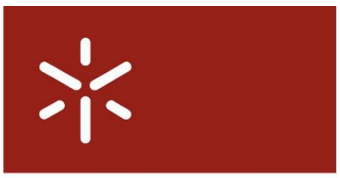


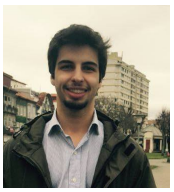
Universidade do Minho

Processamento de Linguagens

MIEI - 3º ano - 2º semestre
Universidade do Minho

Definição de uma linguagem de programação

Yacc - Trabalho Prático nº3



Dinis Peixoto
A75353

Ricardo Pereira
A74185

Marcelo Lima
A75210

16 de Novembro de 2017

Conteúdo

1. *Contextualização*

O presente relatório é o resultado da resolução do terceiro e último trabalho prático posto na unidade curricular de Processamento de Linguagens inserida no curso de Engenharia Informática. Ambientados agora com o sistema operativo *Linux* e com conhecimentos relativos a expressões regulares e filtros de texto em *C Flex* consolidados com a realização dos trabalhos práticos anteriores, seria agora necessária a definição e implementação de uma linguagem de programação imperativa simples.

Ao contrário dos demais trabalhos já realizados na presente unidade curricular, cuja constituição era composta por vários enunciados dos quais o grupo selecionar um ou vários, de modo a conseguir demonstrar os seus conhecimentos, este é constituído por um único enunciado comum e geral e com os objetivos concretamente definidos.

Os objetivos da realização deste trabalho prático passariam essencialmente pelo aumento da experiência dos alunos no âmbito da *engenharia de linguagens* e em *programação generativa*, ou, mais especificamente, no desenvolvimento de processadores de linguagens segundo o método de tradução dirigida pela sintaxe, no desenvolvimento de um compilador gerando código para uma máquina de stack virtual, e ainda, na utilização de geradores de compilares baseados em gramáticas tradutoras, neste caso o Yacc.

2. *Descrição do problema*

2.1 Declaração e manuseamento de variáveis

Primeiramente convém mencionar que a nossa simples linguagem de programação apresentará apenas variáveis simples de um único tipo, podendo, eventualmente, serem formados outros diferentes tipos de variáveis complexas (listas ou matrizes) através desta.

É importante também considerar que as variáveis devem ser declaradas no início do programa de modo a armazenar imediatamente espaço para estas na *stack*, assim, tendo isto em conta deverão ser considerados quaisquer possíveis erros na declaração das mesmas, exigindo inevitavelmente a existência de um mecanismo de deteção de erros durante a inicialização destas.

A existência de variáveis, em qualquer linguagem de programação tem um propósito comum e universal: a atribuição de valores às mesmas, de modo a que possam ser utilizadas em múltiplas ocasiões, como tal, será importante percebermos a necessidade da atribuição de *pointers* a cada uma das variáveis inicializadas, para que possamos, durante a execução do programa realizar as atribuições pretendidas.

2.2 Resolução de instruções algorítmicas básicas

A resolução de instruções algorítmicas básicas é um dos requisitos do presente trabalho, como tal o grupo ponderou, antes de dar início à implementação, quais as operações básicas permitidas e como efetuar a resolução das mesmas. Segue-se a listagem das operações que o grupo considerou imprescindíveis para a formulação da linguagem de programação, e dentro de parênteses a identificação destas na nossa linguagem.

Operações aritméticas

- $+$ $(+)$: soma;
- $-$ $(-)$: subtração;
- $/$ $(/)$: divisão inteira;
- $*$ $(*)$: multiplicação;
- $\%$ $(\%)$: resto da divisão inteira.

Operações lógicas

- > (>) : maior que;
- < (<) : menor que;
- >= (>>) : maior ou igual que;
- <= (<<) : menor ou igual que;
- && (&) : conjunção;
- || (|) : disjunção.

Tal como nas restantes linguagens de programação com as quais já estamos conformados, as expressões aritméticas/lógicas que envolvam várias expressões devem ser agrupas com **()**.

2.3 Standard input/output

A leitura do *standard input* e a apresentação de um *output* depois de executado um programa é algo que a nossa linguagem de programação deve também permitir, como tal, deve ser realizada a conversão de *string* para inteiro e vice-versa. Para o *standard input* deve ser possível a operação de leitura de inteiros, atribuindo o valor deste a uma determinada variável. Quanto ao *standard output* deve permitir a escrita de inteiros ou *strings*, escrevendo no *stdout* o(s) parâmetro(s) passado(s) na chamada da operação.

