Processamento de Linguagens (3º ano do MiEI)

Trabalho Prático 2

Compilador usando Flex e Yacc - Linguagem Tuga

Gustavo Andrez (A27748)

Rogério Moreira (A74634)

Samuel Ferreira (A76507)

11 de Junho de 2017

Resumo

O presente trabalho tem como objetivo aumentar a experiência em engenharia de linguagens e em programação generativa (gramatical), com o propósito de desenvolver processadores de linguagens segundo o método da tradução dirigida pela sintaxe, a partir de uma gramática tradutora, para uma máquina de stack virtual. A ferramenta usada foi o Yacc. Todos os objetivos inicialmente propostos foram cumpridos.

Conteúdo

Capítulo 1

Introdução

Introdução

O presente relatório tem como objetivo documentar o processo de desenvolvimento de uma linguagem de programação imperativa simples, e o respetivo compilador, que deverá ser capaz de gerar pseudo-código Assembly para máquina virtual VM, utilizada neste projeto. Para este fim, é necessário criar uma gramática independente de contexto que defina a linguagem, e estabelecer as regras de tradução para o Assembly da VM fornecido pela equipa docente. O grupo decidiu criar uma linguagem em que todas as instruções são descritas em português de Portugal, apelidada de *Tuga*.

Enunciado

Pretende-se que se comece por definir uma linguagem de programação imperativa simples. Deve-se ter em consideração que essa linguagem terá de permitir:

- declarar e manusear variáveis atómicas do tipo inteiro, com os quais se podem realizar as habituais operações aritméticas, relacionais e lógicas;
- declarar e manusear variáveis estruturadas do tipo array (a 1 ou 2 dimensões) de inteiros, em relação aos quais é apenas permitida a operação de indexação (índice inteiro);
- efetuar instruções algorítmicas básicas como a atribuição de expressões a variáveis;
- ler do standard input e escrever no standard output.
- efetuar instruções para controlo do fluxo de execução — condicional e cíclica — que possam ser aninhadas;
- as variáveis deverão ser declaradas no início do programa e não pode haver re-declarações, nem utilizações sem declaração prévia. Se nada for explicitado, o valor da variável após a declaração é 0 (zero);

Pretende-se então desenvolver um compilador para a linguagem definida com base na GIC criada acima e com recurso ao Gerador Yacc/Flex. O compilador deve gerar pseudo-código Assembly da Máquina Virtual VM.

Descrição do problema

O problema proposto no segundo trabalho prático da disciplina de Processamento de Linguagens compreende a realização de uma linguagem LIPS, com uma gramática própria e capaz de ser executada numa máquina virtual com Assembly próprio previamente fornecida pela equipa docente. Posto isto, o problema tem três fases principais:

1. Desenho de uma gramática tradutora (Yacc);
2. Tradução das instruções da gramática para o Assembly da máquina virtual (Flex);
3. Elaboração de programas de teste na linguagem elaborada;

Estrutura do Relatório

O relatório está dividido em três capítulos, correspondentes ao desenho da linguagem e gramática, às regras de tradução criadas, e ao resultado dos testes efetuados com programas exemplo, por esta ordem. No primeiro capítulo, *Desenho da linguagem/gramática*, são expostos tanto os requisitos do problema apresentado, e discutidas as estratégias utilizadas para a consequente implementação da solução do problema. O capítulo dois, *Regras da tradução para Assembly da VM*, apresenta para além das operações de tradução da linguagem para o Assembly próprio da VM, são também descritas a especificação das estruturas de dados e outras notas importantes. Por fim, o capítulo *Testes e Resultados* expõe o resultado dos testes requeridos à demonstração do funcionamento da linguagem e compilador desenvolvidos durante o projeto. Para isto foram criados alguns programas testes como pedido no enunciado do trabalho prático.

Capítulo 2

Desenho da linguagem/gramática

2.1 Símbolos terminais e não terminais

```
T={INT , VAR , OPINC , OPDEC ,
OPATRMAIS , OPATRMENOS ,
OPATRPROD , OPATRDIV , OPATRMOD , OPAND ,
OPOR , OPGT , OPLT , OPGE , OPLE , OPEQ , OPNE ,
OPARITMAIS , OPARITMENOS , OPARITPROD ,
OPARITDIV , OPARITMOD , STRING , SCAN ,
PRINT , IF , ELSE , NOT , ERRO , WHILE ,
( , ) , [ , ] , { , } , = , " , ; , & }

 OPINC ::- \+\+
 OPDEC ::- \-\-

 OPATRMAIS ::- \+\=
 OPATRMENOS ::- \-\=
 OPATRPROD ::- \*\=
 OPATRDIV ::- \/\=
 OPATRMOD ::- \%\=

 SCAN ::- (?i:ler)

 PRINT ::- (?i:escrever)
 STRING ::- \"([^"]|\\\")*\"

 IF ::- (?i:se)
 ELSE ::- (?i:senao)
 WHILE ::- (?i:enquanto)

 OPGE ::- \>\=
 OPLE ::- \<\=
 OPEQ ::- \=\=
 OPNE ::- \!\=
 NOT ::- \!
 OPGT ::- \>
 OPLT ::- \<
```

```
OPAND ::- \&\&
OPOR ::- \|\|

OPARITMENOS ::- \-
OPARITMAIS ::- \+
OPARITPROD ::- \*
OPARITDIV ::- \/
OPARITMOD ::- \%

INT ::- [0-9]+
VAR ::- [a-z_][a-zA-Z0-9_]*

ERRO ::- .
```

