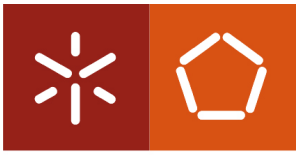

Universidade do Minho

LEI — Licenciatura de Engenharia Informática

Processamento de Linguagens

Compilador de uma LPIS

Orlando Costa - a67705, Paulo Araujo - a58925, Rui Oliveira - a67661



Braga, 7 de Junho de 2015

Resumo

Este relatório descreve o desenvolvimento de um compilador para uma linguagem de programação imperativa simples (LPIS).

A linguagem desenvolvida foi baseada na linguagem de programação C, e suporta:

- Variáveis globais
- Ciclos: for, while, do while
- Estruturas de Condição: If .. Else
- Expressões Aritméticas e lógicas
- Funções com argumentos
- Declaração de variáveis locais dentro das funções

O compilador foi desenvolvido com recurso ao analisador léxico Flex e ao analisador sintático Yacc.

Conteúdo

Capítulo 1

Introdução

O presente trabalho enquadra-se na unidade curricular de Processamento de Linguagens da Licenciatura em Engenharia Informática da Universidade do Minho. O trabalho pretende aumentar a experiência em engenharia de linguagens, assim como incentivar o desenvolvimento de processadores de linguagens e compiladores em ambiente Linux. As ferramentas que suportam a sua implementação consistem no conjunto flex-yacc, sendo estes um gerador de analisadores léxicos e um gerador de analisadores sintáticos/semânticos, respetivamente.

Inicialmente é definida uma linguagem de programação imperativa simples, a qual deve permitir manusear variáveis do tipo inteiro assim como realizar operações básicas. As variáveis devem sem declaradas no início do programa e não pode haver re-declarações. Após a validação da linguagem criada com o docente, é desenvolvido um compilador para esta linguagem, com base na GIC criada e com recurso ao Gerador Yacc/Flex. O compilador da linguagem deve gerar pseudo-código Assembly da Máquina Virtual fornecida.

1.1 Linguagem de programação imperativa simples

Previamente ao desenvolvimento do compilador existe a necessidade de definir uma linguagem sobre a qual este atua, com base numa qualquer linguagem imperativa. Neste sentido e por simplicidade e familiaridade, a linguagem de programação C é a selecionada. Esta linguagem foi simplificada por forma a adaptar-se aos requisitos propostos, sofrendo as seguinte modificações na sua estrutura:

- Apenas permite manusear variáveis do tipo inteiro (escalar ou array).
- Suporta apenas as instruções vulgares de controlo de fluxo de execução (condicional e cíclica), tais como if-else, for, while e do-while.
- As instruções que controlam inserção e output de valores (tipicamente printf e scanf) estão adaptadas para suportar apenas inteiros, e então estão renomeadas (printi e scani).
- As expressões lógicas devem estar rodeadas por parentises para facilitar a sua distinção e ordem quando em conjunto com expressões aritméticas.

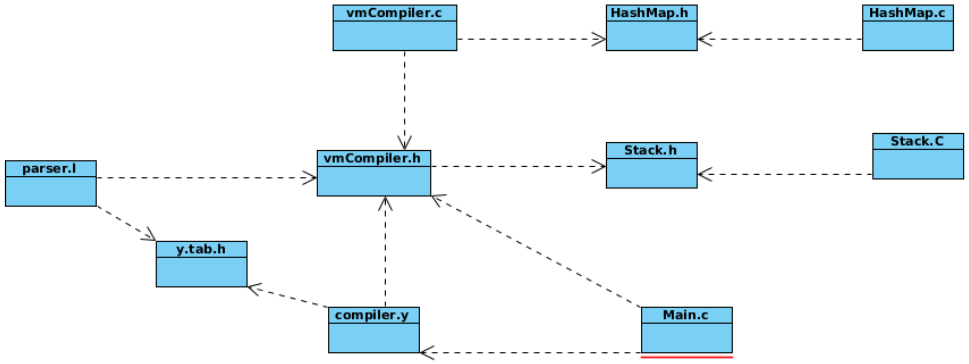

Figura 1.1: Diagrama das pedendencias dos ficheiros

1.2 Arquitetura

O sistema desenvolvido é principalmente constituído por 2 modelos: parser.l e compiler.y, que são respetivamente o analisador léxico e analisador sintático.

Na Figura 1.1, é possível observar as dependências entre os diversos ficheiros.

O analisador sintático utiliza o ficheiro vmCompiler.h, sendo que este modulo responsável pelo tratamento das variáveis e funções existentes (adicionar/consultar variáveis).

O sistema utiliza também duas estruturas de dados: uma HashMap e uma Stack. A hashmap é utilizada para guardar as variáveis e as funções, enquanto que a stack permite o controlo das labels dos ciclos durante a compilação.

Na figura 1.2 é possível observar as estruturas utilizadas em vmCompiler:

- Scope - possui uma map com a informação das variáveis, onde a chave é nome da variável e o valor é um EntryVar;
- EntryFun - guarda a informação sobre o tipo de uma função (argumentos de entrada e tipo de retorno);
- EntryVar - guarda o tipo, nome o endereço relativo de uma dada variável.

