Universidade do Minho

Processamento de Linguagens

Trabalho Prático II

Nelson Mota nº38573

Filipe Ribeiro nº64315

Junho de 2015

Resumo

O presente trabalho foi desenvolvido no âmbito da unidade curricular de Processamento de Linguagens e tem como principal objetivo o aumentar o conhecimento e a capacidade de escrever GIC (Gramáticas Independentes de Contexto), bem como a utilização de ferramentas como o *flex* e *yacc* para a desenvolvimento de um compilador. Ao longo do mesmo relatório iremos apresentar os principais aspetos do funcionamento da aplicação, demonstrar e explicar as estruturas de dados que foram implementadas, bem como todas as diferentes decisões tomadas ao longo deste projeto.

Conteúdo

1 Introdução

Pretende-se, para este projeto que inicialmente se comece por definir uma simples linguagem de programação imperativa (LPIS), que seja capaz de manusear variáveis do tipo inteiro, escalares ou arrays, ter a capacidade de realizar as diferentes operações aritméticas e lógicas, bem como input e output. Além disso teria de ser capaz de lidar com instruções de controlo de fluxo, como *If's* e *While's*.

Posterior ao desenvolvimento da (LPIS), é proposto desenvolver um compilador que com base na gramática criada, com a utilização de ferramentas como o *Yacc* e o *Flex*, fosse capaz de gerar pseudo-código *Assembly* para uma maquina virtual.

2 Linguagem de Programação

A linguagem de programação criada teve como inspiração duas outras já existentes, o C, sendo esta umas das linguagem com grande ênfase no decorrer o curso, e como o Pascal, pela sua forma estruturada de definir um programa.
A linguagem criada em a seguinte estrutura:

```
program NOME_PROGRAMA;
var
    DECLARAÇÃO ;
begin
    CÓDIGO ;
end
```

2.1 GIC

Tendo como base o que é pretendido e a estrutura da linguagem criada, definimos a seguinte gramática;

```
1  Program      :   PROGRAM PROGNAME ';' Corpo
2               ;
3  Corpo        :   Declaracoes BBEGIN Codigo END
4               ;
5  Declaracoes  :
6               |   VAR VarIds ';'
7               ;
8  VarIds       :   Var
9               |   VarIds ',' Var
10              ;
11 Var          :   id
12              |   id'['num']'
13              |   Atribuicao
14              ;
15 Codigo       :
16              |   Codigo Instrucao
17              ;
```

```
18 Instrucao    :   Atribuicao ';'
19              |   Input ';'
20              |   Output  ';'
21              |   Condicao
22              |   Ciclo
23              ;
24 Atribuicao   :   id '=' ExpSimples
25              |   id'['num']' '=' ExpSimples
26              ;
27 Input         :   '?' id
28              |   '?' id'['num']'
29              ;
30 Output        :   '!' string
31              |   '!' id
32              |   '!' id'['num']'
33              ;
34 Condicao     :   IF '(' Expressao ')' '{' Codigo '}' ElseIf
35              ;
36 ElseIf       :
37              |   ELSE '{' Codigo '}'
38              ;
39 Ciclo        :   WHILE '(' Expressao ')' '{' Codigo '}'
40              ;
41 Expressao    :   ExpSimples
42              |   Expressao OPR ExpSimples
43              ;
44 ExpSimples   :   Termo
45              |   ExpSimples '+' Termo
46              |   ExpSimples '-' Termo
47              ;
48 Termo        :   Factor
49              |   Termo '*' Factor
50              |   Termo '/' Factor
51 Factor       :   num
52              |   id
53              |   id '['num']'
54              |   '(' ExpSimples ')'
55              ;
```

2.2 Flex

Para processar os ficheiros que contem o programa a executar, à semelhança do trabalho pratico anterior foi utilizado o *Flex*, que comunicando com a ficheiro *yacc* nos permitir a leitura e interpretação do programa.
De forma a organizar o ficheiro *flex*, recorremos à utilização dos contextos,

```
%x declaracoes programName var code
```

onde cada um deles representa diferentes partes do código.
Alem da utilização dos contextos, para conseguir determinar a corrente linha do ficheiro de entrada usamos:

```
%option yylineno
```

O ficheiro do analisador criado, *(analex.l)*, encontra-se em anexo.

2.3 Yacc

Para as diferentes produções da gramática desenvolvida, gerar o pseuso-código Assembly correspondente, o mais complexo foi a geração para instruções de controlo de fluxo, estruturas condicionais e ciclos, como conseguir gerar e guardar as flags para os respetivos saltos. Para isso recorremos a utilização de stacks, quando encontra uma destas instruções faz push de uma label para a stack, quando sair, faz pop, podendo assim aninhar diferentes instruções de fluxo.