O conjunto dos símbolos não terminais da gramática é o que se segue:

```
NT={ Programa , Declaracoes , Instrucoes ,
Declaracao , Instrucao , Atribuicao ,
Escrita , Leitura , Condicional ,
Ciclo , Exp }
```

2.2 Gramática

Na linguagem desenhada os programas seguem a seguinte estrutura: nas primeiras linhas são declaradas todas as variáveis (inteiros ou arrays de inteiros) que irão ser necessárias durante o programa, depois aparece o símbolo '' e por fim as instruções do programa. (Programa : Declaracoes '&' Instrucoes).
A gramática elaborada é a seguinte:

```
//--- CABECA ---//

Declaracoes : Declaracoes Declaracao
     |
    ;

Declaracao : VAR ';'
    | VAR '[' INT ']' ';'
    ;

//--- CORPO ---//
Instrucoes : Instrucoes Instrucao
    | Instrucao
    ;

Instrucao : Atribuicao
       | Escrita
       | Leitura
       | Condicional
       | Ciclo
       ;

//--- ATRIBUICAO ---//
Atribuicao : VAR '=' Exp ';'
    | VAR '[' Exp ']' '=' Exp ';
    | VAR OPINC ';'
```

```
    | VAR OPDEC ';'
    | VAR OPATRMAIS Exp ';'
    | VAR OPATRMENOS Exp ';'
    | VAR OPATRPROD Exp ';'
   | VAR OPATRDIV Exp ';'
   | VAR OPATRMOD Exp ';'
    ;

//--- LEITURA ---//
Leitura : SCAN VAR ';'
     | SCAN VAR '[' Exp ']' ';'
     ;

//--- ESCRITA ---//
Escrita : PRINT Exp ';'
     | PRINT STRING ';'
     ;

//--- CONDICIONAL IF ---//
Condicional : IF '(' Exp ')' '{' Instrucoes '}' Else
     ;

Else : ELSE '{' Instrucoes '}'
     |
     ;

//--- CICLO WHILE ---//
Ciclo : WHILE '(' Exp ')' '{' Instrucoes '}'
      ;

//--- EXPRESSAO ---//
Exp : NOT Exp
    | Exp OPLT Ex
    | Exp OPGT Exp
    | Exp OPLE Exp
    | Exp OPGE Exp
    | Exp OPEQ Exp
    | Exp OPNE Exp
    | Exp OPAND Exp
    | Exp OPOR Exp
    | Exp OPARITMAIS Exp
    | Exp OPARITMENOS Exp
    | Exp OPARITPROD Exp
    | Exp OPARITDIV Exp
    | Exp OPARITMOD Exp
    | VAR
    | VAR '[' Exp ']'
    | INT
    | '(' Exp ')'
    ;
```