Analisadas as estruturas referidas, de notar as variáveis criadas em vmCompiler, representadas na figura 1.3, onde é possível observar duas variáveis do tipo Scope, uma para o contexto global e outra para o contexto interior a uma função.

Finalmente, existe um map de EntryFun (mFuncMap) onde a chave é o nome da função, e a variável DecFunAux consiste num apontador temporário para uma função declarada.

Figura 1.2: Diagrama das estruturas usadas em vmCompiler

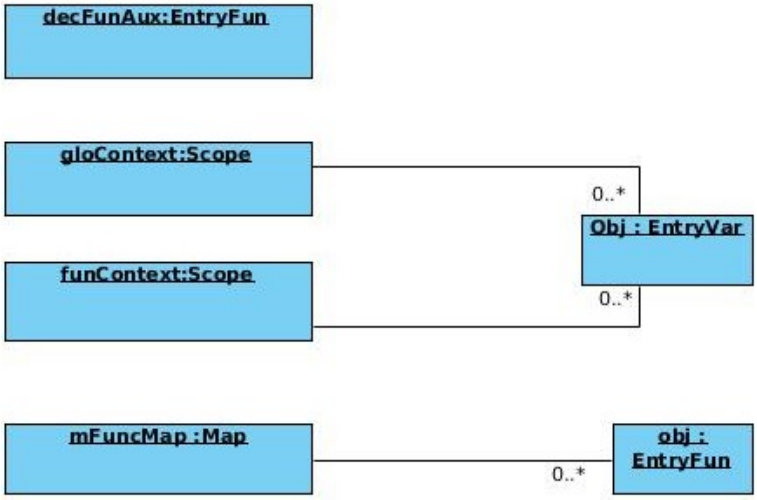

Figura 1.3: Diagrama dos objetos existente em vmCompiler

1.3 Estruturas de dados

1.3.1 Stack

De forma a evitar confusão na atribuição de *labels* relativas a *ifs* e *loops* é utilizado um contador de condições. À medida que é encontrada uma instrução que implique o uso de uma condição, este contador é incrementado e o seu valor é colocado numa stack. Deste modo, o valor que se encontra no topo da stack é relativo ao último *ciclo/if* encontrado. Sempre que é encontrado o final de uma condição, o valor no topo da stack é removido. Através do uso de um contador e de uma stack, é muito mais simples gerir as *labels* e as operações de controlo, como *JUMPs* e *JZs*. A stack utilizada implementa apenas as funções necessárias para a sua inicialização, inserção, remoção e consulta. Com as operações de push/pop são inseridos/removidos valores no topo da stack, e com a operação de get apenas é consultado o valor no topo da stack, sem que este seja removido. Esta última operação é útil para a geração de instruções 'JZ' na geração de código VM.

1.3.2 HashMap

Como foi referido anteriormente, é utilizada uma *hashmap* com o objetivo de guardar as variáveis e as funções. Quanto às variáveis, é necessário guardar e aceder a informação como o seu endereço e tipo. Sendo assim, foi criada uma estrutura de dados auxiliar para armazenar essa informação. O nome da variável é utilizado como chave e, a partir dela, conseguimos aceder à sua informação correspondente na hashmap. Relativamente às funções, é necessário guardar e aceder a informação como o seu nome, informação relativa aos seus argumentos de entrada e o tipo de dados de saída. Para esse feito, também foram necessárias estruturas de dados auxiliares como uma lista ligada capaz de armazenar informação relativa aos argumentos de entrada e uma outra que contem a informação relativa ao tipo de dados de saída, dados de entrada e nome da função. Neste caso, o nome da função funciona como chave na *hashmap* e o seu valor é a estrutura que contem a informação mais geral sobre a função. A *hashmap* utilizada implementa funções necessárias para a sua inicialização, inserção, remoção e consulta, sendo que também contém outras funções que não foram utilizadas no desenvolvimento deste projeto.

Capítulo 2

Compilador

2.1 Analisador léxico

O analisador léxico encontra-se desenvolvido com o suporte da ferramenta 'ylex' e deteta todos os símbolos terminais da linguagem (palavras reservadas, sinais e variáveis). Este analisador efetua também a deteção de comentários (linhas precedidas pelos símbolos '//'), ignorando o texto neles contidos. De forma a facilitar a deteção os erros de sintaxe, o parser conta as linhas que já interpretou. Este funcionalidade permite ao analisador sintático informar a linha onde ocorrer a anomalia em caso de erro de processamento. A passagem de valores é efetuada através do yylval.

2.2 Analisador sintático/semântico

O analisador sintático/semântico é o responsável por processar os tokens obtidos através do analisador léxico, utilizando as produções definidas para calcular a instrução a escrever no ficheiro de output final. O ficheiro onde se encontram todas as produções denomina-se compiler.y,

Definição dos tokens

Previamente à definição das produções, estão definidos os diversos tokens assim como o tipo das diferentes variáveis presentes nas produções. Encontram-se também definidas as precedências correspondentes às operações aritméticas, assim como a produção inicial (Prog).