2.4 Resolução de instruções para controlo de fluxo de execução

As cláusulas de controlo de fluxo de execução, são, também um ponto importantíssimo em qualquer linguagem de programação da atualidade, é com estas que conseguimos definir condições verdadeiras/falsos e ainda ciclos, de modo a evitar a repetição explícita de instruções. De seguida, abordaremos conceitos relativos às instruções em *Assembly* que traduzem as cláusulas de controlo de fluxo que a nossa linguagem de programação apresentará.

IF/IF ELSE

Estas cláusulas são implementadas em Assembly com um conjunto de diferentes saltos (*jumps*), isto é, conforme as condições apresentadas o programa seguirá diferentes rumos saltando (ou não) para operações distintas.

WHILE

Esta será a única cláusula iterativa na nossa linguagem, e a sua aplicação em *Assembly* é, em parte, semelhante à cláusula anterior, assim, do mesmo modo, é verificada uma condição e o programa saltará (ou não) para operações distintas.

Neste caso enquanto o programa se encontrar no processo iterativo estará dentro do mesmo conjunto de instruções, voltando para o início destas a cada iteração do ciclo e saíndo deste conjunto quando a condição exposta se deixar de verificar.

2.5 Invocação de subprogramas

A declaração de funções é fundamental em qualquer linguagem de programação, não pretendíamos que a nossa fugisse à regra por isso foi implementada da maneira mais *caseira* possível, isto é, as funções devem ser declaradas antes do início da execução do programa principal, e devem ser chamadas neste durante a execução do mesmo, agora já tendo conhecimento da existência das funções em cima declaradas.

3. *Decisões e Implementação*

3.1 Gramática Independente de Contexto

Para o probelma em questão, é necessário uma linguagem que permita carregar os dados a processar. Estes dados são obtidos apartir de um ficheiro **.i*, sendo a extensão do ficheiro idealizada por nós.

Como tal, recorremos a uma Gramática Independente de Contexto, *GIC*, que especifique a linguagem pretendida. No processo de desenvolvimento da gramática, tivemos o cuidado de apresentar um estilo apresentável e legível, que se tenta aproximar o mais possível com a sintaxe das linguagens imperativas existentes acutalmente.

A *GIC* é a seguinte:

```
T = {ID, NUM, STR, F, iWrite, iRead, iLoop, exe, ';', ',', '[',
']', '{' , '}', '(', ')', '"', '=', '!', '?', '+', '-', '*',
'/', '%', '>', '<', '&', '|'}

N = {iPL, intIDs, intID, ints, funcs, func, insts, inst, portion,
elem, expr}

S = iPL

P = {
p0: iPL -> ints funcs START insts END

p1: ints -> Int intIDs ';'

p2: intIDs -> intID
p3:         | intIDs ',' intID

p4: intID -> ID
p5:        | ID '=' NUM
p6:        | ID '[' NUM ']'
p7:        | ID '[' NUM ']' '[' NUM ']'

p8: funcs -> func
p9:        | funcs func
p10:       | &
```

```
p11: func -> F STR '{' insts '}'

p12: insts -> inst
p13:       | insts inst

p14: inst -> iWrite '(' elem ')' ';'
p15:      | iWrite '(' '"' STR '"' ')'';'
p16:      | iRead '(' var ')' ';'
p17:      | var '=' expr ';'
p18:      | '?''('expr')' '{' insts '}'
p19:      | '?''('expr')''{' insts '}''!''?''{' insts '}'
p20:      | iLoop '('expr')' '{' insts '}'
p21:      | exe STR ';'
p22:      | &

p23: var -> ID
p24:     | ID '[' expr ']'
p25:     | ID '[' expr ']' '[' expr ']'

p26: expr -> portion
p27:      | expr '+' portion
p28:      | expr '-' portion

p29: portion -> portion '*' elem
p30:         | portion '/' elem
p31:         | portion '%' elem
p32:         | portion '>' elem
p33:         | portion '<' elem
p34:         | portion '>''>' elem
p35:         | portion '<''<' elem
p36:         | portion '!''=' elem
p37:         | portion '=''=' elem
p38:         | portion '&' elem
p39:         | portion '|' elem
p40:         | elem

p41: elem -> NUM
p42:      | ID
p44:      | ID '[' expr ']'
p45:      | ID '[' expr ']''[' expr ']'
p46:      | '(' expr ')'
}
```

3.2 Declaração e manuseamento de variáveis

O processo de declaração de variáveis ocorre apenas na fase inicial, antes da declaração de funções. As produções responsáveis para este processo são:

```
p1: ints -> Int intIDs ';'

p2: intIDs -> intID
p3:         | intIDs ',' intID

p4: intID -> ID
p5:        | ID '=' NUM
p6:        | ID '[' NUM ']'
p7:        | ID '[' NUM ']' '[' NUM ']'
```

O simbolo *ints*, representa o conjunto total das declarações de variáveis. Como podemos ver este pode representar um conjunto de simbolos *intID*, ou apenas um único. Por sua vez, o simbolo *intID* pode respresentar os diferentes tipos de váriaveis aceites pela linguagem, isto é:

- **Valor único** - Ex. Int a;
- **Array de 1 dimensão** - Ex. Int array[10];
- **Array de 2 dimensões** - Ex. Int array[10][2];

A atribuição de valores às variáveis é efectuada nesta fase, na definição de uma função ou entre as cláusulas *START* e *END*.