Capítulo 3

Regras da tradução para Assembly da VM

Em seguida é apresentado um pseudocódigo que descreve o processo de conversão da linguagem desenvolvida (Tuga) para linguagem Assembly da VM.

```
Programa : Declaracoes { adicionar instrução "start" à pilha de instruções }
     '&' Instrucoes { adicionar instrução "stop" à pilha de instruções }
        ;

Declaracoes : Declaracoes Declaracao
     |
    ;

Declaracao : VAR ';' {
     se (o nome de variável em $1 já foi atribuído)
     erro
     senao
     registar variável na estrutura de dados (nome, tipo, endereço)
     adicionar instrução "pushi 0" à pilha de instruções
     gp++
     }
    | VAR '[' INT ']' ';' {
         se (o nome de variável em $1 já foi atribuído)
     erro
     senao
     registar variável na estrutura de dados (nome, tipo, endereço)
     adicionar instrução "pushn X" à pilha de instruções, em que X=$3
     gp+=$3
         }
    ;

Instrucoes : Instrucoes Instrucao
    | Instrucao
    ;

Instrucao : Atribuicao
       | Escrita
       | Leitura
       | Condicional
       | Ciclo
       ;
```

```
Atribuicao : VAR '=' Exp ';' {
     se(não existe a variável em $1 || a variável é do tipo inteiro)
     erro
     senao
     adicionar a instrução "storeg X" à pilha de instruções, em que X=endereço($1)
     }
    | VAR {
        se(não existe a variável em $1 || a variavel é do tipo array)
     erro
     senao
     adicionar a instrução "pushgp" à pilha de instruções
     adicionar a instrução "pushi X" à pilha de instruções, em que X=endereço($1)
     adicionar a instrução "padd" à pilha de instruções
     }
     '[' Exp ']' '=' Exp ';'{
     adicionar a instrução "storen" à pilha de instruções
     }

    | VAR OPINC ';'{
        se(não existe a variável em $1 || a variavel é do tipo array)
     erro
     senao
     // incrementar a variável $1
     adicionar a instrução "pushg X" à pilha de instruções, em que X=endereço($1)
     adicionar a instrução "puhi 1" à pilha de instruções
     adicionar a instrução "add" à pilha de instruções
     adicionar a instrução "storeg X" à pilha de instruções, em que X=endereço($1)
         }
    | VAR OPDEC ';'{ processo análogo ao anterior, mas decrementa em vez de incrementar    }
    (1)->  | VAR {
        se(não existe a variável em $1 || a variavel é do tipo array)
     erro
     senao
     adicionar a instrução "pushg X" à pilha de instruções, em que X=endereço($1)
     }
     OPATRMAIS Exp ';'{
     adicionar a instrução "add" à pilha de instruções
     adicionar a instrução "storeg X" à pilha de instruções, em que X=endereço($1)
         }
   | VAR OPATRMENOS Exp ';'{ análogo ao processo em (1) }
   | VAR OPATRPROD Exp ';'{ análogo ao processo em (1) }
   | VAR OPATRDIV Exp ';'{ análogo ao processo em (1) }
   | VAR OPATRMOD Exp ';'{ análogo ao processo em (1) }
    ;

Leitura : SCAN VAR ';' {
     se(não existe a variável em $1 || a variavel é do tipo array)
     erro
     senao
     adicionar a instrução "read" à pilha de instruções
     adicionar a instrução "atoi" à pilha de instruções
     adicionar a instrução "storeg X" à pilha de instruções, em que X=endereço($2)
     }
```

```
     | SCAN VAR {
     se(não existe a variável em $1 || a variável é do tipo inteiro)
     erro
     senao
     adicionar a instrução "pushgp" à pilha de instruções
     adicionar a instrução "pushi X" à pilha de instruções, em que X=endereço($2)
     adicionar a instrução "padd" à pilha de instruções
     }
     '[' Exp ']' ';'    {
     adicionar a instrução "read" à pilha de instruções
     adicionar a instrução "atoi" à pilha de instruções
     adicionar a instrução "storen" à pilha de instruções
     }
     ;

Escrita : PRINT Exp ';' { adicionar a instrução "writei" à pilha de instruções }
     | PRINT STRING ';'{
     adicionar a instrução "pushs X" à pilha de instruções, em que X= endereço($2)
     adicionar a instrução "writes" à pilha de instruções
     }
     ;

Condicional : IF '(' Exp ')' {
     adicionar endereço da instrução à stack de se's
     adicionar a instrução "jz ifX" à pilha de instruções, em que X=ifs
     }
      '{' Instrucoes '}' { fazer pop da stack de se's }
       Else
     ;

Else : ELSE {
     adicionar endereço da instrução à stack de se's
     adicionar a instrução "jump elseX" à pilha de instruções, em que X=elses
     adicionar a label "ifX:" à pilha de instruções, em que X= ifs
     incrementar ifs
     }
     '{' Instrucoes '}' {
     fazer pop da stack de se's
     adicionar a label "elseX:" à pilha de instruções, em que X=elses
     incrementar variável elses
     }
     |  {    adicionar a label "ifX:" à pilha de instruções, em que X=ifs
     incrementar variável ifs }
     ;

Ciclo : WHILE '(' {
     $1=whiles;
     incrementar a variável ""whiles" em 2 unidades
     adicionar a label "whileX :" à pilha de instruções, em que X=$1
 }
      Exp ')' {
     adicionar a instrução "jz whileX" à pilha de instruções, em que X=$1+1
     }
      '{' Instrucoes {
```

```
            adicionar a instrução "jump whileX" à pilha de instruções, em que X=$1
            adicionar a label "whileX :" à pilha de instruções, em que X=$1+1
      }
      '}'
       ;

//--- EXPRESSAO ---//
Exp : NOT Exp { adicionar a instrução "pushi 0" à pilha de instruções
       adicionar a instrução "equal" à pilha de instruções }
     | Exp OPLT Exp { adicionar a instrução "sup" à pilha de instruções }
     | Exp OPGT Exp { adicionar a instrução "pushgp" à pilha de instruções }
     | Exp OPLE Exp { adicionar a instrução "infeq" à pilha de instruções }
     | Exp OPGE Exp { adicionar a instrução "supeq" à pilha de instruções }
     | Exp OPEQ Exp { adicionar a instrução "equal" à pilha de instruções }
     | Exp OPNE Exp { adicionar a instrução "pushi 0" à pilha de instruções
       adicionar a instrução "equal" à pilha de instruções }
     | Exp OPAND Exp { adicionar a instrução "mul" à pilha de instruções    }
     | Exp OPOR Exp { adicionar a instrução "add" à pilha de instruções }
     | Exp OPARITMAIS Exp { adicionar a instrução "add" à pilha de instruções }
     | Exp OPARITMENOS Exp { adicionar a instrução "sub" à pilha de instruções }
     | Exp OPARITPROD Exp { adicionar a instrução "mul" à pilha de instruções }
     | Exp OPARITDIV Exp { adicionar a instrução "div" à pilha de instruções }
     | Exp OPARITMOD Exp { adicionar a instrução "mod" à pilha de instruções }
     | VAR {
      se(não existe a variável em $1 || a variavel é do tipo array)
      erro
      senao
      adicionar a instrução "pushg X" à pilha de instruções, em que X=endereço($1)
      }
     | VAR {
      se(não existe a variável em $1 || a variável é do tipo inteiro)
      erro
      senao
      adicionar a instrução "pushgp" à pilha de instruções
      adicionar a instrução "pushi X" à pilha de instruções, em que X=endereço($1)
      adicionar a instrução "padd" à pilha de instruções
      }
     '[' Exp ']'{
      adicionar a instrução "loadn" à pilha de instruções
      }

     | INT { adicionar a instrução "pushs X" à pilha de instruções, em que X=$1    }
     | '(' Exp ')'
     ;
```