Produções iniciais

O processamento é iniciado na produção Prog, correspondente ao começo do programa. São realizadas as inserções iniciais no ficheiro de output, e as produções a serem efetuadas podem consistir numa lista de declarações, numa lista de funções, ou numa lista de instruções. Caso se encontre uma lista de declarações, é realizado o controlo de saltos de

instruções, através da inserção da instrução "JUMP init"no ficheiro de output. Isto possibilita a declaração de variáveis antes da declaração das funções, de forma a que as funções possuam acesso às variáveis globais declaradas.

Produções de Funções

A declaração de funções é precedida pelo símbolo terminal '#', de forma a facilitar a sua leitura e distinção com outras instruções. O armazenamento das funções declaradas é realizado através da utilização de uma hashmap, e permite guardar o contexto de estas, ou seja, as declarações efetuadas dentro da função são tidas como variáveis locais, e apenas acessíveis dentro da função. No final da declaração da função, o contexto é encerrado. Por simplicidade, não é possível utilizar chamadas a funções como argumento de outra função.

Produções de declarações

As declarações de variáveis podem ocorrer dentro do contexto de uma função (i.e. variáveis locais) ou fora de qualquer contexto (i.e. variáveis globais). Desta forma, ao declarar uma variável, é efetuada o teste que verifica se está está declarada ou não dentro de uma função. Além disso, não é permitido re-declarar variáveis locais com o mesmo nome que variáveis globais já existentes, sendo apresentado um erro quando isso ocorre.

Produções de instruções e atribuições

São suportadas diversas instruções, no entanto vale a pena mencionar a instrução "return". Esta instrução calcula o endereço a retornar a partir do número de argumentos de entrada da função na qual se insere e do frame pointer salvaguardado. As atribuições permitem efetuar o incremento/decremento de variáveis através da sintaxe var++/var–, assim como atribuir o valor de expressões a variáveis escalares ou vetoriais.

Produções de input/output

As produções referentes à leitura e escrita no stdin. A produção de escrita suporta a impressão de expressões, sendo que a produção de leitura armazena as variáveis atribuídas conforme o contexto definido.

Produções de controlo de fluxo de execução condicional

A instrução de controlo de fluxo de execução condicional está definida como if...else, suportando um conjunto de instruções no seu contexto.

Produções de controlo de fluxo de execução cíclico

As produções correspondentes ao controlo de fluxo de execução cíclico suportam ciclos 'while', 'do while' e 'for'. Estas produções são suportadas por uma estrutura de dados (stack), sendo graças a esta que é possível calcular o saltos a efetuar.

Produções de cálculo de expressões

O compilador possui a capacidade de processar expressões aritméticas e lógicas, tanto em forma de declaração como em forma de cálculo do índice de um array. Estas expressões possibilitam também a utilização de parêntesis aninhados.

Nota: Na compilação do código referente ao programa yacc, é apresentado um erro de compilação do tipo shift-reduce. No entanto, este erro não provoca qualquer comportamento indesejado no programa dado que por defeito, quando em dúvida, o yacc efetua um shift ao invés de um reduce, sendo este o comportamento pretendido para este caso.

2.3 Geração de código máquina

A cada regra da gramática, são associadas ações a serem executadas à medida que estas são reconhecidas. Assim sendo, é realizada uma tradução da linguagem desenvolvida para a linguagem *assembly* da VM, à medida que cada instrução ou expressão é identificada. A maioria destas ações implica uma instrução de escrita no ficheiro de output. Todas estas acções que implicam escrita no ficheiro são triviais, existindo apenas algumas exceções como o caso do ciclo 'for' (figura 2.1). No ciclo 'for', é necessária a utilização de instruções 'JUMP', de forma a ser possível seguir o seu fluxo de execução normal. Após a identificação e execução das ações associadas à expressão lógica presente no ciclo 'for', é gerado o salto condicional respetivo, assim como um salto para as instruções associadas ao corpo do ciclo. Além disso, é gerada uma *label* que irá corresponder ao incremento do ciclo que irá ser identificado de seguida. Identificado o final do corpo do ciclo, é efetuado um salto para a *label* correspondente ao incremento do ciclo, que para além das respetivas instruções, conterá outro 'JUMP' para o teste da expressão lógica.

2.3.1 Funções

De forma a implementar adequadamente o processamento de funções, é necessário resolver certas implicações, sendo que esta funcionalidade obriga a tratar de diversos contextos dentro do programa (global e local).

As declarações das funções são efetuadas após as declarações das variáveis globais para que dentro das funções seja possível aceder às variáveis globais.

As declarações de variáveis são feitas no inicio da função, e no hash table das variáveis é guardado não apenas o endereço mas também o contexto (local ou global). Desta forma, no acesso às variáveis, utiliza-se 'PUSHG' ou 'PUSHL' seja respetivamente variável global ou local.

A passagem de argumentos para a função é tratada como uma declaração especial na qual o endereço é negativo. Já o retorno da função é colocado também num endereço negativo que foi previamente alocado na chamada da função.

