# Laboratórios de Informática III

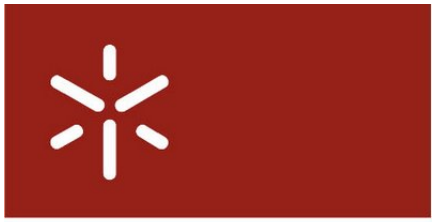
# Relatório de Desenvolvimento

André Gonçalves (A75625)

José Silva (A74576)

Ricardo Certo (A75315)

19 de Junho de 2017



Universidade do Minho

**Resumo**

O problema proposto consiste na criação de uma linguagem imperativa simples, a respetiva formulação da sua GIC e desenvolvimento de um compilador que gere código de *"baixo nível"* para uma máquina virtual.

# Conteúdo

# 1 | Introdução

A realização deste segundo trabalho prático da unidade curricular de Processamento de Linguagens consiste na implementação de uma ***Linguagem Imperativa Simples***, através do desenvolvimento de um compilador que gera código para uma máquina de stack virtual. Para isso utilizamos os conteúdos lecionados nas aulas, funcionando este último trabalho prático como uma junção de conhecimentos adquiridos tais como o uso das ferramentas Yacc e Flex, assim como a consolidação de conhecimentos acerca de gramáticas, expressões regulares, analisadores léxicos e sintáticos.

Ao longo deste relatório vamos apresentar todas as etapas e decisões tomadas durante todo o processo de elaboração da linguagem que desenvolvemos, iniciando com uma breve descrição do problema e enunciado, seguida da formulação da solução que levou à nossa linguagem apresentando exemplos da sua utilização e na parte dos anexos, será colocado o código fonte dos exercícios desenvolvidos.

# 2 | Análise e Especificação

## 2.1 Enunciado

Pretende-se que comece por definir uma linguagem de programação imperativa simples, a seu gosto. Apenas deve 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 de fluxo de execução-condicional e cíclica- que possam ser aninhadas.
- definir e invocar subprogramas sem parâmetros mas que possam retornar um resultado atómico (opcional).

As variáveis deverão ser declaradas no inicio do programa, não podendo haver re-declarações nem utilizações sem declaração prévia. Se nada for explicitado a variável terá o valor 0.

Desenvolva, então, um compilador para essa linguagem com base na GIC criada acima e com recurso ao Gerador Yacc/Flex. O compilador deve gerar pseudo-código, Assembly da Máquina Virtual VM cuja documentação completa está disponível no Bb.

## 2.2 Descrição do Problema

Como descrito no enunciado referido acima havia o desafio de desenvolver uma linguagem simples, capaz de desempenhar uma lista de funcionalidades consideradas básicas para o conceito existente de linguagem imperativa como o C. Após tal decisão é necessário criar a respetiva GIC terminando com a implementação do gerador de pseudo-código Assembly para uma máquina virtual onde os nossos programas irão correr. Dadas estas necessidades começamos então por definir a linguagem tendo em conta os requisitos explicitados para que os próximos passos possam ser desempenhados, começando assim de seguida a descrição de todo este processo aqui descrito.

# 3 | Desenho e Implementação da Solução

## 3.1 Desenho da Linguagem

Dada a análise do enunciado e o correspondente problema proposto, ficou presente a necessidade de definir uma linguagem que preenchesse os requisitos impostos pelo professor, mas que ao mesmo tempo fosse simples quer ao nível da sua compreensão quer no seu desenvolvimento posterior. Para tal foram tidas em conta as seguintes considerações:

### 3.1.1 Estrutura Geral da Linguagem

Sendo explicitado que não poderiam haver declarações de variáveis no corpo do programa foi decidido que todas as declarações seriam efetuadas no inicio do ficheiro, existindo uma palavra reservada que determinaria o fim destas mesmas e daria inicio ao corpo do programa, tendo **HICODE** sido a palavra escolhida para o efeito. Foi atribuída também uma palavra reservada que define o termino do programa no seu todo, tendo sido **KTHKBYE** a palavra escolhida. Todas as instruções na nossa linguagem terminam com o carácter '**;**' à semelhança da linguagem C. Assim sendo terminamos com a seguinte estrutura geral do programa na nossa linguagem:


Figura 3.1: Estrutura geral de um programa

O corpo do código, parte que determina a funcionalidade do programa, é um conjunto de instruções, existindo os seguintes tipo de instruções na nossa linguagem:

- Atribuição
- Condicionais
- Cíclicas
- Escrita
- Leitura

Tais instruções serão descritas em profundidade nos seguintes tópicos do capitulo.

