Módulo 5.5 - **Resumo dos módulos 5.1 a 5.4, exemplos e exercícios**

Resumo dos assuntos abordados nos módulos
- Módulo 5.1 - **Não só cria, o Scheme também modifica…**
- Módulo 5.2 - **Filas, Pilhas e Tabelas são estruturas mutáveis de grande utilidade**
- Módulo 5.3 - **Vectores e cadeias de caracteres são dados mutáveis do Scheme**
- Módulo 5.4 - **Pesquisas, Ordenações e Árvores binárias**

Após o resumo são apresentados alguns exemplos e exercícios para treino da matéria relacionada com os módulos acima referidos.

A utilização do Scheme é fundamental neste módulo.

Resumo dos assuntos abordados nos módulos 5.1 a 5.4

Neste conjunto de quatro módulos, o objectivo principal é mostrar que o Scheme também permite a modificação de dados, *dados mutáveis*, disponibilizando para tal os chamados *modificadores*. Neste contexto surgiram as *listas mutáveis* e com elas construíram-se importantes abstracções, *filas*, *pilhas* e *tabelas*. Também se analisaram abstracções de dados mutáveis disponibilizadas pelo próprio Scheme, *vectores* e *cadeias de caracteres*. No seguimento das cadeias de caracteres, os *ficheiros* de texto são também considerados. Com os ficheiros, os dados produzidos numa sessão de trabalho podem ser guardados para utilização em sessões futuras. A parte final foi organizada em torno das á*rvores binárias*, dando-se um relevo especial às árvores de *pesquisa binária.*

As grandes diferenças entre *vectores* e *listas mutáveis*, com as quais se implementaram as *filas de espera*, as *pilhas* e as *tabelas*, são as seguintes: As listas são estruturas de dados dinâmicas, de grande flexibilidade, pois é possível juntar-lhes ou retirar-lhes elementos. As listas têm um acesso do tipo *sequencial*, pois para chegar a um dos seus elementos obriga a passar por todos os elementos que o antecedem. Os vectores são normalmente estruturas estáticas, de dimensão fixa, mas todos os seus elementos são igualmente acessíveis, através do respectivo índice. As *listas mutáveis* aplicam-se sobretudo em situações em que a dimensão da estrutura de dados varia no tempo e o respectivo processamento adequa-se a um acesso do tipo sequencial. Os *vectores* encontram um campo de aplicação favorável quando o acesso aos seus elementos não privilegia uns em relação a outros e não é importante ter uma estrutura com dimensão variável.
As *árvores de pesquisa binária* surgem como a estrutura que reúne o melhor dos dois mundos, que suporta a pesquisa binária (como o vector), mas com a facilidade de crescer ou diminuir que caracteriza uma estrutura dinâmica (como a lista mutável).

Módulo 5.1
Até aqui o Scheme não permitiu modificar as entidades criadas, obrigando a recorrer a artifícios do tipo - a modificação de uma entidade é simulada criando uma nova entidade com o mesmo nome da entidade que se pretendia modificar. Este tipo de solução tem por base um *construtor*, pode não ser a mais adequada e nem sempre resulta. A forma correcta para realizar este tipo de operação recorre aos chamados *modificadores* que o Scheme disponibiliza para modificar entidades já existentes, sejam elas simples, com o modificador *set!*, sejam elas entidades compostas, com os modificadores *set-car!*, *set-cdr!* e *append!*. Surge assim a *lista mutável.*

Módulo 5.2
Uma forma correcta de trabalhar com entidades que admitam alterações, *dados mutáveis*, recorre aos chamados *modificadores* que o Scheme disponibiliza para alterar entidades já existentes, sejam elas

simples, com o modificador *set!*, sejam elas entidades compostas, com os modificadores *set-car!*, *set-cdr!* e *append!*. Com estes modificadores pode criar abstracções de dados mutáveis com alguma complexidade, nomeadamente, *filas de espera*, *pilhas* e *tabelas*. As abstracções *filas de espera*, *pilhas* e *tabelas* são analisadas e implementadas através de *listas mutáveis*. Todas elas são *estruturas dinâmicas*, uma vez que a respectiva dimensão pode variar, pela introdução ou eliminação de elementos. As duas primeiras organizam os seus elementos em função do tempo, por exemplo, pela ordem de chegada, enquanto que as últimas constituem-se em *registos*, cada um deles caracterizado por uma *chave* e um *valor associado*, e o acesso a qualquer um dos elementos é feito através da sua *chave*.