3.2.1 Código Assembly

Uma vez que o mecanismo tem de reconhecer a gramática realizada, falta uma etapa para o processo final, a conversão para código *Assembly* da respectiva instrução. Para o caso da declaração de variáveis recorremos às seguintes operações:

- **ID** - pushi 0 : empilha na stack a variável com o valor 0.
- **ID = NUM** - pushi NUM : empilha na stack a variável com o valor de NUM.
- **ID [NUM]** - pushn NUM : empilha na stack NUM variáveis com o valor 0.
- **ID [NUM][NUM]** - pushn NUM*NUM : empilha na stack NUM*NUM variáveis com o valor 0.

Desta forma, as váriaveis no inicio do programa, são inicializadas, ficando empilhadas na stack, a 0 ou com o valor atribuido.

3.3 Resolução de instruções algorítmicas básicas

Como já foi enunciado no capítulo anterior, o nosso programa aceita as variadissimas operacões aritméticas. Para tal foi necessário definir na nossa gramática as seguintes produções:

```
p26: expr -> portion
p27:      | expr '+' portion
p28:      | expr '-' portion

p29: portion -> portion '*' elem
p30:         | portion '/' elem
p31:         | portion '%' elem
p32:         | portion '>' elem
p33:         | portion '<' elem
p34:         | portion '>''>' elem
p35:         | portion '<''<' elem
p36:         | portion '!''=' elem
p37:         | portion '=''=' elem
p38:         | portion '&' elem
p39:         | portion '|' elem
p40:         | elem

p41: elem -> NUM
p42:      | ID
p44:      | ID '[' expr ']'
p45:      | ID '[' expr ']''[' expr ']'
p46:      | '(' expr ')'
```

3.3.1 Código Assembly

Com as produções apresentadas anteriormente, podemos definir várias expressões, como de uma variável apenas, ou até mesmo várias expressões, umas agrupadas noutras. Como é de salientar foi necessário tomar medidas para poder efectuar certas operações. As operações em questão são:

- **!=** : Operação que indica que se uma expressão é diferente de outra. Para esta operação, foi necessário definir um algoritmo capaz de devolver o valor pretendido da operação (0-falso, 1-verdadeiro). Em *Assembly*, definimos a seguinte sequência de instruções:

```
        OP1
        OP2
        equal
        pushi 1
        inf
```

Desta forma, a operação de igualdade (*equal*) tem que ser inferior a 1, para indicar que os valores OP1 e OP2 são diferentes.

- & : Operação que indica a conjunção de duas expressões. Como o exemplo anterior, também foi necessário definir um algoritmo capaz de devolver o valor pretendido da operação (0-falso, 1-verdadeiro). Em *Assembly*, definimos a seguinte sequência de instruções:

```
OP1
OP2
add
pushi 2
equal
```

Desta forma, como estamos perante duas operações lógicas, ambas podem ter como valor 0 ou 1. Como tal, para que ambas sejam 1, e verificar que a conjunção é válida, a soma dos seus valores tem que ser igual a 2. Para isso, aplicamos a instrução *add* para somar os dois valores dos operadores, e por fim comparamos se é igual a 2. A partir deste momento, o valor da operação & vai depender do valor da operação *equal*.

- | : Operação que indica a disjunção de duas expressões. Dentro do mesmo género dos outros exemplos, apresenta a necessidade de um algoritmo que nos devolva o valor pretendido da operação (0-falso, 1-verdadeiro). Em *Assembly*, definimos a seguinte sequência de instruções:

```
OP1
OP2
add
pushi 0
sup
```

Como era de esperar, para a disjunção de duas expressões a soma dos valores lógicos que ambas devolvem, tem que ser superior a 0, para que a espressão seja verdadeira. Logo para tal, efectuamos a soma das duas expressões lógicas e verificamos se este é maior a 0, o que indica que é verdadeiro.