### 3.1.2 Atribuição e Manuseamento de Variáveis

Antes do corpo do programa começar a ser escrito, ou seja da palavra reservada **HICODE** ser declarada, temos a possibilidade de efetuar a definição das variáveis que iremos utilizar bem como a atribuição de valores às mesmas. A nossa linguagem apenas permite variáveis do tipo inteiro declaradas com a palavra **Int**. Dentro destas podem ser declaradas da seguinte forma:

- Sob a forma de variável elementar sem atribuição de valor declarando o tipo e o nome da variável que ficará por default com o valor 0. Exemplo: **Int x;**
- Sob a forma de variável elementar com atribuição de valor declarando o tipo e efetuando a atribuição através do caracter '=' seguido do valor numérico que pretendemos para a variável. Exemplo: **Int x = 63;**
- Sob a forma de um array, declarando o tipo, seguido do nome do array e dos caracteres '**[**' e '**]**', dentro dos quais deverá ser explicitado o tamanho do array sob a forma de um valor numérico. Exemplo: **Int array[16];**

É possivel no corpo do programa atribuir valores às variáveis quer sejam elementares ou de um array acedendo neste caso à posição através de '**[**' e '**]**' contendo dentro dos caracteres referidos o valor da posição, sendo esta declarada como um valor numérico, uma variável ou a expressão aritmética entre variáveis, tal como nos seguintes exemplos:

- **x = 10;**
- **array[2] = 38;**
- **array[1 + a] = 10;**

### 3.1.3 Expressões entre Variáveis

Na programação o conceito de uma expressão está ligado à matemática, onde um conjunto de variáveis numéricas se relacionam por meio de operadores compondo uma fórmula que após ser avaliada resulta num valor. Para que seja possível o uso de expressões na linguagem por nós desenvolvida foram implementadas as seguintes expressões e os respetivos símbolos.

#### Expressões Aritméticas

As expressões aritméticas surgem do resultado de uma avaliação do tipo numérico, no caso da nossa linguagem para valores inteiros. Apresentam-se na tabela seguinte as expressões implementadas na linguagem desenvolvida.

| **Operação** | **Símbolo** |
|---|---|
| Soma | + |
| Subtração | - |
| Multiplicação | * |
| Divisão | / |
| Resto | % |

Tabela 3.1: Expressões Aritméticas

#### Expressões Relacionais

As expressões relacionais efetuam uma comparação entre dois valores do mesmo tipo, neste caso entre valores inteiros apenas. Apresentam-se na tabela seguinte as expressões implementadas na linguagem desenvolvida.

| **Operação** | **Símbolo** |
|---|---|
| Igual | == |
| Diferente | != |
| Maior | > |
| Menor | < |
| Maior ou Igual | >= |
| Menor ou Igual | <= |

Tabela 3.2: Expressões Relacionais

#### Expressões Lógicas

As expressões lógicas resultam da aplicação de operadores lógicos a relações no caso da nossa linguagem pois apenas temos capacidade para representar variáveis inteiras e não variáveis lógicas. Apresentam-se na tabela seguinte as expressões implementadas na linguagem desenvolvida.

| **Operação** | **Símbolo** |
|---|---|
| E | & |
| Ou | \| |

Tabela 3.3: Operações Lógicas

### 3.1.4 Controlo de Fluxo

#### Condicionais

Uma condição é indicada por '**?( ) { }**' , contendo dentro dos parênteses curvos a condição (ou condições relacionadas com operadores lógicos) necessárias verificar para efetuar as instruções que serão declaradas dentro das chavetas. Caso a condição falhe hà a possibilidade de declarar um *else* sob a forma **_ { }** contendo dentro das chavetas as instruções.

#### Cíclicas

Um ciclo é definido pela palavra reservada **$( ) { }** contendo dentro dos parênteses curvos a condição (ou condições relacionadas com operadores lógicos) necessárias verificar para entrar no ciclo e efetuar as instruções que serão declaradas dentro das chaveta.

### 3.1.5 Leituras e Escritas

#### Leituras

Para tornar possível a interação entre o utilizador e o programa é possível pedir input ao utilizador numa instrução através do uso da palavra reservada **LEARNTHZ** seguida da variável na qual iremos colocar o input dado pelo utilizador.