Módulo 5.3
Com os modificadores que o Scheme disponibiliza (*set!*, *set-car!*, *set-cdr!* e *append!*) é possível criar abstracções de dados mutáveis adequadas a situações que delas necessitem. Algumas destas abstracções foram definidas e implementadas, nomeadamente, *filas de espera*, *pilhas* e *tabelas*. Outras são, logo à partida, disponibilizadas pelo Scheme e nelas se incluem os *vectores* e as *cadeias de caracteres* que são neste módulo consideradas.
As soluções recursivas sobre estas estruturas de dados surgem com uma forma que difere da utilizada com as listas, por não existir o equivalente ao *cdr*. No caso dos vectores e das cadeias de caracteres principia-se por calcular o comprimento destes e um índice vai sendo incrementado até alcançar o valor do comprimento, situação que constituía uma das condições de terminação. No entanto, a implementação do procedimento *string-cdr*, um construtor, permitiu encarar a recursividade em cadeias de caracteres de uma forma semelhante à utilizada para as listas.
A propósito das cadeias de caracteres e da necessidade de guardar dados de uma sessão para outra sessão de trabalho, introduzem-se os *ficheiros*.

Módulo 5.4
As operações de *pesquisa* e *ordenação* são as operações em destaque neste módulo, incidindo especialmente sobre *árvores binárias* e *vectores*. Das árvores binárias, distinguiu-se a de *pesquisa de binária* pelo excelente desempenho $O(lg_2 n)$ que oferece quando se encontra equilibrada ou balanceada.

Exercícios e exemplos

São apresentados alguns exercícios e exemplos para consolidação da matéria tratada nos módulos 5.1 a 5.4. Neste sentido, recomenda-se o estudo do *Anexo A*, no que se refere a *modificadores* (*set!*, *set-car!*, *set-cdr!* e *append!*), *vectores*, *cadeias de caracteres* e *ficheiros*.

Exercício 1
Utilizando a abstracção *fila de espera*, escreva em Scheme um programa que simule e visualize a fila de espera de uma cantina, de acordo com a interacção que se indica, na qual se considera a seguinte codificação:

e	aluno de Electrotecnia
i	aluno de Informática
c	aluno de Civil
q	aluno de Química
m	aluno de Minas

```
> (cantina)
()
> quem-entra? e
(e)
> quem-entra? q
(e q)
> quem-entra? d
```

```
Nao pode ser!...
(e q)
> quantos-saem? 0
(e q)
> quem-entra? q
(e q q)
> quantos-saem? 2
(q)
> quem-entra? i
(q i)
> quantos-saem? 3
Nao pode ser!...
(q i)
> quem-entra? i
(q i i)
> quantos-saem? -1
Vai terminar...
```

Sabe-se que as perguntas *quem-entra?* e *quantos-saem?* são geradas aleatoriamente pelo programa de simulação, mas a probabilidade da pergunta *quem-entra?* é 3 vezes superior à pergunta *quantos-saem?*.

Espaço para desenvolver e testar o programa *cantina*. O ecrã abre com a abstracção *fila de espera* apresentada no decorrer do módulo.