3.4 Standard input/output

Para resolver o problema de poder escrever/ler do standard output/input, recorremos às seguintes produções:

```
p14: inst -> iWrite '(' elem ')' ';'
p15:      | iWrite '(' '"' STR '"' ')'';'
p16:      | iRead '(' var ')' ';'
```

Como podemos ver, a operação de leitura apenas lê um valor (inteiro), para uma variável atribuída como parâmetro. Quanto à operação de escrita, esta pode imprimir um número inteiro, ou uma *String*.

3.4.1 Código Assembly

- **iWrite** : Para efectuar a operação de escrita efecutamos as seguintes operações em assembly:

```
NUM
writei
```

  Desta forma, basta colocar o valor inteiro, que provém das variáveis ou arrays em questão, no topo da stack, e depois exectuar a instrução *writei*. Para o caso de imprimir uma string, temos a seguinte sequência de instruções:

```
pushs
STR
writes
```

  Como é visível, para este caso, apenas precisamos de colocar, a string apresentada como parâmetro da operação *iWrite*, no topo da stack, e imprimir a mesma.

- **iRead** : Para o processo de leitura é necessário ter maior cuidado. As instruções necessárias a efectuar são as seguintes:

```
VAR.push
read
atoi
VAR.store
```

  Como podemos ver, é necessário ter uma estrutura que nos devolva as instruções de *push*, e ao mesmo tempo as instruções de *store*, que corresponde ao processo de armazenamento dos valores lidos. Este mecanismo é necessário, porque tanto podemos armazenar um valor lido numa variável (a), como numa posição de um array ou matriz (a[3] ou a[2][3]).

3.5 Resolução de instruções para controlo de fluxo de execução

O controlo de fluxo provém das operações *if*, *else* e *while*, que na nossa gramática correspondem a *?*, *!?* e *iLoop*, respectivamente. Para estas operações foram desenvolvidas as seguintes produções:

```
p18:      | '?''('expr')' '{' insts '}'
p19:      | '?''('expr')''{' insts '}''!''?''{' insts '}'
p20:      | iLoop '('expr')' '{' insts '}'
```

3.5.1 Código Assembly

O mecanismo de controlo de fluxo é o que apresenta maior dificuldade ao nível do código *Assembly*.

- **IF ELSE**: Para o caso das condições *if else*, temos as seguintes instruções:

```
CONDICAO
jz LABEL(X)
CODIGO PARA O CASO DE SUCESSO
jump LABEL(Y)
LABEL(X): CODIGO DO ELSE
LABEL(Y):
```

  Esta sequência de instruções definem o processo correspondente a um *if else*. Em primeiro ocorrem as operações lógicas, e consoante o valor obtido, a instrução *jz* salta para LABEL(X) caso a condição não se verifique, ou prossegue a sequência de instruções normalmente caso a condição se verificar. Para este último caso, adicionamos uma instrução *jump*, para saltar a parte correspondente à clausula *else*.

- **LOOP** Por outro lado, para o caso do ciclo, possuimos as seguintes instruções:

```
LABEL(X):
CONDICAO
jz LABEL(Y)
CODIGO PARA O CASO DE SUCESSO
jump LABEL(X)
LABEL(Y):
```

  Neste processo temos algumas diferenças comparadas ao caso anterior, pois caso a condição se verificar, é necessário efectuar um salto para o início para voltar a repetir a sequência de instruções correspondentes ao interior do ciclo. Para tal, usamos uma LABEL(X), para indicar o local onde efectuar uma nova verificar da condição do ciclo. Com isto, é necessário que se efectue as operações lógicas, para verificar se entramos dentro do ciclo ou não. Caso a condição se verificar, a instrução *jz* não nos faz saltar para lado nenhum, fazendo-nos prosseguir na execução da sequência de instruções. Ao fim de executar as instruções, este efectua um salto (*jump*), para a LABEL(X), que é o inicio estabelecido. Quando a condição já não se verifica, ocorre um salto para a LABEL(Y).

3.6 Invocação de subprogramas

O processo de invocação corresponde, basicamente, em agrupar pedaços de instruções do programa geral. Para isso possui as seguintes produções:

```
p8: funcs -> func
p9:        | funcs func
p10:       | &

p11: func -> F STR '{' insts '}'
```

A declaração de uma função ocorre antes da execução do programa geral. É indentificado pela letra "F"e o nome da função STR. Desta forma, basta indicar as instruções que a função vai executar.