Em forma de exemplo, assumindo a chamada de uma função com 2 argumentos. Como é possível observar na figura 2.2, os endereços são negativos ao fp, e o endereço onde a função colocará o retorno é $fp - 3$. Após a execução da função é feito o 'pop' dos argumentos e assim o valor de retorno da função está no topo da stack.

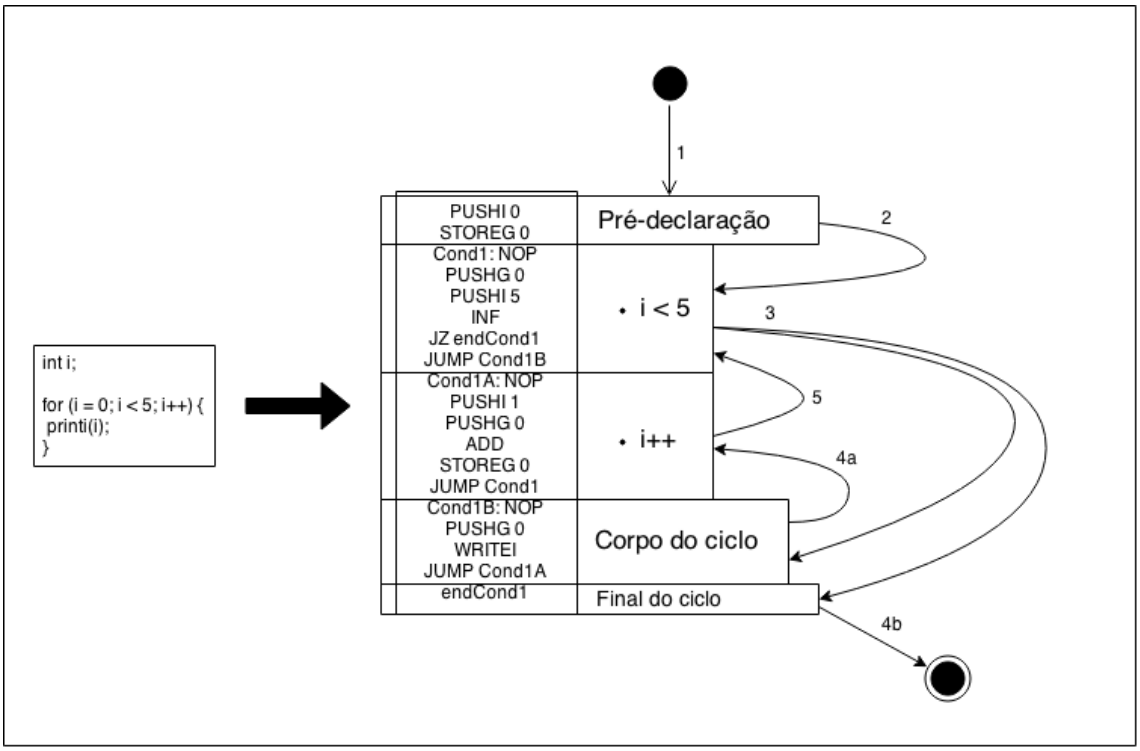

Figura 2.1: Código maquina gerado para ciclo 'for'

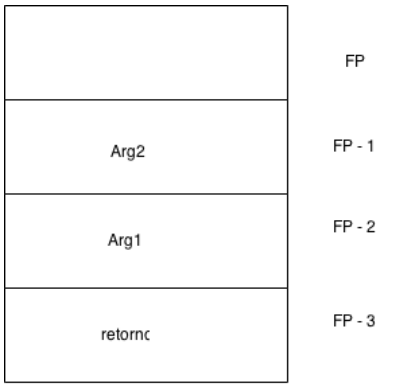

Figura 2.2: Chamada de uma função

Capítulo 3

Testes

3.0.2 Teste 1

Input

```
int i;

i=0;
do {
        i++;
        printi(i);
} while(i<100);
```

Output

```
START
PUSHI 0
JUMP init
init:NOP
PUSHI 0
STOREG 0
Cond1: NOP
PUSHI 1
PUSHG 0
ADD
STOREG 0
PUSHG 0
WRITEI
PUSHG 0
PUSHI 100
INF
JZ endCond1
JUMP Cond1
endCond1: NOP
```

```
STOP
```

3.0.3 Teste 2

Input

```
int i;
int read;

for(i=0; i<10; i++){
        scani(read);
        printi(read);
}
```

Output

```
START
PUSHI 0
PUSHI 0
JUMP init
init:NOP
PUSHI 0
STOREG 0
Cond1: NOP
PUSHG 0
PUSHI 10
INF
JZ endCond1
JUMP Cond1B
Cond1A: NOP
PUSHI 1
PUSHG 0
ADD
STOREG 0
JUMP Cond1
Cond1B: NOP
READ
ATOI
STOREG 1
PUSHG 1
WRITEI
JUMP Cond1A
endCond1
STOP
```

