

UNIVERSIDADE DO MINHO  
MESTRADO INTEGRADO EM ENGENHARIA INFORMÁTICA



---

Processamento de Linguagens - TP2

---

GRUPO 29

*Trabalho realizado por:*

Adriana Martins Gonçalves  
Carlos Filipe Coelho Ferreira  
Joel Salgueiro Martins

*Número*

A75119  
A89542  
A89575

# Conteúdo

<b>1</b>	<b><i>Introdução</i></b>	<b>2</b>
<b>2</b>	<b><i>Desenvolvimento de compilador via Yacc/Lex</i></b>	<b>3</b>
2.1	Enunciado . . . . .	3
2.2	Descrição do problema . . . . .	4
2.3	Decisões de implementação . . . . .	4
2.3.1	compilador_lex . . . . .	4
2.3.2	compilador_yacc . . . . .	6
2.3.3	parsetab . . . . .	14
2.4	Resultados obtidos . . . . .	15
2.4.1	Ler 4 números e dizer se podem ser os lados de um quadrado . . . . .	15
2.4.2	Ler um inteiro N, depois ler N números e escrever o menor deles . . . . .	16
2.4.3	Ler N (constante do programa) números e calcular e imprimir o seu produtório . . . . .	18
2.4.4	Contar e imprimir os números ímpares de uma sequência de números naturais . . . . .	20
2.4.5	Ler e armazenar N números num array; imprimir os valores por ordem inversa . . . . .	22
2.4.6	Invocar e usar num programa seu uma função 'potencia()', que começa por ler do input a base B e o expoente E e retorna o valor $B^E$ . . . . .	23
<b>3</b>	<b><i>Conclusão</i></b>	<b>25</b>

## 1 *Introdução*

No âmbito da unidade curricular de Processamento de Linguagens foi-nos proposto a elaboração de um trabalho prático cuja finalidade é desenvolver um compilador para uma linguagem imperativa à nossa escolha com base na GIC e com recurso aos módulos Yacc/Lex do PLY/Python. Esse compilador deve ser capaz de gerar pseudo-código, Assembly da Máquina Virtual VM.

Dos 3 ciclos possíveis, o que nos foi atribuído como sendo de implementação obrigatória foi o 2, referente ao ciclo **for-do**, sendo decidido também suportar as funcionalidade sobre **arrays**.

Ao longo do relatório iremos explicar as decisões e abordagens que tomamos ao longo do desenvolvimento do projeto, de forma a obter o resultado final desejado.

## 2 *Desenvolvimento de compilador via Yacc/Lex*

### 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* variáveis atômicas do tipo *inteiro*, com os quais se podem realizar as habituais operações aritméticas, relacionais e lógicas.
- *efetuar* instruções algorítmicas básicas como a *atribuição do valor de expressões numéricas a variáveis*.
- *ler* do *standard input* e *escrever* no *standard output*.
- *efetuar* instruções *condicionais* para controlo do fluxo de execuções.
- *efetuar* instruções *cíclicas* para controlo do fluxo de execução, permitindo o seu aninhamento.

Note que deve implementar pelo menos o ciclo **while-do**, **repeat-until** ou **for-do** conforme o Número do seu Grupo módulo 3 seja 0, 1 ou 2.

Adicionalmente deve ainda suportar, à sua escolha, uma das duas funcionalidades seguintes:

- *declarar e manusear* variáveis do tipo *array* (a 1 ou 2 dimensões) de *inteiros*, em relação aos quais é apenas permitida a operação de indexação (índice inteiro).
- *definir e invocar subprogramas* sem parâmetros mas que possam retornar um resultado do tipo inteiro.

Como é da praxe neste tipo de linguagens, as variáveis deverão ser declaradas no início do programa e não pode haver re-declarações, nem utilizações sem declaração prévia. Se nada for explicado, o valor da variável após a declaração é 0 (zero).

Desenvolva, então, um compilador para essa linguagem com base na GIC criada acima e com recurso aos módulos Yacc/Lex do PLY/Python.

O compilador deve gerar **pseudo-código**, Assembly da Máquina Virtual VM.

Muito Importante:

Para a entrega do TP deve preparar um conjunto de testes (programas-fonte escritos na sua linguagem) e mostrar o código Assembly gerado bem como o programa a correr na máquina virtual VM. Esse conjunto terá de conter, no mínimo, os 4 primeiros exemplos abaixo e um dos 2 últimos conforme a sua escolha acima:

- ler 4 números e dizer se podem ser lados de um quadrado.
- ler um inteiro N, depois ler N números e escrever o menor deles.
- ler N (constante do programa) números e calcular e imprimir o seu produtório.
- contar e imprimir os números ímpares de uma sequência de números naturais.

- ler e armazenar  $N$  números num array; imprimir os valores por ordem inversa.
- invocar e usar num programa seu uma função 'potencia()', que começa por ler do input a base  $B$  e o expoente  $E$  e retorna o valor  $B^E$ .

## 2.2 Descrição do problema

Tal como referido no enunciado, todo este trabalho se resume à criação de um compilador para uma linguagem imperativa à nossa escolha, recorrendo à utilização dos módulos *yacc/lex* do *Phyton*.

Começamos então por criar um ficheiro *lex*, onde estão definidas as especificações da linguagem e é responsável por gerar analisador léxico. Seguidamente, passamos para a criação do ficheiro *yacc*. Este, depois de esperar pelas funções do *lex* para percorrer o texto a ser compilado, gera o *parser*. Finalmente o *parser* vai processar toda a informação permitindo fazer os testes necessário para comprovar a eficácia do compilador.

## 2.3 Decisões de implementação

Este capítulo contém as explicações de todas as decisões tomadas para o desenvolvimento e criação do compilador. Para tal foi necessário a criação de um ficheiro *lex* e outro *yacc*.

### 2.3.1 compilador\_lex

Este ficheiro *lex*, é um compilador de compiladores cuja função recai em gerar o analisador léxico. Neste sentido, o *lex* necessita da especificação da linguagem na forma de expressões regulares e ainda de um conjunto de funções adicionais para ser possível manipular os *tokens* gerados, particularmente funções com operações e fatores lógicos.