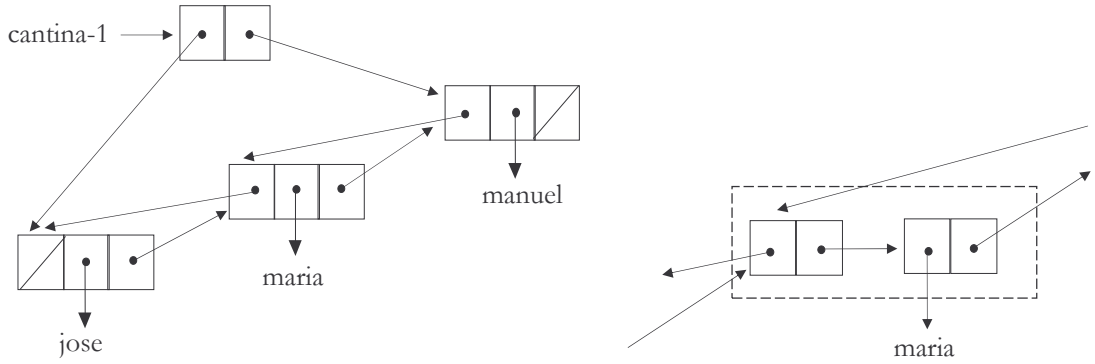
Exercício 2

Na fila de espera modelada como se fosse um par, as operações de saída (do início) e de entrada (no fim) apresentavam-se com um comportamento *O(1)*.

Imagine que estava agora perante uma fila de espera especial, que tinha também operações de entrada no início e de saída no fim!

Acha que estruturar este tipo especial de fila de espera como se fosse um par, manteria todas as operações com um comportamento O(1)? *Se a resposta for negativa, apresente uma solução em que todas as operações serão O(1).*

Para que também estas operações se apresentem com um comportamento *O(1)*, seria necessário recorrer a uma *lista duplamente ligada* ou ligada nos dois sentidos, como se pode ver na figura, em que cada elemento da lista duplamente ligada é modelada por dois pares.



1- Desenvolva o procedimento para visualizar o conteúdo das filas de espera, *visualiza-fila-dupla*, num formato semelhante ao já utilizado.

```
> (visualiza-fila-dupla cantina-1)
(fila-dupla: jose maria manuel)
```

2- Desenvolva, para as filas duplamente ligadas, os seguintes procedimentos:

Construtor

`(cria-fila-dupla)`
devolve uma fila duplamente ligada vazia.

Selectores

`(fila-dupla-vazia? fila)`
devolve #*t*, se fila vazia, ou #*f*, no caso contrário.

`(frente-da-fila-dupla fila)`
devolve o primeiro elemento da fila, mas não a altera. Devolve erro se fila vazia.

`(tras-da-fila-dupla fila)`
devolve o último elemento da fila, mas não a altera. Devolve erro se fila vazia.

Modificadores

`(entra-frente-da-fila-dupla! fila item)`
insere *item* no início de *fila* e devolve o símbolo *ok*.

`(entra-tras-da-fila-dupla! fila item)`
insere *item* no fim de *fila* e devolve o símbolo *ok*.

`(sai-frente-da-fila-dupla! fila)`
retira o primeiro elemento de *fila* e devolve o símbolo *ok*. Devolve erro se fila vazia.

`(sai-tras-da-fila-dupla! fila)`
retira o último elemento de *fila* e devolve o símbolo *ok*. Devolve erro se fila vazia.

Espaço para desenvolver e testar a abstracção *fila de espera duplamente ligada*. O ecrã abre com a abstracção *fila de espera* apresentada no decorrer do módulo.

Exercício 3
Escreva em Scheme o procedimento *procura-no-vector* com dois parâmetros, o vector *vec* e o objecto *obj*, que devolve o índice da primeira ocorrência de *obj* em *vec*. Devolve -1 se *obj* não for um dos elementos de *vec*.

```
> (procura-no-vector '#(g n p r a d l b s) 'a)
4
> (procura-no-vector '#(29 13 96 -5 24 11 9 11 2) 11)
5
> (procura-no-vector '#(29 13 96 -5 24 11 9 11 2) 10)
-1
```

Espaço para desenvolver e testar o procedimento *procura-no-vector*. O ecrã abre vazio.

Exercício 4
Os preços unitários dos produtos adquiridos por um cliente foram registados no vector *precos*, enquanto que as quantidades adquiridas foram registadas no vector *quantidades*. Escreva em Scheme o procedimento *gasto-total* que espera como argumentos dois vectores, um do tipo *precos* e outro do tipo *quantidades* e responde como a seguir se indica.

```
> (define precos (vector 15.50 8.95 12.00))
> (define quantidades (vector 2 5 3))
> (gasto-total precos quantidades)
111.75
```

Espaço para desenvolver e testar o procedimento pedido.
O ecrã abre vazio.

