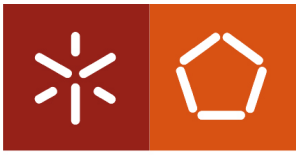

Universidade do Minho

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 Linguangens da Licenciatura em Engenharia Informática da Universidade do Minho. O trabalho pretende aumentar a experiência em engenharia de linguagens, e motivar a utilização de ferramentas de compilação de compiladores e análise léxica.

Para isso era pretendido criar uma linguagem de programação imperativa simples, que será descrita à frente. O compilador desta linguagem foi desenvolvido desde a analise léxica até à geração de código. O Código gerado foi assembly para a maquina virtual fornecida por o docente.

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.

<Programa exemplo (programa exemplo que mostre de forma simples todas as funcionalidades - funçoes, atribuiçoes, ciclos, expressoes logicas e aritmeticas)>

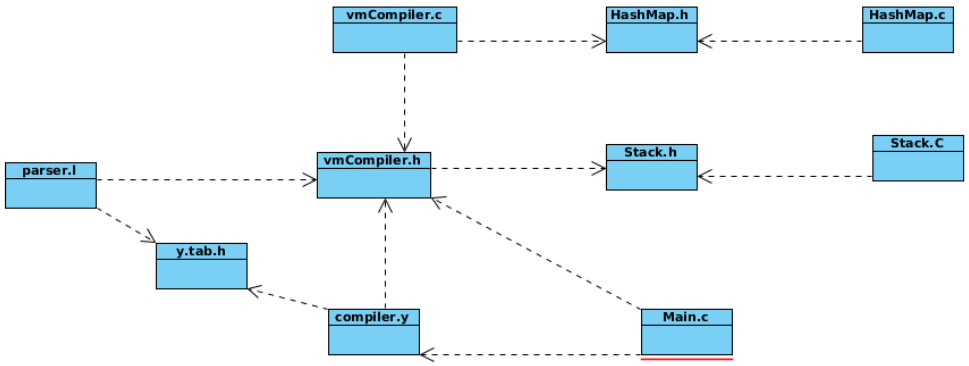

Figura 1.1: Diagrama das pedendencias dos ficheiros

1.2 Arquitetura

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

Na Figura 1.1, podemos observar as dependências entre os ficheiros.

O analisador sintático utiliza o ficheiro vmCompiler.h, sendo que este o 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 respetivamente 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 - possuem 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;

Com as estruturas anteriores em mente, as variáveis criadas em vmCompiler são as representadas na figura 1.3, onde se podem ver duas variáveis do tipo Scope, uma para o contexto global e outra para o contexto interior a uma função.

Existe um map de EntryFun (mFuncMap) onde a chave é o nome da função.

A variável DecFunAux consiste num apontador temporário para uma função declarada.

Figura 1.2: Diagrama das estruturas usadas em vmCompiler

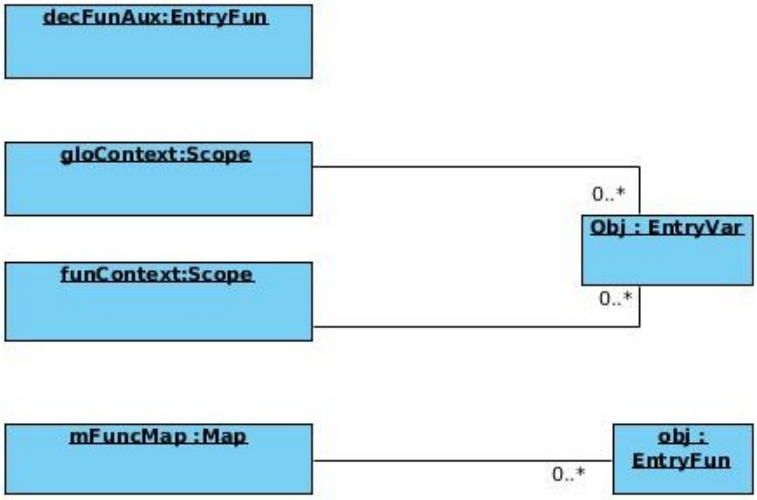

Figura 1.3: Diagrama dos objetos existente em vmCompiler

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 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.

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 sintatico/semantico

As funções são declaradas depois das variáveis globais para as funções terem acesso às variáveis globais.

