Programar em Visual Basic

António Ramires Fernandes
Departamento de Informática
Universidade do Minho
2001

1 Definições Básicas

Programar consiste em definir uma sequencia de comandos para o computador executar.

Tipos de dados simples

tipos	valores possíveis
Boolean	True, False
Integer	números inteiros de -32768 a 32767
Long	números inteiros de –2.147.483.648 a 2.147.483.647
Single	números reais com limites muito grandes
Variant	qualquer valor
String	texto de comprimento variável

1.1 Variáveis

identificadores definidos pelo programador para armazenar valores

Uma variável tem sempre associado, explicitamente ou implicitamente, um tipo de dados.

Os valores armazenados nas variáveis podem ser alterados durante o programa.

Os identificadores das variáveis podem ser compostos por letras e números e alguns caracteres especiais. Os caracteres não permitidos são o espaço, !,@,&,$ e #. Para alem disso todos identificadores de variáveis devem começar por uma letra

Exemplos de **identificadores inválidos**:

A minha morada	utiliza espaços
1aluno	começa por um número
a#2	utiliza o caracter #

Exemplo de **identificadores válidos**:

aMinhaMorada

a_minha_morada

um_aluno

umAluno

a_2

Para efeitos de legibilidade do programa os identificadores devem ser escolhidos de forma a indicar qual o significado dos valores armazenados nas variáveis associadas. Por

exemplo ao definir uma variável para armazenar um número de aluno, os seguintes identificadores têm sentido:

numAluno

num

Ao usar um dos indicadores acima a legibilidade do programa aumenta. Por outro lado se utilizássemos identificadores como por exemplo *a* ou *x12* a dificuldade de leitura do programa seria aumentada desnecessariamente.

1.1.1 Declaração de variáveis

Por regra de boa programação deve-se sempre declarar as variáveis no inicio da sequencia de instruções. Por exemplo caso se pretenda utilizar um texto no programa deve-se escrever:

Dim oMeuTexto **As** String

Para declarar uma variável que contenha como valor um numero escreve-se

Dim umNumero **As** Integer

Sintaxe:

Dim identificador **As** *tipo*

em que tipo tem de representar um tipo de dados válido. **Dim** e **As** são palavras reservadas do vocabulário do Visual Basic.

1.2 Constantes

identificadores definidos pelo programador para armazenar valores fixos

Uma constante não pode ser alterada durante a execução do programa.

A utilização de constantes permite escrever programas mais legíveis e mais fáceis de alterar.

1.2.1 Declaração de constantes

A declaração de uma constante implica, para além da indicação do identificador e do seu tipo, a atribuição de um valor.

Sintaxe:

Public Const identificador **As** *tipo* = valor

Public, Const e **As** são palavras reservadas do vocabulário do Visual Basic.

Exemplos:

Public Const maxAlunosPráticas **As Integer** = 40

1.3 Expressões

Em termos genéricos uma expressão é um conjunto de operações ou simplesmente um valor. O tipo de uma expressão é o tipo de dados do valor do seu resultado.

Note-se que os tipos de dados das variáveis que integram a expressão não têm necessariamente de ter o mesmo tipo do resultado da expressão. Por exemplo uma expressão que calcule o comprimento de um texto devolve um resultado inteiro, sendo portanto do tipo **Integer** ou **Long**. No entanto a variável presente na expressão, cujo valor é o texto do qual se pretende saber o comprimento, é do tipo de dados **String**.

1.3.1 Expressões numéricas

Uma expressão pode ser simplesmente um valor:

12723

Uma expressão pode também representar uma operação

b * 2

O símbolo * representa o operador da multiplicação.

Uma expressão pode conter varias operações, por exemplo:

b * 2 + c

No exemplo acima apresentado o resultado é ambíguo. Por um lado pode-se interpretar a expressão como sendo

o valor de *c* mais o dobro do valor de *b*

ou então

o valor de *b* multiplicado pela soma de *2* e *c*

As duas interpretações podem levar a resultados diferentes. Por exemplo considerando que o valor de *b* é 3 e o valor de *c* igual a 2 teríamos como resultados 8 e 12 para a primeira e segunda interpretações respectivamente.

O Visual Basic, perante uma expressão com várias operações, executa as operações pela ordem determinada pela precedência dos operadores, a ordem em que as operações são descritas é irrelevante. Para o Visual Basic a multiplicação e divisão tem precedência sobre a adição e subtracção, sendo assim a interpretação que o Visual Basic adopta na expressão acima é a primeira: primeiro realiza a multiplicação e depois a adição.

Põe-se agora a seguinte questão: e se a interpretação pretendida fosse a segunda? Nesse caso é necessário utilizar parêntesis para definir a expressão:

a = b * (2 + c)

O Visual Basic interpreta os parêntesis como tendo precedência máxima, ou seja primeiro realiza as operações definidas dentro dos parêntesis.

Outros operadores úteis são o módulo, **mod**, e a divisão inteira \. Estes operadores permitem realizar operações sobre números inteiros (caso sejam utilizados números reais estes são arredondados automaticamente para efeitos de cálculo).

O operador \ realiza a divisão inteira entre dois número, por exemplo:

5 \ 2 dá como resultado 2

10 \ 6 dá como resultado 1

O operador **mod** devolve como resultado da operação o resto da divisão inteira, por exemplo:

5 mod 2 dá como resultado 1

10 mod 6 dá como resultado 4

Tabela dos operadores aritméticos:

operador	significado
+	adição
-	subtracção
*	multiplicação
/	divisão
\	divisão inteira
mod	resto da divisão inteira

1.3.2 Expressões Booleanas

As expressões booleanas são extremamente utilizadas na programação. Estas expressões baseiam-se nos operadores de comparação e nos operadores booleanos. Os operadores booleanos mais relevantes são os seguintes:

operador	significado
NOT	Negação
AND	Conjunção (e)
OR	Disjunção (ou)
XOR	Disjunção Exclusiva

O operador NOT é aplicável a uma variável ou expressão booleana. O seu resultado é a negação do valor do seu argumento.

NOT x

NOT expressão booleana

são expressões válidas.

O resultado será **True** se o argumento for **False**, e vice-versa.

Em Português o correspondente seria a palavra NÃO. Ou seja, a frase "não fui ao cinema" é falsa se de facto fui ao cinema, ou seja se o argumento "fui ao cinema" for verdadeiro.

O operador **AND** é um operador binário, isto é realiza uma operação com duas expressões booleanas. Supondo que *x* e *y* são variáveis ou expressões booleanas a expressão

x **AND** y

é uma expressão válida. O resultado desta expressão depende do valor de *x* e *y*, só sendo **True** (Verdade) no caso em que ambos *x* e *y* sejam **True**. A tabela seguinte apresenta o resultado da expressão para os valores possíveis de *x* e *y*.

		valor de y	
		False	True
valor de x	False	False	False
	True	False	True

Em Português o correspondente seria a palavra E. Ou seja a frase "A Maria comprou o casaco e o guarda-chuva" só é verdade se de facto ambos o casaco e o guarda-chuva foram comprados. Sendo falsa se nada se comprou, ou se só um dos artigos foi comprado.

O operador **OR**, à semelhança do operador **AND**, também é um operador binário. O resultado desta operação é **True** se pelo menos uma das expressões for **True**. A sua tabela de resultados é

x OR y

		valor de y	
		False	True
valor de x	False	False	True
	True	True	True

Em Português o correspondente mais próximo seria a palavra **OU**. A frase "o João vai ao cinema ou ao pub" será verdade se o João for ao cinema ou se o João for ao pub. Note-se que no entanto que a interpretação é relativamente ambígua no caso em que o João visitou ambos os locais, já que a frase pode implicitamente querer dizer que o João irá somente a um local. Por exemplo uma mãe pode dizer a um filho "levas o Homem Aranha ou o Power Ranger", querendo implicitamente dizer que ambos os brinquedos estão fora de questão mas que um deles está garantido. O significado da frase depende do

contexto em que ela é proferida. No caso dos brinquedos, o operador apropriado em Visual Basic é o **XOR**.

O operador **XOR** também é um operador binário. O seu resultado é **True** quando um e só um dos seus argumentos for **True**. Se ambos forem **False**, ou se ambos forem **True** o resultado é **False**. A sua tabela de resultados é a seguinte:

x XOR y

		valor de y	
		False	True
valor de x	False	False	True
	True	True	False

Relativamente às precedências a ordem é a seguinte:

1. NOT
2. AND
3. OR
4. XOR

Seguem-se alguns exemplos:

- Na expressão

  NOT b AND c

  primeiro calcula-se a negação de b (NOT), e seguidamente a conjunção(AND).

- Na expressão

  a OR b AND c

  primeiro calcula-se a conjunção (b AND c) e depois a disjunção (OR).

1.3.2.1 Operadores de Comparação

Uma expressão booleana pode também ser construída à custa de operadores de comparação. Alguns dos operadores de comparação definidos são os seguintes:

operador	exemplo	resultado
<	a < b	Verdade se *a* for menor que *b*
<=	a <= b	Verdade se *a* for menor ou igual a *b*

>	a > b	Verdade se *a* for maior que *b*
<=	a >= b	Verdade se *a* for maior ou igual a *b*
<>	a <> b	Verdade se *a* for diferente de *b*
=	a = b	Verdade se *a* for igual a *b*

Alguns exemplos:

c * 2 > 3

c <= a + b

Note-se que os operadores aritméticos têm precedência sobre todos os operadores de comparação. Ou seja primeiro efectuam-se os cálculos e depois as comparações.

É possível combinar os operadores booleanos e de comparação numa expressão booleana. Por exemplo:

c < b AND c < a

a expressão acima só será verdade se *c* for o menor dos três valores: *a*, *b* e *c*.

Exemplos: definir a expressão que dado uma variável *nota* inteira do tipo **Integer** que representa uma nota devolva um resultado booleano que indique se a variável é maior ou igual a 10.

nota > 10

1.3.3 Expressões do tipo String

Utilizando operadores, só existe um tipo de operação sobre Strings: concatenação. A concatenação consiste em juntar dois textos.

"olá" & " mundo" tem como resultado "olá mundo"

A concatenação pode também ser realizada com o operador "+".

2 Programação

Programar consiste em definir uma sequência de instruções, ou comandos. As instruções são seguidas à letra pelo computador por isso é necessário muito cuidado ao escrevê-las.

Dentro dos tipos de instruções existentes, os seguintes serão abordados nesta cadeira:

- Atribuições
- Condicionais
- Cíclicas

2.1 Atribuições

Instrução que atribui a uma variável o valor de uma expressão. O resultado da expressão tem de ser do mesmo tipo que a variável.

Sintaxe

identificador = *expressão*

As seguintes atribuições são válidas

```
Dim a As Integer
Dim b As Integer
a = 3
b = a + 2
```

2.1.1 Exemplos

- Definir a atribuição de forma a que a variável *a* fique com o dobro do valor de *b*

```
Dim a As Integer
Dim b As Integer
a = 2 * b
```

- Definir uma instrução que atribua à variável *positiva* o valor True se a variável *nota* for superior a 9

```
Dim positiva As Boolean
Dim nota As Integer
positiva = nota > 9
```

- Definir uma instrução que atribua à variável preçoFinal a expressão que multiplica um determinado preço de custo, representado pela variável preçoCusto, por uma margem de lucro de 10%, e pelo IVA de 17%.

```
Dim preçoFinal As Single
preçoFinal = preçoCusto * 1.1 * 1.17
```

Por razões de clareza neste último exemplo poderíamos optar por uma solução mais longa mas mais legível:

```
Dim preçoFinal As Single
Public Const IVA As Single = 1.17
Public Const MargemLucro As Single = 1.1
preçoFinal = preçoCusto * Margem * IVA
```

- Cálculo da nota final de programação Informática

```
Dim tp1, tp2, nt1, nt2 As Integer
Dim notaFinal As Integer
notaFinal = (tp1 + tp2 + nt1 + nt2) / 4
```

Nota: este cálculo não corresponde completamente à realidade da actual cadeira.

2.2 Comentários

Os comentários na programação são pequenas notas em linguagem corrente incluídas no texto do programa que facilitam a leitura do programa pelos programadores. A utilização dos comentários é uma prática que se recomenda vivamente.

Os comentários não fazem parte do programa propriamente dito, sendo inclusivamente ignorados pelo compilador ou interpretador. No entanto a sua utilização é fundamental para a boa programação.

Para inserir um comentário em Basic escreve-se o símbolo ' seguido do texto do comentário. O Basic ao encontrar este símbolo ignora o texto dessa linha a partir do local onde o símbolo foi encontrado. Para escrever um comentário com várias linhas deve-se colocar o símbolo em todos as linhas.

Exemplo:

```
' Isto é um comentário
Dim a As Integer
'O meu segundo comentário
' tem duas linhas
a = 3
```

Através de comentários é possível documentar o texto do programa de forma a facilitar a sua leitura. A titulo de exemplo vejamos o seguinte excerto de um programa sem comentários:

```
n = (t1+t2+p1+p2) / 4
```

em seguida é apresentada a versão comentada:

```
' o número médio de alunos na sala é a média dos números das aulas
' teóricas e práticas para os dois turnos
n = (t1+t2+p1+p2) / 4
```

2.3 Instruções Condicionais

Por vezes é conveniente garantir que determinada sequencia de instruções só é executada em determinadas circunstancias. Por exemplo, podemos dizer que os alunos que obtiveram uma nota teórica superior a 18 merecem um bónus de 1 valor. Sendo assim a atribuição do bónus estaria condicionada pelo valor da nota teórica.