3.0.4 Teste 3

Input

```
int i;
int array[100];

i=0;
while(i<100) {
        array[i]=i;
}

for(i=0; i<100; i++){
        if(array[i]<50){
                printi(i);
        }
        else {
                array[i]=0;
        }
}
```

Output

```
START
PUSHI 0
PUSHN 100
JUMP init
init:NOP
PUSHI 0
STOREG 0
Cond1: NOP
PUSHG 0
PUSHI 100
INF
JZ endCond1
PUSHGP
PUSHG 1
PADD
PUSHG 0
PUSHG 0
STOREN
JUMP Cond1
endCond1
PUSHI 0
STOREG 0
Cond2: NOP
PUSHG 0
PUSHI 100
INF
JZ endCond2
JUMP Cond2B
Cond2A: NOP
PUSHI 1
```

```
PUSHG 0
ADD
STOREG 0
JUMP Cond2
Cond2B: NOP
PUSHGP
PUSHG 1
PADD
PUSHG 0
LOADN
PUSHI 50
INF
JZ endCond3
PUSHG 0
WRITEI
endCond3
PUSHGP
PUSHG 1
PADD
PUSHG 0
PUSHI 0
STOREN
JUMP Cond2A
endCond2
STOP
```

3.0.5 Teste 4

Input

```
// asd
int i;
int j;

# int fun(int a, int b) {
    return a + b;
}

i = 3;
j = 4;
i = fun(i + 2,j);
```

Output

```
START
PUSHI 0
PUSHI 0
JUMP init
fun:NOP
```

```
PUSHL -2
PUSHL -1
ADD
STOREL -3
RETURN
init:NOP
PUSHI 3
STOREG 0
PUSHI 4
STOREG 1
nPUSHI 0
PUSHG 0
PUSHI 2
ADD
PUSHG 1
CALL fun
POP 2
STOREG 0
STOP
```

3.0.6 Teste 5

Input

```
// asd
int i;
int j;

# int fun(int a) {
    int ret;
    if (a > 0) {
        ret = fun(a-1);
    }
    return ret;
}

printi(fun(i + 2));
```

Output

```
START
PUSHI 0
PUSHI 0
JUMP init
fun:NOP
PUSHI 0
PUSHL -1
PUSHI 0
SUP
```

```
JZ endCond1
nPUSHI 0
PUSHL -1
PUSHI 1
SUB
CALL fun
POP 1
STOREL 1
endCond1
PUSHL 1
STOREL -2
RETURN
init:NOP
nPUSHI 0
PUSHG 0
PUSHI 2
ADD
CALL fun
POP 1
WRITEI
STOP
```

Capítulo 4

Conclusão

Finalizado o desenvolvimento do trabalho, é possível analisar o resultado final e o impacto que as diversas decisões tiveram sobre este. Um dos principais pontos positivos consiste na implementação do processamento e compilação de funções, sendo esta a funcionalidade mais trabalhosa e sobre a qual recaiu maior parte do tempo despendido. Relativamente a estas, de notar uma mudança na forma como estas foram implementadas. Anteriormente, a estrutura relativa ao armazenamento de dados de uma função possuía a capacidade de dar acesso às variáveis declaradas dentro do seu contexto, no entanto decidiu-se que esta funcionalidade era desnecessária para o funcionamento do compilador, sendo esta informação descartada. A implementação das expressões de controlo de execução possuíram também uma dificuldade acrescida, obrigando à utilização de estruturas de dados mais complexas, tais como hashmaps e stacks. Creemos ter alcançado os objetivos definidos aquando da proposta do trabalho, tendo desenvolvido um compilador capaz de processar uma LPIS, com a possibilidade de dar feedback sobre o código de input definido e criar o ficheiro com instruções em Assembly correspondentes.

Capítulo 5

Anexos

5.1 parser.l

```
%{
#include "vmCompiler.h"
#include "y.tab.h"

int ccLine = 1;

%}


%%

int             {return(INT);}
while           {return(WHILE);}
for             {return(FOR);}
if              {return(IF);}
else            {return(ELSE);}
return          {return(RETURN);}
void            {return(VOID);}
printi          {return(PRINTI);}
scani           {return(SCANI);}
true            {return(TRUE);}
false           {return(FALSE);}
do              {return(DO);}
\=              {return('=');}
\.              {return('.');}
\;              {return(';');}
\(              {return('(');}
\)              {return(')');}
\{              {return('{');}
```

```
\}              {return('}');}
\[              {return('[');}
\]              {return(']');}
\,              {return(',');}
\<              {return('<');}
\>              {return('>');}
\+              {return('+');}
\-              {return('-');}
\*              {return('*');}
\/              {return('/');}
\%              {return('%');}
\#              {return('#');}
\|              {return('|');}
\&              {return('&');}
[a-zA-Z]+       {yylval.var_name = strdup(yytext); return(var);}
[0-9]+          {yylval.value = atoi(yytext); return(num);}
[\n]            { ccLine++;}
\/\/.*          { ; }
.               { ; }
%%

int yywrap()
{ return(1); }
```