Exercício 5

1- Escreva em Scheme o procedimento *soma-elementos-de-vector* que toma como argumento um vector de elementos numéricos e devolve a soma desses elementos.

```
> (soma-elementos-de-vector (vector 1 3 5 7 9 11))
36
> (soma-elementos-de-vector '#(10 30 50))
90
```

2- Escreva em Scheme o procedimento *multiplica-elementos-de-vector*, semelhante ao procedimento *soma-elementos-de-vector*, mas que devolve o produto dos elementos do vector.

```
> (multiplica-elementos-de-vector (vector 1 3 5 7 9))
945
```

3- Os procedimentos *soma-elementos-de-vector* e *multiplica-elementos-de-vector*, das alíneas anteriores, pela estrutura semelhante que apresentam, sugerem a definição de um procedimento de ordem mais elevada que vamos designar por *acumulador-de-vector*. Com *acumulador-de-vector* a definição daqueles procedimentos seria a seguinte:

```
(define soma-de-elementos-de-vector
  (lambda (vec)
    ((acumulador-de-vector + 0) vec)))

(define multiplica-de-elementos-de-vector
  (lambda (vec)
    ((acumulador-de-vector * 1) vec)))
```

Escreva em Scheme o procedimento *acumulador-de-vector* o qual, como se observa nas duas definições anteriores, espera dois argumentos, um operador e o respectivo valor neutro, e devolve um outro procedimento que tem um vector como único parâmetro.

Espaço para desenvolver e testar os procedimentos pedidos, seguindo a ordem das 3 alíneas.
O ecrã abre vazio.
Indique o que resulta de `(acumulador-de-vector cons '())`.
Procure outras aplicações para *acumulador-de-vector*.

Exercício 6

Escreva em Scheme o procedimento *vector-map* em duas versões, *construtor* e *modificador*, sabendo que na versão construtor responde da seguinte maneira:

```
> (vector-map add1 (vector 10 11 12))
#(11 12 13)
> (vector-map even? (vector 10 11 12 13))
#(#t #f #t #f)
> (vector-map (lambda (el)(if (even? el)
                              (list 'par el)
                              (list 'impar el)))
              (vector 10 11 12 13 14))
#((par 10) (impar 11) (par 12) (impar 13) (par 14))
```

Espaço para desenvolver e testar as duas versões do procedimento pedido.

O ecrã abre vazio.

Exemplo 1
Pretende-se desenvolver uma abstracção designada *equipa-de-futebol*, baseada nos procedimentos:

Construtor

`(cria-equipa)`
Cria e devolve uma equipa, completamente vazia.

Selector

`(visu-equipa equipa)`
Visualiza a constituição de *equipa*.

Modificador

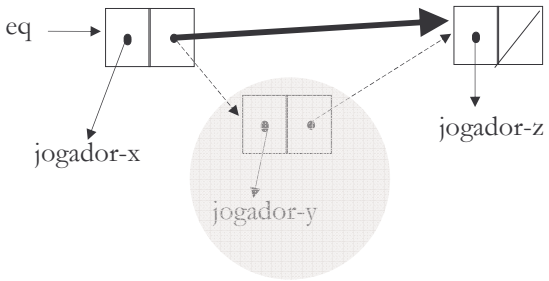
`(entra-equipa! equipa nome-jogador)`
Em *equipa* entra o jogador designado por *nome-jogador*. Se já fizer parte da equipa, esta não sofre qualquer alteração. Em ambos os casos devolve o símbolo *ok*.

`(sai-equipa! equipa nome-jogador)`
De *equipa* sai o jogador designado por *nome-jogador*. Se não fizer parte da equipa, esta não sofre qualquer alteração. Em ambos os casos devolve o símbolo *ok*.