<erro de compilaçao nao tem problema pq fazer o shift por default é que é o necessario fazer>

2.3 Geração de codigo máquina

A cada regra da gramática, associamos acções a serem executadas à medida que estas são reconhecidas. Assim sendo, é feita uma tradução da linguagem desenvolvida para a linguagem *assembly* da VM, à medida que cada instrução ou expressão é identificada. A maioria destas acções implica uma instrução de escrita no ficheiro de output. Todas estas acções que implicam escrita no ficheiro são triviais, existindo apenas algumas excepções como o caso do ciclo 'for'. No ciclo 'for', foi necessária a utilização de instruções 'JUMP', de forma a ser possível seguir o seu fluxo de execução normal. Após a identificação e execução das acções associadas à expressão lógica presente no ciclo 'for', é gerado o salto condicional respectivo, assim como um salto para as instruções associadas ao corpo do ciclo. Além disso, é gerada uma *label* que irá corresponder

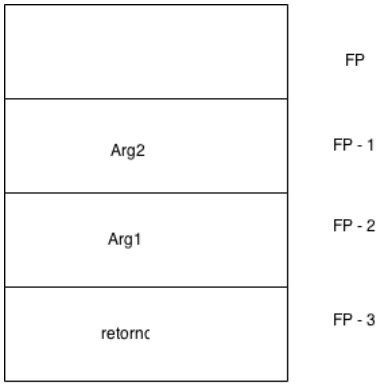

Figura 2.1: Chamada de uma função

ao incremento do ciclo que irá ser identificado de seguida. Identificado o final do corpo do ciclo, é feito 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

Para a implementação de funções existiram algumas implicações que tiveram que ter ser resolvidas, teve-se que passar a lidar com mais do que um contexto, ou seja as variáveis podem ser globais ou locais.

As declarações das funções vem logo após às declarações das variáveis globais para que dentro das funções já se possa aceder às variáveis globais.

As declarações de variáveis são feitas no inicio da função, e no hash table das variáveis guarda-se não só o endereço mas também o contexto (ou seja se é local ou global). Com isto nos acessos ás variáveis utiliza-se 'PUSHG' ou 'PUSHL' mediante seja respetivamente variável global ou local.

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

Para perceber melhor vamos utilizar um exemplo, assumindo que estamos a chamar uma função com 2 argumentos. Como podemos ver na figura 2.1, 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.

Capítulo 3

Testes

3.0.2 Teste 1

Input

Output

3.0.3 Teste 2

Input

Output

3.0.4 Teste 3

Input

Output

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
            '{' 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",
             ;

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

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

ConjInst:    |'{' ListInst '}';

Inst:        If
             | While
             | DoWhile
             | For
             | Atrib ';'
             | Printi';'
             | Scani ';'
             | RETURN Exp ';'                                {fprintf(f,"STOREL_%d\n",decF
             | ELSE                                          {yyerror("'Else'_sem_um_'If'_
             ;

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

Atrib:       VarAtr '=' Exp                                  {Addr a = getAddr($1.var_name
                                                             if(a.type == _INTS)
                                                                 fprintf(f,"STORE%c_%d\n",
                                                             else yyerror("Tipos_íincompat
                                                             }
             | VarAtr '+''+'                                 {Addr a = getAddr($1.var_name
                                                             if(a.type == _INTS)
                                                                 fprintf(f,"PUSHI_1\nPUSH%
                                                                         a.scope,a.addr,a.
                                                             else yyerror("Tipos_íincompat
                                                             }
             | VarAtr                                        {Addr a = getAddr($1.var_name
                                                             fprintf(f,"PUSH%cP\nPUSH%c_%d
```

```
                '[' 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%
              ;
// ###### IF THEN ELSE ######

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

Else:         | ELSE ConjInst ;

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

While:        WHILE                                    {total++; push(s,total); fpri
              TestExpL                                 {fprintf(f,"JZ endCond%d\n",
              ConjInst                                 {fprintf(f,"JUMP Cond%d\nendC
              ;

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

DoWhile:      DO                                       {total++; push(s,total); fpri
              ConjInst WHILE TestExpL                  {fprintf(f,"JZ endCond%d\nJUM
              ;

// ####### FOR ###########

For:          FOR ForHeader ConjInst                   {fprintf(f,"JUMP Cond%dA\nend
              ;

ForHeader:    '(' ForAtrib ';'                         {total++; push(s,total); fpri
              ExpL ';'                                 {fprintf(f,"JZ endCond%d\nJUM
              ForAtrib ')'                             {fprintf(f,"JUMP Cond%d\nCond
              ;

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.add
              | VarAtr                               {Addr a = getAddr($1.var_name);
                                                     fprintf(f,"PUSH%cP\nPUSH%c %d\nPADD\n
              '[' Exp ']'                            {fprintf(f,"LOADN\n");}
              | var                                  {expFun($1); fprintf(f,"nPUSHI 0\n");
              '(' FunArgs')'                         {fprintf(f,"CALL %s\n",$1); fprintf(f
              ;


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\
              | 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 end
              '&''&' '(' ExpL ')'                      {fprintf(f,"PUSHI 1\nEQUAL\nJZ end
              | '(' ExpL ')' '|''|' '(' ExpL ')'       {fprintf(f,"ADD\nJZ endCond%d:NOP\

%%

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;
}
```