5.2 compiler.y

```
%{
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "vmCompiler.h"
#include "stack.h"
#include "y.tab.h"


void yyerror(char *s);
extern ccLine;
FILE *f;
static int total;
static Stack s;

%}

%union{
     char* var_name;
     int value;
     Type type;
     struct sVarAtr
```

```
    {
        char* var_name;
        int value;
        int size;
    } varAtr;
}

%token INT WHILE FOR IF ELSE RETURN VOID PRINTI SCANI TRUE FALSE DO END
%token var nomefuncao num

%type <var_name> var
%type <value> num
%type <varAtr> VarAtr
%type <varAtr> Atrib
%type <type> Tipo

%left '+' '-'
%left '*' '/' '%'
%left '&' '|'

%start Prog

%%
// ##################### PROGRAMA #####################################

Prog:                                   {fprintf(f,"START\n");}
            ListaDecla                  {fprintf(f,"JUMP_init\n");}
            ListaFun                    {fprintf(f,"init:NOP\n");}
            ListInst                    {fprintf(f,"STOP\n");}
            ;

ListaFun:   ListaFun Funcao
            |
            ;

ListaDecla: ListaDecla Decla
            |
            ;

ListInst:   ListInst Inst
            | Inst
            ;
// ##################### FUNCAO ###########################################

Funcao:     '#' Tipo var                {decFun($2,$3);}
            '(' ListaArg ')'            {decFunArgRefresh();
                                         fprintf(f,"%s:NOP\n",$3);}
            '{' ListaDecla ListInst '}' {endDecFun();}
            ;
```

```
ListaArg:    | ListaArg2 ;

ListaArg2:   Tipo var                        {decAddFunArg($1,$2);}
             | ListaArg2  ','  Tipo var      {decAddFunArg($3,$4);}
             ;


// ################### DECLARACAO ###################################

Decla:       INT var ';'                     {decVar($2, 1);
                                              fprintf(f,"PUSHI 0\n");}
             | INT var '[' num ']' ';'       {decVar($2, $4);
                                              fprintf(f,"PUSHN %d\n",$4);}
             ;

Tipo:        VOID                            {$$ = _VOID;}
             | INT                           {$$ = _INTS;}
             ;

// ###################### INSTRUCAO ###################################

ConjInst:    |'{' ListInst '}';

Inst:        If
             | While
             | DoWhile
             | For
             | Atrib ';'
             | Printi';'
             | Scani ';'
             | RETURN Exp ';' {fprintf(f,"STOREL %d\n",decFunRetAddr());
                               fprintf(f,"RETURN\n");}
             | ELSE           {yyerror("'Else' sem um 'If' anteriormente");}
             ;

VarAtr:      var             {Addr a = getAddr($1);   $$.var_name=strdup($1);
                                                      $$.size=1;}
             ;
// ###### ÇATRIBUIAO ######

Atrib:       VarAtr '=' Exp          {Addr a = getAddr($1.var_name);
                                     if(a.type == _INTS)
                                         fprintf(f,"STORE%c %d\n",a.scope,a.addr);
                                     else yyerror("Tipos íincompatveis");
                                     }
             | VarAtr '+''+'         {Addr a = getAddr($1.var_name);
                                     if(a.type == _INTS)
                                         fprintf(f,"PUSHI 1\nPUSH%c %d\nADD\n"
```

```
                                                    "STORE%c %d\n", a.scope,
                                                    a.addr,a.scope,a.addr);
                                        else yyerror("Tipos incompatveis");
                                        }
            | VarAtr                    {Addr a = getAddr($1.var_name);
                                        fprintf(f,"PUSH%cP\nPUSH%c %d\nPADD\n",
                                            (a.scope=='L')? 'F':'G',a.scope,a.addr);}
             '[' Exp ']' '=' Exp        {fprintf(f,"STOREN\n");}
            ;

// ###### PRINT SCAN ######

Printi:     PRINTI '(' Exp ')'          {fprintf(f,"WRITEI\n");}
            ;

Scani:      SCANI '(' VarAtr ')'        {Addr a = getAddr($3.var_name);
                                        fprintf(f,"READ\nATOI\nSTORE%c %d\n",
                                                a.scope,a.addr);}
            ;
// ###### IF THEN ELSE ######

If:         IF                          {total++; push(s,total);}
            TestExpL                    {fprintf(f,"JZ endCond%d\n", get(s));}
            ConjInst                    {fprintf(f,"endCond%d\n", pop(s));}
            Else
            ;

Else:       | ELSE ConjInst ;

// ####### WHILE ###########

While:      WHILE                       {total++; push(s,total);
                                        fprintf(f,"Cond%d: NOP\n", get(s));}
            TestExpL                    {fprintf(f,"JZ endCond%d\n", get(s));}
            ConjInst                    {fprintf(f,"JUMP Cond%d\nendCond%d\n",
                                         get(s), get(s)); pop(s);}
            ;

// ####### DO WHILE ###########

DoWhile:    DO                           {total++; push(s,total);
                                         fprintf(f,"Cond%d: NOP\n", get(s));}
            ConjInst WHILE TestExpL      {fprintf(f,"JZ endCond%d\nJUMP Cond%d\n"
                                                  "endCond%d: NOP\n",get(s),
                                                  get(s) ,get(s)); pop(s);}
            ;

// ####### FOR ###########
```

```
For:        FOR ForHeader ConjInst    {fprintf(f,"JUMP Cond%dA\nendCond%d\n",
                                            get(s), get(s)); pop(s);}
            ;

ForHeader:  '(' ForAtrib ';'          {total++; push(s,total);
                                       fprintf(f,"Cond%d: NOP\n", get(s));}
            ExpL ';'                  {fprintf(f,"JZ endCond%d\nJUMP Cond%dB\n"
                                            "Cond%dA: NOP\n", get(s),
                                             get(s), get(s));}
            ForAtrib ')'              {fprintf(f,"JUMP Cond%d\nCond%dB: NOP\n",
                                            get(s), get(s));}
            ;

ForAtrib:   Atrib | ;

// ################################ CALCULO DE EXPRESSOES ######################

Exp:         Exp '+' Exp              {fprintf(f,"ADD\n");}
           | Exp '-' Exp              {fprintf(f,"SUB\n");}
           | Exp '%' Exp              {fprintf(f,"MOD\n");}
           | Exp '*' Exp              {fprintf(f,"MUL\n");}
           | Exp '/' Exp              {fprintf(f,"DIV\n");}
           | '(' Exp ')'
           | num                      {fprintf(f,"PUSHI %d\n", $1);}
           | VarAtr                   {Addr a = getAddr($1.var_name);
                                       fprintf(f,"PUSH%c %d\n",a.scope,a.addr);}
           | VarAtr                   {Addr a = getAddr($1.var_name);
                                        fprintf(f,"PUSH%cP\nPUSH%c %d\nPADD\n",
                                              (a.scope=='L')? 'F':'G',a.scope,a.addr);}
           '[' Exp ']'                {fprintf(f,"LOADN\n");}
           | var                      {expFun($1); fprintf(f,"nPUSHI 0\n");}
           '(' FunArgs')'             {fprintf(f,"CALL %s\n",$1);
                                         fprintf(f, "POP %d\n",expFunNArgs());}
           ;


FunArgs:   | FunArgs2 ;
FunArgs2:  Exp                        {expFunNextArg(_INTS);}
           | FunArgs2 ',' Exp         {expFunNextArg(_INTS);}
           ;

TestExpL:  '(' ExpL ')'
           ;

ExpL:        Exp '=''=' Exp           {fprintf(f,"EQUAL\n");}
           | Exp '!''=' Exp           {fprintf(f,"EQUAL\nPUSHI 0\nEQUAL\n");}
           | Exp '>''=' Exp           {fprintf(f,"SUPEQ\n");}
           | Exp '<''=' Exp           {fprintf(f,"INFEQ\n");}
           | Exp '<' Exp              {fprintf(f,"INF\n");}
```

```
              | Exp '>' Exp            {fprintf(f,"SUP\n");}
              | '(' ExpL ')'           {fprintf(f,"PUSHI 1\nEQUAL\nJZ endCond%d:NOP\n",
                                                get(s));}
              '&''&' '(' ExpL ')'      {fprintf(f,"PUSHI 1\nEQUAL\nJZ endCond%d:NOP\n",
                                                get(s));}
              | '(' ExpL ')' '|''|' '(' ExpL ')'
                                       {fprintf(f,"ADD\nJZ endCond%d:NOP\n", get(s));}

%%

void yyerror(char *s){
    fprintf(stderr,"ERRO: Syntax LINHA: %d MSG: %s\n",ccLine,s);
    exit(0);
}

void init()
{
    s = initStack();
    total = 0;
    f = fopen("assembly.out", "w");
}
```