#### Escritas

Para que possamos ver o output do nosso programa torna-se necessária a apresentação de resultados no ecrã. Em progz tal pode ser declarado através da palavra reservada **VISIVEL**

seguida da variável a imprimir no ecrã, ou então seguida de "" contendo dentro das mesmas a string a imprimir.

## 3.2 Desenho da Gramática

Após ter sido efetuado o desenho da linguagem, é necessário definir a gramática da mesma, definindo todas as regras referidas numa GIC.

Assim sendo obtemos os seguintes símbolos terminais:

**T: Int**, **HICODE**, **KTHKBYE**, **VISIVEL**, **LEARNTHZ**, **f**, **call**, '**;**', '**{**', '**}**', '(', ')', '**$**', '**_**', '**?**', '**=**'.

E os seguintes símbolos não terminais:

**NT: siplp**, **intvars**, **intvar**, **funcs**, **func**, **insts**, **inst**, **data**, **expr**, **parcel**, **factor**.

Assim sendo obtemos a seguinte gramática:

```
1  siplp: intvars funcs HICODE insts KTHKBYE { printf("%sjump inic\n%sstart\
       ninic: %sstop\n", $1, $2, $4); }
2        ;
3  intvars: Int intvar ';'              { $$ = $2; }
4         | Int intvar ';' intvars      { asprintf(&$$, "%s%s", $2, $4); }
5         ;
6  intvar: VAR                          { asprintf(&$$, "\tpushi 0\n"); addVar(
       $1); }
7         | VAR '=' NUM                 { asprintf(&$$, "\tpushi %d\n", $3);
             addVar($1); }
8         | VAR '[' NUM ']'             { asprintf(&$$, "\tpushn %d\n", $3);
             addVector($1, $3, 0);}
9         | VAR '[' NUM ']' '[' NUM ']' { asprintf(&$$, "\tpushn %d\n", $3 * $6)
             ; addVector($1, $3, $6);}
10        ;
11 funcs: func                         { $$ = $1; }
12      | funcs func                   { asprintf(&$$, "%s%s", $1, $2); }
13      |                              { $$ = ""; }
14      ;
15 func: f STRING '{' insts '}'        { asprintf(&$$, "%s: nop\n%s\treturn\n",
       $2, $4); }
16     ;
17 insts: inst                         { $$ = $1; }
18      | insts inst                   { asprintf(&$$, "%s%s", $1, $2); }
19
20 inst: VISIVEL factor ';'                 { asprintf(&$$, "%s\twritei\n", $2)
       ;}
21     | VISIVEL '"' STRING '"' ';'         { asprintf(&$$, "\tpushs \"%s\"\n\
          twrites\n", $3); }
22     | LEARNTHZ  data  ';'                { asprintf(&$$, "%s\tread\n\tatoi\n
          \%s", $2.begin, $2.end); }
```

```
23      | data '=' expr ';'                   { asprintf(&$$, "%s%s%s", $1.begin,
            $3, $1.end); }
24      | '?''('expr')' '{' insts '}'         { asprintf(&$$, "%s\tjz label%d\n%
            slabel%d: ", $3, label,
25                                              $6, label); label++; }
26      | '?''('expr')''{' insts '}''_''{' insts '}'    { asprintf(&$$, "%s\tjz
            label%d\n%sjump label%d\nlabel%d: %slabel%d: ",
27                $3, label, $6, label + 1, label, $10, label + 1); label += 2;
                  }
28      | '$''('expr')' '{' insts '}'                   { asprintf(&$$, "label%d
            : %s\tjz label%d\n%sjump label%d\nlabel%d: ",
29       label, $3, label + 1, $6, label, label + 1); label += 2; }
30      | call STRING ';'               { asprintf(&$$, "\tpusha %s\n\tcall\n\
            tnop\n", $2); }
31      |                               { $$ = ""; }
32      ;
33  data: VAR                            { asprintf(&$$.begin, "");
34                                         asprintf(&$$.end, "\tstoreg %d\n",
                                               getVar($1)); }
35      | VAR '[' expr ']'                { asprintf(&$$.begin, "\tpushgp\n\tpushi
            %d\n\tpadd\n%s", getVector($1), $3);
36                                         asprintf(&$$.end, "\tstoren\n"); }
37      | VAR '[' expr ']' '[' expr ']' { asprintf(&$$.begin, "\tpushgp\n\tpushi
            %d\n\tpadd\n%s\tpushi %d\n\tmul\n%s\tadd\n", getVector($1), $3,
           getVectorCol($1), $6);
38                                         asprintf(&$$.end, "\tstoren\n"); }
39  expr: parcel                  { $$ = $1; }
40      | expr '+' parcel         { asprintf(&$$, "%s%s\tadd\n", $1, $3); }
41      | expr '-' parcel         { asprintf(&$$, "%s%s\tsub\n", $1, $3); }
42  parcel: parcel '*' factor     { asprintf(&$$, "%s%s\tmul\n", $1, $3); }
43        | parcel '/' factor     { asprintf(&$$, "%s%s\tdiv\n", $1, $3); }
44        | parcel '%' factor     { asprintf(&$$, "%s%s\tmod\n", $1, $3); }
45        | parcel '>' factor     { asprintf(&$$, "%s%s\tsup\n", $1, $3); }
46        | parcel '<' factor     { asprintf(&$$, "%s%s\tinf\n", $1, $3); }
47        | parcel '>''=' factor { asprintf(&$$, "%s%s\tsupeq\n", $1, $4); }
48        | parcel '<''=' factor { asprintf(&$$, "%s%s\tinfeq\n", $1, $4); }
49        | parcel '!''=' factor { asprintf(&$$, "%s%s\tequal\npushi 1\ninf\n",
            $1, $4); }
50        | parcel '=''=' factor { asprintf(&$$, "%s%s\tequal\n", $1, $4); }
51        | parcel '&' factor     { asprintf(&$$, "%s%s\tadd\n\tpushi 2\n\tequal\
            n", $1, $3); }
52        | parcel '|' factor     { asprintf(&$$, "%s%s\tadd\n\tpushi 0\n\tsup\n"
            , $1, $3); }
53        | factor                { $$ = $1; }
54  factor: NUM                           { asprintf(&$$, "\tpushi %d\n", $1); }
55        | VAR                           { asprintf(&$$, "\tpushg %d\n", getVar
            ($1)); }
56        | VAR '[' expr ']'              { asprintf(&$$, "\tpushgp\n\tpushi %d\
            n\tpadd\n%s\tloadn\n", getVector($1), $3); }
57        | VAR '[' expr ']' '[' expr ']' { asprintf(&$$, "\tpushgp\n\tpushi %d\
            n\tpadd\n%s\tpushi %d\n\tmul\n%s\tadd\n\tloadn\n", getVector($1),
            $3, getVectorCol($1), $6); }
58        | '(' expr ')'                  { $$ = $2; }
59        ;
```

