\n", instruction->type);
```

```
      break;
    }
    instruction = instruction->next;
  }

  fprintf(file_target_code, "\n\t \t firstClock <= 0;\n");
  fprintf(file_target_code, "\t \t end\n");
  fprintf(file_target_code, "\t end\n\n");
  fprintf(file_target_code, "\t assign iRAMOutput = instructionsRAM[
     address];\n");
  fprintf(file_target_code, "endmodule // simpleInstructionsRAM\n");

  fclose( file_target_code );
}
```