5.3 vmCompiler.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "hashmap.h"
#include "vmCompiler.h"

#define OK 0
#define ERRO_VAR_ALREADY_EXIST -1
#define ERRO_VAR_DONT_EXIST -2
#define ERRO_VAR_INVALID_TYPE -3
#define ERRO_FUN_DONT_EXIST -4
#define ERRO -5

void yyerror(char *s);

struct sEntryVar{
    Type type;
    char *name;
    int memAdr;
};

typedef struct sFunArg
{
    struct sEntryVar* v;
```

```
    struct sFunArg *next;
}* FunArgL;

struct sEntryFun{
    Type type;
    char *name;
    FunArgL args;
    FunArgL argsEnd;
    int nargs;
};

typedef struct sScope{
    map_t vars;
    int addrCount;
}* Scope;

static EntryFun inUseFun;
static EntryFun decFunAux;
static Scope gloContext;
static Scope funContext;

static map_t mFuncMap;


EntryVar containsVar(Scope fun, char* varName);


int initVarMap()
{
    gloContext = (Scope) malloc(sizeof(struct sScope));
    gloContext->vars = hashmap_new();
    gloContext->addrCount = 0;
        //addressCounter = 0;
        //mVarMap = hashmap_new();
        mFuncMap = hashmap_new();
    return 0;
}