## 3.3 Geração do Pseudo-Código Assembly

Para que seja gerado o código assembly para a máquina virtual é necessário definir as ações no ficheiro Yacc. As ações referidas, escritas em código C, são ativadas quando é reconhecida a expressão das mesmas.

Para efetuarmos um controlo das variáveis no nosso programa recorremos ao seu armazenamento em duas estruturas, uma para as variáveis simples e outra para os vetores, e a geração do código assembly para alocação de memória das mesmas. Quando é reconhecida a declaração de um inteiro simples este é adicionado na sua estrutura com o nome da variável e o endereço na stack da mesma sendo este incrementado. O mesmo acontece para arrays, tirando que para um controlo no registo dos mesmos é necessário adicionar uma estrutura com a informação como o número de linhas e colunas em vez do endereço da stack apenas, sendo no final o stackpointer incrementado de acordo com o número de linhas.

Quando é reconhecida a palavra **HICODE** é gerada a instrução assembly para o inicio do corpo do programa através de uma label init. Nas instruções de atribuição sem valor é efetuado um **pushi** de 0, enquanto que nas atribuições com valor definido é efetuado um **pushi** com o valor indicado.

Na escrita de valores é gerada a instrução **pushs** seguida da string a imprimir caso seja dada uma string e da instrução **writes**. Se for caso de imprimir uma variável então é gerada a instrução **pushg** do número de declaração da mesma começando em 0 seguido de um **writei** da variável.