Assim sendo, o nosso ficheiro está organizado da seguinte forma:

- **reserved:** Define quais os caracteres reservados e de que forma estes estão definidos. Neste caso temos: 'if' como 'IF', 'else' como 'ELSE', 'while' como 'WHILE', 'print' como 'PRINT', 'int' como 'INT', 'read' como 'READ', 'true' como 'TRUE', 'false' como 'FALSE' e 'for' como 'FOR'.
- **tokens:** Contém um conjunto de parâmetros como por exemplo 'CONJ', 'DISJ', 'LOWEREQUAL', 'GREATEREQUAL', 'NOTEQUAL', 'EQUALS', 'ID', 'NUM', e ainda uma lista dos caracteres reservados ('list(reserved.values())' exemplificados no ponto acima.
- **literals:** Os literals permitem escrever um valor de forma mais fácil e sucinta. Estes representam as escolhas possíveis de tipos primitivos para a nossa linguagem. Algumas das opções de tipos literais que definimos foram: '!', '"', '>', '<', '(', ')', ',', '+', '-', '\*', '/', '=', '[]' e '%'.
  - **ER:** Este ficheiro tem definido 8 expressões regulares (ER), nomeadamente:
    - t\_NUM** = **r'\d+'** representa um ou mais dígitos entre 0-9;
    - t\_EQUALS** = **r'=='** representa a igualdade entre caracteres/funções;
    - t\_LOWEREQUAL** = **r'<='** verifica se um caracter/função é menor ou igual a outro;

`t_GREATEQUAL = r'>='` verifica se um caracter/função é maior ou igual a outro;  
`t_NOTEQUAL = r'!='` representa a desigualdade entre caracteres/funções;  
`t_CONJ = r'&&'` representa o operador lógico 'e';  
`t_DISJ = r'\\|\\|'` representa o operador lógico 'ou';

- **t\_ID:** Define o ID através da expressão regular (ER) `r'[a-z][a-zA-Z_0-9]*'`, que verifica uma palavra que no seu conjunto de caracteres pode ter 'a' a 'z' (no caso de ser minúscula) 'A' a 'Z' (no caso de ser maiúscula), ou ainda um dígito entre 0 e 9 e até mesmo o carácter '\_'. O ID tem de começar obrigatoriamente com uma letra minúscula. Verifica ainda que este não pode ser uma das palavras reservadas.
- **t\_error:** Responsável por imprimir a mensagem de erro 'Caracter ilegal' caso o caracter não seja válido.
- **t\_ignore:** Responsável por ignorar espaços e tabs.

Na imagem abaixo, é possível visualizar de que forma foi definido o ficheiro *compilador\_lex.py*.

```
import ply.lex as lex

reserved = {
    'if': 'IF',
    'else': 'ELSE',
    'while': 'WHILE',
    'print': 'PRINT',
    'int': 'INT',
    'read': 'READ',
    'true': 'TRUE',
    'false': 'FALSE',
    'for': 'FOR',
}

tokens = ['CONJ', 'DISJ', 'LOWEREQUAL', 'GREATEQUAL', 'NOTEQUAL', 'EQUALS', 'ID', 'NUM'] + list(reserved.values())
literals = ['!', '{', '}', '>', '<', '(', ')', ',', '+', '-', '*', '/', '=', ';', '[', ']', '%']

t_NUM = r'\d+'
t_EQUALS = r'=='
t_LOWEREQUAL = r'<='
t_GREATEQUAL = r'>='
t_NOTEQUAL = r'!='
t_CONJ = r'&&'
t_DISJ = r'\\|\\|'

def t_ID(t):
    r'[a-z][a-zA-Z_0-9]*'
    t.type = reserved.get(t.value, 'ID') # Check for reserved words
    return t

t_ignore = " \n\t"

def t_error(t):
    print('Carater ilegal: ', t.value[0])
    t.lexer.skip(1)

# Controlo o lexer
lexer = lex.lex()
```

Figura 1: Código do *compilador\_lex*

### 2.3.2 compilador\_yacc

O ficheiro *yacc* é onde se encontra definida a nossa gramática, para que com a ajuda do *flex* já definido possa criar o *parser* e assim gerar o pseudo-código *Assembly* para a VM.

A primeira função é a *p\_Programa* e é composta por um bloco declarativo onde se deve declarar todas as variáveis e arrays que serão usadas durante a execução do programa, e um bloco instrutivo com todas as instruções como por exemplo ciclos, ifs, prints ou operações aritméticas. Ambos os blocos são separados pelas instruções 'START' e 'STOP' do pseudo-código Assembly.

```
def p_Programa(p):  
    "Programa : BlocoDeclarativo BlocoInstrutivo"  
    p[0] = p[1] + "START\n" + p[2] + "STOP\n"  
    #pass
```

Figura 2: Código da função *p\_Program*

Na imagem abaixo encontram-se as funções relativas aos blocos declarativos funcionando de forma semelhante a uma lista. Temos a *p\_BlocoDeclarativo\_fim* que é neste caso a lista vazia e a *p\_BlocoDeclarativo* que é composta por um bloco declarativo, ou seja, uma lista mais um elemento declaração.

```
def p_BlocoDeclarativo_fim(p):  
    " BlocoDeclarativo : "  
    p[0] = ""  
  
def p_BlocoDeclarativo(p):  
    " BlocoDeclarativo : BlocoDeclarativo Declaracao ';' "  
    p[0] = p[1] + p[2]
```

Figura 3: Código das funções relativas aos blocos declarativos

Na imagem abaixo encontram-se as funções relativas aos blocos instrutivos. Implementadas de forma semelhante aos blocos declarativos.

```
def p_BlocoInstrutivo_fim(p):  
    " BlocoInstrutivo : "  
    p[0] = ""  
  
def p_BlocoInstrutivo(p):  
    " BlocoInstrutivo : BlocoInstrutivo Instrucao ';' "  
    p[0] = p[1] + p[2]
```

Figura 4: Código das funções relativas aos blocos instrutivos

Quanto às funções relativas às declarações, temos a *p\_Declaracao* e a *p\_Declaracao\_array*. A diferença entre estas duas funções é que a segunda função serve para declarar um array de números usando a instrução 'PUSHN'. A primeira é usada para declarar variáveis que como só ocupam um espaço na stack usa-se a função 'PUSHI 0'. Para cada variável declarada

é necessário guardar a sua posição em relação ao global point num dicionário para futuras operações.

```
def p_Declaracao_d(p):
    "Declaracao : INT ID"
    parser.declaracoes[p[2]] = parser.totDeclar
    parser.totDeclar += 1
    p[0] = "PUSHI 0" + "\n"

def p_Declaracao_array(p):
    "Declaracao : INT ID '[' NUM ']"
    parser.declaracoes[p[2]] = parser.totDeclar
    parser.totDeclar += int(p[4])
    p[0] = "PUSHN " + p[4] + "\n"
```

Figura 5: Código das funções relativas às declarações

Como primeira função definida temos a *p\_Instrucao\_atrib* em que nos diz que uma atribuição é uma instrução. As próximas duas funções são responsáveis por executar as condições 'IF/THEN/ELSE', sendo que em *Assembly* estas necessitarão do uso de Labels aos quais devem ser etiquetadas para impedir a existência de 2 labels com o mesmo nome e também instruções do tipo 'JZ' e 'JUMP' para permitir o salto com destino as labels. Explicando um pouco o funcionamento de saltos como podemos ver pela *p\_Instrucao\_if* após aplicar a condição e ter o resultado verdade ou falso no topo da stack, se este for falso ou seja igual a 0 salta para a label end do ciclo.