3.1 Main do programa

Aquando da execução do programa, depois de validar se existem argumentos, executam-se os seguintes passos:

- redirecionar o stdin do analisador sintático para um descritor do ficheiro fornecido como argumento;
- executar a função yyparse (analisador sintático). Durante o decorrer desta função, a variável listaInstrucoes vai ser preenchida para posterior apresentação;
- abrir para escrita um ficheiro com o mesmo nome do argumento fornecido, mas com a extensão .vm;

- Verificar se existiram erros de compilação e:
  - em caso afirmativo, escrever para o descritor do ficheiro .vm uma mensagem de erro;
  - em caso negativo, imprime cada uma uma das instruções na lista de instruções, separadas por um “”;

O código associado a este procedimento é apresentado em seguida.

```
int main(int argc, char *argv[]){
    if(argc < 2){
     printf("Introduzir ficheiro para compilação como argumento!\n");
     exit(0);
    }
    //--- PARSE ---//
    yyin = fopen(argv[1],"r");
    yyparse();
   //--- IMPRIMR OUTPUT ---//
   filename = strdup(argv[1]);
    filename[strrchr(filename,'.')-filename] = '\0';
    strcat(filename,".vm");
    FILE* f = fopen(filename,"w");
    if(!erros){
     while(listaInstrucoes){
     if(listaInstrucoes->instrucao){
     fprintf(f,"%s\n",listaInstrucoes->instrucao);
     }
     listaInstrucoes = listaInstrucoes->next;
     }
    }
    else {
     printf("Houve erros durante a compilação\n");
    }
    fclose(f);
    return 0;
}
```

3.2 Estruturas de Dados

Foi necessário criar estruturas de dados capazes de albergar a informação para a correta execução dos programas. As estruturas criadas foram as seguintes:

```
struct variavel
{
char *nome;
char *tipo;
int endereco;
struct variavel *next;
};
```

Estrutura que armazena toda a informação respeitante à declaração de uma variável. O `nome` é o nome da variável, o `tipo` é o tipo da variável (ex: INT), o `endereco` é o endereço onde a variável está na VM e o `next` é o apontador para a próxima variável. Trata-se de uma lista ligada de variáveis.

```
struct instrucao
{
```

```
int endereco;
char *instrucao;
struct instrucao *next;
};
```

Estrutura que armazena uma instrução. O `endereco` é um endereço no qual a instrução está localizada na VM, a `instrucao` é a instrução em causa e o `next` é, à semelhança da declaração de variáveis, o apontador para a próxima instrução. Trata-se de uma lista ligada.

```
struct ifAddr
{
int jz;
struct ifAddr *next;
};
```

Estrutura respeitante a uma condição If. A variável `jz` na estrutura armazena o salto condicional e a variável `next` armazena o endereço da próxima estrutura na lista ligada.

```
struct whileAddr
{
int jump;
int jz;
struct whileAddr *next;
};
```

Estrutura respeitante a um ciclo while. A variável `jump` armazena o salto do início do while, a variável `jz` armazena o salto condicional e a variável `next` armazena o endereço da próxima estrutura na lista ligada.