Para que fosse possível uma correta implementação da gramática:

```
1 %union{
2     char* pal;
3     int inteiro;
4     struct  SFactor
5     {
6         int type;
7         int valueI;
8         char *valueS;
9     }factor;
```

```
10 }
11
12
13 %token <inteiro> num
14 %token <pal> id string OPR
15 %token ERROR PROGRAM PROGNAME VAR BBEGIN END IF ELSE WHILE
16
17 %type <factor> Termo
18 %type <factor> Factor
19 %type <factor> ExpSimples
20
21 %start Program
```

O ficheiro *yacc* desenvolvido *(anasyn.y)* encontra-se em anexo.

2.3.1 Geração de Erros

A deteção e identificação de erros sempre foi uma grande ajuda para os programadores, para isso decidimos definir oito tipos de erros, identificados de seguida:

```
1 #define ERR_SINTAX "ERROR: Sintax error"
2 #define ERR_NOT_DEC "ERROR: Variable not defined"
3 #define ERR_VAR_EXISTS "ERROR: Variable already declared"
4 #define ERR_INDEX_OOB "ERROR: Array index out of bounds"
5 #define ERR_TYPE_ERROR "ERROR: Wrong type variable"
6 #define ERR_INIT_ERROR "ERROR: Can't initialize arrays in
      declaration"
7 #define ERR_MISS_OP_BRACK " (maybe missing opening brackets?)"
8 #define ERR_MISS_CL_BRACK " (maybe missing closing brackets?)"
```

Ao compilar um programa, o código Assembly só é apresentado ao utilizador se não existir qualquer tipo de erro, caso contrario será apresentado através do terminal uma descrição do mesmo, bem como a identificação e a respetiva linha onde este foi detetado. Se a compilação foi concluída com sucesso é exibido no terminal o código assembly gerado, sendo possível redirecionar para um ficheiro caso seja pretendido.

Se não foi introduzido nenhum ficheiro como parâmetro, o programa inicia em

modo interativo através do terminal, e em caso de erro, este será de imediato apresentado, não gerando o Assembly dessa mesma instrução, tendo o programador a oportunidade de corrigir a instrução e continuar o programa.

2.4 Estruturas de Dados Auxiliares

2.4.1 Tabela de Identificadores

Para guarda a informação toda a informação necessária das variáveis declaradas no programa, decidimos utilizar uma estrutura em forma de *hashtable* para o mesmo.

Para cada variável será guardado o seu tipo (int ou array), o seu endereço, a sua categoria e o seu tamanho, 1 em caso de inteiro, n em caso de array, onde n é o tamanho do mesmo.

Uma das opções que implementamos, ao compilar um um determinada programa, usando a opção -v, alem de ser apresentado o respetivo código assembly será também apresentado a sua tabela de identificadores.

```
1  typedef struct svariable {
2    int tipo;
3    int addr;
4    int cat;
5    int tam;
6  } variable, *Variable;
7
8  typedef struct entry {
9    char id[MAX_ID];
10   Variable var;
11   UT_hash_handle hh;
12 } *HashTable;
```

2.4.2 Assembly

Como já foi mencionado, o pseudo-assembly do programa pretendido apenas é apresentado caso não exista qualquer erro, para isso é necessário percorrer o programa todo, assim, enquanto compila o programa guarda a o assembly gerado numa estrutura que segue uma filosofia FIFO, implementada através de uma simples lista ligada. Caso não exista nenhum erro, é apresentado o assembly gerado.

Para conseguirmos garantir que os erros sejam apresentados interativamente, o código assembly gerado é guardado num buffer temporário que, caso a instrução seja valida, acrescenta ao assembly definitivo já gerado.

```
1 typedef struct assembly *Assembly;
2 struct assembly{
3   char *code;
4   struct assembly *next;
5 };
```

2.4.3 Stack

Para conseguir gerir o controlo das labels das instruções de controlo de fluxo usamos uma simples estrutura stack, quando deteta o inicio de uma dessas instruções faz push de uma label, que é incrementada posteriormente, quando sai faz pop, mantendo assim a consistência.

```
typedef struct sStack *Stack;
struct sStack {
    int     data[STACK_MAX];
    int     size;
};
```