Quanto à *p\_Instrucao\_for* e *p\_Instrucao\_while* funcionam de forma semelhante ao qual estamos todos habituados usando os mesmos mecanismos utilizados na condição IF/THEN/ELSE.

A funções do tipo *p\_Instrucao\_print* são responsáveis por fazer a escrita de variáveis, números ou elementos de um array, como podemos ver pela primeira vez, para conhecer os valores das variáveis representadas por um ID é necessário conhecer a sua posição relativa ao global point, e tal é possível devido a sua introdução no dicionário durante a sua declaração. Qualquer variável que não esteja contida no dicionário é porque não foi declarada sendo por isso necessário mostrar um erro de compilação.



```

def p_Instrucao_atrib(p):
    "Instrucao : Atribuicao"
    p[0] = p[1]

def p_Instrucao_if_else(p):
    "Instrucao : IF '(' Condicao ')' '{' BlocoInstrutivo '}' ELSE '{' BlocoInstrutivo '}'"
    parser.totEtiquetas += 1
    x = str(parser.totEtiquetas)
    p[0] = p[3] + "JZ else" + x + "\n" + p[6] + "JUMP end" + x + "\n" + "else" + x + ":\n" + p[10] + "end" + x + ":\n"

def p_Instrucao_if(p):
    "Instrucao : IF '(' Condicao ')' '{' BlocoInstrutivo '}'"
    parser.totEtiquetas += 1
    x = str(parser.totEtiquetas)
    p[0] = p[3] + "JZ end" + x + "\n" + p[6] + "end" + x + ":\n"

def p_Instrucao_for(p):
    "Instrucao : FOR '(' Atribuicao ';' Condicao ';' Atribuicao ')' '{' BlocoInstrutivo '}'"
    parser.totEtiquetas += 1
    x = str(parser.totEtiquetas)
    p[0] = p[3] + "startFor" + x + ":\n" + p[5] + "JZ endFor" + x + "\n" + p[10] + p[7] + "JUMP startFor" + x + "\n" + "endFor" + x + ":\n"

def p_Instrucao_while(p):
    "Instrucao : WHILE '(' Condicao ')' '{' BlocoInstrutivo '}'"
    parser.totEtiquetas += 1
    x = str(parser.totEtiquetas)
    p[0] = "startWhile" + x + ":\n" + p[3] + "JZ endWhile" + x + "\n" + p[6] + "JUMP startWhile" + x + "\n" + "endWhile" + x + ":\n"

def p_Instrucao_print(p):
    "Instrucao : PRINT '(' ID ')'"
    if parser.declaracoes.__contains__(p[3]):
        x = parser.declaracoes[p[3]]
        p[0] = "PUSHG " + str(x) + "\nWRITEI\n"
    else:
        p[0] = ""
        print("Erro ", p[3], "' nao esta declarado!")

def p_Instrucao_print_num(p):
    "Instrucao : PRINT '(' Exp ')'"
    p[0] = p[3] + "WRITEI\n"

def p_Instrucao_printArray(p):
    "Instrucao : PRINT '(' ID '[' Exp ']' ')'"
    if parser.declaracoes.__contains__(p[3]):
        x = parser.declaracoes[p[3]]
        p[0] = "PUSHGP\n" + "PUSHI " + str(x) + "\n" + p[5] + "ADD\n" + "LOADN\n" + "WRITEI\n"
    else:
        p[0] = ""
        print("Erro ", p[3], "' nao esta declarado!")

```

Figura 6: Código das funções relativas às instruções

Relativamente às funções das atribuições, a *p\_Atribuicao\_exp* faz a atribuição de uma expressão a uma certa variável, por outras palavras, atualiza uma variável para um novo valor que tem de ser um inteiro, a *p\_Atribuicao\_expArray* atribui um valor a uma dada posição no array.

A *p\_Atribuicao\_read* e *p\_Atribuicao\_readArray* são ambas também atribuições, mas o valor ao qual devem atualizar a variável não provem de contas ou cálculos, mas são dados como input pelo utilizador.

```

def p_Atribuicao_exp(p):
    " Atribuicao : ID '=' Exp"
    if parser.declaracoes.__contains__(p[1]):
        x = parser.declaracoes[p[1]]
        p[0] = p[3] + "STOREG " + str(x) + "\n"
    else:
        p[0] = ""
        print("Erro ", p[1], "' nao esta declarado!")

def p_Atribuicao_expArray(p):
    " Atribuicao : ID '[' Exp ']' '=' Exp"
    if parser.declaracoes.__contains__(p[1]):
        x = parser.declaracoes[p[1]]
        p[0] = "PUSHGP\n" + "PUSHI " + str(x) + "\n" + p[3] + "ADD\n" + p[6] + "STOREN\n"
    else:
        p[0] = ""
        print("Erro ", p[1], "' nao esta declarado!")

def p_Atribuicao_read(p):
    " Atribuicao : ID '=' READ '(' ' )'"
    if parser.declaracoes.__contains__(p[1]):
        x = parser.declaracoes[p[1]]
        p[0] = "READ\n" + "ATOI\n" + "STOREG " + str(x) + "\n"
    else:
        p[0] = ""
        print("Erro ", p[1], "' nao esta declarado!")

def p_Atribuicao_readArray(p):
    " Atribuicao : ID '[' Exp ']' '=' READ '(' ' )'"
    if parser.declaracoes.__contains__(p[1]):
        x = str(parser.declaracoes[p[1]])
        p[0] = "PUSHGP\n" + "PUSHI " + x + "\n" + p[3] + "ADD\n" + "READ\n" + "ATOI\n" + "STOREN\n"
    else:
        p[0] = ""
        print("Erro ", p[1], "' nao esta declarado!")