Estas são as estruturas principais, para além destas foram também criadas outras estruturas mas que fazem uso destas.

```
struct variavel* insertVariavel(struct variavel *variavel, struct variavel *listaVariaveis);
int existeVariavel(char *variavel, struct variavel *listaVariaveis);
int enderecoVariavel(char *variavel, struct variavel *listaVariaveis);
char* tipoVariavel(char *variavel, struct variavel *listaVariaveis);

struct instrucao* insertInstrucao(int endereco, char *instrucao, struct instrucao *listaInstrucoes);

struct ifAddr* pushIfAddr(int endereco, struct ifAddr *pilhaIfAddr);
struct ifAddr* popIfAddr(struct ifAddr *pilhaIfAddr);

struct whileAddr* pushWhileAddr(int endereco, int whileAddr, struct whileAddr *pilhaWhileAddr);
struct whileAddr* popWhileAddr(struct whileAddr *pilhaWhileAddr);

void ifJump(int endereco, struct ifAddr *pilhaIfAddr, struct instrucao *pilhaInstrucoes);
void elseJump(int endereco, struct ifAddr *pilhaIfAddr, struct instrucao *pilhaInstrucoes);
int whileJump(int endereco, struct whileAddr *pilhaWhileAddr, struct instrucao *pilhaInstrucoes);
```

Capítulo 4

Programas Exemplo

4.1 Ordenação descendente - Bubble Sort

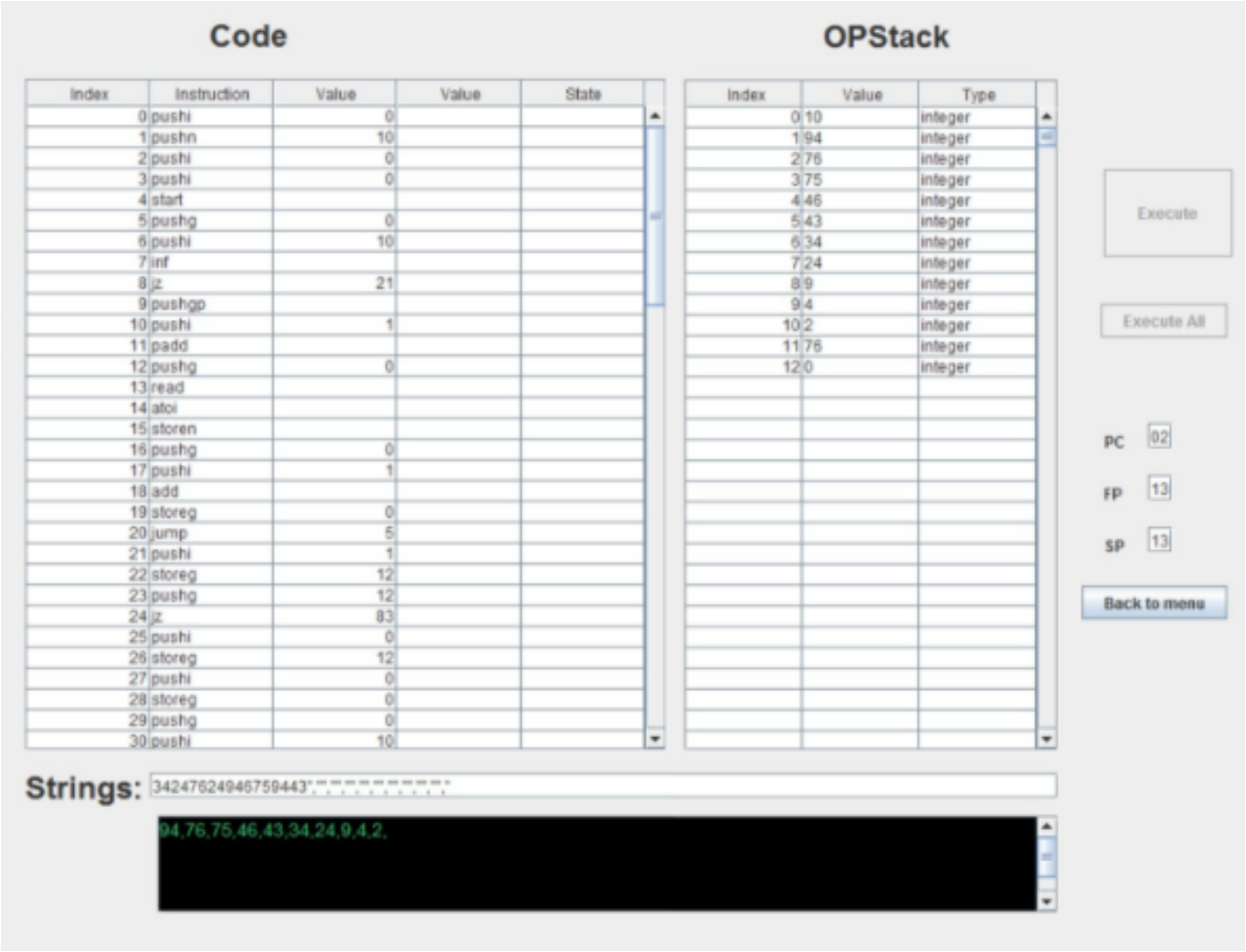
Descrição: programa que lê 10 números naturais e os escreve por ordem descendente.