Na solução que se apresenta, uma equipa é modelada por uma lista mutável com cabeça, de comprimento variável.

```
> (define fcp (cria-equipa))
> (visu-equipa fcp)
()
> fcp
(equipa)
> (entra-equipa! fcp 'jardel)
ok
> (visu-equipa fcp)
(jardel)
> (entra-equipa! fcp 'rui-correia)
ok
> (visu-equipa fcp)
(rui-correia jardel)
> (entra-equipa! fcp 'rui-barros)
ok
> (visu-equipa fcp)
(rui-barros rui-correia jardel)
> (sai-equipa! fcp 'folha)
ok
> (visu-equipa fcp)
(rui-barros rui-correia jardel)
> (sai-equipa! fcp 'jardel)
ok
> (visu-equipa fcp)
(rui-barros rui-correia)
```

Antes de passar à análise da implementação da abstracção, observe a figura que mostra o que acontece quando se retira um *jogador-y* de uma equipa, com o modificador *sai-equipa!*.



```
(define cria-equipa
  (lambda ()
    (list 'equipa)))

(define entra-equipa!
  (lambda (equipa jogador)
    (if (not (member jogador (cdr equipa)))
        (set-cdr! equipa               ; ainda não faz parte da equipa...
                  (cons jogador (cdr equipa))))  ; então entra novo jogador
    'ok))                              ; em qualquer dos casos, devolve ok

(define sai-equipa!
  (lambda (equipa jogador)
    (letrec ((aux
              (lambda (eq)
                (cond
                  ((null? (cdr eq)) 'ok); não faz parte da equipa.
                  ;                      Nada a fazer.
                  ((equal? (cadr eq) jogador)
                   (set-cdr! eq                ; faz parte da equipa...
                             (cddr eq))        ; então o jogador é retirado
                   'ok)
                  (else                ; não foi encontrada neste ciclo...
                   (aux (cdr eq)))))))  ; nova tentativa
      ;
      (aux equipa))))

(define visu-equipa
  (lambda (equipa)
    ... ... ... para completar
```

Espaço para completar e testar a abstracção *equipa-de-futebol*.
O ecrã abre com os procedimentos da abstracção, alguns para completar.

Exercício 7

Estude o exemplo anterior e desenvolva uma nova versão dessa abstracção, agora designada por *equipa-de-futebol-listas*, em que a equipa é estruturada como se fosse uma lista de 3 elementos, inicialmente com a seguinte composição:

```
((? ? ? ? ? ? ? ? ? ? ?) (? ? ? ? ? ? ?) ?)
```

O primeiro elemento será preenchido com os 11 jogadores da equipa principal, o segundo com 6 jogadores suplentes e o terceiro com o treinador. Os procedimentos para criação e manipulação das equipas são os seguintes:

`(cria-equipa-fute)`
Cria uma equipa, completamente vazia, com o formato acima indicado.

`(entra-equi! equipa lugar nome-jogador)`
Em *equipa* entra um jogador, designado por *nome-jogador* para um certo lugar da equipa principal, especificado por *lugar* (inteiro de 1 a 11). Se *lugar* já estiver ocupado, será desocupado, para entrar o novo jogador. Devolve o símbolo *ok*. Se lugar > 11, visualiza mensagem de erro.

`(entra-sup! equipa lugar nome-jogador)`
Em *equipa* entra um jogador, designado por *nome-jogador* para um certo lugar da equipa suplente, especificado por *lugar* (inteiro de 1 a 6). Se *lugar* já estiver ocupado, será desocupado, para entrar o novo jogador. Devolve o símbolo *ok*. Se lugar > 6, visualiza mensagem de erro.

`(entra-treina! equipa nome-treinador)`
O treinador designado por *nome-treinador* ocupa o seu lugar em *equipa*. Se *lugar* já estiver ocupado será desocupado, para entrar o novo treinador. Devolve o símbolo *ok*.

`(sai-equi! equipa nome)`

Em *equipa*, o elemento designado por *nome* é procurado, primeiro na equipa principal, depois na equipa de suplentes e, finalmente, no lugar do treinador, e retirado da equipa, ficando o lugar vago. Devolve o símbolo *ok*. Se não for encontrado, visualiza mensagem de erro.