```

Figura 7: Código das funções relativas às atribuições

Para suportar a parte lógica necessária ao uso dos IF/THEN/ELSE, WHILE e FOR, foi necessário então a criação de duas novas entidades Condição e Afirmação.

A Condição é a entidade encarregue de operações sobre uma ou mais afirmações com um respectivo valor lógico. A *p\_Condicao\_igual* por exemplo verifica se 2 valores lógicos são equivalentes.

Todas estas operações devem ser transformadas em linguagem Assembly, a *p\_Condicao\_conj*, por exemplo, é realizado usando a operação MUL que multiplica 2 valores do topo da stack. Isto resulta, pois se um dos 2 valores for 0 "falso" então o resultado é 0 "falso". Podemos ver um processo semelhante em *p\_Condicao\_disj* onde a adição de 2 valores da stack só devolve um valor positivo "verdadeiro" se pelo menos um dos valores for maior que zero.

Por fim, uma condição também pode ser somente uma afirmação de valor logico verdadeiro ou falso como podemos ver na função *p\_Condicao\_afirm*.

```

def p_Condicao_igual(p):
    "Condicao : Condicao EQUALS Afirmacao"
    p[0] = p[1] + p[3] + "EQUAL\n"

def p_Condicao_conj(p):
    "Condicao : Condicao CONJ Afirmacao"
    p[0] = p[1] + p[3] + "MUL\n"

def p_Condicao_disj(p):
    "Condicao : Condicao DISJ Afirmacao"
    p[0] = p[1] + p[3] + "ADD\n"

def p_Condicao_desigual(p):
    "Condicao : Condicao NOTEQUAL Afirmacao"
    p[0] = p[1] + p[3] + "EQUAL\n" + "NOT\n"

def p_Condicao_afirm(p):
    "Condicao : Afirmacao"
    p[0] = p[1]

```

Figura 8: Código das funções relativas às condições

Quanto às funções relativas a afirmações, foi necessário definir 9.

A primeira função é a *p\_Afirmacao\_Negacao*, cuja finalidade é negar essa mesma afirmação usando a instrução 'NOT'.

A segunda função é a *p\_Afirmacao\_True* e corresponde ao operador 'True' através da instrução lógico 'PUSHI 1' onde 1 como é positivo corresponde a verdadeiro.

A terceira função é a *p\_Afirmacao\_False* e segue o mesma linha de raciocínio da função anterior mas corresponde ao operador 'False' usando 'PUSHI 0' que como é nulo corresponde a falso.

A quarta e a quinta funções são a *p\_Afirmacao\_Greater* e a *p\_Afirmacao\_Lower*, e verificam se uma expressão ou seja um inteiro é maior ou menor (respetivamente) ao outro.

As funções *p\_Afirmacao\_GreaterEQ* e *p\_Afirmacao\_LowerEQ* diferem das duas anteriores apenas por verificarem se, para além de serem maiores ou menores, são ainda igual.

As duas últimas funções são a *p\_Afirmacao\_Equals* e *p\_Afirmacao\_notEquals* e servem para verificar se duas expressões, ou seja, no cenário da Maquina Virtual dois valores no topo da stack, são iguais ou não. Enquanto que no caso de igualdade a operação lógica a usar em assembly será 'EQUALS', no caso de testar desigualdade será operação 'EQUALS' seguido de "NOT" a operação da negação.

Na imagem abaixo encontram-se definidas todas as funções explicadas anteriormente.

```

def p_Afirmacao_Negacao(p):
    "Afirmacao : '!' Afirmacao"
    p[0] = p[2] + "NOT\n"

def p_Afirmacao_True(p):
    "Afirmacao : TRUE"
    p[0] = "PUSHI 1\n"

def p_Afirmacao_False(p):
    "Afirmacao : FALSE"
    p[0] = "PUSHI 0\n"

def p_Afirmacao_Greater(p):
    "Afirmacao : Exp '>' Exp"
    p[0] = p[1] + p[3] + "SUP\n"

def p_Afirmacao_Lower(p):
    "Afirmacao : Exp '<' Exp"
    p[0] = p[1] + p[3] + "INF\n"

def p_Afirmacao_GreaterEQ(p):
    "Afirmacao : Exp GREATEREQUAL Exp"
    p[0] = p[1] + p[3] + "SUPEQ\n"

def p_Afirmacao_LowerEQ(p):
    "Afirmacao : Exp LOWEREQUAL Exp"
    p[0] = p[1] + p[3] + "INFEQ\n"

def p_Afirmacao_Equals(p):
    "Afirmacao : Exp EQUALS Exp"
    p[0] = p[1] + p[3] + "EQUAL\n"

def p_Afirmacao_notEquals(p):
    "Afirmacao : Exp NOTEQUAL Exp"
    p[0] = p[1] + p[3] + "EQUAL\n" + "NOT\n"

```

Figura 9: Código das funções relativas às afirmações

Na imagem abaixo encontram-se as funções relativas às soma e subtração de expressões. A primeira é a ***p\_Exp\_add***, cuja função é fazer a adição entre expressões, em *Assembly* terá como resultado um conjunto de instruções formado pelo resultado da derivação da Expressão, seguido do resultado da derivação do Termo, seguido de um 'ADD', que irá adicionar os dois valores do topo da stack. A ***p\_Exp\_sub*** usa a mesma lógica anterior, no entanto, substitui agora o ADD por SUB, pois queremos subtrair os dois últimos valores da stack. A ***p\_Exp*** é usada nos casos em que uma Expressão deriva num Termo, não acrescentando qualquer instrução.

```

def p_Exp_add(p):
    " Exp : Exp '+' Termo"
    p[0] = p[1] + p[3] + "ADD\n"

def p_Exp_sub(p):
    " Exp : Exp '-' Termo"
    p[0] = p[1] + p[3] + "SUB\n"

def p_Exp(p):
    "Exp : Termo"
    p[0] = p[1]