Código-Fonte:

```
i;
a[10];
troca;
trocado;
&
enquanto(i<10){
    ler a[i];
    i++;
}
trocado=1;
enquanto(trocado){
    trocado=0;
    i=0;
    enquanto(i<10 -1 ){
     se(a[i]<a[i+1]){
     troca=a[i];
     a[i]=a[i+1];
     a[i+1]=troca;
     trocado=1;
     }
     i++;
    }
}
i=0;
enquanto(i<10){
    escrever a[i];
    escrever ",";
    i++;
}
```

Assembly gerado:
pushi 0 pushn 10 pushi 0 pushi 0 start while0: pushg 0 pushi 10 inf jz while1 pushgp pushi 1 padd pushg 0 read atoi storen pushg 0 pushi 1 add storeg 0 jump while0 while1: pushi 1 storeg 12 while2: pushg 12 jz while3 pushi 0 storeg

12 pushi 0 storeg 0 while4: pushg 0 pushi 10 pushi 1 sub inf jz while5 pushgp pushi 1 padd pushg 0 loadn pushgp pushi 1 padd pushg 0 pushi 1 add loadn inf jz if0 pushgp pushi 1 padd pushg 0 loadn storeg 11 pushgp pushi 1 padd pushg 0 pushgp pushi 1 padd pushg 0 pushi 1 add loadn storen pushgp pushi 1 padd pushg 0 pushi 1 add pushg 11 storen pushi 1 storeg 12 if0: pushg 0 pushi 1 add storeg 0 jump while4 while5: jump while2 while3: pushi 0 storeg 0 while6: pushg 0 pushi 10 inf jz while7 pushgp pushi 1 padd pushg 0 loadn writei pushs ","; writes pushg 0 pushi 1 add storeg 0 jump while6 while7: Stop



4.2 Séries de Fibbonacci

Descrição: programa que lê um número natural e escreve a sua sequência de fibbonacci

Código-Fonte:

```
i;
j;
k;
t;
n;
&
ler n;
j+=1;
```

```
escrever j;
k++;
enquanto(k<=n){
    t = i + j;
i = j;
j = t;
    k++;
    escrever j;
    escrever " , ";
}
```

Assembly gerado:

```
pushi 0
     pushi 0
     pushi 0
     pushi 0
     pushi 0
start
     read
     atoi
 storeg 4
     pushg 1
     pushi 1
     add
     storeg 1
     pushg 1
     writei
     pushg 2
     pushi 1
     add
     storeg 2
while0:
     pushg 2
     pushg 4
     infeq
     jz while1
     pushg 0
     pushg 1
     add
     storeg 3
     pushg 1
     storeg 0
     pushg 3
     storeg 1
     pushg 2
     pushi 1
     add
     storeg 2
     pushg 1
     writei
     pushs " , ";
     writes
     jump while0
```

```
while1:
stop
```

4.3 Números Ímpares

Descrição: programa que lê uma sequência de números terminada por 0 e imprime a contagem e os números ímpares da sequência.

Código-Fonte:

```
numero;
conta;
&
numero=1;
escrever "IMPARES: ";
enquanto(numero){
    ler numero;
    se(numero%2){
     escrever(numero);
     escrever " , ";
     conta++;
    }
}
escrever "\nCONTAGEM:";
escrever conta;
```

Assembly gerado:

```
pushi 0
     pushi 0
start
     pushi 1
     storeg 0
     pushs "IMPARES: ";
     writes
while0:
     pushg 0
     jz while1
     read
     atoi
     storeg 0
     pushg 0
     pushi 2
     mod
     jz if0
     pushg 0
     writei
     pushs " , ";
     writes
     pushg 1
     pushi 1
     add
     storeg 1
if0:
```

```
     jump while0
while1:
     pushs "\nCONTAGEM:";
     writes
     pushg 1
     writei
stop
```

4.4 Ordem Inversa

Descrição: programa que lê uma sequência de 5 numeros naturais e imprime-os na ordem inversa.