3 Makefile

Para facilitar a compilação da aplicação criamos a seguinte Makefile:

```
1  compiler : lex.yy.o y.tab.o stack.o hash.o assembly.o
2   gcc -ggdb -o compiler y.tab.o lex.yy.o stack.o hash.o assembly.o -
        ll
3
4  hash.o : hash.c hash.h uthash.h
5   gcc -c hash.c
6
7  stack.o : stack.c stack.h
8   gcc -c stack.c
9
10 assembly.o : assembly.c assembly.h
11  gcc -c assembly.c
12
13 y.tab.o : y.tab.c
14  gcc -c y.tab.c
15
16 lex.yy.o : lex.yy.c
17  gcc -c lex.yy.c
18
19 y.tab.c, y.tab.h : anasyn.y
20  yacc -d -t -v anasyn.y
21
22 lex.yy.c : analex.l y.tab.h
23  flex analex.l
```

4 Exemplo de Utilização - Programas Válidos

4.1 Exemplo 1

```
program prog1;
    var a,b=1,c,d[10];
begin
    a=b*(75/(6-2));
    d[5] = a;
    if(a>1){
        c=a/b;
    } else {
        c=0;
    }
end
```

$ ./compiler -v program0.txt

```
 Nome - Endereco - Tipo - Categoria - Tamanho -
| a        0        0        0           1    |
| b        1        0        0           1    |
| c        2        0        0           1    |
| d        3        1        0           10   |
 - - - - - - - - - - - - - - - - - - - - - - -
PUSHN 1
PUSHI 1
STOREG 1
PUSHN 1
PUSHN 10
START
PUSHG 1
PUSHI 75
PUSHI 6
PUSHI 2
SUB
DIV
MUL
STOREG 0
PUSHG 0
STOREG 8
PUSHG 0
PUSHI 1
SUP
JZ L0001
PUSHG 0
PUSHG 1
DIV
STOREG 2
JUMP L0002
L0001:
PUSHI 0
STOREG 2
L0002:
STOP
```

4.2 Exemplo 2

```
program prog2;
var a;
begin
    a=0;
    if(a==0){
        while(a<10){
            !"Teste!";
            a= a + 1;
        }
    }
    else{
        !"ElseTeste!";
    }
end
```

$ ./compiler program1.txt

```
PUSHN 1
START
PUSHI 0
STOREG 0
PUSHG 0
PUSHI 0
EQUAL
JZ L0001
L0002:
PUSHG 0
PUSHI 10
INF
JZ L0003
PUSHS "Teste!"
WRITES
PUSHG 0
PUSHI 1
ADD
STOREG 0
JUMP L0002
L0003:
JUMP L0004
L0001:
PUSHS "ElseTeste!"
WRITES
L0004:
STOP
```

5 Exemplo de Utilização - Programas Inválidos

5.1 Exemplo 1

```
1 program prog3;
2 var a,b=1,a,d[10];
3 begin
4     a = b;
5     c = 1;
6     d[11] = 1;
7 end
```

$ ./compiler programERRO_0.txt

```
ERROR: Variable already declared ( programERRO_0.txt : 2 )

    var a,b=1,a,d[10];

ERROR: Variable not defined ( programERRO_0.txt : 5 )

    c = 1;

ERROR: Array index out of bounds ( programERRO_0.txt : 6 )

    d[11] = 1;
```

5.2 Exemplo 2

```
1 program prog4;
2 var a = 1,b[10];
3 begin
4     b = a;
5     a = 2 * (3/(1+2);
6 end
```

$ ./compiler programERRO_1.txt

```
ERROR: Wrong type variable ( programERRO_1.txt : 4 )

    b = a;

ERROR:
Sintax error (maybe missing closing brackets?) ( programERRO_1.txt : 5 )

    a = 2 * (3/(1+2);
```

6 Conclusão

Entre outros o principal objetivo deste trabalho era de conhecer e criar uma linguagem gramatical, bem como a utilização de ferramentas como yacc e do flex.

O flex, dado que já tinha sido praticado no trabalho pratico anterior não causou problemas, por outro lado, o yacc revelou-se uma ferramenta bastante útil e ao mesmo tempo complexa no inicio, pois não foi simples implementar uma gramática que não apresentasse conflitos, bem como garantir que todo o código gerado estava correto.