3.6.1 Código Assembly

Desta forma para converter este processo em *assembly*, apenas precisamos de definir uma *label*, com o nome da função, e agrupar as suas instruções nessa *label*. Com isto, posteriormente apenas é preciso chamar a função indicando a *label*, e este apresenta as instruções a executar. A sequência em assembly obtida é a seguinte:

```
NOME DA FUNCAO:
nop
CODIGO DAS INSTRUCOES DA FUNCAO
return
```

4. *Deteção de erros*

Para efetuar a deteção de erros aquando da compilação dos programas desenvolvidos na nossa linguagem de programação foi utilizada uma arquitetura idealizada pelo grupo. Foram criadas duas *HashTables*, importadas da glib.h, que servem para armazenar as variáveis, desta forma, se forem realizadas atribuições ou quaisquer tipos de operações com variáveis não declaradas previamente, uma vez que estas não se irão encontrar na estrutura de dados será lançado um erro de compilação instruindo o utilizador da razão causadora de tal. A diferença entre as duas *HashTables* é, nada mais nada menos que a distinção entre armazenagem de variáveis simples e complexas, isto é, simples inteiros, ou conjuntos de inteiros (arryays ou matrize).

5. *Demonstração dos resultados obtidos*

Nesta secção apresentaremos alguns exemplos utilizados durante os testes da linguagem de programação produzida, assim como os respectivos resultados capazes de reproduzir o objetivo final pretendido.

5.1 Identificação de números ímpares

O programa que se segue lê do utilizador (*stdin*) uma sequência de 10 números inteiros e armazena numa lista os que forem ímpares.

5.1.1 Programa fonte

```
Int n,i,count,lido,num[10];

F contador{
        n=10;
        i=0;
        count=0;
        iLoop(i<n){
                iRead(lido);
                ?((lido%2)!=0){
                        num[count]=lido;
                        count = count + 1;
                }
                i = i+1;
        }
}

START
        exe contador;
        i=0;
        iWrite("Os numeros impares sao: ");
        iLoop(i<count){
                iWrite(num[i]);
                iWrite(" ");
                i = i+1;
        }
```

```
END
```

5.1.2 Código fonte produzido

```
        pushi 0
        pushi 0
        pushi 0
        pushi 0
        pushn 10
jump inic
contador: nop
        pushi 10
        storeg 0
        pushi 0
        storeg 1
        pushi 0
        storeg 2
LABEL1:         pushg 1
        pushg 0
        inf

        jz LABEL2
        read
        atoi
        storeg 3
        pushg 3
        pushi 2
        mod
        pushi 0
        equal
pushi 1
inf
        jz LABEL0
        pushgp
        pushi 4
        padd
        pushg 2
        pushg 3
        storen
        pushg 2
        pushi 1
        add
        storeg 2
LABEL0:         pushg 1
        pushi 1
        add
```

```
        storeg 1
        jump LABEL1
LABEL2:         return
start
inic:         pusha contador
        call
        nop
        pushi 0
        storeg 1
        pushs "Os numeros impares sao: "
        writes
LABEL3:         pushg 1
        pushg 2
        inf

        jz LABEL4
        pushgp
        pushi 4
        padd
        pushg 1
        loadn
        writei
        pushs " "
        writes
        pushg 1
        pushi 1
        add
        storeg 1
        jump LABEL3
LABEL4: stop
```

5.1.3 Input/Output

Input
Para testar o programa anterior foi utilizada um conjunto de números inteiros muito simples, a sequência de 0 até 9, inclusive, fazendo um total de 10 números tal como pretendido.

```
0123456789
```

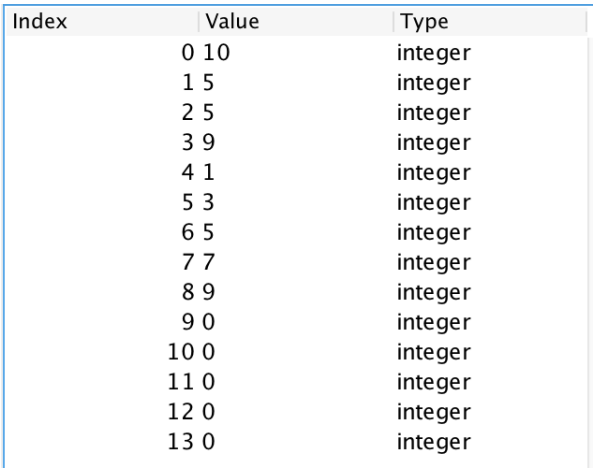
Output
Facilmente conseguimos reparar que o resultado obtido é o resultado esperado, uma vez que os números 1, 3, 5, 7, 9 são, de facto, os números ímpares no intervalo

considerado.

```
Os numeros impares sao: 1 3 5 7 9
```

Stack

Pela observação da *stack* identificamos facilmente os valores presentes em cada um dos elementos dos seus índices.