A leitura de valores é um pouco mais complexa. No caso de se tratar de uma leitura para uma variável simples é gerada a instrução **read** para efetuar a leitura, seguida da instrução **atoi** para conversão num inteiro, finalizando com um **storeg** com o endereço da variável. No caso dos arrays, é necessário efetuar um **pushgp** seguido de um **pushi** e um **padd** para obter a variável do array. O processo de leitura e escrita da variável é igual apenas tendo diferença na instrução final na qual é usada a instrução assembly **storen**.

Nos ciclos são usadas labels para marcar o inicio e fim dos mesmos, sendo efetuado um **jumpz** que verifica a falsidade da condição e salta para o fim do ciclo caso se verifique. No fim das instruções do ciclo existe um jump com a condição que se for validada como verdadeira retornará para a label de inicio de ciclo.

As expressões condicionais recorrem também às labels para marcar o fim das instruções contidas no if ou para a label referente ao else caso a mesma exista. A verificação é efetuada com recurso à instrução **jz** que salta para o fim caso seja falsa a verificação da condição.

Para representar operações aritméticas simples foi feito o parse dos operadores dado pelo utilizador da linguagem. Para a maior parte das operações existia uma operação assembly correspondente, porém, para operações lógicas 'e'() e ou(|) estas operações não estão definidas e portanto tiveram de ser definidas por nós através de junção de outras operações.

## 3.4 Estruturas de Dados

Para um controlo das variáveis declaradas no inicio do programa fizemos recurso à biblioteca **GLIB** mais propriamente à implementação de **HashTables** da mesma, permitindo-nos assim uma associação entre o nome da variável e o endereço na stack da mesma, ou no caso dos arrays a associação entre o nome da variável e a seguinte estrutura:

```
typedef struct VectorI {
    int add;
    int lin;
    int col;
} VectorI;
```

Esta permite marcar o endereço no campo add, bem como o número de linhas e colunas do array.

Para manter registo das variáveis nas atribuições usamos a seguinte estrutura:

```
typedef struct SString {
    char *begin;
    char *end;
} SString;
```

O campo **begin** é um apontador para o inicio do nome da variavel, e o campo **end** é um apontador para o fim da mesma.

# 4 | Exemplos de Utilização

De seguida apresentamos exemplos de ficheiros de entrada de programas escritos na nossa linguagem e o respetivo ficheiro do pseudo-código Assembly gerado.

## 4.1 Logico

**Ficheiro de Entrada:**

```
Int A;
Int B;
Int E;

HICODE

A = 3;
B = 5;
E = 1;

?((A >= E) & (E <= B)) { VISIVEL "True AND\n"; }
?((A >= E) | (E <= B)) { VISIVEL "True OR1\n"; }
?((A >= E) | (E > B))  { VISIVEL "True OR2\n"; }
?((A <= E) & (E < = B)){ VISIVEL "False AND\n";}
?((A < E) | (E > B))   { VISIVEL "False OR\n"; }

KTHKBYE
```

**Ficheiro de Saída:**

```
pushi 0
```

```
pushi 0
pushi 0

jump inic

start

inic:  pushi 3

storeg 0
pushi 5
storeg 1
pushi 1
storeg 2
pushg 0
pushg 2
supeq
pushg 2
pushg 1
infeq
add
pushi 2
equal
jz label0
pushs "True AND\n"
writes

label0:  pushg 0

pushg 2
supeq
pushg 2
pushg 1
infeq
add
pushi 0
sup
jz label1
pushs "True OR1\n"
writes

label1:  pushg 0

pushg 2
supeq
```

```
pushg 2
pushg 1
sup
add
pushi 0
sup
jz label2
pushs "True OR2\n"
writes

label2:  pushg 0

pushg 2
infeq
pushg 2
pushg 1
infeq
add
pushi 2
equal
jz label3
pushs "False AND\n"
writes

label3:  pushg 0

pushg 2
inf
pushg 2
pushg 1
sup
add
pushi 0
sup
jz label4
pushs "False OR\n"
writes

label4: stop
```

## 4.2 Menor

**Ficheiro de Entrada:**

```
Int i;
Int MINZ[2];

HICODE

VISIVEL "HOW NUMBZ?\n";

LEARNTHZ i;

$(i != 0){

VISIVEL "MIZ THZ MHUZ NUMB";
VISIVEL i;
VISIVEL " \n";
MINZ[0] = i;
LEARNTHZ MINZ[1];

?( MINZ[1] < MINZ[0] ){ MINZ[0] = MINZ[1]; }

i = i - 1;
}

VISIVEL "MINZ ITZ: ";
VISIVEL MINZ[0];

KTHKBYE
```