7 Anexo

```
alphadigit              [a-zA-Z0-9_]
alpha                   [a-zA-Z]
digit                   [0-9]

%x declaracoes programName var code
%option yylineno

%{

#include <stdio.h>
#include "y.tab.h"

extern int p;

%}


%%

<*>[ \t\n]                              ;
<*>(?i:program)                         { BEGIN programName; return(PROGRAM); }
<*>[\[\]\{\}\?\!+\-*\/=]                { return (*yytext); }
<*>\(                                   { p++; return (*yytext); }
<*>\)                                   { p--; return (*yytext); }
<*>{digit}+                             { yylval.inteiro = atoi(yytext); return (num);   }

<programName>[a-zA-Z0-9]*               { return(PROGNAME); }
<programName>;.*\n                      { BEGIN declaracoes; return(';'); }

<declaracoes>"var"                      { BEGIN var; return(VAR); }
<declaracoes>(?i:begin)                 { BEGIN code; return (BBEGIN);}
<var>;                                  { BEGIN declaracoes; return(';'); }
<var>,                                  { return(','); }
<var>{alphadigit}+                      { yylval.pal = strdup(yytext); return(id); }


<code>\/\/.*                            ;
<code>;                                 { return (';'); }
<code>(?i:end)                          { return(END); }
<code>"if"                              { return (IF);    }
<code>"else"                            { return (ELSE);  }
<code>"while"                           { return (WHILE); }
<code>(==)|(>=)|(<=)|(<)|(>)            { yylval.pal = strdup(yytext); return (OPR);   }
<code>{alphadigit}+                     { yylval.pal = strdup(yytext); return (id); }
<code>\"(\\.|[^"])*\"                   { yylval.pal = strdup(yytext); return (string); }

<*>.                                    { return (ERROR); }
%%
```

```
%{

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include "stack.h"
#include "assembly.h"
#include "hash.h"


#define MAX_SIZE 100
#define INT 0
#define ARRAY 1
#define VARIAVEL 2
#define INT_SIZE 1
#define TRUE 1
#define FALSE 0
#define ERRO -1

#define ERR_SINTAX "ERROR: Sintax error"
#define ERR_NOT_DEC "ERROR: Variable not defined"
#define ERR_VAR_EXISTS "ERROR: Variable already declared"
#define ERR_INDEX_OOB "ERROR: Array index out of bounds"
#define ERR_TYPE_ERROR "ERROR: Wrong type variable"
#define ERR_INIT_ERROR "ERROR: Can't initialize arrays in declaration"
#define ERR_MISS_OP_BRACK " (maybe missing opening brackets?)"
#define ERR_MISS_CL_BRACK " (maybe missing closing brackets?)"

int yydebug=0;

// declaração de funções
extern int yylex();
extern int yylineno;
extern FILE * yyin;
int yyerror(char *);
char* getOperator(char* simb);
int isValidId(char* c);
int isValidArray(char* c, int n);
Stack Stack_Init(Stack);


//inicializações

HashTable ht = NULL;
Stack stlabel = NULL;
Assembly codeBuffer = NULL;
Assembly code = NULL;

int sp=0;
int lblCounter=1;

// flag to detect declaration block
int declBlock=1;

int addr=-1;
int tam=-1;
int tipo=-1;

// ficheiro de leitura
char* infile;
int interactive=0;

// usado para copiar string para a lista ligada
char* buffer;

// flag para determinar se houve erro
int error=0;;

// contador de parenteses abertos
int p=0;

// modo verbose
int verbose=0;
// flag com ultimo comando gerado
int cType;
```

```
%}

%union{
        char* pal;
        int inteiro;
        struct  SFactor
        {
          int type;
          int valueI;
          char *valueS;
        }factor;
}
%token <inteiro> num
%token <pal> id string OPR
%token ERROR PROGRAM PROGNAME VAR BBEGIN END IF ELSE WHILE
%type <factor> Termo
%type <factor> Factor
%type <factor> ExpSimples

%start Program


%%

Program         : PROGRAM PROGNAME ';' Corpo    { insertStop(&code); if(verbose)
                                                  printTabelaVariaveis(&ht);
                                                  if(!error) printAssemblyToScreen(code) ; return(0); }
                ;
Corpo           : Declaracoes BBEGIN            { insertStart(&codeBuffer);
                                                  assemblyCat(&code , codeBuffer); codeBuffer=NULL; }
                  Codigo END
                ;

Declaracoes     :                               { declBlock=0; }
                | VAR VarIds ';'                { declBlock=0; }
                ;

VarIds          : Var
                | VarIds ',' Var
                ;

Var             : id                            { if (declBlock) { if(getEndereco(ht,$1)!=-1)