2.3.1 Instruções IF

Em Visual Basic existe uma forma de definir instruções condicionais utilizando a instrução IF. A sintaxe desta instrução é a seguinte:

If *expressão booleana* **Then**

... sequencia de instruções

End If

Nesta instrução, a sequencia de instruções definida no seu interior só é executada quando a expressão booleana for verdadeira.

Assumindo que temos o valor da nota teórica e prática nas variáveis *teo* e *prat*, respectivamente, podemos expressar a expressão booleana que traduz o facto "nota teórica superior a 18" por

teo > 18

O excerto de código para realizar esta atribuição condicional poderia ser o seguinte:

```
...
nota = (teo + prat) / 2

If teo > 18 Then
        nota = nota + 1
End If
```

Note-se que no exemplo acima é possível obter uma nota incorrecta, se um aluno tiver 20 valores de nota final, com o bónus ficara com 21. Ao programar é necessário prever estes casos que potencialmente podem gerar erros e resolve-los atempadamente.

Para um bom funcionamento de um programa é crucial prever <u>todas</u> as situações passíveis de gerar erros.

No exemplo acima poder-se-ia resolver o problema de duas formas:

1. atribuir o bónus aos alunos cuja nota teórica seja superior a 18 obtendo assim a nota final bonificada. Caso esta nota seja superior a 20 valores então o valor da nota bonificada seria reduzida para 20 valores garantindo assim que no final da operação a nota é válida.
2. atribuir o bónus aos alunos cuja nota teórica fosse superior a 18, somente nos casos em que a nota final seja menor que 20 valores garantindo assim que a nota nunca assume valores incorrectos.

O código correspondente à solução 1 é o seguinte:

```
nota = (teo + prat) / 2

' Primeiro atribui-se o bónus

If teo > 18 Then
        nota = nota + 1
End If

' Depois verifica-se se a nota é válida

If nota > 20 Then
        nota = 20
End If
```

O código da solução 2 é de seguida apresentado:

```
nota = (teo + prat) / 2

' Aqui só se altera a nota nos casos em que a alteração não conduz a
' uma nota inválida

If (teo > 18) AND (nota < 20) Then
        nota = nota + 1
End If
```

Note-se que neste caso é trivial definir o caso em que a atribuição de bónus leva a uma nota inválida, no entanto nem sempre é assim. Por vezes a solução preventiva (solução 2) é demasiado complexa, sendo preferível a solução "curativa" (solução 1).

Considere-se agora a expressão para cálculo do preço de venda de um determinado produto. Anteriormente foi apresentada uma expressão da seguinte forma:

```
Dim preçoFinal, preçoCusto As Single
Public Const IVA As Single = 1.17
Public Const MargemLucro As Single = 1.1
preçoFinal = preçoCusto * Margem * IVA
```

Assuma agora que o preço final pode ter um desconto de 5 ou 10 por cento, conforme o valor da variável preçoFinal seja inferior a 100.000$, caso em que o desconto é de 5%, ou o preçoFinal seja igual ou superior a 100.000$, caso em que o desconto é de 10%.

Pode-se reescrever o programa da seguinte forma utilizando instruções condicionais:

```
Dim preçoFinal, preçoCusto, desconto As Single
Public Const IVA As Single = 1.17
Public Const MargemLucro As Single = 1.1
preçoFinal = preçoCusto * Margem * IVA

' Se o valor for igual ou superior a 100 contos o
' desconto é de 10%
```

```
If (preçoFinal >= 100000) Then
        desconto = 0.9
End If

' Se o valor for inferior a 100 contos o
' desconto é de 5%

If (preçoFinal < 100000) Then
        desconto = 0.95
End If

' Cálculo do preço final com desconto

preçoFinal = preçoFinal * desconto
```

Note-se no entanto que a atribuição de descontos é exclusiva, isto é uma venda tem um desconto de 10% ou de 5%, mas nunca de ambos. A analise das condições de ambas as instruções condicionais reflecte este facto. Se uma das condições for True então a outra será False.

A exclusividade no exemplo acima é implicitamente definida pelas expressões condicionais utilizadas. Nestes casos, convém tornar a exclusividade explicita, ou seja em vez de escrever:

se o preço for inferior a 100000 então o desconto é de 5%

se o preço for superior ou igual a 100000 então o desconto é de 10%

deve-se escrever:

se o preço for inferior a 100000 então o desconto é de 5%, senão o desconto é de 10%

Em Visual Basic a ultima alternativa corresponde a escrever:

```
If (preçoFinal < 100000) Then
        desconto = 0.95
Else
        desconto = 0.90
End If
```

A sintaxe geral é a seguinte:

```
If expressão Then
        sequencia de instruções A
Else
        sequencia de instruções B
End If
```

A primeira sequencia de instruções, A, é executada quando a *expressão* definida for verdadeira. A segunda sequencia de instruções, B, é executada sempre que a *expressão* definida seja falsa.

Considere-se agora que as condições de atribuição de descontos eram as seguintes:

preço final	desconto
< 100 contos	0
>= 100 e < 1000	5%
>= 1000	10%

O excerto de código para cálculo do desconto poderia ser escrito da seguinte forma:

```
        ' Sem desconto
1       desconto = 1
        ' Atribuição dos descontos para valores superiores a 100 contos
2       If (preço >= 1000) Then
3               desconto = 0.90
4       Else If (preço >= 100) Then
5               desconto = 0.95
6       End If

7       preçoFinal = preço * desconto
```

Nota: os números de linha não devem ser escritos no Visual Basic.

Note-se que no exemplo acima a instrução **Else** também leva uma expressão booleana. Esta expressão traduz uma condição parcial para atribuição do desconto de 5%. Note-se que a expressão completa implicaria testar se o valor é inferior a 1000 contos. No entanto a escrita da expressão completa é desnecessária uma vez que a segunda condição (preço >= 100) só será testada caso a expressão booleana definida na primeira condição (preço >= 1000) seja falsa.

De seguida ir-se-á simular o comportamento do Visual Basic ao executar o excerto de código acima apresentado. Para tal define-se uma tabela com a seguinte estrutura:

- número de linha actual
- valor actual das variáveis *desconto* e *preçoFinal* após a execução da linha (um traço representa *indefinido*).
- observações

Considera-se neste exemplo que o valor de *preço* é 5000 contos.

num	desconto	preçofinal	obs.
1	1	-	atribuição à variável desconto do valor 1
2	1	-	Teste se o valor de preço é maior ou igual a 1000. Esta condição é verdadeira., logo a linha seguinte é a linha 3
3	0.9	-	atribuição à variável desconto do valor 0.9. Após a execução desta instrução terminou-se o bloco

			de código para a condição (preço >= 1000) a instrução **If** é terminada e a linha seguinte é a linha que contem **End If**
6	-	-	
7	0.9	4500	atribuição à variável preçoFinal

Neste caso o Visual Basic executaria as linhas 1,2,3,6 e 7. A exclusão das restantes linhas deveu-se ao facto da a primeira condição da instrução **If** ser verdadeira.

A tabela seguinte representa a simulação da execução do Visual Basic no caso em que o *preço* é de 300 contos.

num	desconto	preçofinal	obs.
1	1	-	atribuição à variável desconto do valor 1
2	1	-	Teste se o valor de preço é maior ou igual a 1000. Esta condição é falsa., logo a linha seguinte é a linha 4, onde se definiu a segunda condição.
4	1	-	Esta condição é verdadeira, 300 >= 100, logo o bloco de instruções associado vai ser executado
5	0.95	-	atribuição à variável desconto do valor 0.95. Após a execução desta instrução terminou-se o bloco de código para a condição (preço >= 1000) a instrução **If** é terminada e a linha seguinte é a linha que contem **End If**
6	0.95	-	
7	0.95	285	atribuição à variável preçoFinal

Finalmente considere-se o caso em que o valor da variável *preço* é de 50 contos.

num	desconto	preçofinal	obs.
1	1	-	atribuição à variável desconto do valor 1
2	1	-	Teste se o valor de preço é maior ou igual a 1000. Esta condição é falsa., logo a linha seguinte é a linha 4, onde se definiu a segunda condição.
4	1	-	Esta condição também é falsa, uma vez que 50 < 100. Sendo assim, e por não estarem definidas mais condições a instrução If termina sem nunca executar nenhum dos blocos de instruções nela definidos.
6	1	-	
7	1	50	atribuição à variável preçoFinal

Como se pode ver pelas três simulações acima apresentadas o percurso de execução do Visual Basic depende da verificação de condições presentes em instruções If, podendo

executar um só bloco, ou nenhum no caso em que todas as expressões booleanas definidas sejam falsas.

Retornando ao exercício apresentado acima, uma outra alternativa para a escrita do código seria:

```
If (preço >= 1000) Then
        desconto = 0.90
Else If (preço >= 100) Then
        desconto = 0.95
Else
        desconto = 1
End If
```

O ultimo **Else** não apresenta nenhuma condição. A sequencia de instruções associada (desconto = 1) é executada sempre que todas as condições especificadas anteriormente (preço >= 1000 e preço >= 100) sejam falsas.

No caso genérico a sintaxe da instrução **If** pode ser sumariada da seguinte forma:

```
If condição1 Then
        sequencia de instruções 1
Else If condição 2 Then
        sequencia de instruções 2
Else If condição 3 Then
        sequencia de instruções 3
...

Else If condição n Then
        sequencia de instruções n
Else
        sequencia de instruções n+1
End If
```

sendo o ultimo **Else** opcional. Note-se que só o ultimo **Else** pode prescindir de expressão.

No caso geral em que existe um **Else** sem ter expressão definida é sempre executada uma e só uma sequencia de instruções. Caso o ultimo **Else** também tenha uma expressão então é possível que nenhuma sequencia de instruções seja executada se todas as condições forem falsas.

Finalmente convém referir que existe a possibilidade de construir instruções condicionais dentro de instruções condicionais. A titulo de exemplo vejamos uma outra forma de escrever a instrução If no caso dos descontos:

```
If (preço >= 1000) Then
        desconto = 0.90
Else
' Vamos agora calcular o desconto caso o preço seja inferior a 1000
        If (preço >= 100) Then
                desconto = 0.95
```

```
        Else
                desconto = 1
        End If
End If
```

2.3.2 Instrução SELECT CASE

O exemplo seguinte apresenta um excerto de código para determinar qual o número de dias de um determinado mês ignorando o caso dos anos bissextos. Assume-se que a variável mês é um inteiro cujo valor se situa entre 1 e 12 inclusive.

```
If (mes=1 OR mes=3 OR mes=5 OR mes=7 OR mes=8 OR mes=10 OR mes=12) Then
        dias = 31
Else If mes=4 OR mes=6 OR mes=9 OR mes=11 Then
        dias = 30
Else
        dias = 29
End If
```

Note-se que o exemplo acima não está escrito da forma mais eficiente. Como exercício procure reescreve-lo de uma forma mais eficiente

Neste caso a instrução **If** está dependente de uma variável, ou seja todas as expressões se referem à variável mês. Nestes casos é possível utilizar uma outra instrução fornecida pelo Visual Basic de forma a obter um excerto de código mais compacto e simultaneamente mais legível.

```
1       Select Case (mes)
2           Case 1, 3, 5, 7, 8, 10, 12: dias = 31
3           Case 4, 6, 9, 11: dias = 30
4           Case Else: dias = 29
5       End Select
```

A primeira linha define qual a variável que será utilizada nas expressões definidas dentro da instrução Select Case. As linhas 2 e 3 utilizam o valor desta variável e comparam-no com os valores definidos nas listas. Se o valor da variável mês for igual a um dos valores da lista a instrução correspondente é executada e o programa termina a instrução **Select Case**. A ultima linha (**Case Else**) é aplicada nos casos em que o valor da variável mes não se encontra em nenhuma das listas apresentadas.