```

Figura 10: Código das funções relativas às expressões

A *p\_Termo\_mul*, responsável pela derivação de uma multiplicação, portanto, de um termo multiplicado por um fator. Gera para a VM um conjunto de instruções resultante da derivação do termo e do fator, acrescentado da instrução MUL, que irá multiplicar os dois últimos valores da stack. A *p\_Termo\_div* faz uso da lógica anterior, só que agora com a instrução 'DIV', que irá dividir os dois últimos valores da stack. A *p\_Termo\_mod* também irá dividir os dois últimos valores, no entanto, será agora uma divisão inteira, portanto com a instrução MOD. Finalmente, a *p\_Termo* diz-nos apenas que um termo deriva num fator, não acrescentando qualquer instrução.

```

def p_Termo_mul(p):
    "Termo : Termo '*' Fator"
    p[0] = p[1] + p[3] + "MUL\n"

def p_Termo_div(p):
    "Termo : Termo '/' Fator"
    p[0] = p[1] + p[3] + "DIV\n"

def p_Termo_mod(p):
    "Termo : Termo '%' Fator"
    p[0] = p[1] + p[3] + "MOD\n"

def p_Termo(p):
    "Termo : Fator"
    p[0] = p[1]

```

Figura 11: Código das funções relativas aos termos

A *p\_Fator\_varArray* cria o conjunto de instruções de forma a que seja apresentado o valor que está num determinado array, numa determinada posição. Para isso, fazemos o push do global pointer, seguidamente do push do pointer correspondente ao início do array, seguidamente colocamos o resultado da expressão que irá ditar a posição do array, fazemos a soma desse valor com o pointer do array e, por fim, fazemos um load do valor que está na posição resultante da soma do pointer (global pointer) que está na penúltima posição da stack, com o valor que está na última posição da stack, sendo adicionada assim, no topo da stack o valor que estava na respetiva posição do array. A *p\_Fator\_num* faz push do número resultante da derivação do fator, a *p\_Fator\_var* faz push do endereço da variável derivada do fator e, por fim, a *p\_Fator* é a derivação de fator numa expressão, pelo que não passa nenhuma instrução para a VM.

```

def p_Fator_varArray(p):
    "Fator : ID '[' Exp ']'"
    x = parser.declaracoes[p[1]]
    p[0] = "PUSHGP\n" + "PUSHI " + str(x) + "\n" + p[3] + "ADD\n" + "LOADN\n"

def p_Fator_num(p):
    "Fator : NUM"
    p[0] = "PUSHI " + p[1] + "\n"

def p_Fator_var(p):
    "Fator : ID"
    x = parser.declaracoes[p[1]]
    p[0] = "PUSHG " + str(x) + "\n"

def p_Fator(p):
    "Fator : '(' Exp ')'"
    p[0] = p[2]

```

Figura 12: Código relativo às funções dos fatores

No caso de ocorrer algum erro ao nível da sintaxe, será informado através da mensagem 'Erro sintatico' com o respetivo erro, e não será possível gerar o *parser* com sucesso.

```

def p_error(p):
    print('Erro sintatico: ', p)
    parser.sucesso = False

```

Figura 13: Código responsável por informar se existem erros sintáticos

Na imagem abaixo está representado o código responsável pela construção do *parser*. Caso tudo corra bem, deverá aparecer a mensagem 'Válido', o *input* dado e o respetivo resultado, caso contrário surgirá a mensagem 'Inválido' e o *input* correspondente.

```

# Constroi o parser
parser = yacc.yacc()

parser.sucesso = True
parser.totDeclar = 0
parser.totEtiquetas = 0
parser.declaracoes = {}

# Consome input
inp = ""
for l in sys.stdin:
    inp += l

parser.sucesso = True
result = parser.parse(inp)

if parser.sucesso:
    print('Valido: ', inp)
    print(result)
else:
    print('Invalido: ', inp)

```

Figura 14: Código responsável por criar o *parser*

### 2.3.3 parsetab

No final da criação dos ficheiros *compilador\_flex* e *compilador\_yacc*, onde foi definida toda a gramática da linguagem e foi gerado o analisador léxico depois de especificada a linguagem sob a forma de expressões regulares, o *parser* é gerado automaticamente sob a forma de ficheiro *parsetab*.