                                                    yyerror(ERR_VAR_EXISTS);
                                                  else { insereVariavel(&ht,$1,sp,INT,INT_SIZE);
                                                    sp+=INT_SIZE; insertPushN(&codeBuffer, 1); }}
                                                }

                | id'['num']'                   { if (declBlock) { if(getEndereco(ht,$1)!=-1)
                                                    yyerror(ERR_VAR_EXISTS);
                                                  else { insereVariavel(&ht,$1,sp,ARRAY, $3); sp+=$3;
                                                    insertPushN(&codeBuffer, $3); }}
                                                }
                | Atribuicao

Codigo          :
                | Codigo Instrucao              {  if(codeBuffer!=NULL) assemblyCat(&code , codeBuffer);
                                                 codeBuffer=NULL; }
                ;

Instrucao       : Atribuicao ';'
                | Input ';'
                | Output ';'
                | Condicao
                | Ciclo
                ;

Atribuicao      : id '=' ExpSimples             { if(declBlock) { insereVariavel(&ht,$1,sp,INT,INT_SIZE);
                                                  sp+=INT_SIZE; }
                                                  if(isValidId($1)) { insertStoreG(&codeBuffer, addr); }
                                                }

                | id'['num']' '=' ExpSimples    { if(declBlock) { yyerror(ERR_INIT_ERROR); }
                                                  if(isValidArray($1,$3)) {
                                                    insertStoreG(&codeBuffer, (getEndereco(ht,$1)+$3)); }
                                                }
```

```
    ;

Input       : '?' id                    { if(isValidId($2)) { insertRead(&codeBuffer);
                                            insertStoreG(&codeBuffer, getEndereco(ht,$2)); }}
            | '?' id'['num']'           { if(isValidArray($2,$4)) { insertRead(&codeBuffer);
                                            insertStoreG(&codeBuffer, (getEndereco(ht,$2)+$4)); }}
            ;

Output      : '!' string                { insertPushS(&codeBuffer,$2);insertWriteS(&codeBuffer);}
            | '!' id                    { if(isValidId($2)) {
                                          insertPushG(&codeBuffer, getEndereco(ht,$2));
                                          insertWriteI(&codeBuffer); }}
            | '!' id'['num']'           { if(isValidArray($2,$4)) {
                                          insertPushG(&codeBuffer, getEndereco(ht,$2)+$4);
                                          insertWriteI(&codeBuffer); }}
            ;

Condicao    : IF '(' Expressao ')'      { Stack_Push(stlabel,lblCounter++);
                                          insertJZ(&codeBuffer, Stack_Top(stlabel)); }
              '{' Codigo '} ElseIf
            ;

ElseIf      :                           { insertLabel(&codeBuffer, Stack_Pop(stlabel)); }
            | ELSE                      { Stack_Push(stlabel,lblCounter++);
                                          insertJump(&codeBuffer, Stack_Top(stlabel));
                                          int previous = Stack_Pop(stlabel);
                                          insertLabel(&codeBuffer, Stack_Pop(stlabel));
                                          Stack_Push(stlabel,previous); }
              '{' Codigo '}'            { insertLabel(&codeBuffer, Stack_Pop(stlabel));  }
            ;

Ciclo       : WHILE                     { Stack_Push(stlabel,lblCounter++);
                                          insertLabel(&codeBuffer, Stack_Top(stlabel));  }
              '(' Expressao ')'         { Stack_Push(stlabel,lblCounter++);
                                          insertJZ(&codeBuffer, Stack_Top(stlabel)); }
              '{' Codigo '}'            { int previous = Stack_Pop(stlabel);
                                          insertJump(&codeBuffer,  Stack_Pop(stlabel));
                                          Stack_Push(stlabel,previous);
                                          insertLabel(&codeBuffer, Stack_Pop(stlabel));
                                        }
            ;

Expressao   : ExpSimples
            | Expressao OPR ExpSimples  { insertCode(&codeBuffer, getOperator($2)); }
            ;

ExpSimples  : Termo
            | ExpSimples '+' Termo      { insertAdd(&codeBuffer); }