**Ficheiro de Saída:**

```
pushi 0
pushn 2

jump inic

start

inic:  pushs "HOW NUMBZ?\n"

writes
read
atoi
storeg 0

label1:  pushg 0
```

```
pushi 0
equal

pushi 1

inf

jz label2
pushs "MIZ THZ MHUZ NUMB"
writes
pushg 0
writei
pushs " \n"
writes
pushgp
pushi 1
padd
pushi 0
pushg 0
storen
pushgp
pushi 1
padd
pushi 1
read
atoi
storen
pushgp
pushi 1
padd
pushi 1
loadn
pushgp
pushi 1
padd
pushi 0
loadn
inf
jz label0
pushgp
pushi 1
padd
pushi 0
pushgp
```

```
pushi 1
padd
pushi 1
loadn
storen

label0:  pushg 0

pushi 1
sub
storeg 0

jump label1

label2:  pushs "MINZ ITZ: "

writes
pushgp
pushi 1
padd
pushi 0
loadn
writei

stop
```

## 4.3 Ola

**Ficheiro de Entrada:**

```
Int a;

HICODE

VISIVEL "HELLOZ WORLDZS!!";

KTHKBYE
```

**Ficheiro de Saída:**

```
pushi 0
```

```
jump inic

start

inic:  pushs "HELLOZ WORLDZS!!"

writes

stop
```

## 4.4 Ordenar

**Ficheiro de Entrada:**

```
Int QUANTZ = 10;
Int INPUTZ[10];
Int OTPUTZ[10];
Int ITERZ;
Int MXITERZ;
Int MAXPLAZ;
Int MAXZ;

func MAXMILIANZ {

MAXZ = INPUTZ[0];

MAXPLAZ = 0;

MXITERZ = 0;

$( MXITERZ != QUANTZ ){

?( INPUTZ[MXITERZ] > MAXZ ){

MAXZ = INPUTZ[MXITERZ];
MAXPLAZ = MXITERZ;

}

MXITERZ = MXITERZ + 1;
}
```

```
}

HICODE



VISIVEL "TELZ";
VISIVEL QUANTZ;
VISIVEL " NUMBZ \ITERZ";

$( ITERZ != QUANTZ ){

VISIVEL "INPUTZ";
VISIVEL ITERZ;
VISIVEL " \n";
LEARNTHZ INPUTZ[ITERZ];
ITERZ = ITERZ +1;

}

ITERZ = 0;

$( ITERZ != QUANTZ ){

call MAXMILIANZ;
OTPUTZ[ITERZ] = INPUTZ[MAXPLAZ];
INPUTZ[MAXPLAZ] = 0;
ITERZ = ITERZ + 1;

}

ITERZ = 0;

$( ITERZ != QUANTZ ){

VISIVEL OTPUTZ[ITERZ];
VISIVEL " \ITERZ";
ITERZ = ITERZ +1;

}

KTHKBYE
```

**Ficheiro de Saída:**

```
pushi 10
pushn 10
pushn 10
pushi 0
pushi 0
pushi 0
pushi 0
jump inic

MAXMILIANZ: nop

pushgp
pushi 1
padd
pushi 0
loadn
storeg 24
pushi 0
storeg 23
pushi 0
storeg 22

label1:  pushg 22

pushg 0
equal

pushi 1

inf
jz label2
pushgp
pushi 1
padd
pushg 22
loadn
pushg 24
sup
jz label0
pushgp
pushi 1
padd
pushg 22
loadn
storeg 24
```

```
pushg 22
storeg 23

label0:  pushg 22

pushi 1
add
storeg 22

jump label1

label2:  return

start

inic:  pushs "TELZ"

writes
pushg 0
writei
pushs " NUMBZ \ITERZ"
writes

label3:  pushg 21

pushg 0
equal

pushi 1

inf

jz label4
pushs "INPUTZ"
writes
pushg 21
writei
pushs " \n"
writes
pushgp
pushi 1
padd
pushg 21
read
atoi
```

```
storen
pushg 21
pushi 1
add
storeg 21

jump label3

label4:  pushi 0

storeg 21

label5:  pushg 21

pushg 0
equal

pushi 1

inf
jz label6
pusha MAXMILIANZ
call
nop
pushgp
pushi 11
padd
pushg 21
pushgp
pushi 1
padd
pushg 23
loadn
storen
pushgp
pushi 1
padd
pushg 23
pushi 0
storen
pushg 21
pushi 1
add
storeg 21
```

```
jump label5

label6:  pushi 0

storeg 21

label7:  pushg 21

pushg 0
equal

pushi 1

inf

jz label8
pushgp
pushi 11
padd
pushg 21
loadn
writei
pushs " \ITERZ"
writes
pushg 21
pushi 1
add
storeg 21

jump label7

label8: stop
```