É agora apresentado uma simulação de execução considerando que o mês é Abril, ou seja 4.

num	dias	obs.
1	-	Inicio da instrução Select Case
2	-	O valor 4 não se encontra na enumeração desta linha logo passa-se à linha seguinte.
3	30	O valor 4 encontra-se presente nesta enumeração logo a instrução associada é executada. Uma vez que o valor se encontra presente na enumeração após a execução da

		instrução associada a instrução **Select Case** termina.
5	1	Fim da instrução Select Case

É agora apresentado uma simulação de execução considerando que o mês é Fevereiro, ou seja 2.

num	dias	obs.
1	-	Inicio da instrução Select Case
2	-	O valor 4 não se encontra na enumeração desta linha logo passa-se à linha seguinte.
3	-	O valor 4 não se encontra na enumeração desta linha logo passa-se à linha seguinte.
4	29	Esta linha especifica a sequencia de instruções a ser executada caso todas as outras condições Case não sejam satisfeitas. Após a execução da instrução associada a instrução **Select Case** termina.
5	1	Fim da instrução Select Case

A sintaxe deste comando permite um elevado número de variações. Por exemplo, é possível definir um número entre 1 e 5 de duas formas:

Enumerando os possíveis valores	Case 1,2,3,4,5: sequencia de instruções
Definindo um intervalo	Case 1 To 5: sequencia de instruções

A titulo de exemplo considere-se que se pretende traduzir um valor numérico, correspondente a uma nota, para um texto que atribui uma categoria à classificação obtida:

```
1       Select Case (nota)
2               Case 1 To 5: categoria = "Medíocre"
3               Case 6 To 9: categoria = "Insuficiente"
4               Case 10 To 13: categoria = "Suficiente"
5               Case 14 To 17: categoria = "Bom"
6               Case 18 To 20: categoria = "Muito Bom"
7       End Select
```

Considere-se o exemplo hipotético em que a nota era 21. Apresenta-se em seguida a simulação da execução do Visual Basic para este caso.

num	categoria	obs.
1	-	Inicio da instrução Select Case

2	-	O valor 21 não se encontra no intervalo desta linha logo passa-se à linha seguinte.
3	-	O valor 21 não se encontra no intervalo desta linha logo passa-se à linha seguinte.
4	-	O valor 21 não se encontra no intervalo desta linha logo passa-se à linha seguinte.
5	-	O valor 21 não se encontra no intervalo desta linha logo passa-se à linha seguinte.
6	-	O valor 21 não se encontra no intervalo desta linha logo passa-se à linha seguinte.
7	-	Fim da instrução Select Case

Neste caso o valor de categoria permanece indefinido já que nenhuma das instruções definidas no interior da instrução **Select Case** foi executada.

É possível misturar numa mesma expressão Select Case vários tipos de condições: por enumeração e intervalo. O exemplo seguinte utiliza ambas as construções para reescrever o exemplo anterior.

```
Select Case (nota)
        Case 1 To 5 : categoria = "Medíocre"
        Case 6 To 9: categoria = "Insuficiente"
        Case 10 To 13: categoria = "Suficiente"
        Case 14 To 17: categoria = "Bom"
        Case 18,19,20 : categoria = "Muito Bom"
End Select
```

As vantagens desta instrução sobre a instrução **If** são evidentes neste caso. O código necessário utilizando instruções **If** seria muito mais longo e menos claro. A titulo de exercício reescreva utilizando instruções **If** o código acima apresentado.

2.4 Funções

Até agora todos os exemplos apresentados continham expressões que trabalhavam somente com operadores. Embora o conjunto de operadores definidos seja significativo, a especificação de uma tarefa pode tornar-se longa e pouco clara.

À medida que os programas crescem, a sua legibilidade diminui forçosamente. A estratégia para lidar com este tipo de situações consiste em dividir o problema original em problemas mais pequenos, repetindo se necessário o processo de divisão de forma a que no final cada um dos sub problemas identificados seja simples e fácil de implementar. Através desta estratégia consegue-se escrever programas para resolver problemas extremamente complicados. Devido à divisão do problema original em sub problemas a legibilidade do programa aumenta consideravelmente permitindo assim que os erros de implementação sejam eliminados de uma forma mais fácil.

Outra situação que ocorre geralmente na implementação da solução de problemas consiste na reutilização de excertos de código. Por exemplo um determinado excerto de código pode ser utilizado diversas vezes no mesmo programa. É desejável que se possa escrever esse mesmo excerto uma só vez e reutiliza-lo sempre que necessário.

Quase todas as linguagens de programação oferecem um mecanismo que permite agrupar uma sequencia de instruções de forma a que esta possa ser utilizada durante o programa de forma simples e clara. Em Visual Basic um destes mecanismos são as funções.

Através da utilização de funções pode-se dividir o código de um programa em fragmentos, sendo cada um deles idealmente de dimensão reduzida e portanto potencialmente de melhor legibilidade.

A sintaxe geral da construção de funções é a seguinte:

```
Function nome()
        sequencia de instruções
End Function
```

Uma função recebe um número de argumentos e devolve um e um só resultado.

O exercício do cálculo da nota é agora revisitado de forma a solucionar o mesmo incorporando funções. O objectivo é dividir o problema original em sub problemas e utilizar uma função para resolver cada um dos sub problemas identificados. Apresenta-se de seguida o código utilizado anteriormente para resolver este problema.

```
If ( frequência > exame ) Then
        teórica = frequência
Else
        teórica = exame
End If
nota = (teórica + prática) / 2
```

O problema original pode-se dividir em dois sub problemas:

1. cálculo da nota teórica: máximo da nota de frequência e exame
2. cálculo da nota final: média da nota teórica e prática.

Sendo assim os dois sub problemas identificados são calcular o máximo de dois números e calcular a média de dois números.

Considere-se a função para calcular o máximo dos valores contidos em duas variáveis. Neste caso os parâmetros serão as variáveis, o resultado será o máximo dos valores aceites como parâmetro.

```
Function máximo(a, b)
' definir uma variável para armazenar o valor pretendido
        Dim aux
' cálculo do máximo de dois valores
        If (a > b ) Then
```

```
                aux = a
        Else
                aux = b
        End If

' o resultado pretendido esta armazenado em aux
' para devolve-lo atribui-se ao nome da função o valor de aux

        máximo = aux
End Function
```

Como se pode ver no exemplo acima apresentado o Visual Basic utiliza o nome da função dentro do código com um objectivo particular: indicar qual o valor a ser devolvido.

Não esquecer que uma função tem sempre um e um só resultado. A sintaxe para indicar qual o resultado da função em Visual Basic consiste em atribuir ao nome da função o valor pretendido.

A função para calcular a média de dois números também tem dois argumentos. O código é apresentado a seguir:

```
Function média(a, b)
        média = (a + b) / 2
End function
```

É agora apresentado o código que utiliza as funções acima definidas para resolver o problema original:

```
teórica = máximo(frequência, exame)
nota = média(teórica, prática)
```

Note-se que ao definir as funções máximo e média utilizou-se variáveis com nomes distintos da sua utilização. Ao definir uma função, os nomes das variáveis são da escolha do programador, não tendo relação nenhuma com os nomes que posteriormente se utilizam na sua invocação. Mesmo que se os nomes fossem iguais, para o Visual Basic seriam variáveis distintas.

O Visual Basic estabelece a relação necessária entre os argumentos da utilização e parâmetros da definição ao utilizar a função. Por exemplo na primeira linha do ultimo excerto de código apresentado o Visual Basic irá "invocar" a função máximo com os argumentos *frequência* e *exame*. Esta função tem dois parâmetros *a* e *b*. Neste caso o Visual Basic estabelece a relação entre *frequência* e *a*, e entre *exame* e *b*.

O ultimo excerto de código apresentado poderia por sua vez ser também parte de uma função:

```
Function nota( frequência, exame, prática)
        teórica = máximo(frequência, exame)
        nota = média(teórica, prática)
End Function
```

Sendo assim de onde vem os valores das variáveis argumento da função? Talvez de outra função que por sua vez utilize esta função, ou então de uma folha de cálculo Excel. Embora existam outras alternativas, o seu estudo não será realizado no âmbito desta cadeira.

A função *nota* acima apresentada em cima, apresenta um grau de abstracção elevado para o problema original, o cálculo de uma nota com base na *frequência*, *exame* e *prática*. Assumindo que os nomes atribuídos às funções tem algum significado, ao ler a função sabe-se que a nota teórica é o máximo entre a *frequência* e o *exame*. Sabe-se também que a nota final é a média da nota *teórica* e *prática*. Tudo isto sem entrar em pormenores de como calcular o máximo e a média. Ou seja a leitura da função acima apresentada fornece tanta informação como a leitura do código completo, apresentado de seguida, sem utilizar funções.

```
If ( frequência > exame ) Then
        teórica = frequência
Else
        teórica = exame
End If
nota = (teórica + prática) / 2
```

Este é um dos objectivos das funções: permitir a escrita de programas facilmente legíveis.

Outro objectivo também muito importante é permitir a utilização de um excerto de código parametrizado (ou seja uma função com parâmetros) múltiplas vezes num programa sem reescrever o excerto utilizado.

Para ilustrar este ponto ir-se-á aumentar o exercício do cálculo da nota definido anteriormente. Considere que a nota de frequência é calculada com base nas frequências de Fevereiro e Junho. assuma que as classificações obtidas nas frequências está armazenada nas variáveis *fev* e *jun* para a primeira e segunda frequência respectivamente. Considere ainda que a nota prática é a média dos dois trabalhos práticos, cujas classificações estão armazenadas nas variáveis *tp1* e *tp2*.

Sendo assim o seguinte excerto de código sem funções resolve o problema:

```
prática = (tp1 + tp2) / 2
frequência = (fev + jun) / 2
If ( frequência > exame ) Then
        teórica = frequência
Else
        teórica = exame
End If
nota = (teórica + prática) / 2
```

Utilizando as duas funções definidas acima, média e máximo, o código resultante seria:

```
prática = média (tp1,tp2)
frequência = média(fev, jun)
teórica = máximo(frequência, exame)
nota = média(teórica, prática)
```

O código utilizando funções é claramente mais legível, aproximando a escrita em Visual Basic da nossa formulação do problema, e eliminando de uma primeira leitura pormenores desnecessários. Caso subsistam duvidas sobre o cálculo da nota pode-se sempre recorrer à leitura do código das funções utilizadas. No entanto se os nomes atribuídos às funções reflectirem o seu funcionamento, este passo é geralmente desnecessário, salvo em caso de erro.

Como ultima nota pretende-se salientar que os programas não se medem à linha! A clareza de um determinado excerto de código não tem relação directa com o número de linhas escritas. Para ilustrar este ponto apresenta-se a seguinte linha que produz exactamente o mesmo resultado que ambos os excertos apresentados acima:

```
nota = média(máximo(média(fev, jun),exame),média(tp1,tp2))
```

Embora esta ultima versão seja decididamente mais curta, ela representa o pesadelo de um programador: é de muito difícil leitura! Uma troca de parêntesis e está tudo perdido...

2.5 Colecções

Até agora os tipos de dados definidos permitiam armazenar um só valor. Em Visual Basic existe o conceito de colecção para permitir armazenar numa só variável uma sequencia de valores.

A sintaxe para definir uma colecção é a seguinte:

Dim x **As New Collection**

Em Visual Basic, Collection é na realidade um *objecto* e não um tipo de dados. A distinção entre estes dois conceitos está fora do âmbito desta cadeira. As diferenças na sua utilização ao nível sintáctico serão no entanto apresentadas de forma a permitir a sua utilização.

Note-se que por x ser um *objecto* é necessário utilizar a palavra **New** na sua declaração para o inicializar.

Para adicionar um elemento a uma colecção o Visual Basic fornece a função **Add** que adiciona ao fim da colecção um determinado valor:

```
Dim x As New Collection
x.Add(327)
x.Add(425)
```

Note-se que para *objectos*, a invocação de uma função é precedida do nome do objecto sobre o qual se irá realizar a operação.

Para consultar o valor do elemento na posição *i* de uma colecção utiliza-se a função **Item**.

```
x.Item(i)
```

Exemplo: após a execução do seguinte excerto de código o valor da variável *a* é 425.

```
Dim x As New Collection
```

```
x.Add(327)
x.Add(425)
a = x.Item(2)
```

Note-se que uma consulta a uma posição para além dos número de elementos da colecção produz um erro em Visual Basic terminando a execução do programa.

Para determinar qual o número de elementos de uma colecção o Visual Basic fornece a função **Count**.

Exemplo: após a execução do seguinte excerto de código o valor da variável *a* é 2.

```
Dim x As New Collection
x.Add(327)
x.Add(425)
a = x.Count()
```

2.6 Instruções Cíclicas

Para trabalhar com colecções é necessário um tipo de instrução que permita aplicar uma sequencia de instruções aos elementos de uma colecção. As instruções para este efeito denominam-se instruções cíclicas.

Dentro das instruções fornecidas pelo Visual Basic as mais relevantes serão de seguida apresentadas, terminando esta secção com uma análise comparativa entre as instruções apresentadas.

2.6.1 Ciclos FOR EACH

A instrução **FOR EACH** permite percorrer completamente uma colecção realizando uma sequencia de operações para cada elemento da mesma.

A sintaxe para esta instrução é a seguinte:

```
For Each elemento In colecção
        sequencia de instruções
Next elemento
```

Um exemplo apropriado para esta instrução é realizar a soma de todos os elementos de uma colecção. Para tal define-se uma função que recebe como parâmetro uma colecção.

Este tipo de processos iterativos realiza-se por acumulação, ou seja, inicialmente o valor de *soma* é zero, ou seja, antes de se iniciar a operação o total é zero. Seguidamente queremos acumular no valor de *soma* o primeiro elemento da colecção. A operação repete-se até ao ultimo elemento da colecção, altura em que *soma* conterá como valor o somatório de todos os elementos.

Para cada elemento da colecção, a instrução que realiza a acumulação do seu valor com a soma anterior é:

$$soma = soma + elemento$$

Sendo assim a função completa ficará:

```
        Function total(colecção)
1               soma = 0
2               For Each elem In colecção
3                       soma = soma + elem
4               Next elem
5               total = soma
        End Function
```

A instrução **Next** fornece-nos o próximo elemento da colecção caso ainda não se tenha atingido o fim da colecção, caso contrário o resultado desta instrução é a saída do ciclo. Note-se que o nome da variável nesta instrução tem de coincidir com o nome utilizado na definição **For Each**.