EntryFun containsFun(char* varName)
{
    EntryFun varEntry; //= (Entry) malloc(sizeof(Entry));

    if(!(hashmap_get(mFuncMap, varName, (any_t*) &varEntry) == MAP_OK))
        varEntry = NULL;
    return varEntry;
}

EntryVar containsVar(Scope fun, char* varName)
```

```
{
        EntryVar varEntry; //= (Entry) malloc(sizeof(Entry));
    if (fun==NULL)
        fun = gloContext;

        if(!(hashmap_get(fun->vars, varName, (any_t*) &varEntry) == MAP_OK))
                varEntry = NULL;
        return varEntry;
}

int decFun(Type type,char* funName){
    int ret;
    if(!containsFun(funName)) {
        EntryFun newFun = (EntryFun) malloc(sizeof(struct sEntryFun));
        newFun->name = strdup(funName);
        newFun->type = type;
        newFun->args = NULL;
        newFun->argsEnd = NULL;
        newFun->nargs  = 0;

        hashmap_put(mFuncMap, funName, (any_t) newFun);
        decFunAux = newFun;

        funContext = (Scope) malloc(sizeof(struct sScope));
        funContext->vars = hashmap_new();
        funContext->addrCount = 0;
        ret = OK;
    } else {
        yyerror("áVarivel áj declarada anteriormente");
        ret = ERRO_VAR_ALREADY_EXIST;
    }
    return ret;
}

int decAddFunArg(Type type, char* name){
    if(decFunAux->argsEnd == NULL){
        decFunAux->argsEnd = (FunArgL) malloc(sizeof(struct sFunArg));
        decFunAux->args = decFunAux->argsEnd;
    } else {
        decFunAux->argsEnd->next = (FunArgL) malloc(sizeof(struct sFunArg));
        decFunAux->argsEnd = decFunAux->argsEnd->next;
    }
    decFunAux->argsEnd->next = NULL;
    //funContext->argsEnd->v
    int err = decVar(name,1);
    if (err == OK){
        (decFunAux->nargs)++;
        hashmap_get(funContext->vars, name, (any_t*) &(decFunAux->argsEnd->v));
    }
```

```
    return err;
}

void decFunArgRefresh(){
    FunArgL i = decFunAux->args;
    while(i != NULL){
        i->v->memAdr -= decFunAux->nargs;
        i = i->next;
    }
}

int decFunRetAddr(){
    return -(decFunAux->nargs) - 1;
}

void endDecFun(){
    decFunAux = NULL;
    funContext = NULL;
}

int expFun(char * fun){
    inUseFun = containsFun(fun);
    inUseFun->argsEnd = inUseFun->args;

    if(inUseFun == NULL) {
        yyerror("ERROR Funtion don't exist");
        return ERRO_FUN_DONT_EXIST;
    }
}
int expFunNextArg(Type type){
    if(inUseFun->argsEnd == NULL){
        yyerror("ERROR invalid number of arguments");
        return ERRO;
    }

    if(type != inUseFun->argsEnd->v->type){
        yyerror("ERROR types don't match");
        return ERRO_FUN_DONT_EXIST;
    }


    inUseFun->argsEnd = inUseFun->argsEnd->next;
    return OK;
}
int expFunNArgs(){
    if(inUseFun->argsEnd != NULL){
        yyerror("ERROR invalid number of arguments");
        return ERRO;
    }
```

```
    return inUseFun->nargs;
}

int decVar(char* varName, int size)
{
    Scope context;
    int err = 0;
    if (funContext == NULL) {
        context = gloContext;
        err += (containsVar(gloContext, varName) != NULL);
    } else {
        context = funContext;
        err += (containsVar(funContext, varName) != NULL);
        err += (containsVar(gloContext, varName) != NULL);
    }

    if(!err)
        {
                EntryVar newVar = (EntryVar) malloc(sizeof(struct sEntryVar));
                if(size>1)
                {
                        newVar->type = _INTA;
                }
                else
                {
            newVar->type = _INTS;
                }
                newVar->name = strdup(varName);
                newVar->memAdr = context->addrCount;
                context->addrCount+=size;

                hashmap_put(context->vars, varName, (any_t) newVar);

                return OK;
        }
    yyerror("áVarivel áj declarada anteriormente");
        return ERRO_VAR_ALREADY_EXIST;
}


Addr getAddr(char* varName)
{
    EntryVar varEntry;
        int memAddr;
    char scope;
    Type type;

    if (funContext == NULL) {
        varEntry = containsVar(gloContext,varName);
```

```
        scope = 'G';
    } else {
        varEntry = containsVar(funContext,varName);
        scope = 'L';
        if(varEntry == NULL) {
            varEntry = containsVar(gloContext,varName);
            scope = 'G';
        }
    }

        if(varEntry != NULL) {
                memAddr = varEntry->memAdr;
        type = varEntry->type;
        } else {
        memAddr = ERRO_VAR_DONT_EXIST;
        yyerror("áVarivel ãno declarada");
    }


    Addr ret = {memAddr,scope,type};

        return ret;
}
```