Nesta abstracção não é controlado o facto de um mesmo jogador ocupar mais do que um lugar na equipa.

```
> (define fcp (cria-equi-fute))
> fcp
((? ? ? ? ? ? ? ? ? ? ?) (? ? ? ? ? ?) ?)
> (entra-equi! fcp 1 'correia)
ok
> fcp
((correia ? ? ? ? ? ? ? ? ? ?) (? ? ? ? ? ?) ?)
> (entra-equi! fcp 11 'artur)
ok
> fcp
((correia ? ? ? ? ? ? ? ? ? artur) (? ? ? ? ? ?) ?)
> (entra-equi! fcp 10 'jardel)
ok
> fcp
((correia ? ? ? ? ? ? ? ? jardel artur) (? ? ? ? ? ?) ?)
> (entra-sup! fcp 2 'folha)
ok
> fcp
((correia ? ? ? ? ? ? ? ? jardel artur) (? folha ? ? ? ?) ?)
> (entra-treina! fcp 'oliveira)
ok
> fcp
((correia ? ? ? ? ? ? ? ? jardel artur) (? folha ? ? ? ?) oliveira)
> (sai-equi! fcp 'folha)
ok
> fcp
((correia ? ? ? ? ? ? ? ? jardel artur) (? ? ? ? ? ?) oliveira)
> (entra-equi! fcp 15 's-conceicao)
lugar estranho!...
> (entra-equi! fcp 5 's-conceicao)
ok
> fcp
((correia ? ? ? s-conceicao ? ? ? ? jardel artur) (? ? ? ? ? ?) oliveira)
> (sai-equi! fcp 'artur)
ok
> fcp
((correia ? ? ? s-conceicao ? ? ? ? jardel ?) (? ? ? ? ? ?) oliveira)
> (sai-equi! fcp 'oliveira)
ok
> fcp
((correia ? ? ? s-conceicao ? ? ? ? jardel ?) (? ? ? ? ? ?) ?)
> (sai-equi! fcp 'folha)
este jogador nao esta na equipa!...
> fcp
((correia ? ? ? s-conceicao ? ? ? ? jardel ?) (? ? ? ? ? ?) ?)
```

Espaço para desenvolver e testar a abstracção *equipa-de-futebol-listas*.
O ecrã abre vazio.
Introduza as alterações necessárias para evitar que o mesmo jogador possa ocupar, ao mesmo tempo, mais do que um lugar na equipa.

Exercício 8

Ainda no contexto dos exercícios anteriores, considere agora a modelação de uma equipa como uma lista de 3 elementos, sendo vectores os 2 primeiros. A equipa é inicializada da seguinte maneira: `(#(? ? ? ? ? ? ? ? ? ? ?) #(? ? ? ? ? ?) ?)`.

Nesta abstracção, *equipa-de-futebol-vectores*, não é controlado o facto de um mesmo jogador ocupar mais do que um lugar na equipa.

```
> (define fcp (cria-equi-fute-vec))
> fcp
(#(? ? ? ? ? ? ? ? ? ? ?) #(? ? ? ? ? ?) ?)
> (entra-equi-vec! fcp 11 'artur)
ok
> fcp
(#(? ? ? ? ? ? ? ? ? ? artur) #(? ? ? ? ? ?) ?)
> (entra-equi-vec! fcp 1 'rui-correia)
ok
> fcp
(#(rui-correia ? ? ? ? ? ? ? ? ? artur) #(? ? ? ? ? ?) ?)
> (entra-sup-vec! fcp 4 'folha)
ok
> fcp
(#(rui-correia ? ? ? ? ? ? ? ? ? artur) #(? ? ? folha ? ?) ?)
> (entra-treina-vec! fcp 'oliveira)
ok
> fcp
(#(rui-correia ? ? ? ? ? ? ? ? ? artur) #(? ? ? folha ? ?) oliveira)
> (sai-equi-vec! fcp 'folha)
ok
> fcp
(#(rui-correia ? ? ? ? ? ? ? ? ? artur) #(? ? ? ? ? ?) oliveira)
```

Espaço para desenvolver e testar a abstracção *equipa-de-futebol-vectores*. O ecrã abre vazio. Introduza as alterações necessárias para evitar que o mesmo jogador ocupe, ao mesmo tempo, mais do que um lugar na equipa.