            | ExpSimples '-' Termo      { insertSub(&codeBuffer); }
            ;


Termo       : Factor
            | Termo '*' Factor          {  insertMul(&codeBuffer);      }
            | Termo '/' Factor          {  insertDiv(&codeBuffer);      }
            ;


Factor      : num                       { $$.valueI = $1; $$.type=INT;
                                          insertPushI(&codeBuffer, $1); }
            | id                        { if(isValidId($1)) { $$.valueS = $1; $$.type=VARIAVEL;
                                            insertPushG(&codeBuffer, getEndereco(ht, $1));}}
            | id '['num']'              { if(isValidArray($1,$3)) { $$.valueS = $1;
                                            $$.valueI = $3;
                                            $$.type=ARRAY;
                                            insertPushG(&codeBuffer, (getEndereco(ht, $1)+$3));}
                                        }
            | '(' ExpSimples ')'        { if($2.type==INT) { $$.type=INT; $$.valueI = $2.valueI;}
                                          else if($2.type==ARRAY) { $$.type=ARRAY;
                                            $$.valueI = $2.valueI; $$.valueS = $2.valueS; }
                                          else if($2.type==VARIAVEL) { $$.type=VARIAVEL;
                                            $$.valueS = $2.valueS; }
                                        }
            ;
```

```
%%

int isValidId(char* c) {
        addr =  getEndereco(ht,c); tipo = getTipo(ht,c);
        if(addr==-1) { yyerror(ERR_NOT_DEC); return 0; }
        else if(tipo!=INT) { yyerror(ERR_TYPE_ERROR); return 0; }
        else return 1;
}
int isValidArray(char* c, int n) {
        addr =  getEndereco(ht,c); tipo = getTipo(ht,c); tam = getTamanho(ht,c);
        if(addr==-1) { yyerror(ERR_NOT_DEC); return 0; }
        else if(tipo!=ARRAY) { yyerror(ERR_TYPE_ERROR); return 0; }
        else if( n >(tam-1)) { yyerror(ERR_INDEX_OOB); return 0; }
        else return 1;
}

char* getOperator(char* simb) {
        if(strcmp(simb,"==")==0) return "EQUAL\n";
        if(strcmp(simb,">=")==0) return "SUPEQ\n";
        if(strcmp(simb,"<=")==0) return "INFEQ\n";
        if(strcmp(simb,"<")==0) return "INF\n";
        if(strcmp(simb,">")==0) return "SUP\n";
        return "";
}


int yyerror(char *msg) {

    char * message = strdup("");

    if(strcmp("syntax error",msg)==0) {
            msg = strdup(ERR_SINTAX);
            if(p!=0) {
                    if(p>0) { message = strdup(ERR_MISS_CL_BRACK); }
                    else { message = strdup(ERR_MISS_OP_BRACK); }
            }
    }

    printf("%s%s",msg,message);
    if(!interactive) {
            error=1;
            printf(" ( %s : %d )\n\n", infile, yylineno);
            char command[50];
            sprintf(command, "awk 'NR==%d' %s", yylineno,infile);

            system(command);
    }
    printf("\n");

    codeBuffer=NULL;

    return 0;
}

int main( int argc, char **argv )
{
    buffer = malloc(sizeof(char)*MAX_SIZE);
    stlabel = Stack_Init(stlabel);

    ++argv, --argc;

    if ( argc > 0 ) {
            if( strcmp(argv[0],"-v")==0) {
                    verbose=1;
                    infile=strdup(argv[1]);
            } else {
                    infile=strdup(argv[0]);
            }
            yyin = fopen( infile, "r" );
            if(yyin==0) { printf("%s - File not found, entering interactive mode!\n", infile);
            interactive=1; }
            else interactive=0;
    }
    else {
       yyin = stdin;
```

```
        interactive=1;
      }
      yyparse();
      fclose(yyin);
      return 0;
}
```

```
#ifndef _ASSEMBLY_
#define _ASSEMBLY_

typedef struct assembly *Assembly;

void insertCode(Assembly *l, char* c);
int lenghtAssembly(Assembly m);
void printAssemblyToFile(Assembly l,char* filename);
void printAssemblyToScreen(Assembly l);
void assemblyCat(Assembly *l1, Assembly l2);

void insertStart(Assembly *l);
void insertStop(Assembly *l);
void insertRead(Assembly *l);
void insertWriteI(Assembly *l);
void insertWriteS(Assembly *l);
void insertPushI(Assembly *l, int i);
void insertPushN(Assembly *l, int i);
void insertPushG(Assembly *l, int i);
void insertPushS(Assembly *l, char* s);
void insertStoreG(Assembly *l, int i);
void insertJump(Assembly *l, int i);
void insertJZ(Assembly *l, int i);
void insertLabel(Assembly *l, int i);
void insertAdd(Assembly *l);
void insertSub(Assembly *l);
void insertMul(Assembly *l);
void insertDiv(Assembly *l);

#endif
```