Index	Value	Type
0	10	integer
1	5	integer
2	5	integer
3	9	integer
4	1	integer
5	3	integer
6	5	integer
7	7	integer
8	9	integer
9	0	integer
10	0	integer
11	0	integer
12	0	integer
13	0	integer

Índice	Valor	Variável	Significado
0	10	n	Número de números a serem lidos
1	5	i	Valor utilizado para iterar (fica a 5 porque é o número de ímpares encontrados)
2	5	count	Número de valores ímpares encontrados (sempre que a condição é verificada este é incrementado)
3	9	lido	Número de valores lidos, este é incrementado até a condição do ciclo deixar de se verificar, isto é, enquanto for menor que 10, daí o seu valor final ser 9.
4	1	num[0]	Índice 0 do array num[], este contém o valor 1, o primeiro valor ímpar encontrado.
5	3	num[1]	Índice 1 do array num[], este contém o valor 3, o segundo valor ímpar encontrado.
6	5	num[2]	Índice 2 do array num[], este contém o valor 5, o terceiro valor ímpar encontrado.
7	7	num[3]	Índice 3 do array num[], este contém o valor 7, o quarto valor ímpar encontrado.
8	9	num[4]	Índice 4 do array num[], este contém o valor 9, o quinto valor ímpar encontrado.
9	0	num[5]	Índice 5 do array num[], este contém o valor 0 porque não foram encontrados mais valores, como tal não foi preenchido.
10	0	num[6]	Índice 6 do array num[], este contém o valor 0 porque não foram encontrados mais valores, como tal não foi preenchido.
11	0	num[7]	Índice 7 do array num[], este contém o valor 0 porque não foram encontrados mais valores, como tal não foi preenchido.
12	0	num[8]	Índice 8 do array num[], este contém o valor 0 porque não foram encontrados mais valores, como tal não foi preenchido.
13	0	num[9]	Índice 9 do array num[], este contém o valor 0 porque não foram encontrados mais valores, como tal não foi preenchido.

5.2 Representação inversa dos números de uma lista

O programa que se segue lê um conjunto de três números imprimindo estes posteriormente ao utilizador pela sua ordem inversa.

5.2.1 Programa fonte

```
Int i,num[3];

F ler{
        iWrite("Introduza uma sequencia de numeros: ");
        i=0;
        iLoop(i<3){
                iRead(num[i]);
                i = i+1;
        }
}

START
        exe ler;
        iWrite("Array por ordem inversa: ");
        i = i-1;
        iLoop(i>>0){
                iWrite(num[i]);
                iWrite(" ");
                i=i-1;
        }
END
```

5.2.2 Código fonte produzido

```
        pushi 0
        pushn 3
jump inic
ler: nop
        pushs "Introduza uma sequencia de numeros: "
        writes
        pushi 0
        storeg 0
LABEL0:         pushg 0
        pushi 3
        inf

        jz LABEL1
        pushgp
        pushi 1
        padd
        pushg 0
        read
        atoi
        storen
```

```
        pushg 0
        pushi 1
        add
        storeg 0
        jump LABEL0
LABEL1:         return
start
inic:         pusha ler
        call
        nop
        pushs "Array por ordem inversa: "
        writes
        pushg 0
        pushi 1
        sub
        storeg 0
LABEL2:         pushg 0
        pushi 0
        supeq

        jz LABEL3
        pushgp
        pushi 1
        padd
        pushg 0
        loadn
        writei
        pushs " "
        writes
        pushg 0
        pushi 1
        sub
        storeg 0
        jump LABEL2
LABEL3: stop
```

5.2.3 Input/Output

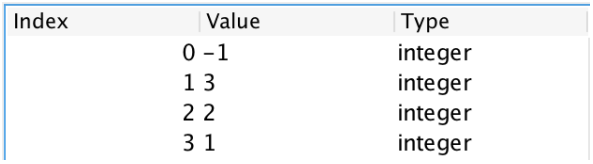
Input
Para testar o programa anterior foi utilizada um conjunto de números inteiros muito simples, a sequência 3,2,1 fazendo um total de 3 números tal como pretendido.

"Introduza uma sequencia de numeros: "321"

Output
Sem grandes dificuldades percebemos que o resultado obtido é, de facto, a sequência inversa da inicialmente fornecida pelo *utilizador*.

```
Array por ordem inversa: 1 2 3
```

Stack
Pela observação da *stack* identificamos facilmente os valores presentes em cada um dos elementos dos seus índices.



Index	Value	Type
0	-1	integer
1	3	integer
2	2	integer
3	1	integer

Índice	Valor	Variável	Significado
0	-1	i	Variável utilizada para iterações. O valor final é -1 uma vez que se trata de um ciclo em que esta é decrementada até cumprir a condição menor ou igual que 0.
1	3	num[0]	Índice 0 do array num, este contém o primeiro valor lido.
2	2	num[1]	Índice 1 do array num, este contém o segundo valor lido.
3	1	num[2]	Índice 2 do array num, este contém o terceiro valor lido.