Na imagem abaixo podemos ver de que forma está estruturado o *parser*.

```

_lr_method = 'LALR'

_lr_signature = "CONJ DISJ ELSE EQUALS FALSE FOR GREATEREQUAL ID IF INT LOWEREQUAL NOTEQUAL NUM PRINT READ TRUE WHILEPrograma : BlocoDeclarativo E"

_lr_action_items = {'INT':([0,2,13],[2,5,-3]),'IF':([0,2,3,13,15,70,88,92,95,107,108,109,110],[2,-4,8,-3,-5,-4,-4,8,8,-4,-4,8,8]),'FOR':([0,2,3,13,15,70,88,92,95,107,108,109,110],[2,-4,8,-3,-5,-4,-4,8,8,-4,-4,8,8]),'IF':([0,2,3,13,15,70,88,92,95,107,108,109,110],[2,-4,8,-3,-5,-4,-4,8,8,-4,-4,8,8]),'FOR':([0,2,3,13,15,70,88,92,95,107,108,109,110],[2,-4,8,-3,-5,-4,-4,8,8,-4,-4,8,8])}

_lr_action = {}
for _k, _v in _lr_action_items.items():
    for _x, _y in zip(_v[0], _v[1]):
        if not _x in _lr_action:
            _lr_action[_x] = {}
        _lr_action[_x][_k] = _y
del _lr_action_items

_lr_goto_items = {'Programa':([0],[1]),'BlocoDeclarativo':([0],[2]),'BlocoInstrutivo':([2,70,88,107,108],[3,92,95,109,110]),'Declaracao':([2,70,88,107,108],[3,92,95,109,110])}

_lr_goto = {}
for _k, _v in _lr_goto_items.items():
    for _x, _y in zip(_v[0], _v[1]):
        if not _x in _lr_goto:
            _lr_goto[_x] = {}
        _lr_goto[_x][_k] = _y
del _lr_goto_items

_lr_productions = [
    ("S' -> Programa", "S'", 1, None, None, None),
    ('Programa -> BlocoDeclarativo BlocoInstrutivo', 'Programa', 2, 'p_Programa', 'compilador_yacc.py', 39),
    ('BlocoDeclarativo -> <empty>', 'BlocoDeclarativo', 0, 'p_BlocoDeclarativo_fim', 'compilador_yacc.py', 44),
    ('BlocoDeclarativo -> BlocoDeclarativo Declaracao ;', 'BlocoDeclarativo', 3, 'p_BlocoDeclarativo', 'compilador_yacc.py', 49),
    ('BlocoInstrutivo -> <empty>', 'BlocoInstrutivo', 0, 'p_BlocoInstrutivo_fim', 'compilador_yacc.py', 54),
    ('BlocoInstrutivo -> BlocoInstrutivo Instrucao ;', 'BlocoInstrutivo', 3, 'p_BlocoInstrutivo', 'compilador_yacc.py', 59),
    ('Declaracao -> INT ID', 'Declaracao', 2, 'p_Declaracao_d', 'compilador_yacc.py', 64),
    ('Declaracao -> INT ID [ NUM ]', 'Declaracao', 5, 'p_Declaracao_array', 'compilador_yacc.py', 70),
    ('Instrucao -> Atribuicao', 'Instrucao', 1, 'p_Instrucao_atrib', 'compilador_yacc.py', 76),
    ('Instrucao -> IF ( Condicao ) { BlocoInstrutivo } ELSE { BlocoInstrutivo }', 'Instrucao', 11, 'p_Instrucao_if_else', 'compilador_yacc.py', 81),
    ('Instrucao -> IF ( Condicao ) { BlocoInstrutivo }', 'Instrucao', 7, 'p_Instrucao_if', 'compilador_yacc.py', 87),
    ('Instrucao -> FOR ( Atribuicao ; Condicao ; Atribuicao ) { BlocoInstrutivo }', 'Instrucao', 11, 'p_Instrucao_for', 'compilador_yacc.py', 93),
    ('Instrucao -> WHILE ( Condicao ) { BlocoInstrutivo }', 'Instrucao', 7, 'p_Instrucao_while', 'compilador_yacc.py', 100),
    ('Instrucao -> PRINT ( ID )', 'Instrucao', 4, 'p_Instrucao_print', 'compilador_yacc.py', 107),
    ('Instrucao -> PRINT ( Exp )', 'Instrucao', 4, 'p_Instrucao_print_num', 'compilador_yacc.py', 116),
    ('Instrucao -> PRINT ( ID [ Exp ] )', 'Instrucao', 7, 'p_Instrucao_printArray', 'compilador_yacc.py', 121),
    ('Atribuicao -> ID = Exp', 'Atribuicao', 3, 'p_Atribuicao_exp', 'compilador_yacc.py', 130),
    ('Atribuicao -> ID [ Exp ] = Exp', 'Atribuicao', 6, 'p_Atribuicao_expArray', 'compilador_yacc.py', 140),
    ('Atribuicao -> ID = READ ( )', 'Atribuicao', 5, 'p_Atribuicao_read', 'compilador_yacc.py', 149),
    ('Atribuicao -> ID [ Exp ] = READ ( )', 'Atribuicao', 8, 'p_Atribuicao_readArray', 'compilador_yacc.py', 159),
    ('Condicao -> Condicao EQUALS Afirmacao', 'Condicao', 3, 'p_Condicao_igual', 'compilador_yacc.py', 168),
    ('Condicao -> Condicao CONJ Afirmacao', 'Condicao', 3, 'p_Condicao_conj', 'compilador_yacc.py', 172),
    ('Condicao -> Condicao DISJ Afirmacao', 'Condicao', 3, 'p_Condicao_disj', 'compilador_yacc.py', 176),
    ('Condicao -> Condicao NOTEQUAL Afirmacao', 'Condicao', 3, 'p_Condicao_desigual', 'compilador_yacc.py', 181),
    ('Condicao -> Afirmacao', 'Condicao', 1, 'p_Condicao_afirm', 'compilador_yacc.py', 186),
    ('Afirmacao -> ! Afirmacao', 'Afirmacao', 2, 'p_Afirmacao_Negacao', 'compilador_yacc.py', 191),
    ('Afirmacao -> TRUE', 'Afirmacao', 1, 'p_Afirmacao_True', 'compilador_yacc.py', 196),
    ('Afirmacao -> FALSE', 'Afirmacao', 1, 'p_Afirmacao_False', 'compilador_yacc.py', 201),
    ('Afirmacao -> Exp > Exp', 'Afirmacao', 3, 'p_Afirmacao_Greater', 'compilador_yacc.py', 206),
    ('Afirmacao -> Exp < Exp', 'Afirmacao', 3, 'p_Afirmacao_Lower', 'compilador_yacc.py', 211),
    ('Afirmacao -> Exp GREATEREQUAL Exp', 'Afirmacao', 3, 'p_Afirmacao_GreaterEQ', 'compilador_yacc.py', 216),
    ('Afirmacao -> Exp LOWEREQUAL Exp', 'Afirmacao', 3, 'p_Afirmacao_LowerEQ', 'compilador_yacc.py', 221),
    ('Afirmacao -> Exp EQUALS Exp', 'Afirmacao', 3, 'p_Afirmacao_Equals', 'compilador_yacc.py', 226),
    ('Afirmacao -> Exp NOTEQUAL Exp', 'Afirmacao', 3, 'p_Afirmacao_notEquals', 'compilador_yacc.py', 232),
    ('Exp -> Exp + Termo', 'Exp', 3, 'p_Exp_add', 'compilador_yacc.py', 238),
    ('Exp -> Exp - Termo', 'Exp', 3, 'p_Exp_sub', 'compilador_yacc.py', 243),
    ('Exp -> Termo', 'Exp', 1, 'p_Exp', 'compilador_yacc.py', 248),
    ('Termo -> Termo * Fator', 'Termo', 3, 'p_Termo_mul', 'compilador_yacc.py', 253),
    ('Termo -> Termo / Fator', 'Termo', 3, 'p_Termo_div', 'compilador_yacc.py', 258),
    ('Termo -> Termo % Fator', 'Termo', 3, 'p_Termo_mod', 'compilador_yacc.py', 263),
    ('Termo -> Fator', 'Termo', 1, 'p_Termo', 'compilador_yacc.py', 267),
    ('Fator -> ID [ Exp ]', 'Fator', 4, 'p_Fator_varArray', 'compilador_yacc.py', 272),
    ('Fator -> NUM', 'Fator', 1, 'p_Fator_num', 'compilador_yacc.py', 277),
    ('Fator -> ID', 'Fator', 1, 'p_Fator_var', 'compilador_yacc.py', 281),
    ('Fator -> ( Exp )', 'Fator', 3, 'p_Fator', 'compilador_yacc.py', 287),
]

```

Figura 15: Código do ficheiro *parsetab*

## 2.4 Resultados obtidos

Depois de explicado o processo de desenvolvimento de todo o projeto, nesta secção iremos apresentar os resultados obtidos.