Para a simulação de execução que se segue considere que a colecção fornecida como parâmetro à função, tem os seguintes elementos: < 4, 3, 5, 2>.

num	elem	soma	obs.
1	-	0	Inicialmente o valor de *soma* é zero.
2	4	0	*elem* toma o valor do primeiro elemento da colecção, 4.
3	4	0 + 4 = 4	é atribuído a *soma* a acumulação do seu valor prévio com o valor de *elem*.
4	3	4	*elem* toma o valor do próximo elemento da colecção, neste caso o segundo.
3	3	4 + 3 = 7	é atribuído a *soma* a acumulação do seu valor prévio com o valor de *elem*.
4	5	7	*elem* toma o valor do próximo elemento da colecção, neste caso o terceiro.
3	5	7 + 5 = 12	é atribuído a *soma* a acumulação do seu valor prévio com o valor de *elem*.
4	2	12	*elem* toma o valor do próximo elemento da colecção, neste caso o quarto.
3	2	12 + 2 = 14	é atribuído a *soma* a acumulação do seu valor prévio com o valor de *elem*.
4	-	14	Não existem mais elementos na colecção, logo o ciclo termina. Note-se que o valor de elem é indefinido após esta instrução.
5	-	14	instrução que define que o valor de soma é o resultado da função

De seguida é apresentado um outro exemplo de utilização do ciclo FOR EACH para cálculo do valor máximo de uma colecção.

O calculo do máximo de uma determinada colecção é um processo iterativo em que se vai comparando os elementos da colecção com um valor máximo calculado até à altura, ou seja, se o ciclo se encontrar no terceiro elemento, deve existir uma variável que contenha

o máximo dos dois primeiros elementos. Essa variável é então comparada com o terceiro elemento e caso seja inferior o seu valor é actualizado para o valor do terceiro elemento. O processo continua iterativamente percorrendo os restantes elementos da colecção substituindo o valor da variável que contem o valor máximo sempre que seja encontrado um valor superior.

```
        Function máximo(colecção)
1       máximo = 0
2       For Each elem In colecção
3               If elem > máximo Then
4                       máximo = elem
5               End If
6       Next elem
        End Function
```

Para a simulação de execução que se segue considere que a colecção fornecida como parâmetro à função, tem os seguintes elementos: < 4, 3, 5, 2>.

num	elem	máximo	obs.
1	-	0	Inicialmente o valor de *máximo* é zero.
2	4	0	*elem* toma o valor do primeiro elemento da colecção, 4.
3	4	0	O valor de *máximo* é inferior a *elem*, logo a condição da instrução **If** é verdadeira e por conseguinte a instrução definida dentro do **If** será executada.
4	4	4	*máximo* toma o valor de *elem*.
5	4	4	
6	3	4	*elem* toma o valor do próximo elemento da colecção, neste caso o segundo.
3	3	4	O valor de *máximo* é superior a *elem*, logo a condição da instrução **If** é falsa e por conseguinte a instrução definida dentro do **If** não será executada.
5	3	4	
6	5	4	*elem* toma o valor do próximo elemento da colecção, neste caso o terceiro.
3	5	4	O valor de *máximo* é inferior a *elem*, logo a condição da instrução **If** é verdadeira e por conseguinte a instrução definida dentro do **If** será executada.
4	5	5	*máximo* toma o valor de *elem*.
5	5	5	
6	2	5	*elem* toma o valor do próximo elemento da colecção, neste caso o quarto e último.
3	2	5	O valor de *máximo* é superior a *elem*, logo a

			condição da instrução **If** é falsa e por conseguinte a instrução definida dentro do **If** não será executada.
5	2	5	
6	-	5	Não existem mais elementos em *colecção* e por isso termina o ciclo.

Note-se que na linha 1 da função acima definida atribui-se o valor 0 à variável *máximo*. O valor inicial desta variável é importante já que pode influenciar o resultado. A titulo de exemplo considere que na linha 1 da função acima definida em vez de

```
1       máximo = 0
```

se tinha escrito

```
1       máximo = 10
```

Neste caso o valor da variável *máximo* permaneceria inalterado durante todo o ciclo, sendo o seu valor final 10, o que é claramente errado! A razão pela qual o valor de *máximo* permanece inalterado é que a condição especificada na instrução IF, linha 3,

```
3               If elem > máximo Then
```

nunca se verifica e portanto nunca se executa a instrução da linha 4.

```
4                       máximo = elem
```

Caso o valor inicial de *máximo* seja superior a todos os elementos da colecção então o valor final da variável *máximo* é incorrecto.

Sendo assim qual a regra para atribuir o valor inicial à variável *máximo* de forma a obter um resultado correcto no final do ciclo, i.e. o valor máximo presente na colecção? O valor inicial da variável *máximo* deve ser tal que exista sempre pelo menos um valor na colecção superior ao valor inicial escolhido. Desta forma garante-se que a condição especificada na instrução If, linha 3, será verificada para pelo menos um valor e portanto o valor final de *máximo* é realmente o máximo da colecção e não um elemento maior que todos os elementos da colecção.

Se considerarmos que a colecção contem preços, então é seguro dizer que o máximo é zero, já que não existem preços inferiores a esse valor. Caso se trate de uma colecção que contenha ambos números positivos e números negativos então a única solução seria inicializar a variável *máximo* com o menor número possível, no caso de números inteiros o seu valor seria –32768.

O problema inverso verifica-se com o cálculo do valor mínimo. Neste caso o valor inicial da variável que conterá o valor mínimo da colecção deve ser inicializada com um valor superior ao mínimo actual da colecção. Como no caso geral não se conhece o valor

mínimo de uma colecção então a forma segura de inicializar o valor mínimo e atribuir-lhe o maior valor possível, no caso dos inteiros este valor é 32767.

Uma vez que o valor inicial destas variáveis depende da colecção em causa, uma possível solução é inicializar a variável com o primeiro valor da colecção, ou seja, por exemplo

máximo = A.item(1)

ou

mínimo = A.item(2)

2.6.2 Ciclos FOR

O segundo tipo de ciclos que aqui será estudado são os ciclos FOR. Este tipo de ciclos é mais poderoso que os ciclos FOR EACH uma vez que permite especificar a forma como uma colecção é percorrida e quais os elementos a considerar.

A sintaxe para este ciclo é a seguinte:

```
For i = limite_inicial TO limite_final
        sequencia de instruções
Next i
```

A instrução **Next** *i* corresponde a incrementar o valor de *i* por uma unidade e de seguida, caso o valor de *i* não seja superior a *limite_final* então voltar de novo para dentro do ciclo.

Considere de novo o exemplo do calculo da soma de todos os elementos de uma colecção. Neste caso o *limite_inicial* é 1, ou seja o primeiro elemento, e o *limite_final* a posição do ultimo elemento, ou seja o número total de elementos. O Visual Basic fornece uma função que determina o número total de elementos de uma colecção, **count()**, ver secção referente às colecções. O esqueleto da função utilizando a função **count()** será:

```
        Function total(colecção)
1               soma = 0
2               For i  = 1 To coleção.count()
3                       instrução para acumular o valor de soma
4               Next i
5               total = soma
        End Function
```

A instrução para acumular o valor da soma, linha 3 da função acima apresentada, pode ser definida informalmente como:

```
        soma = soma + valor do elemento que se encontra na posição i
```

Para determinar o valor de um elemento numa determinada posição i, utiliza-se a função **item(i)**, ver secção referente às colecções.

A função completa é em seguida apresentada:

```
        Function total(colecção)
1               soma = 0
2               For i  = 1 To coleção.count()
3                       soma = soma + coleção.item(i)
4               Next i
5               total = soma
        End Function
```

Para a simulação da execução desta função considere que a colecção tem os seguintes elementos < 2 4 3>.

num	i	soma	obs.
1	-	0	Inicialmente o valor de *soma* é zero.
2	1	0	Atribui-se a *i* o valor do limite inicial do ciclo, ou seja 1.
3	1	0 + 2 = 2	é atribuído a *soma* a acumulação do seu valor prévio,0, com o valor do elemento da colecção na posição *i*, ou seja o primeiro elemento da colecção.
4	2	2	*i* é incrementado de uma unidade. Uma vez que *i* não é superior ao limite final, coleção.**count**() a simulação prossegue na linha 3.
3	2	2 + 4 = 6	é atribuído a *soma* a acumulação do seu valor prévio com o valor do elemento da colecção na posição *i*, ou seja o segundo elemento da colecção.
4	3	6	*i* é incrementado de uma unidade. Uma vez que *i* não é superior ao limite final, coleção.**count**() a simulação prossegue na linha 3.
3	3	6 + 3 = 9	é atribuído a *soma* a acumulação do seu valor prévio com o valor do elemento da colecção na posição *i*, ou seja o terceiro elemento da colecção.
4	4	9	*i* é incrementado de uma unidade. Uma vez que *i* é superior ao limite final, coleção.**count**() o ciclo é terminado e a simulação prossegue na linha 5.
5	-	9	instrução que define que o valor de soma é o resultado da função

Pelo exemplo apresentado acima é evidente que para percorrer completamente uma colecção a escrita de uma função utilizando o ciclo FOR é definitivamente mais complicada que a mesma função utilizando o ciclo FOR EACH. De facto para percorrer completamente uma colecção os ciclos FOR EACH são os mais apropriados.

No entanto os ciclos FOR permitem realizar operações que os ciclos FOR EACH não permitem como por exemplo percorrer somente os *n* primeiros elementos de uma colecção. Ou seja nos ciclos FOR existe a liberdade de seleccionar limites iniciais e limites finais diferentes de 1 e o numero total de elementos respectivamente. De facto o ciclo FOR EACH é um caso particular dos ciclos FOR em que os limites inicias e finais são respectivamente o primeiro e ultimo elemento de uma colecção.

Por exemplo a seguinte função realiza a soma dos elementos que se encontram entre a quinta e a décima posição numa colecção.

```
        Function total(colecção)
1               soma = 0
2               For i  = 5 To 10
3                       soma = soma + coleção.item(i)
4               Next i
5               total = soma
        End Function
```

Note-se que se a colecção não possuir pelo menos dez elementos a função provoca um erro de "Index Out of Bounds", ou seja tentativa de acesso a um elemento numa posição fora da gama de posições da colecção.

Considere agora um outro exemplo: a procura de um determinado valor numa colecção. Esta função terá de ter dois parâmetros: o elemento a encontrar, e a colecção onde se pretende procurar. Sendo assim o esqueleto da função poderá ser algo do género:

```
Function procura(elem, colecção)
...
End Function
```

Basicamente o que se pretende é determinar para cada elemento da colecção verificar se este é igual ou diferente ao elemento que se pretende procurar. Caso seja igual o resultado final deve ser false, caso contrário este resultado será true.

Apresenta-se agora uma primeira tentativa para a resolução deste problema:

```
        Function procura(elem, colecção)
1               For i = 1 To coleção.count()
2                       If (elem = coleção.item(i)) Then
3                               procura = true
4                       Else
5                               procura = false
6                       End If
7               Next i
        End Function
```

Considerando que a colecção é <8,6,7,5> e sendo 6 o valor procurado procede-se de seguida a uma simulação de execução para esta função.

num	i	procura	obs.

1	1	-	Atribui-se à variável i o valor do limite inicial do ciclo. A variável procura, que também é o resultado da função permanece indefinido.
2	1	-	Testa se elem = coleção.item(1), ou seja 8. A condição é falsa logo a linha seguinte a executar é a linha 5.
5	1	false	
6	1	false	Termina a instrução If
7	2	false	*i* é incrementado de uma unidade. Uma vez que *i* não é superior ao limite final, coleção.**count**() a simulação prossegue na linha 3.
2	2	false	Testa se elem = coleção.item(2), ou seja 6. A condição é verdadeira logo a linha seguinte a executar é a linha 4.
4	2	true	é atribuído a *procura* o valor true. Terminou o bloco de instruções para a condição verdadeira, logo a próxima instrução a executar encontra-se na linha 6
6	2	true	Termina a instrução If
7	3	true	*i* é incrementado de uma unidade. Uma vez que *i* não é superior ao limite final, coleção.**count**() a simulação prossegue na linha 3.
2	3	true	Testa se elem = coleção.item(3), ou seja 7. A condição é falsa logo a linha seguinte a executar é a linha 5.
5	3	false	
6	3	false	Termina a instrução If
7	4	false	*i* é incrementado de uma unidade. Uma vez que *i* não é superior ao limite final, coleção.**count**() a simulação prossegue na linha 3.
2	4	false	Testa se elem = coleção.item(4), ou seja 5. A condição é falsa logo a linha seguinte a executar é a linha 5.
5	4	false	
6	4	false	Termina a instrução If
7	5	false	*i* é incrementado de uma unidade. Uma vez que *i* é superior ao limite final termina o ciclo FOR.

O resultado desta função é *false*, no entanto o valor 6 encontra-se presente na colecção passada como parâmetro. Podemos então concluir que esta função está mal construída já que o resultado correcto seria *true*.

Exercício: Existem dois casos particulares em que a função acima apresentada dará um resultado correcto: dê exemplos de parâmetros para os quais a função de resultados correctos, e identifique as razões que levam a função a devolver resultados correctos.