5.3 Soma de duas matrizes com a mesma dimensão

O programa seguinte corresponde ao somatório de duas matrizes com a mesma dimensão (2x2), resultando numa matriz da mesma dimensão que as anteriores.

5.3.1 Programa fonte

```
Int i,j,dimensaoi, dimensaoj, matriz[2][2], matrizz[2][2],
matrizresultante[2][2];

F input{
        i = 0;
        j = 0;
        dimensaoi = 2;
        dimensaoj = 2;
        iLoop(i<dimensaoi){
```

```
                j = 0;
                iLoop(j<dimensaoj){
                        iRead(matriz[i][j]);
                        j = j+1;
                }
                i = i+1;
        }
        i = 0;
        j = 0;
        iLoop(i<dimensaoi){
                j = 0;
                iLoop(j<dimensaoj){
                        iRead(matrizz[i][j]);
                        j = j+1;
                }
                i = i+1;
        }
}

START
        iWrite("Introduza uma sequencia de numeros para as
        duas matrizes (2x2): ");
        exe input;
        i = 0;
        iWrite(" A matriz resultante é: \n");
        iLoop(i<dimensaoi){
                iWrite("[ ");
                j = 0;
                iLoop(j<dimensaoj){
                        matrizresultante[i][j] =
                            matriz[i][j] + matrizz[i][j];
                        iWrite(matrizresultante[i][j]);
                        iWrite(" , ");
                        j = j+1;
                }
                iWrite(" ]\n");
                i = i+1;
        }
END
```

5.3.2 Código fonte produzido

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

```
        pushn 4
        pushn 4
        pushn 4
jump inic
input: nop
        pushi 0
        storeg 0
        pushi 2
        storeg 2
        pushi 2
        storeg 3
LABEL2:         pushg 0
        pushg 2
        inf

        jz LABEL3
        pushi 0
        storeg 1
LABEL0:         pushg 1
        pushg 3
        inf

        jz LABEL1
        pushgp
        pushi 4
        padd
        pushg 0
        pushi 2
        mul
        pushg 1
        add
        read
        atoi
        storen
        pushg 1
        pushi 1
        add
        storeg 1
        jump LABEL0
LABEL1:         pushg 0
        pushi 1
        add
        storeg 0
        jump LABEL2
LABEL3:         pushi 0
        storeg 0
LABEL6:         pushg 0
```

```
        pushg 2
        inf

        jz LABEL7
        pushi 0
        storeg 1
LABEL4:         pushg 1
        pushg 3
        inf

        jz LABEL5
        pushgp
        pushi 8
        padd
        pushg 0
        pushi 2
        mul
        pushg 1
        add
        read
        atoi
        storen
        pushg 1
        pushi 1
        add
        storeg 1
        jump LABEL4
LABEL5:         pushg 0
        pushi 1
        add
        storeg 0
        jump LABEL6
LABEL7:         return
start
inic:         pushs "Introduza uma sequencia de numeros para as
duas matrizes (2x2): "
        writes
        pusha input
        call
        nop
        pushi 0
        storeg 0
        pushs " A matriz resultante é: \n"
        writes
LABEL11:         pushg 0
        pushg 2
        inf
```

```
        jz LABEL12
        pushs "[ "
        writes
        pushi 0
        storeg 1
LABEL9:         pushg 1
        pushg 3
        inf

        jz LABEL10
        pushgp
        pushi 12
        padd
        pushg 0
        pushi 2
        mul
        pushg 1
        add
        pushgp
        pushi 4
        padd
        pushg 0
        pushi 2
        mul
        pushg 1
        add
        loadn
        pushgp
        pushi 8
        padd
        pushg 0
        pushi 2
        mul
        pushg 1
        add
        loadn
        add
        storen
        pushgp
        pushi 12
        padd
        pushg 0
        pushi 2
        mul
        pushg 1
        add
```

```
        loadn
        writei
        pushg 1
        pushg 3
        pushi 1
        sub
        equal
pushi 1
inf
        jz LABEL8
        pushs " , "
        writes
LABEL8:         pushg 1
        pushi 1
        add
        storeg 1
        jump LABEL9
LABEL10:         pushs " ]\n"
        writes
        pushg 0
        pushi 1
        add
        storeg 0
        jump LABEL11
LABEL12: stop
```