### 2.4.1 Ler 4 números e dizer se podem ser os lados de um quadrado

Na *query* 1 era pedido que depois de ler 4 números, fosse possível saber se se tratavam de lados de um quadrado. Para isso, foi necessário serem lidos 4 números (através do **'read'**) e verificar se são iguais para serem considerados lados de um quadrado. Na imagem abaixo encontra-se o *input* para o parser ler de forma a gerar o pseudo-código e responder a esta *query*.

```
int atual; int i; int verdade; int anterior;

verdade = 1;
anterior = read();

for (i = 0; i<3; i = i + 1) {
    atual = read();
    if (atual != anterior) {
        verdade = 0;
    };
};

print(verdade);
```

Figura 16: *Input* a ler pelo parser de forma a verificar se os números fornecidos correspondem a lados de um quadrado

Como podemos verificar na próxima imagem, o compilador conseguiu gerar o pseudo-código que corre na máquina virtual (VM), com todas as instruções em *Assembly*.



```

output.vm
PUSHI 0
PUSHI 0
PUSHI 0
PUSHI 0
START
PUSHI 1
STOREG 2
READ
ATOI
STOREG 3
PUSHI 0
STOREG 1
startFor2:
PUSHG 1
PUSHI 3
INF
JZ endFor2
READ
ATOI
STOREG 0
PUSHG 0
PUSHG 3
EQUAL
NOT
JZ end1
PUSHI 0
STOREG 2
end1:
PUSHG 1
PUSHI 1
ADD
STOREG 1
JUMP startFor2
endFor2:
PUSHG 2
WRITEI
STOP

```

Figura 17: Pseudo-código gerado *query* 1

#### 2.4.2 Ler um inteiro N, depois ler N números e escrever o menor deles

Na *query* 2, começamos por ler um inteiro N (através de '**read**'), e através do ciclo '**for**' vai sempre avaliando qual o menor elementos que esta a ser lido comparativamente a todos os números lidos. Na imagem abaixo é possível ver esse código para posteriormente ser lido pelo parser.

```

int x; int i; int num; int menor;

num = read();

for (i = 0; i < num; i = i + 1) {
    x = read();

    if (i == 0) {
        menor = x;
    } else {
        if (x < menor) {
            menor = x;
        };
    };
};

print(menor);

```

Figura 18: *Input* a ser lido pelo parser de forma a devolver o menor de N números

Tal como aconteceu na *query* 2, o compilador conseguiu gerar o pseudo-código que corre na máquina virtual (VM) de forma a responder a esta questão.

```

output.vm
PUSHI 0
PUSHI 0
PUSHI 0
PUSHI 0
START
READ
ATOI
STOREG 2
PUSHI 0
STOREG 1
startFor3:
PUSHG 1
PUSHG 2
INF
JZ endFor3
READ
ATOI
STOREG 0
PUSHG 1
PUSHI 0
EQUAL
JZ else2
PUSHG 0
STOREG 3
JUMP end2
else2:
PUSHG 0
PUSHG 3
INF
JZ end1
PUSHG 0
STOREG 3
end1:
end2:
PUSHG 1
PUSHI 1
ADD
STOREG 1
JUMP startFor3
endFor3:
PUSHG 3
WRITEI
STOP

```

Figura 19: Pseudo-código gerado *query 2*

### 2.4.3 Ler N (constante do programa) números e calcular e imprimir o seu produto

Para a *query 3* é pedido que se leia N números e imprimir o seu produto. Neste caso, cada vez que se lê um número (através de '**read**'), faz-se o produto com o valor anterior até não haver mais números para serem lidos (através de '**prod=prod\*num**').

```

int x; int i; int num; int prod;

prod = 1;
x = read();

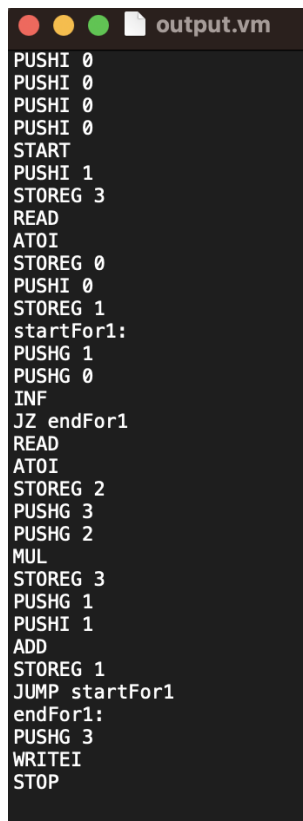
for (i = 0; i<x; i = i + 1) {
    num = read();
    prod = prod * num;
};

print(prod);

```

Figura 20: *Input* a ser lido pelo parser de forma a calcular o produto de N números

Na imagem abaixo encontra-se o código que foi gerado pelo compilador na linguagem *Assembly* a ser corrido na máquina virtual.



```

PUSHI 0
PUSHI 0
PUSHI 0
PUSHI 0
START
PUSHI 1
STOREG 3
READ
ATOI
STOREG 0
PUSHI 0
STOREG 1
startFor1:
PUSHG 1
PUSHG 0
INF
JZ endFor1
READ
ATOI
STOREG 2
PUSHG 3
PUSHG 2
MUL
STOREG 3
PUSHG 1
PUSHI 1
ADD
STOREG 1
JUMP startFor1
endFor1:
PUSHG 3
WRITEI
STOP

```

Figura 21: Pseudo-código gerado *query 3*

#### 2.4.4 Contar e imprimir os números ímpares de uma sequência de números naturais

A *query* 4 consiste em contar e imprimir os números ímpares de uma sequência de números naturais (inteiros não-negativos). Para tal, o programa começa por verificar quais dos números são ímpares (através de '**num % 2!=0**'), depois devolve a sequência desses números (através de '**impares[cont] = num**') e posteriormente verifica quantos números no total se enquadram nessa premissa (através de '**cont = cont +1**'). O código a ler pelo parser encontra-se na imagem abaixo.

```
int impares[100]; int i; int num; int cont; int x;

x = read();

for (i = 0; i<x; i = i + 1) {
    num = read();

    if (num % 2 != 0) {
        impares[cont] = num;
        cont = cont + 1;
    };
};

for (i = 0; i<cont; i = i + 1) {
    print(impares[i]);
};

print(cont);
```

Figura 22: *Input* a ser lido pelo parser de forma a contar e imprimir os ímpares de uma sequência de números