## 4.5 Vetor

**Ficheiro de Entrada:**

```
Int NUMBR;
Int VECZ[5];

HICODE
```

```
NUMBR = 35;
VECZ[1+1] = 1 + 2;
VECZ[0] = 5;
VECZ[1] = NUMBR;
VECZ[4] = VECZ[1] + VECZ[0];
VISIVEL VECZ[4];

KTHKBYE
```

**Ficheiro de Saída:**

```
pushi 0

pushn 5

jump inic

start

inic:  pushi 35

storeg 0
pushgp
pushi 1
padd
pushi 1
pushi 1
add
pushi 1
pushi 2
add
storen
pushgp
pushi 1
padd
pushi 0
pushi 5
storen
pushgp
pushi 1
padd
pushi 1
pushg 0
storen
```

```
pushgp
pushi 1
padd
pushi 4
pushgp
pushi 1
padd
pushi 1
loadn
pushgp
pushi 1
padd
pushi 0
loadn
add
storen
pushgp
pushi 1
padd
pushi 4
loadn
writei

stop
```

## 4.6 Inverso

**Ficheiro de Entrada:**

```
Int i = 10;
Int a[10];
Int n;

HICODE

VISIVEL "TELZ";
VISIVEL i;
VISIVEL " NUMBZ \n";

$( n != i ){

VISIVEL "INPUTZ";
```

```
VISIVEL n;
VISIVEL " \n";
LEARNTHZ a[n];
n = n +1;

}

n = i-1;

$( n >= 0 ){

VISIVEL a[n];
VISIVEL "  ";
n = n -1;

}

KTHKBYE
```

**Ficheiro de Saída:**

```
pushi 10
pushn 10
pushi 0

jump inic

start

inic:  pushs "TELZ"

writes
pushg 0
writei
pushs " NUMBZ \n"
writes

label0:  pushg 11

pushg 0
equal

pushi 1
```

```
inf

jz label1
pushs "INPUTZ"
writes
pushg 11
writei
pushs " \n"
writes
pushgp
pushi 1
padd
pushg 11
read
atoi
storen
pushg 11
pushi 1
add
storeg 11

jump label0

label1:  pushg 0

pushi 1
sub
storeg 11

label2:  pushg 11

pushi 0
supeq
jz label3
pushgp
pushi 1
padd
pushg 11
loadn
writei
pushs "  "
writes
pushg 11
pushi 1
sub
```

```
storeg 11
jump label2

label3: stop
```

# 5 | Conclusão

Após descrevermos todos os processos que envolvem a elaboração deste trabalho, desde a elaboração do desenho da nossa linguagem até á geração do código em Assembly, sem esquecer a construção da respetiva gramática e os respetivos testes que usamos para pormos á prova a viabilidade da linguagem desenvolvida pelo grupo, só nos resta apresentar uma breve conclusão sobre todo o processo.

Para construirmos esta linguagem baseamo-nos em dois tipos de linguagens, uma é uma linguagem imperativa e a outra é uma linguagem isotérica. A nossa linguagem vai resultar numa linguagem com características isotéricas mas com as funcionalidades imperativas. Atualmente o projeto encontra-se totalmente funcional, pois recebe um ficheiro com um programa escrito na nossa linguagem e gera sempre um pseudo-código Assembly para uma máquina virtual onde os nossos programas irão correr.

No entanto, como trabalho futuro, seria interessante implementarmos funções sobre floats e novas funcionalidades tais como abreviaturas para a linguagem tal como a++ ou a+=n;.

Pessoalmente achamos que a realização deste projeto muito cativante não só pelo incentivo á nossa criatividade mas também por acharmos todo o conceito de criar um compilador muito importante, pois adquirimos uma melhor noção de como um compilador realmente funciona. No inicio, parecia-nos um projeto bastante complexo mas depois de percebermos o seu funcionamento tornou-se mais acessível.