5.3.3 Input/Output

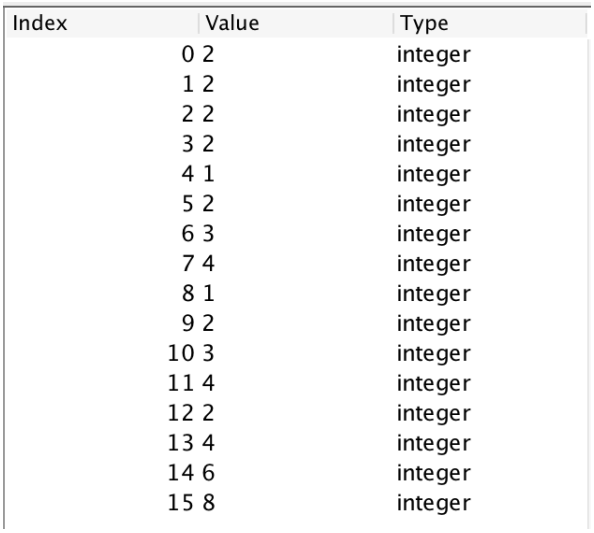
Input
De modo a estar o programa anterior utilizamos duas matrizes semelhantes constituídas pelos números 1, 2, 3, 4

```
"Introduza uma sequencia de numeros para as duas matrizes (2x2): "12341234"
```

Output
Sem grandes dificuldades percebemos que o resultado obtido é, de facto, a sequência inversa da inicialmente fornecida pelo *utilizador*.

```
Introduza uma sequencia de numeros para as duas matrizes (2x2):  A matriz resultante é:
[ 2 , 4 ]
[ 6 , 8 ]
```

Stack
Pela observação da *stack* identificamos facilmente os valores presentes em cada um dos elementos dos seus índices.



Index	Value	Type
0	2	integer
1	2	integer
2	2	integer
3	2	integer
4	1	integer
5	2	integer
6	3	integer
7	4	integer
8	1	integer
9	2	integer
10	3	integer
11	4	integer
12	2	integer
13	4	integer
14	6	integer
15	8	integer

Índice	**Valor**	**Variável**	**Significado**
0	2	i	Variável utilizada para iterar as linhas das matrizes.
1	2	j	Variável utilizada para iterar as colunas das matrizes.
2	2	dimensaoi	Número de linhas das matrizes.
3	2	dimensaoj	Número de colunas das matrizes.
4	1	matriz[0][0]	Posição (0,0) da primeira matriz.
5	2	matriz[0][1]	Posição (0,1) da primeira matriz.
6	3	matriz[1][0]	Posição (1,0) da primeira matriz.
7	4	matriz[1][1]	Posição (1,1) da primeira matriz.
8	1	matrizz[0][0]	Posição (0,0) da segunda matriz.
9	2	matrizz[0][1]	Posição (0,1) da segunda matriz.
10	3	matrizz[1][0]	Posição (1,0) da segunda matriz.
11	4	matrizz[1][1]	Posição (1,1) da segunda matriz.
12	2	resultante[0][0]	Posição (0,0) da matriz resultante.
13	4	resultante[0][1]	Posição (0,1) da matriz resultante.
14	6	resultante[1][0]	Posição (1,0) da matriz resultante.
15	8	resultante[0][1]	Posição (1,1) da matriz resultante.

6. *Conclusão*

Este que foi o último trabalho prático da unidade curricular de Processamento de Linguagens, foi, sem dúvida o que exigiu mais planeamento por parte do grupo, comparativamente aos restantes. Durante a realização deste, várias foram as etapas em que encontramos dificuldades, que tiveram de ser ultrapassadas para chegar ao resultado final conseguido.

A realização deste trabalho exigiu a aplicação de todos os conhecimentos consolidados anteriormente com o desenvolvimento dos demais trabalhos práticos, mais especificamente do segundo trabalho prático cujo tema passaria essencialmente pela elaboração de filtros de texto, em *Flex*.

Apesar do reduzido tempo, tendo em conta a fase do semestre em que nos encontramos, o grupo deu o seu melhor e apresenta um produto final do qual se consegue orgulhar, não descartando a possibilidade de, com mais tempo de trabalho, realizar algumas melhorias e implementar ainda mais e melhores funcionalidades à linguagem de programação final.

Contudo, e concluindo no geral, estamos satisfeitos com o trabalho desenvolvido, na medida que mesmo dentro de um tema com o qual os alunos do presente curso se encontram acostumados, as linguagens de programação, permitiu um contacto com estas de um modo que jamais tiveram tido, a sua construção. De um modo geral, consideramos que este trabalho prático se revelou bastante útil no aperfeiçoamento dos nossos conhecimentos neste âmbito, permitindo-nos adquirir bastante experiência no que toca ao tema em questão.