Tal como aconteceu nas *queries* anteriores, o compilador conseguiu gerar o pseudo-código que corre na máquina virtual (VM) de forma a responder a esta questão.

```

output.vm
PUSHN 100
PUSHI 0
PUSHI 0
PUSHI 0
PUSHI 0
START
READ
ATOI
STOREG 103
PUSHI 0
STOREG 100
startFor2:
PUSHG 100
PUSHG 103
INF
JZ endFor2
READ
ATOI
STOREG 101
PUSHG 101
PUSHI 2
MOD
PUSHI 0
EQUAL
NOT
JZ end1
PUSHGP
PUSHI 0
PUSHG 102
ADD
PUSHG 101
STOREN
PUSHG 102
PUSHI 1
ADD
STOREG 102
end1:
PUSHG 100
PUSHI 1
ADD
STOREG 100
JUMP startFor2
endFor2:
PUSHI 0
STOREG 100
startFor3:
PUSHG 100
PUSHG 102
INF
JZ endFor3
PUSHGP
PUSHI 0
PUSHG 100
ADD
LOADN
WRITEI
PUSHG 100
PUSHI 1
ADD
STOREG 100
JUMP startFor3
endFor3:
PUSHG 102
WRITEI
STOP

```

Figura 23: Pseudo-código gerado *query* 4

#### 2.4.5 Ler e armazenar N números num array; imprimir os valores por ordem inversa

Na *query* 5 era pedido que se lesse e armazenasse N números num array e posteriormente imprimir esses valores por ordem inversa. Para isso, o programa começa por ler os números através do **'read'**, depois armazena-os num array **'numeros[cont]'** e finalmente devolve-os pela ordem inversa devido à condição for definida. Podemos ver de que forma esta criada na imagem abaixo.

```
int numeros[100]; int i; int num; int cont; int x;

x = read();

for (i = 0; i<x; i = i + 1) {
    num = read();
    numeros[cont] = num;
    cont = cont + 1;
};

for (i = cont - 1; i>=0; i = i - 1) {
    print(numeros[i]);
};
```

Figura 24: *Input* a ser lido pelo parser de forma a armazenar N números num *array* e imprimi-los pela ordem inversa

Na imagem abaixo encontra-se o código que foi gerado pelo compilador na linguagem *Assembly* a ser corrido na máquina virtual.

```

output.vm
PUSHN 100
PUSHI 0
PUSHI 0
PUSHI 0
PUSHI 0
START
READ
atoi
STOREG 103
PUSHI 0
STOREG 100
startFor1:
PUSHG 100
PUSHG 103
INF
JZ endFor1
READ
atoi
STOREG 101
PUSHGP
PUSHI 0
PUSHG 102
ADD
PUSHG 101
STOREN
PUSHG 102
PUSHI 1
ADD
STOREG 102
PUSHG 100
PUSHI 1
ADD
STOREG 100
JUMP startFor1
endFor1:
PUSHG 102
PUSHI 1
SUB
STOREG 100
startFor2:
PUSHG 100
PUSHI 0
SUPEQ
JZ endFor2
PUSHGP
PUSHI 0
PUSHG 100
ADD
LOADN
WRITEI
PUSHG 100
PUSHI 1
SUB
STOREG 100
JUMP startFor2
endFor2:
STOP

```

Figura 25: Pseudo-código gerado *query* 5

#### 2.4.6 Invocar e usar num programa seu uma função 'potencia()', que começa por ler do input a base B e o expoente E e retorna o valor $B^E$

A última *query* pedia que fosse invocado uma função potencia(), tendo o grupo escolhido a realização de arrays em vez da capacidade de invocar funções, optou-se por realizar o equiva-



lente. Através do 'read' lê-se então a base ('base') e o expoente ('exp') e após realizada a condição 'for' o valor resultante é apresentado, como podemos verificar na imagem abaixo.

```
int base; int exp; int i; int num;

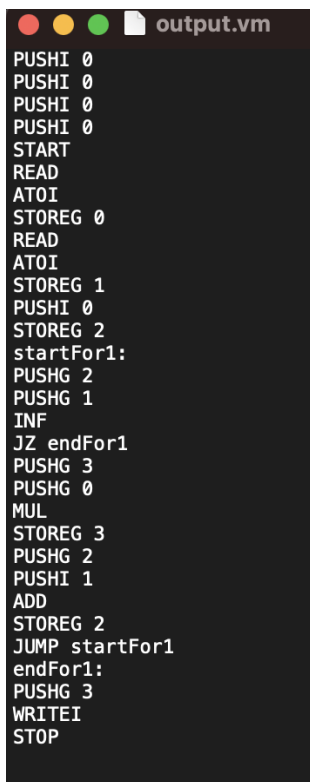
base = read();
exp = read();

for (i = 0; i<exp; i = i + 1) {
    num = num * base;
};

print(num);
```

Figura 26: *Input* a ser lido pelo parser de forma a calcular  $B^E$

A ultima *query* não foge à regra e, através da imagem abaixo, podemos verificar que o compilador conseguiu gerar o pseudo-código que corre na máquina virtual (VM), com todas as instruções em *Assembly*.



```
output.vm
PUSHI 0
PUSHI 0
PUSHI 0
PUSHI 0
START
READ
ATOI
STOREG 0
READ
ATOI
STOREG 1
PUSHI 0
STOREG 2
startFor1:
PUSHG 2
PUSHG 1
INF
JZ endFor1
PUSHG 3
PUSHG 0
MUL
STOREG 3
PUSHG 2
PUSHI 1
ADD
STOREG 2
JUMP startFor1
endFor1:
PUSHG 3
WRITEI
STOP
```

Figura 27: Pseudo-código gerado *query* 6

### 3 *Conclusão*

Durante a elaboração desta segunda e última fase do trabalho prático da unidade curricular de Processamento de Linguagens, foram vários os desafios com que nos deparamos na tentativa de obter um resultado final o mais fidedigno e consistente possível.

Para a obtenção destes resultados, tentamos seguir os ensinamentos lecionados nas aulas teóricas e práticas tendo estes permitido uma maior consolidação de conhecimentos no que diz respeito à escrita de Expressões Regulares (ER), definição de gramáticas e ao processamento de ficheiros.

Assim, concluímos que nosso desempenho ao longo deste trabalho foi positivo, visto que conseguimos encontrar forma de ultrapassar os desafios apresentados durante o desenvolvimento do mesmo.