Analisando a simulação de execução da função pode-se concluir o seguinte: ao encontrar o elemento pretendido é atribuído ao valor da variável *procura* o valor *true*. No entanto, logo de seguida o valor da variável *procura* passa a *false* mal a função se depare com um elemento da colecção que não seja igual ao elemento procurado.

O problema reside no facto de se alterar para todos os elementos o valor da variável procura e aceitar o ultimo valor como sendo o resultado real.

Esta função deve ser reescrita da seguinte forma:

1. inicialmente define-se que o valor da variável procura é false, ou seja, antes de se procurar nada se pode encontrar!
2. seguidamente realiza-se o ciclo FOR tal com na função anterior sendo a única diferença o facto de só se alterar o valor da variável procura case se encontre o elemento pretendido.

Sendo assim o código da função fica:

```
        Function procura(elem, colecção)
1               procura = false
2               For i = 1 To coleção.count()
3                       If (elem = coleção.item(i)) Then
4                               procura = true
5                       End If
6               Next i
        End Function
```

Apresenta-se de seguida a simulação de execução da função.

num	i	procura	obs.
1	-	false	
2	1	false	Atribui-se à variável i o valor do limite inicial do ciclo. A variável procura, que também é o resultado da função permanece indefinido.
3	1	false	Testa se elem = coleção.item(1), ou seja 8. A condição é falsa logo a linha seguinte a executar é a linha 5.
5	1	false	Termina a instrução If
6	2	false	*i* é incrementado de uma unidade. Uma vez que *i* não é superior ao limite final, coleção.**count**() a simulação

			prossegue na linha 3.
3	2	false	Testa se elem = coleção.item(2), ou seja 6. A condição é verdadeira logo a linha seguinte a executar é a linha 4.
4	2	true	é atribuído a *procura* o valor true. Terminou o bloco de instruções para a condição verdadeira, logo a próxima instrução a executar encontra-se na linha 6
5	2	True	Termina a Instrução If.
6	3	true	*i* é incrementado de uma unidade. Uma vez que *i* não é superior ao limite final, coleção.**count**() a simulação prossegue na linha 3.
3	3	true	Testa se elem = coleção.item(3), ou seja 7. A condição é falsa logo a linha seguinte a executar é a linha 5.
5	3	true	Termina a instrução If
6	4	true	*i* é incrementado de uma unidade. Uma vez que *i* não é superior ao limite final, coleção.**count**() a simulação prossegue na linha 3.
3	4	true	Testa se elem = coleção.item(4), ou seja 5. A condição é falsa logo a linha seguinte a executar é a linha 5.
5	4	true	Termina a instrução If
6	5	true	*i* é incrementado de uma unidade. Uma vez que *i* é superior ao limite final termina o ciclo FOR.

Esta função devolve o resultado correcto. É fácil ver que caso o elemento não pertença à colecção o resultado será sempre false. A titulo de exercício construa uma simulação de execução tomando como parâmetros a colecção <9,8,7,6> e o elemento 5.

O código desta função não é muito *inteligente*. Considere a seguinte analogia para ilustrar a falta de eficiência desta função. Um gestor de uma empresa pede ao seu assistente uma cópia de um relatório sobre a situação financeira da empresa. Na secretária do assistente encontra-se uma pilha de relatórios na qual se deve encontrar o relatório pretendido. O assistente começa a procura do relatório pelo topo da pilha, e vai procurando até encontrar o dito relatório. Quando encontra, suspira aliviado, posa o relatório na beira da secretária, e... continua a sua operação de procura até chegar ao fim da pilha!?

Porque continuar a procurar após encontrar? Porque na função acima definida não há maneira de especificar que a busca termina após *procura* assumir o valor true. Os ciclos FOR só permitem especificar os limites inicial e final. Atingir o limite final é a única saída possível do ciclo. Na analogia acima o assistente só poderia terminar a sua busca

após procurar em toda a pilha de relatórios, mesmo que encontre o relatório pretendido a meio do processo.

Na secção seguinte vamos apresentar um novo tipo de ciclos que permite resolver este tipo de problemas de forma adequada.

2.6.3 Ciclos WHILE

Os ciclos que agora são apresentados, ciclos WHILE, são os mais versáteis de todos. Como foi mencionado anteriormente, os ciclos FOR EACH são um caso particular dos ciclos FOR. Por sua vez os ciclos FOR são um caso particular dos ciclos WHILE.

Os ciclos WHILE tem a seguinte sintaxe:

```
While condição
        sequencia de instruções
Wend
```

Sendo *condição* uma expressão booleana que determina enquanto o ciclo deve prosseguir, ou seja o ciclo não termina enquanto a expressão for verdadeira. Considere de novo a função soma. Neste caso pretende-se percorrer a colecção completa desde o primeiro elemento até ao último. Ao definir o ciclos FOR para esta função os limites eram especificados na própria definição da instrução. Para os ciclos WHILE, a situação é diferente. Só é possível especificar uma condição de continuação do ciclo, ou seja, o ciclo irá ser executado enquanto a posição do elemento actual não exceder o total de elementos. Apresenta-se de seguida o código desta função:

```
        Function soma(col)
1               soma = 0
2               i = 1
3               While i <= col.count()
4                       soma = soma + col.item(i)
5                       i = i + 1
6               Wend
        End function
```

A variável *i* é utilizada para percorrer a colecção, indicando a posição do elemento actual. A condição do ciclo WHILE

$i <=$ col.**count**()

indica que o ciclo será executado enquanto a posição do elemento actual não exceder o total de elementos da colecção. Note-se que na definição do ciclo WHILE não existe referencia ao valor inicial da variável *i*. Esta definição de valor inicial deve ser feita antes do ciclo começar.

Ao terminar o ciclo é necessário explicitar que se pretende passar ao elemento seguinte, ou seja ao elemento que se encontra na posição i+1, dai a instrução

$i = i + 1$

Ao terminar um ciclo FOR com a instrução Next i, está-se de facto a realizar duas operações:

1. incrementar o valor de i por uma unidade, ou seja i = i + 1
2. definir o fim da instrução FOR

Como se pode inferir do exemplo anterior, nos ciclos WHILE é necessário realizar o incremento da variável i explicitamente. Os ciclos WHILE são portanto mais complexos, no entanto essa complexidade permite realizar operações de uma forma mais eficiente. Antes de se prosseguir para o próximo exemplo considere a seguinte simulação de execução da função soma assumindo que a colecção passada como parâmetro tem os seguintes elementos:<2,4,3>.

num	i	soma	obs.
1	-	0	Inicialmente o valor de *soma* é zero.
2	1	0	
3	1	0	Uma vez que a condição é verdadeira, ou seja o valor de i não é superior a col.count() as instruções dentro do ciclo serão executadas.
4	1	0+2 = 2	é atribuído a *soma* a acumulação do seu valor prévio com o valor do elemento na posição *i*.
5	2	2	a variável *i* é incrementada por uma unidade.
6	2	2	a instrução Wend determina o fim da instrução WHILE e provoca um salto para a linha onde se encontra o inicio da definição desta instrução, ou seja a linha 3.
3	2	2	Uma vez que a condição é verdadeira, ou seja o valor de i não é superior a col.count() as instruções dentro do ciclo serão executadas.
4	2	2+4 = 6	é atribuído a *soma* a acumulação do seu valor prévio com o valor do elemento na posição *i*.
5	3	6	a variável *i* é incrementada por uma unidade.
6	3	6	Salto para a linha onde se encontra o inicio da definição desta instrução, ou seja a linha 3.
3	3	6	Uma vez que a condição é verdadeira, ou seja o valor de i não é superior a col.count() as instruções dentro do ciclo serão executadas.
4	3	6+3 = 9	é atribuído a *soma* a acumulação do seu valor prévio com o valor do elemento na posição *i*.
5	4	9	a variável *i* é incrementada por uma unidade.
6	4	9	Salto para a linha onde se encontra o inicio da definição desta instrução, ou seja a linha 3.
3	4	9	A condição é falsa, ou seja o valor de i é superior a col.count() o ciclo termina e a próxima instrução será a instrução imediatamente a seguir à instrução Wend. Neste caso como não

			existem mais instruções a função termina.

Considere agora o caso em que se pretende realizar uma procura de um determinado elemento numa colecção, sendo ambos os valores passados como parâmetro. O esqueleto da função é equivalente ao definido anteriormente para os ciclos FOR, ou seja:

```
Function procura(elem, col)
        ...
End Function
```

Apresenta-se de seguida uma versão da função *procura* semelhante à apresentada com ciclos FOR mas desta vez utilizando ciclos WHILE.

```
Function procura(elem, col)
        procura = false
        i = 1
        While (i <= col.count())
                If elem = col.Item(i) Then
                        procura = true
                End If
                i = i + 1
        Wend
End Function
```

A função acima apresentada sofre do mesmo problema que a sua homónima com ciclos FOR. No entanto com ciclos WHILE é possível resolver o problema da ineficiência presente na solução com ciclos FOR.

A essência da questão é definir quando sair do ciclo WHILE, ou por outras palavras, enquanto permanecer dentro do ciclo. A função acima define como condição de permanência no ciclo o facto de a variável i referenciar uma posição dentro dos limites da colecção. No entanto a variável procura é completamente ignorada nesta condição. Ou seja traduzindo para Português corrente actualmente a condição especificada é:

Procurar até ao fim, ignorando o facto de se encontrar a meio

Na realidade o que se pretende é a conjunção das duas condições apresentadas a seguir:

1. Procurar até ao fim

2. Procurar até encontrar.

A condição *1* já se encontra definida. Resta agora definir a condição *2*. De acordo com o código acima apresentado sabe-se que enquanto o elemento não for encontrado na colecção a variável *procura* será false. Sendo assim a segunda condição poderá ser:

procura = false

O código completo com a nova condição é agora apresentado:

```
Function procura(elem, col)
        procura = false
        i = 1
        While (i <= col.count()) AND (procura = false) Then
                If elem = col.Item(i) Then
                        procura = true
                End If
                i = i + 1
        Wend
End Function
```

O código acima pode ser reescrito de forma a simplificar o ciclo. Considere a seguinte função como uma primeira tentativa para resolver o problema:

```
        Function procura(elem, col)
1               i = 1
2               While (i <= col.count()) AND (elem <> col.Item(i)) Then
3                       i = i + 1
4               Wend
End Function
```

A condição desta ultima função especifica que o ciclo será executado enquanto:

1. a posição actual, variável i, estiver dentro dos limites da colecção
2. o elemento for diferente do elemento actual

A segunda condição garante que mal se encontre um elemento na colecção igual ao elemento passado como parâmetro o ciclo é terminado.

A função não está completa uma vez que nunca é definido qual o resultado da função, ou seja não existe nenhuma atribuição à variável *procura* (nome da função). Duas simulações de execução são agora apresentadas, a primeira em que o elemento existe na colecção, a seguinte em que o elemento não se encontra na colecção.

Simulação de execução da ultima versão da função *procura* com os seguintes parâmetros: *elem* = 2, *col* = <3,2,4>

num	i	col.item(i)	obs.
1	1	3	
2	1	3	Condição verdadeira.
3	2	2	
4	2	2	Salto para a linha 3
2	2	2	A condição é falsa porque elem = col.item(i). Salto para o fim da função.

O valor de *i* é 2, ou seja indica a posição em que *elem* se encontra na colecção.

Considere agora que os parâmetro eram: *elem* = 5, *col* = <3,2,4>.

num	i	col.item(i)	obs.
1	1	3	
2	1	3	Condição verdadeira.
3	2	2	
4	2	2	Salto para a linha 3.
2	2	2	Condição verdadeira.
3	3	4	
4	3	4	Salto para a linha 3.
2	3	4	Condição verdadeira.
3	4	-	
4	4	-	Salto para a linha 3.
2	4	-	A condição é falsa porque 4 é superior a col.count(), ou seja a posição actual, 4, está fora dos limites da colecção. Salto para o fim da função.

Neste caso o valor da variável i é 4. Este valor indica que o elemento procurado não se encontra na colecção, uma vez que o processo de procura foi terminado por termos esgotado os elementos da colecção.

Pode-se então concluir que, ao terminar o ciclo, caso a variável i esteja dentro dos limites da colecção então o elemento existe na colecção, caso contrário, ou seja se i for maior que o número de elementos da colecção, então o elemento não existe na colecção.

Sendo assim deve-se acrescentar as seguintes linhas a seguir ao ciclo:

```
        If i <= col.count() Then
                procura = true
        Else
                procura = false
        End If
```

O código completo é:

```
Function procura(elem, col)
        i = 1
        While (i <= col.count()) AND (elem <> col.Item(i)) Then
                i = i + 1
        Wend
        If i <= col.count() Then
                procura = true
        Else
                procura = false
        End If
End Function
```

Um ultimo pormenor de interesse nesta função reside na ordem das condições do ciclo WHILE. Caso a ordem fosse inversa ou seja,

```
        While (elem <> col.Item(i)) AND (i <= col.count()) Then
```

então o Visual Basic terminaria a execução em erro caso o elemento não existisse na colecção. O erro deve-se ao facto de o valor final de i ser superior ao número de elementos da colecção (porque o elemento não existe), logo

col.Item(i)

não se encontra definido. Na ordem original,

```
        While (i <= col.count()) AND (elem <> col.Item(i)) Then
```

este problema não se levanta porque a primeira condição é falsa, e ao realizar a conjunção de condições (AND), desde que uma das condições seja falsa a conjunção também é falsa e logo não é necessário calcular as restantes condições. Ou seja, para o valor final de i, como a condição *i <= col.count()* é falso o VB não calcula a outra condição, uma vez que a conjunção de uma condição falsa com outra condição é sempre falsa, independentemente do valor da outra condição. Resumindo o VB só calcula o estritamente necessário para obter um resultado. A esta estratégia dá-se o nome de "lazy evaluation".