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "assembly.h"

#define MAX_DIGITS 10
#define LABEL 0
#define OTHER 1
#define LABEL_PREFIX "L000"

extern int cType;

struct assembly{
  char *code;
  struct assembly *next;
};

/*---------API-----------*/

void insertCode(Assembly *l, char* c){
        Assembly noActual = *l;

        Assembly no = (Assembly)malloc(sizeof(struct assembly));
        no->code = strdup(c);
        no->next = NULL;

        if(noActual == NULL) *l = no;
        else{
                while(noActual->next != NULL)
                        noActual = noActual->next;
                noActual->next = no;
        }
}

void assemblyCat(Assembly *l1, Assembly l2){
        Assembly noActual = *l1;
        if(noActual == NULL) *l1 = l2;
        else{
                while(noActual->next != NULL)
                        noActual = noActual->next;
                noActual->next = l2;
        }
}

int lenghtAssembly(Assembly m){
  Assembly aux = m;
  int t= 0;
  while(aux != NULL){
    t++;
    aux = aux->next;
  }
  return t;
}

void printAssemblyToFile(Assembly l,char* filename){
        FILE* f;
        f = fopen(filename, "w+");
        Assembly aux = NULL;
        aux = l;
        while(aux!=NULL){
                fprintf(f, "%s",aux->code);
                aux = aux->next;
        }
}

void printAssemblyToScreen(Assembly l){
        Assembly aux = NULL;
        aux = l;
        while(aux!=NULL){
                printf("%s",aux->code);
                aux = aux->next;
        }
}

/*-----------INSERTS-----------*/
```

```
void insertStart(Assembly *l) {
	insertCode(l,"START\n");
	cType=OTHER;
}
void insertStop(Assembly *l) {
	insertCode(l,"STOP\n");
	cType=OTHER;
}
void insertRead(Assembly *l) {
	insertCode(l,"READ\n");
	cType=OTHER;
}
void insertWriteI(Assembly *l) {
	insertCode(l,"WRITEI\n");
	cType=OTHER;
}
void insertWriteS(Assembly *l) {
	insertCode(l,"WRITES\n");
	cType=OTHER;
}
void insertAdd(Assembly *l) {
	insertCode(l,"ADD\n");
	cType=OTHER;
}
void insertSub(Assembly *l) {
	insertCode(l,"SUB\n");
	cType=OTHER;
}
void insertMul(Assembly *l) {
	insertCode(l,"MUL\n");
	cType=OTHER;
}
void insertDiv(Assembly *l) {
	insertCode(l,"DIV\n");
	cType=OTHER;
}
void insertPushI(Assembly *l, int i) {
	char* command = strdup("PUSHI");

	char* buffer = malloc(sizeof(char)*(strlen(command)+MAX_DIGITS));
	sprintf(buffer, "%s %d\n",command,i);
	insertCode(l,buffer);
	cType=OTHER;
}
void insertPushN(Assembly *l, int i) {
	char* command = strdup("PUSHN");

	char* buffer = malloc(sizeof(char)*(strlen(command)+MAX_DIGITS));
	sprintf(buffer, "%s %d\n",command,i);
	insertCode(l,buffer);
	cType=OTHER;
}
void insertPushG(Assembly *l, int i) {
	char* command = strdup("PUSHG");

	char* buffer = malloc(sizeof(char)*(strlen(command)+MAX_DIGITS));
	sprintf(buffer, "%s %d\n",command,i);
	insertCode(l,buffer);
	cType=OTHER;
}
void insertPushS(Assembly *l, char* s) {
	char* command = strdup("PUSHS");

	char* buffer = malloc(sizeof(char)*(strlen(command)+strlen(s)+3));
	sprintf(buffer, "%s %s\n",command,s);
	insertCode(l,buffer);
	cType=OTHER;
}
void insertStoreG(Assembly *l, int i) {
	char* command = strdup("STOREG");

	char* buffer = malloc(sizeof(char)*(strlen(command)+MAX_DIGITS));
	sprintf(buffer, "%s %d\n",command,i);
	insertCode(l,buffer);
	cType=OTHER;
}
```

```
void insertJump(Assembly *l, int i) {
	char* command = strdup("JUMP");

	char* buffer = malloc(sizeof(char)*(strlen(command)+strlen(LABEL_PREFIX)+MAX_DIGITS));
	sprintf(buffer, "%s %s%d\n",command,LABEL_PREFIX,i);
	insertCode(l,buffer);
	cType=OTHER;
}
void insertJZ(Assembly *l, int i) {
	char* command = strdup("JZ");

	char* buffer = malloc(sizeof(char)*(strlen(command)+MAX_DIGITS));
	sprintf(buffer, "%s %s%d\n",command,LABEL_PREFIX,i);
	insertCode(l,buffer);
	cType=OTHER;
}
void insertLabel(Assembly *l, int i) {
	if(cType==LABEL) {
		insertCode(l,"NOP\n"); cType=OTHER;
	}

	char* command = strdup(LABEL_PREFIX);

	char* buffer = malloc(sizeof(char)*(strlen(command)+MAX_DIGITS));
	sprintf(buffer, "%s%d:\n",command,i);
	insertCode(l,buffer);
	cType=LABEL;
}
```