Código-Fonte:

```
n;
a[5];
i;
&
n=5;
enquanto(i<n){
    ler a[i];
    i++;
}
i=n-1;
enquanto(i>=0){
    escrever a[i];
    ESCREVER " , ";
    i--;
}
```

Assembly gerado:

```
pushi 0
     pushn 5
     pushi 0
start
     pushi 5
     storeg 0
while0:
     pushg 6
     pushg 0
     inf
     jz while1
     pushgp
     pushi 1
     padd
     pushg 6
     read
     atoi
     storen
     pushg 6
     pushi 1
     add
     storeg 6
```

```
     jump while0
while1:
     pushg 0
     pushi 1
     sub
     storeg 6
while2:
     pushg 6
     pushi 0
     supeq
     jz while3
     pushgp
     pushi 1
     padd
     pushg 6
     loadn
     writei
     pushs " , ";
     writes
     pushg 6
     pushi 1
     sub
     storeg 6
     jump while2
while3:
stop
```

4.5 Se's aninhados

Descrição: programa de teste de vários se's aninhados (com e sem senao's)

Código-Fonte:

```
a;
i;
&
se(a==0){
    escrever "SIM,";
    se(a==1){
     escrever "SIM,";
    }
    senao{
     enquanto(i<2){
     escrever "NAO,";
     se(a==1){
     escrever "SIM,";
     se(a==1){
     escrever "SIM,";
     }
     senao{
     escrever "NAO,";
     }
     }
     senao{
```

```
     escrever "NAO,";
     se(a==0){
     escrever "SIM,";
     }
     senao{
     escrever "NAO,";
     }

     }
     i+=1;
     }
    }
}
senao{
    escrever "NAO,";
    se(a==0){
     escrever "SIM,";
    }
    senao{
     escrever "NAO,";
    }

}

escrever " FIM";
```

Assembly gerado:

```
pushi 0
     pushi 0
start
     pushg 0
     pushi 0
     equal
     jz if4
     pushs "SIM,";
     writes
     pushg 0
     pushi 1
     equal
     jz if0
     pushs "SIM,";
     writes
     jump else0
if0:
while0:
     pushg 1
     pushi 2
     inf
     jz while1
     pushs "NAO,";
     writes
     pushg 0
     pushi 1
     equal
```

```
     jz if2
     pushs "SIM,";
     writes
     pushg 0
     pushi 1
     equal
     jz if1
     pushs "SIM,";
     writes
     jump else0
if1:
     pushs "NAO,";
     writes
else0:
     jump else1
if2:
     pushs "NAO,";
     writes
     pushg 0
     pushi 0
     equal
     jz if3
     pushs "SIM,";
     writes
     jump else1
if3:
     pushs "NAO,";
     writes
else1:
else2:
     pushg 1
     pushi 1
     add
     storeg 1
     jump while0
while1:
else3:
     jump else4
if4:
     pushs "NAO,";
     writes
     pushg 0
     pushi 0
     equal
     jz if5
     pushs "SIM,";
     writes
     jump else4
if5:
     pushs "NAO,";
     writes
else4:
else5:
     pushs " FIM";
```

```
     writes
stop
```

4.6 Lógica Clássica

Descrição: programa para testar operações lógicas e relacionais.

Código-Fonte:

```
i;
f;
&
enquanto(i<10){
    se(!f){
     escrever " SIM ,";
    }
    senao{
     escrever " NAO ,";
    }
    se( f==1 ||  i%2==0 && i==5  ){
     escrever ",,JACKPOT,,";
    }
    f=(f+1)%2;
    i++;
}
```

Assembly gerado:

```
pushi 0
     pushi 0
start
while0:
     pushg 0
     pushi 10
     inf
     jz while1
     pushg 1
     pushi 0
     equal
     jz if0
     pushs " SIM ,";
     writes
     jump else0
if0:
     pushs " NAO ,";
     writes
else0:
     pushg 1
     pushi 1
     equal
     pushg 0
     pushi 2
     mod
     pushi 0
```

```
     equal
     add
     pushg 0
     pushi 5
     equal
     mul
     jz if1
     pushs ",,JACKPOT,,";
     writes
if1:
     pushg 1
     pushi 1
     add
     pushi 2
     mod
     storeg 1
     pushg 0
     pushi 1
     add
     storeg 0
     jump while0
while1:
stop
```

4.7 Menor número

Descrição: programa que lê n números (n fornecido pelo utilizador) e escreve o menor deles.

Código-Fonte:

```
n;
aux;
menor;
&
ler n;
ler aux;
menor=aux;
n--;
enquanto(n){
    ler aux;
    se(aux<menor){
     menor=aux;
    }
    n--;
}
escrever menor;
```

Assembly gerado:

```
pushi 0
     pushi 0
     pushi 0
start
     read
```

```
     atoi
     storeg 0
     read
     atoi
     storeg 1
     pushg 1
     storeg 2
     pushg 0
     pushi 1
     sub
     storeg 0
while0:
     pushg 0
     jz while1
     read
     atoi
     storeg 1
     pushg 1
     pushg 2
     inf
     jz if0
     pushg 1
     storeg 2
if0:
     pushg 0
     pushi 1
     sub
     storeg 0
     jump while0
while1:
     pushg 2
     writei
stop
```

4.8 Operações Aritemáticas

Descrição: programa para testar o funcionamento das operações aritméticas e atribuições/leituras dos índices de um array.