Nas funções anteriores o resultado devolvido corresponde em Português a "encontrei" ou "não encontrei". No entanto esta resposta pode não ser a pretendida, muitas vezes pretende-se saber não só se o elemento existe na colecção mas também caso exista em que posição se encontra.

A função apresentada acima pode ser reescrita para devolver false caso o elemento não se encontre na colecção e a sua posição caso o elemento exista. Uma vez que a variável i representa a posição do elemento caso este exista na colecção o código final desta versão será:

```
Function procura(elem, col)
        i = 1
        While (i <= col.count()) AND (elem <> col.Item(i)) Then
                i = i + 1
        Wend
        If i <= col.count() Then
                procura = i
        Else
                procura = false
        End If
End Function
```

2.6.4 Análise comparativa de instruções Cíclicas

A analise realizada nesta secção vai incidir sobre duas vertentes distintas: simplicidade do código, e versatilidade das instruções propriamente ditas.

Considere a função soma definida anteriormente para todos os ciclos.

FOR EACH	FOR	WHILE

Function soma(col)
  soma = 0
  For Each elem **In** col
  soma = soma + elem
  Next elem
End function
	Function soma(col)
	  soma = 0
	  For i = 1 **To** col.**count**()
	  soma = soma + col.**item**(i)
	  Next i
	End function
		Function soma(col)
		  soma = 0
		  i = 1
		  While i <= col.**count**()
		  soma = soma + col.**item**(i)
		  i = i + 1
		  Wend
		End function

A versão que utiliza o ciclo FOR EACH é a mais simples, não é necessário especificar limites para o ciclo. A forma de se obter o elemento também é mais simples utilizando a instrução FOR EACH, não sendo necessário recorrer a índices. A instrução FOR implica a definição dos limites, ou seja col.**count**(). A terceira versão implica a inicialização separada do limite inferior , assim como o incremento manual da variável *i*.

Em termos de simplicidade as três instruções poderiam ser apresentadas pela seguinte ordem, da mais simples à mais complexa:

1. FOR EACH
2. FOR
3. WHILE

Em termos de versatilidade a ordem é exactamente a inversa. Isto é natural uma vez que a simplicidade advém do facto de se restringir a utilização a casos concretos.

A instrução FOR EACH só deve ser utilizada quando se pretende percorrer uma colecção do principio ao fim. A instrução FOR permite definir intervalos diferentes do inicial, 1, e final, **count()** sendo portanto mais versátil.

Por sua vez a instrução WHILE permite especificar qualquer tipo de condição. A vantagem da utilização dos ciclos WHILE é evidente quando se realiza uma operação cujo critério de saída do ciclo não depende de um limite fixo. Um exemplo deste tipo de operações é a procura de um determinado elemento em que não se sabe à partida a sua posição na colecção.

2.6.4.1 Conversão de instruções cíclicas FOR EACH - FOR

É possível realizar a conversão de qualquer instrução cíclica FOR EACH para uma instrução FOR. A regra para a conversão é a seguinte:

Para o esqueleto da instrução a conversão é realizada tal como se apresenta na tabela em baixo:

FOR EACH elem in colecção
  ...
Next elem
	FOR i = 1 to coleção.count()
	  ...
	Next i

Para o corpo de instruções dentro dos ciclos as alterações a realizar prendem-se com o facto de ser necessário substituir as referencias a elem, no caso dos ciclos FOR EACH, por referencias a coleção.**item**(i), para os ciclos FOR. De seguida apresentam-se as funções completas para o caso da soma com as diferenças sublinhadas:

```
Function soma(col)
  soma = 0
  For Each elem In col
   soma = soma + elem
  Next elem
End function
```

```
Function soma(col)
  soma = 0
  For i  = 1 To col.count()
   soma = soma + col.item(i)
  Next i
End function
```

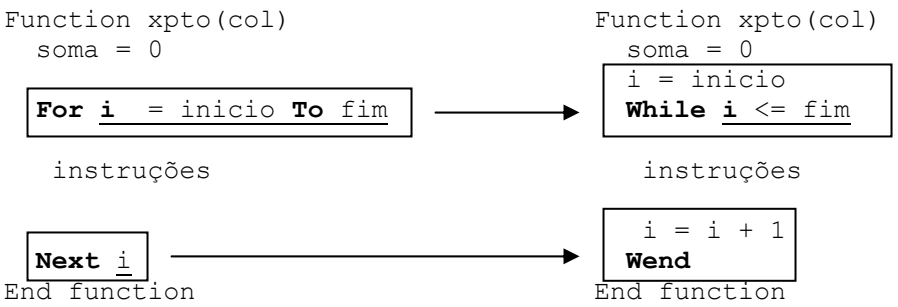
Note-se que a passagem de ciclos FOR para ciclos FOR EACH nem sempre é possível. Só é possível converter um ciclo FOR num ciclo FOR EACH no caso em que os limites iniciais e finais correspondem ao inicio e fim da colecção. Para alem disso é necessário que as indexações a elementos nos ciclos FOR sejam todas do tipo col.**item(**i) sendo *i* a variável utilizada no ciclo.

A titulo de exemplo apresenta-se agora um ciclo FOR que não pode ser convertido para um ciclo FOR EACH.

```
Function soma5a10(col)
soma = 0
  For i  = 5 To 10
   soma = soma + col.item(i)
  Next i
End function
```

2.6.4.2 Conversão de instruções cíclicas FOR - WHILE

Um ciclo FOR pode ser sempre convertido para um ciclo WHILE seguindo um processo automático da seguinte forma:



As instruções dentro do ciclo não sofrem qualquer alteração ao converter o ciclo de FOR para WHILE.

A conversão no sentido inverso nem sempre é possível. Por exemplo a função soma pode ser escrita com um ciclo WHILE ou com um ciclo FOR, no entanto a função procura escrita com um ciclo WHILE não tem tradução directa para um ciclo FOR uma vez que a conjunção de condições não pode ser especificada em ciclos FOR.

3 Aplicação do Visual Basic em Excel

Todos os conceitos apresentados anteriormente tem aplicação na folha de cálculo Excel. Neste capitulo é inicialmente feita uma introdução resumida aos conceitos relevantes do Excel, seguida da aplicação dos conceitos de Visual Basic ao Excel.

3.1 Introdução aos Conceitos do Excel

Basicamente uma folha de cálculo é uma grelha. Cada posição da grelha é denominada por **célula**. Uma célula pode conter um valor introduzido manualmente ou então um resultado de um cálculo.

A titulo de exemplo considere-se a seguinte folha de cálculo onde foram introduzidos os valores de abertura e fecho das cotações de algumas empresas cotadas na Bolsa de Valores de Lisboa para um determinado dia.

	A	B	C	D
1		abertura	fecho	
2	EDP	15	15.1	
3	PT	40	39.8	
4	BRISA	37.3	39	

figura 3.1 - Exemplo de uma folha de cálculo preenchida

O Excel permite introduzir formulas para cálculo de valores a partir dos valores existentes na folha de cálculo. Por exemplo poder-se-ia seleccionar a célula D2 e introduzir a formula para o cálculo da variação diária.

	A	B	C	D
1		abertura	fecho	
2	EDP	15	15.1	=C2-B2
3	PT	40	39.8	
4	BRISA	37.3	39	

figura 3.2 – Introdução de uma formula numa célula

Para escrever uma formula em Excel é necessário digitar o símbolo "=" antes de escrever a formula propriamente dita. A formula da figura 3.2 realiza a subtracção entre os valores das células C2 e B2. Note-se que a convenção no Excel para definir uma célula consiste em especificar primeiro a coluna e depois a linha pretendida.

Escrever a fórmula "=C2-B2" implica que o Excel execute os seguintes passos:

1. calcule o valor dessa expressão
2. e que atribua o valor da expressão à célula activa, neste caso D2.

Para aplicar a formula às outras linhas basta copiá-la para D3,D4,etc... O Excel automaticamente refaz as referencias às células dentro das fórmulas, ou seja, ao copiar a formula da célula D2 para a célula D3, desceu-se uma linha, logo o Excel automaticamente altera todas as referencias da fórmula de forma a que sejam referentes à linha 3. A fórmula resultante da cópia é:

=C3-B3

Considere-se agora um outro exemplo: pretende-se calcular o preço com IVA de um conjunto de artigos, aplicando o IVA ao preço. Na folha de cálculo, figura 3.3, acrescentou-se uma célula com o valor do IVA em vez de utilizar o valor directamente. Com esta estratégia pretende-se a redução das alterações necessárias caso a taxa do IVA seja modificada. Caso se utilizasse o valor do IVA directamente na fórmula, =B2*1.17, seria necessário alterar todas as fórmulas quando a taxa fosse modificada, desta forma basta alterar o valor da célula E1.

Como é ilustrado na figura que se segue, a fórmula correcta para o cálculo do preço com IVA é:

= B2 * E1

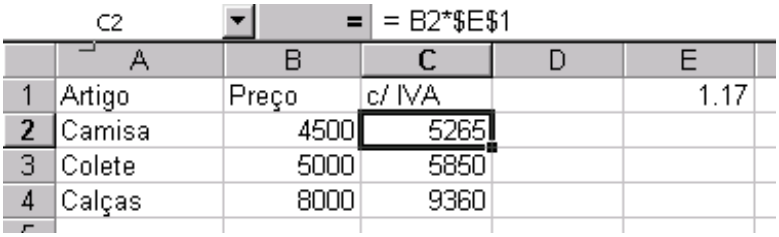


figura 3.3 - Cálculo do IVA em Excel

Os símbolos $ indicam que a referência a esta célula não varia quando a fórmula é copiada, ou seja caso a fórmula seja copiada para a célula C3, a fórmula resultante será:

= B3 * E1

Só a referência à célula B2 foi alterada para B3, a referência a E1 manteve-se.

Em Visual Basic pode-se definir a seguinte função:

```
Function IVA(a, b)
        IVA = a * b
End Function
```

Para invocar esta função escreve-se

= IVA(B2,E1)

tal como ilustrado na figura que se segue.

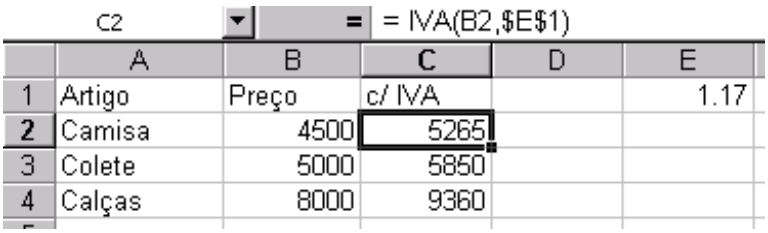


figura 3.4 – Invocação em Excel de uma função definida em VB

Como se pode inferir do código da função acima os conceitos de Visual Basic apresentados anteriormente são directamente aplicáveis ao Excel.

3.1.1 Referências a Células na Folha de Cálculo

No exemplo da secção anterior é desnecessário considerar como argumento a célula E1 uma vez que esta é constante. Como já foi referido também não se pretende utilizar o valor do IVA directamente na fórmula para facilitar a manutenção da folha de cálculo caso o IVA seja alterado.

Sendo assim a nossa função em VB fica só com um argumento, o preço:

```
Function IVA(a)
        IVA = a.Value * valor da célula E1
End Function
```

Em VB o acesso a uma célula é feito segundo regras diferentes das do Excel. Enquanto em Excel se referência uma célula através da sua coluna e linha, em VB utiliza-se a função *Cells* que tem dois argumentos, a linha e a coluna. Note-se que a ordem de referência entre o Excel e o VB é inversa, ou seja em Excel primeiro indica-se a coluna e depois a linha, enquanto que em VB especifica-se primeiro a linha e depois a coluna.

Apresentam-se de seguida alguns exemplos:

Excel	***Visual Basic***
A2	Cells(2,1)
B1	Cells(1,2)
E1	Cells(1,5)

Sendo assim a função em VB fica com a seguinte forma:

```
Function IVA(a)
        IVA = a.Value * Cells(1,5)
End Function
```

Note-se que ao alterar o valor da célula E1 o resultado desta função não é imediatamente alterado. Por omissão, o Excel só actualiza automaticamente as células com formulas que referenciam nos seus parâmetros células que foram alteradas. A ultima versão da função IVA não contem como parâmetro a célula E1, logo não é automaticamente actualizada.

Para actualizar todos os resultados deve-se utilizar a opção "Calculate Full" disponível no Excel 2000. Esta opção realiza as actualizações pretendidas, ou seja, se a célula E1 for alterada, então todas as células às quais a função IVA foi aplicada também serão actualizadas embora não existe nenhuma referencia à célula E1 no cabeçalho da função IVA.

Na ultima versão da formula reduziu-se o número de parâmetros desta de 2 para 1. A questão que se põe é qual é a vantagem da redução do número de parâmetros de uma função?