```c
#ifndef HASH_H
#define HASH_H

#include "uthash.h"

#define MAX_KEY 64

typedef struct svariable {
  int tipo;
  int addr;
  int cat;
  int tam;
} variable, *Variable;

typedef struct entry {
  char key[MAX_KEY];
  Variable var;
  UT_hash_handle hh;
} *HashTable;



Variable hash_get(HashTable hash, char *key);
void insereVariavel(HashTable *hash, char *key, int addr, int tipo, int tam);
int hash_size(HashTable hash);
void hash_clear(HashTable *h);
int getEndereco(HashTable hash, char *key);
int getTipo(HashTable hash, char *key);
int getCategoria(HashTable hash, char *key);
int getTamanho(HashTable hash, char *key);
void hash_print(HashTable *h);
void printTabelaVariaveis(HashTable *h);

#endif
```

```c
#include "hash.h"
#include <stdio.h>
#include <stdlib.h>

#define DEFAULT_CAT 0

void insereVariavel(HashTable *hash, char *key, int addr, int tipo, int tam) {
  struct entry *temp = malloc(sizeof(struct entry)), *old,*temp2;
  Variable v = malloc(sizeof(variable));
  v->tipo=tipo;
  v->addr=addr;
  v->cat=DEFAULT_CAT;
  v->tam=tam;
  strcpy(temp->key, key);
  temp->var = v;
  HASH_REPLACE_STR(*hash, key, temp, old);
  if (old) free(old);
}

int getEndereco(HashTable hash, char *key) {
  Variable v = hash_get(hash, key);
  if(v==NULL) return -1;
  else return v->addr;
}
int getTipo(HashTable hash, char *key) {
  Variable v = hash_get(hash, key);
  if(v==NULL) return -1;
  else return v->tipo;
}
int getCategoria(HashTable hash, char *key) {
  Variable v = hash_get(hash, key);
  if(v==NULL) return -1;
  else return v->cat;
}
int getTamanho(HashTable hash, char *key) {
  Variable v = hash_get(hash, key);
  if(v==NULL) return -1;
  else return v->tam;
}

Variable hash_get(HashTable hash, char *key) {
  struct entry *temp;

  HASH_FIND_STR(hash, key, temp);

  if (temp == NULL) return NULL;
  else return temp->var;
}

int hash_size(HashTable hash) {
  return HASH_COUNT(hash);
}

void hash_clear(HashTable *h) {
    struct entry *entry,*temp;
    HASH_ITER(hh,(*h),entry,temp){
        HASH_DEL((*h),entry);
        free(entry->var);
        free(entry);
    }
}
void hash_print(HashTable *h) {
  struct entry *entry,*temp;
  HASH_ITER(hh,(*h),entry,temp){
    Variable v = entry->var;
    printf("|     %s          %d           %d          %d             %d        |\n",
            entry->key, v->addr, v->tipo, v->cat,  v->tam);
  }
}
void printTabelaVariaveis(HashTable *h) {
    if(h){
      printf("\n\t\tTabela de identificadores\n\n");
      printf(" - - Nome - - Endereco - - Tipo - - Categoria - - Tamanho - - \n");
      hash_print(h);
      printf(" - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -\n");
      printf("\n\n");
```

```
    }
}
```

```c
#ifndef _STACK_
#define _STACK_

#define STACK_MAX 1000


typedef struct sStack *Stack;

Stack Stack_Init(Stack);
int Stack_Top(Stack);
void Stack_Push(Stack, int );
int Stack_Pop(Stack);

#endif
```

```c
#include <stdio.h>
#include <stdlib.h>
#include "stack.h"

struct sStack {
    int     data[STACK_MAX];
    int     size;
};

Stack Stack_Init(Stack S)
{
    if (S==NULL) {
        S = (Stack) malloc(sizeof(struct sStack));
    }
    S->size = 0;
    return S;
}
int Stack_Top(Stack S)
{
    if (S->size == 0) {
        fprintf(stderr, "Error: stack empty\n");
        return -1;
    }

    return S->data[S->size-1];
}
void Stack_Push(Stack S, int d)
{
    if (S->size < STACK_MAX)
        S->data[S->size++] = d;
    else
        fprintf(stderr, "Error: stack full\n");
}
int Stack_Pop(Stack S)
{
    int ret;
    if (S->size == 0) {
        fprintf(stderr, "Error: stack empty\n");
        ret=-1;
    }
    else {
        S->size--;
        return S->data[S->size];
    }
}
```