Código-Fonte:

```
b;
a[5];
&
a[0]=7/7;
a[1]= 5%3;
a[2]= 6-3;
a[3]= 2*2;
a[4]= 10;
escrever (b+1)*100/2;
escrever a[0];
escrever a[1];
escrever a[2];
escrever a[3];
```

```
escrever (a[4]);
```

Assembly gerado:

```
pushi 0
     pushn 5
start
     pushgp
     pushi 1
     padd
     pushi 0
     pushi 7
     pushi 7
     div
     storen
     pushgp
     pushi 1
     padd
     pushi 1
     pushi 5
     pushi 3
     mod
     storen
     pushgp
     pushi 1
     padd
     pushi 2
     pushi 6
     pushi 3
     sub
     storen
     pushgp
     pushi 1
     padd
     pushi 3
     pushi 2
     pushi 2
     mul
     storen
     pushgp
     pushi 1
     padd
     pushi 4
     pushi 10
     storen
     pushg 0
     pushi 1
     add
     pushi 100
     mul
     pushi 2
     div
     writei
     pushgp
     pushi 1
```

```
     padd
     pushi 0
     loadn
     writei
     pushgp
     pushi 1
     padd
     pushi 1
     loadn
     writei
     pushgp
     pushi 1
     padd
     pushi 2
     loadn
     writei
     pushgp
     pushi 1
     padd
     pushi 3
     loadn
     writei
     pushgp
     pushi 1
     padd
     pushi 4
     loadn
     writei
stop
```

4.9 Produtório

Descrição: programa que lê 5 números naturais e imprime o seu produtório.

Código-Fonte:

```
n;
produtorio;
numero;
&
n=5;
produtorio=1;
enquanto(n){
    ler numero;
    produtorio=produtorio*numero;
    n--;
}
escrever produtorio;
```

Assembly gerado:

```
     pushi 0
     pushi 0
     pushi 0
```

```
start
     pushi 5
     storeg 0
     pushi 1
     storeg 1
while0:
     pushg 0
     jz while1
     read
     atoi
storeg 2
     pushg 1
     pushg 2
     mul
     storeg 1
     pushg 0
     pushi 1
     sub
     storeg 0
     jump while0
while1:
     pushg 1
     writei
stop
```

4.10 Quadrado

Descrição: programa que lê 4 números e verifica se podem corresponder aos lados de um quadrado.

Código-Fonte:

```
a;
b;
c;
d;
&
ler a;
ler b;
ler c;
ler d;
se(a==b && b==c && c==d){
    escrever "É um quadrado.\n";
}
senao{
    escrever "Não é um quadrado.\n";
}
```

Assembly gerado:

```
pushi 0
     pushi 0
     pushi 0
     pushi 0
start
```

```
     read
     atoi
     storeg 0
     read
     atoi
     storeg 1
     read
     atoi
     storeg 2
     read
     atoi
     storeg 3
     pushg 0
     pushg 1
     equal
     pushg 1
     pushg 2
     equal
     mul
     pushg 2
     pushg 3
     equal
     mul
     jz if0
     pushs "É um quadrado.\n";
     writes
     jump else0
if0:
     pushs "Não é um quadrado.\n";
     writes
else0:
stop
```

4.11 Números Ímpares

Descrição: programa que lê 5 números naturais e imprime o seu somatório.

Código-Fonte:

```
a;
num;
soma;
&
enquanto(a<5){
    ler num;
    soma+=num;
    a++;
}
escrever "Soma: ";
escrever soma;
```

Assembly gerado:

```
pushi 0
```

```
     pushi 0
     pushi 0
start
while0:
     pushg 0
     pushi 5
     inf
     jz while1
     read
     atoi
     storeg 1
     pushg 2
     pushg 1
     add
     storeg 2
     pushg 0
     pushi 1
     add
     storeg 0
     jump while0
while1:
     pushs "Soma: ";
     writes
     pushg 2
     writei
stop
```

Capítulo 5

Conclusão

No geral, os objetivos esperados para o programa foram alcançados e obtivemos uma solução sólida para o problema apresentado, cumprindo o definido inicialmente. Destacamos ainda alguns pontos:

- a GIC desenvolvida é de simples utilização, intuitiva e bastante completa;
- foram implementadas várias formas de manuseamento de variáveis do tipo inteiro tradicionais: i++, i+=int, arr[(i-int)*int], entre outras.

Como trabalho futuro sugerimos:

- implementação de manipulação de arrays em 2 dimensões;
- definição e invocação de subprogramas sem parâmetros que possam retornar um resultado atómico.