No caso geral se um parâmetro pode ser calculado à custa de outro então é desnecessário. A sua eliminação da lista de parâmetros evita erros de invocação da função como por exemplo:

Preço(B2, E1)

em vez de

Preço(B2, E1)

No entanto em Excel existe potencialmente uma desvantagem de eliminar parâmetros que sejam referencias a células caso se altera a estrutura da folha de cálculo. Ao utilizar o comando "Insert Column" do Excel este desloca todas as colunas desde o ponto de inserção uma coluna para a direita. Sendo assim caso se utilizasse o comando "Insert Column" na coluna C, localização da célula do IVA passaria a F1.

C2 = =IVA(B2,F1)

	A	B	C	D	E	F
1	Artigo	Preço	c/IVA			1.17
2	Camisa	4500	5265			
3	Colete	5000				
4	Calças	8000				

figura 3.5 Resultado da inserção de uma coluna com o comando "Insert Column"

Como se pode ver na figura 3.5 a fórmula utilizando dois parâmetros é automaticamente actualizada. O Excel refaz todas as referencias a células presentes em fórmulas de forma a compensar a operação realizada, neste caso a inserção de uma nova coluna.

No caso da segunda versão, em que só existe um parâmetro: a célula do preço, o Excel não actualiza o código da função onde a referencia é realizada (Cells(1,5)). Para obter um resultado correcto após a inserção da coluna utilizando o comando "Insert Column" seria necessário actualizar manualmente o código da função.

Resumindo: desde que se garante que a estrutura da folha de cálculo não é alterada é vantajoso reduzir o número de parâmetros de forma a evitar erros de invocação como exemplificado acima. No entanto se não se puder garantir que a estrutura não é alterada então é necessário considerar se convém introduzir todos os parâmetros.

Uma vez que o âmbito desta secção é referenciar células de uma folha de cálculo em Visual Basic serão apresentados mais exemplos onde o número de parâmetros é reduzido ao mínimo possível. No entanto ao realizar um trabalho em Excel é necessário ter em atenção as considerações acima enunciadas sobre os prós e contras da redução de parâmetros e analisar caso a caso qual a melhor estratégia.

Considere agora que na folha de cálculo acima definida era acrescentada mais uma coluna com uma percentagem de desconto para cada item. Pretende-se definir uma função em VB para o cálculo do preço final de acordo com a seguinte expressão em Excel:

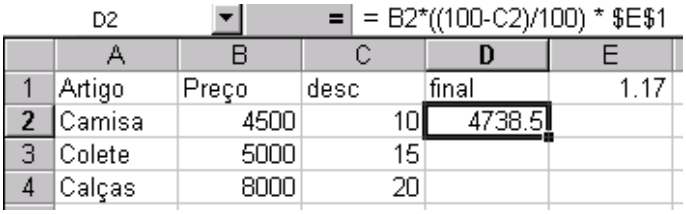


figura 3.6 - Exemplo em Excel

A função em VB é:

```
Function Preço(a, b)
        Preço = a * ((100 - b)/100) * Cells(1,5)
End Function
```

Sendo a sua invocação feita da seguinte forma:

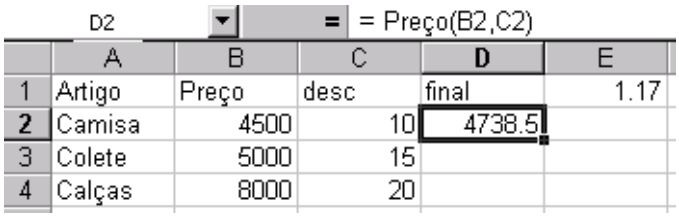


figura 3.7 - Invocação em VB

Por razões de clareza pode-se dividir a função em várias linhas, por exemplo:

```
Function Preço(a, b)
        IVA = Cells(1,5)
        Preço = a * ((100 - b)/100) * IVA
End Function
```

Ou ainda

```
Function Preço(a, b)
        IVA = Cells(1,5)
        PercentagemPagar = (100 - b) / 100
        Preço = a * PercentagemPagar * IVA
End Function
```

Desta forma a leitura da função fica facilitada.

Uma leitura atenta desta ultima função permite concluir que afinal só é necessário um parâmetro. O segundo parâmetro é sempre a célula na coluna seguinte à do primeiro argumento. Pode-se então simplificar a invocação da fórmula de forma a só ter um parâmetro. Ou seja, a invocação seria

= Preço(B2)

É necessário modificar a função em VB de forma a reflectir a alteração desejada. O cabeçalho da função fica portanto alterado de modo a reflectir a redução do número de parâmetros para:

```
Function Preço(a)
....
```

O problema reside no facto de como referenciar a célula que se encontra na coluna a seguir à coluna do primeiro argumento, sendo a linha comum.

A função *Cells* devolve uma referência a uma célula. Por omissão o VB entende a referencia à célula como o seu valor. No entanto uma célula representa um conjunto de propriedades que a definem e não só o seu valor, entre estas temos:

- valor (Value) não é necessário especificar, já que o VB por omissão assume que se pretende o valor da célula
- linha (Row)
- coluna (Column)

Sendo assim, assumindo que o parâmetro da função é *a*, é possível saber a coluna da linha referenciada através da expressão:

a.**Column**

Analogamente para a linha:

a.**Row**

Sendo assim a referência à célula na coluna seguinte à célula do primeiro argumento é

Cells(a.**Row**, a.**Column**+1)

A função em VB é portanto:

```
Function Preço(a)
    Preço = a * ((100-Cells(a.Row, a.Column+1)) / 100) * Cells(1, 5)
End Function
```

Ou, decompondo a função em várias linhas,

```
Function Preço(a)
        IVA = Cells(1, 5)
```

```
        desconto = Cells(a.Row, a.Column + 1)
        PercentagemPagar = ((100 -desconto) / 100)
        Preço = a * PercentagemPagar * IVA
End Function
```

Como pode verificar-se é de mais fácil leitura esta última função.

3.1.2 Referencia a células noutras folhas de cálculo

Em Excel existe o conceito de "Livro", sendo um "Livro" constituído por várias folhas de cálculo. Apresenta-se de seguida um exemplo que utiliza várias folhas de cálculo para guardar informação, ilustrando como escrever expressões que utilizem células em folhas de cálculo distintas.

No exemplo acima considerou-se que o desconto era atribuído artigo a artigo. Mais realisticamente poder-se-ia dizer que existem escalões de descontos. Ou seja em vez de se atribuir ao artigo xpto um desconto de 15%, atribui-se ao artigo um desconto do escalão 1. Para associar os escalões de desconto aos valores a descontar criou-se uma tabela noutra folha de cálculo.

Sendo assim é necessário alterar a folha de cálculo inicial substituindo o valor do desconto pelo escalão de desconto.

	A	B	C	D	E
1	Artigo	Preço	Escalão	PVP	1.17
2	Camisa Xadrez	4500	1	4475.25	
3	Camisa Riscas	4750	3		
4	Colete	5000	2		
5	Casaco	12000	2		
6	Calças	8000	3		
7	Calções	7000	1		

figura 3.8 - Folha de Cálculo Excel com os artigos e respectivos tipos de desconto

Para definir os escalões de descontos utiliza-se uma nova folha de cálculo cuja estrutura é apresentada na figura seguinte. Nesta folha de cálculo o escalão de desconto 1 é apresentado na linha 1, o escalão 2 na linha 2, e assim sucessivamente.

	A
1	15
2	20
3	25

figura 3.9 - Folha de cálculo com os descontos por tipo de artigo

Nota: Os nomes das folhas de cálculo de um "Livro" Excel são apresentados numa barra na parte inferior da aplicação.

figura 3.10 - Barra com os nomes das folhas de cálculo de um "Livro" Excel

Os nomes apresentados são os atribuídos por omissão pelo próprio Excel. Para facilitar a legibilidade das fórmulas e /ou funções VB deve-se alterar esses nomes de forma a atribuir carga semântica.

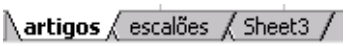


figura 3.11 - Nomes atribuídos para o nosso problema

Nota: Para alterar os nomes de uma folha de cálculo faz-se um duplo clique no nome que pretende alterar e introduz-se o novo nome.

Recapitulando: A folha de cálculo da figura 3.8 é a folha "artigos" e a folha da figura 3.9 é a folha "escalões".

Qual o PVP do artigo "Camisa Riscas"? Sabe-se que o seu preço é de 4750$00, o IVA é 1.17, e o desconto a aplicar é 25% (escalão 3). O desconto é 25% porque se consultou a linha 3 da folha "escalões", ou seja, os descontos estão na folha "escalões", na coluna A e na linha indicada pela coluna "Escalão" na folha "artigos".

Do ponto de vista da escrita da expressão do cálculo do preço em Excel, este exemplo levanta um problema: Como referenciar em Excel a célula na folha de cálculo "escalões" na coluna A e na linha indicada na coluna C da folha "artigos"?

Para resolver este problema é necessário ir por partes:

1. para referenciar uma célula noutra folha de cálculo escreve-se o nome da folha, seguido de ! e da referência à célula. Por exemplo para referenciar a célula A1 da folha "escalões" escreve-se:

    escalões!A1

2. No nosso caso sabe-se que o desconto está na folha "escalões", na coluna A, na linha indicada pela coluna "Escalão" da folha "artigos". Por exemplo, se se considerar o artigo "Camisa Riscas", sabe-se que a referência à célula que contém o desconto a aplicar é:

    escalões!A3

3. Infelizmente não se pode escrever algo do género:

    escalões!A(valor de C3)

4. Pode-se no definir um texto que represente a referência correcta. A parte fixa deste texto é "escalões!A", sendo a parte variável o numero da linha, ou seja supondo que a fórmula está a ser escrita para a segunda linha, o valor da célula C2. Se juntarmos estes dois valores ficamos com o texto

    "escalões!A1"

que representa a referência pretendida.

5. Para juntar um texto a outro texto, ou a um valor, utiliza-se a função CONCATENATE, ou seja a referência textual à célula que contém o desconto para o artigo na linha 2 da folha "artigos" é:

CONCATENATE("escalões!A",C2)

pois o valor de C2 indica a coluna onde se encontra o valor a descontar para o artigo da linha 2.

6. O problema agora resume-se a dada uma referência em formato de texto obter a célula desejada. Para tal utiliza-se a função INDIRECT, ou seja para obter o valor do desconto para o artigo na linha 2 temos de escrever algo do género:

INDIRECT(**CONCATENATE**("escalões!A",C2))

Podemos finalmente escrever a nossa expressão em Excel para o cálculo do PVP como se ilustra na figura seguinte:

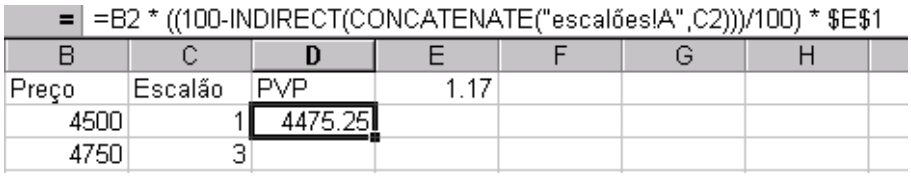


figura 3.12: Fórmula para cálculo do PVP com referência a outra folha de cálculo

Felizmente em VB as coisas são mais simples! Os conceitos já apresentados são quase suficientes para escrever a fórmula, sendo só necessário introduzir uma função para acesso a uma célula noutra folha de cálculo. Ao escrever

Cells(linha, coluna)

por omissão, a referencia é a uma célula na folha de cálculo activa. Para referenciar uma célula noutra folha de cálculo escreve-se

Worksheets(nome_da_folha_de_cálculo).**Cells**(linha, coluna)

Sendo assim a função em VB fica

```
        Function PVP(preço)
1           IVA = Cells(1, 5)
2           escalão = Cells(preço.Row, preço.Column + 1)
3           desconto = Worksheets("escalões").Cells(escalão, 1)
4           PercentagemPagar =  (100 - desconto) / 100
5           PVP = preço.Value * PercentagemPagar  * IVA
        End Function
```

Apresenta-se de seguida a simulação de execução desta função assumindo que é invocada com o parâmetro B2, ou seja PVP(B2).

num	escalão	escalão	desconto	percentagemPagar	obs.
1	-	-	-	-	IVA = valor da célula E1, ou seja 1.17
2	1	1	-	-	*escalão* = valor da célula C2,

					ou seja a célula na coluna a seguir a B2. O valor é 1.
3	1	1	15		desconto = valor da célula A1 (escalão = 1) da folha de cálculo "Escalões", ou seja, 15.
4	1	1	15	0.85	*percentagemPagar* = *(100-15)/100*
5	1	1	15	0.85	Como todos os dados agora calcula-se o PVP, ou seja PVP = 4500 * 0.85 * 1.17

Diferença entre esta versão da função em VB e a anterior em que os descontos eram definidos artigo a artigo é pequena, mas em Excel é enorme!!! A formula em Excel é de muito difícil leitura quando comparada com a função em VB. Para tarefas com um grau de complexidade razoável o VB permite definir a solução de forma mais clara e legível.

3.2 Colecções de Células

Em Excel existe a noção de colecção de células ("range"). Em visual Basic é possível trabalhar com colecções de células da mesma forma que se trabalha com colecções de valores.

Uma colecção de células em Excel tem a seguinte sintaxe

célulaInicial:célulaFinal

Por exemplo

A1:A10 representa todas as células desde a A1 até a A10

A3:D4 representa as células A3,A4,B3,B4,C3,C4,D3,D4

O exemplo aqui apresentado é a soma de uma colecção de células:

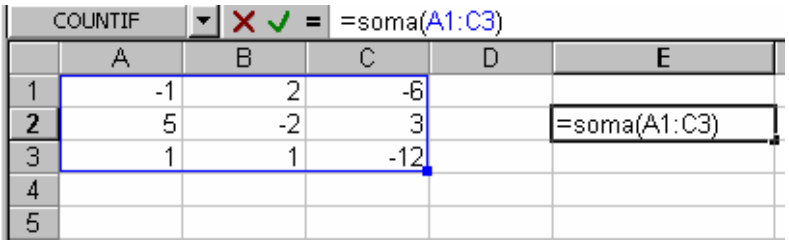


figura 3.13 Soma de uma colecção de células

A figura 3.13 apresenta uma folha de cálculo onde se realiza a invocação de uma função soma que recebe como parâmetro uma colecção de células. O código da função é exactamente igual ao definido para colecções de valores em 2.6.1:Ciclos FOR EACH.

```
Function total(colecção)
        soma = 0
        For Each elem In colecção
                soma = soma + elem
        Next elem
```

```
        total = soma
End Function
```

3.3 Exemplos de Aplicação do VB ao Excel

Considere uma folha de cálculo onde são armazenadas as cotações diárias para empresas cotadas na Bolsa de Valores de Lisboa.

	A	B	C	D
1		EDP	BRISA	PT
2	990104	25.3	51.2	41.2
3	990105	24.9	50.3	42.4
4	990106	25.1	49.2	45.3
5	990107	25.3	48.5	44.2

figura 3.14 Cotações BVL

O primeiro exemplo aqui apresentado consiste em definir uma função que determine em que coluna se encontram os dados de uma determinada empresa. O parâmetro desta função é o nome da empresa.

Para implementar esta função deve-se percorrer a primeira linha da folha de cálculo desde a segunda coluna até encontrar o nome da empresa ou até encontrar uma célula vazia (EMPTY).

```
Function procuraEmp(nome)
    col = 2
    linha = 1
    While (Cells(linha, col) <> Empty And Cells(linha, col) <> nome)
        col = col + 1
    Wend
    If Cells(linha, col) = nome Then
        procuraEmp = col
    Else
        procuraEmp = "Não Existe"
    End If
End Function
```

Resultados desta função serão:

Invocação	Resultado
=procuraEmp("EDP")	2
=procuraEmp("PT")	4
=procuraEmp("EPT")	"Não existe"

De seguida apresenta-se uma função que dada uma coluna onde se encontram os dados de uma empresa devolve a ultima cotação disponível. Para implementar esta função define-se um ciclo que percorra a coluna passada como parâmetro desde a segunda linha, primeira cotação, até encontrar uma célula vazia. O valor da célula na linha acima da linha onde se encontra a célula vazia é o valor da ultima cotação.

```
Function ultimacot(coluna)
```

```
    linha = 2
    While Cells(linha, coluna) <> Empty
        linha = linha + 1
    Wend
    ultimacot = Cells(linha - 1, coluna)
End Function
```

Resultados desta função serão:

Invocação	Resultado
= ultimacot (2)	25.3
= ultimacot (3)	48.5
= ultimacot (4)	44.2

Seguidamente apresenta-se uma função para determinar qual a linha onde se encontra uma determinada data. Esta função é semelhante à apresentada para procurar uma empresa dado o seu nome, só que desta vez a procura é realizada nas linhas e não nas colunas.

```
Function procuraData(data)
    col = 1
    linha = 2
    While (Cells(linha, col) <> Empty And Cells(linha, col) <> data)
        linha = linha + 1
    Wend
    If Cells(linha, col) = data Then
        procuraData = linha
    Else
        procuraData = "Não Existe"
    End If
End Function
```

Finalmente apresenta-se uma função que utilizando as funções acima definidas irá calcular a cotação de uma empresa numa determinada data.

Para tal é necessário determinar em que coluna se encontra a empresa, seguidamente encontrar em que linha se encontra a data e finalmente devolver como resultado o valor da célula na linha e coluna encontradas.

```
Function cotação(emp, dia)
    col = procuraEmp(emp)
    linha = procuraData(dia)
    cotação = Cells(linha, col)
End Function
```

Exemplo: escrever uma função que tem como argumento um índice de uma linha. O resultado desta função é o índice da coluna que contem a primeira célula vazia.

```
Function procuraVazia(linha)
    i = 1
    While (Cells(linha, i) <> Empty)
        i = i + 1
```

```
    Wend
    procuraVazia = i

End Function
```

Considerando a seguinte folha de cálculo, o resultado da invocação

procuraVazia(1)

seria 5.

	A	B	C	D	E	F	G	H
1	2	3	2	4		6	7	
2		1	2	3		3	4	

Caso se invocasse

procuraVazia(2)

Exercício proposto: implemente uma função para determinar a posição da enésima célula vazia. A função deverá receber como parâmetro um índice de uma linha.

Exercício: Escrever uma função que some os elementos de uma linha desde o elemento a seguir à primeira célula vazia até encontrar a segunda célula vazia.

```
Function soma2(linha)
    soma2 = 0
    i = procuraVazia(linha) + 1
    While (Cells(linha, i) <> Empty)
        soma2 = soma2 + Cells(linha, i)
        i = i + 1
    Wend
End Function
```

No caso da folha de cálculo acima apresentada o resultado seria 6+7 = 13.

Exercício: Escreva uma função que determine a soma dos elementos de uma colecção, considerando somente os elementos superiores à média:

Primeiro vamos calcular a média, e seguidamente realizaremos a soma.

```
Function média(A)
    soma = 0
    For Each elem In A
        soma = soma + elem
    Next elem
    média = soma / A.Count()
End Function
```

Esta primeira versão de uma função para calcular a média contabiliza os elementos vazios no cálculo da média. A versão seguinte não conta os elementos vazios.

```
Function média2(A)
    soma = 0
    contador = 0
```

```
    For Each elem In A
        soma = soma + elem
        If elem <> Empty Then
            contador = contador + 1
        End If
    Next elem
    média2 = soma / contador
End Function
```

A função para realizar a soma é

```
Function soma3(A)
    soma3 = 0
    m = média(A)
    For Each elem In A
        If (elem > m) Then
            soma3 = soma3 + elem
        End If
    Next elem
End Function
```

O resultado desta função é 17. Verifique!

Exercício proposto: faça uma simulação de execução para as funções acima apresentadas.

3.4 Exemplos Avançados

Vamos agora imaginar uma função mais complexa: Calcular a variação em escudos (ou juros) da cotação de uma determinada empresa desde um determinado dia até à última data na folha de cálculo.

A função implica os seguintes passos:

1. saber qual a coluna da empresa em questão
2. saber qual a ultima cotação da empresa
3. saber qual a cotação da empresa no dia pretendido
4. calcular a variação

Para saber qual a coluna da empresa podemos utilizar a função procura definida acima. Para saber a última cotação precisamos de saber a linha e a coluna. A coluna é a coluna da empresa devolvida pela função procura. Para saber qual a linha temos de construir uma função que faça uma busca na primeira coluna, a das datas, à procura da ultima casa preenchida, assume-se que não existem casas brancas entre duas datas.

```
Function procuraUltimaData()
        linha = 2         // primeira linha onde aparecem datas
                While (Cells(linha,1) <> Empty)
                        linha = linha + 1
        Wend
        procuraUltimaData = linha -1
End Function
```

Note-se que a variável *linha* ao sair do ciclo indica a primeira linha não preenchida. O resultado da função é a linha anterior, ou seja a última linha preenchida.

Para saber qual a cotação de um determinado dia precisamos também de saber qual a linha e coluna em que o valor se encontra. A coluna obtêm-se como explicado anteriormente, para descobrir a linha temos de fazer uma procura de uma determinada data.

```
Function procuraData(d)
        linha = 2         // primeira linha onde aparecem datas
                While (Cells(linha,1) <> d)
                        linha = linha + 1
        Wend
        procuraData = linha
End Function
```

Nesta função assumimos que a data existe.

Podemos agora construir a função pretendida:

```
        Function variação(empresa, data)
                emp = procura(1,2,empresa)
```

```
                linhaUltimaData = procuraUltimaData()
                linhaData = procuraData(data)
                ultimaCotação = Cells(linhaUltimaData, emp)
                cotaçãoData = Cells(linhaData, emp)
                variação = ultimaCotação - cotaçãoData
        End Function
```

Um exemplo de invocação para determinar a variação da Brisa em escudos desde 6 de Janeiro de 99 seria:

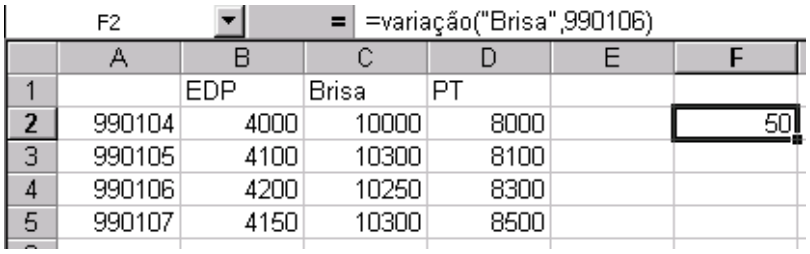


figura 3.15 - Invocação da função variação

Vejamos como seria a função variação se não tivéssemos decomposto o problema, ou seja se escrevêssemos todo o código numa só função:

```
Function variação(empresa, data)

        emp = 2
        While (Cells(1, emp) <> empresa)
           emp = emp + 1
        Wend

        linhaUltimaData = 2
        While (Cells(linhaUltimaData, 1) <> Empty)
            linhaUltimaData = linhaUltimaData + 1
        Wend
        linhaUltimaData = linhaUltimaData - 1

        linhaData = 2
        While (Cells(linhaData, 1) <> data)
               linhaData = linhaData + 1
        Wend
        ultimaCotação = Cells(linhaUltimaData, emp)
        cotaçãoData = Cells(linhaData, emp)
        variação = ultimaCotação - cotaçãoData
End Function
```

Como podemos ver a leitura desta última função é bastante mais complicada, no entanto o resultado é o mesmo, assim como a sua invocação. Ao decompor o problema nos sub problemas que o definem obtemos uma maior legibilidade.

Podemos ainda complicar um pouco mais e pedir qual a coluna da empresa que registou a variação máxima em escudos. Ou seja uma função para calcular o máximo das

variações calculadas anteriormente. Esta função só tem um parâmetro, a data. A empresa não é parâmetro uma vez que todas as empresas vão ser testadas.

```
Function maxVar(data)
        ' col é inicializado com a primeira coluna
        ' onde aparecem empresas
        col = 2

        ' o máximo é colocado a um valor inferior
        ' a qualquer variação que se possa verificar
        máximo = -99999

        ' o ciclo termina quando não existirem mais empresas
        While (Cells(1, col) <> Empty)
                ' calculo da variação da empresa da coluna col
                ' para uma variável auxiliar, var1
                var1 = variação(Cells(1, col), data)

                ' se a variação for superior ao máximo até então,
                ' alterar o máximo e a coluna da empresa
                ' que verificou a variação
                If var1 > máximo Then
                        máximo = var1
                        maxVar = col
                End If
                ' passar à coluna seguinte
                col = col + 1
        Wend
End Function
```

A invocação desta função poderia ser

F3 = =maxvar(990105)

	A	B	C	D	E	F
1		EDP	Brisa	PT		
2	990104	4000	10000	8000		
3	990105	4100	10300	8100		4
4	990106	4200	10250	8300		
5	990107	4150	10300	8500		

figura 3.16- Invocação da função maxvar com a data 5 de Janeiro de 1999-01-23

Note-se que esta função é extremamente ineficiente já que ao invocar a função *variação* para cada empresa estamos sempre a procurar a linha da data assim como da ultima data. Uma versão optimizada da função poderia ser escrita reescrevendo a função variação de modo a retirar as procuras nela existentes:

```
Function variação2(col_empresa, linhaData, linhaUltimaData)
        ultimaCotação = Cells(linhaUltimaData, col_empresa)
        cotaçãoData = Cells(linhaData, col_empresa)
        variação2 = ultimaCotação - cotaçãoData
        End Function
```

Nesta versão da função *variação* temos como parâmetros as linhas e a coluna em que se encontram os valores pretendidos, a função só faz o cálculo da diferença. A função *maxVar* deve então incluir as procuras necessárias para fornecer os parâmetros à função variação

```
Function maxVar(data)
        col = 2
        máximo = -99999
        linhaData = procuraData(data)
        linhaUltimaData = procuraUltimaData()
        While (Cells(1, col) <> Empty)
                var1 = variação2(col, linhaData, linhaUltimaData)
                If var1 > máximo Then
                        máximo = var1
                        maxVar = col
                End If
                col = col + 1
        Wend
End Function
```

A invocação é exactamente igual à da versão anterior, no entanto esta última função é mais eficiente pois só executa uma vez as funções *procuraData* e *procuraUltimaData*.