Exercício 9 - uma reflexão sobre estruturas de dados

O exemplo e os exercícios anteriores sobre equipas de futebol apresentam como principal diferença a estrutura de dados em que se baseia a modelação da equipa. Afinal, o que se pretende mostrar é que, de uma forma geral, qualquer estrutura pode servir para resolver o problema, mas algumas adaptam-se melhor que outras.

Analise as implementações que tenha desenvolvido e identifique o impacto das diferentes estruturas nessas mesmas implementações.

Exercício 10

Escreva em Scheme o programa *ocorrencias-de-digitos* que lê um número inteiro e visualiza, para cada dígito decimal, quantas vezes ocorre nesse número.

```
> (ocorrencias-de-digitos)

Indicar um inteiro: 1999

digito 0: 0    digito 1: 1    digito 2: 0    digito 3: 0
digito 4: 0    digito 5: 0    digito 6: 0    digito 7: 0
digito 8: 0    digito 9: 3
```

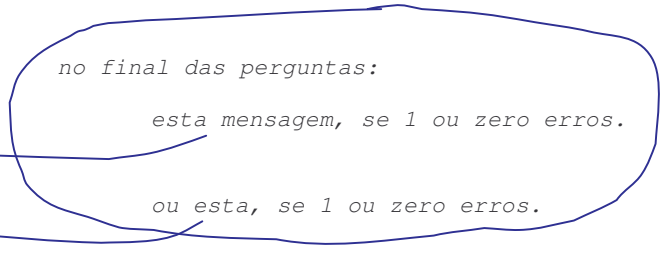
Espaço para desenvolver e testar o procedimento *ocorrencias-de-digitos*. O ecrã abre vazio.

Pista: Utilizar um vector de 10 posições para ir acumulando a ocorrência de cada um dos dígitos à medida que o número fornecido vai sendo analisado.

Exercício 11

Numa outra ocasião, foi proposto um exercício que permitia testar o nível de conhecimento sobre a tabuada de multiplicar. Tratava-se do programa *teste-da-tabuada*, com um só parâmetro, *num*, e que colocava *num* perguntas sobre a tabuada de multiplicar. Os números que surgem nas questões são gerados aleatoriamente.

```
> (teste-da-tabuada 10)              (para um teste com 10 perguntas)

3 x 8 = 24
Resposta certa

2 x 7 = 15
Resposta errada

...

Muito bem

Deve estudar melhor a tabuada
```



Pretende-se agora o programa *teste-da-tabuada-melhorado* que, relativamente à versão referida, difere apenas nas respostas que vai dando. Assim, em vez de *Resposta certa* vai aleatoriamente utilizando uma das seguintes hipóteses: *Muito Bem*; *Excelente*; *Boa resposta*; *Continuar assim*. Por outro lado, em vez de *Resposta errada* vai também escolhendo, aleatoriamente, uma das mensagens: *Errou. Tentar de novo*; *Incorrecto. Tentar novamente*; *E' necessario ter calma, pois a resposta esta' incorrecta*; *Calma, para que o resto corra bem*.

Espaço para desenvolver e testar o programa pedido. O ecrã abre vazio.

Pista: Utilizar dois vectores, um para as respostas certas e outro para as erradas, cujos elementos são as cadeias de caracteres referentes às mensagens indicadas. A selecção aleatória de uma mensagem limita-se à geração de um número aleatório na gama de índices dos vectores.

Exercício 12

Escreva em Scheme o procedimento *cadeia-ao-contrario* que recebe uma cadeia de caracteres como argumento e visualiza-a de trás para a frente.

```
> (cadeia-ao-contrario "ab cdef")
fedc ba
```

Espaço para desenvolver e testar o procedimento pedido. O ecrã abre vazio.

Exercício 13

Escreva em Scheme o predicado *capicua?* que recebe uma cadeia de caracteres como argumento e devolve *#t* se a cadeia recebida tiver a mesma leitura da frente para trás e de trás para a frente.

```
> (capicua "123abba321")
#t
> (capicua "")
#t
```

```
> (capicua "12 3abba321")
#f
```

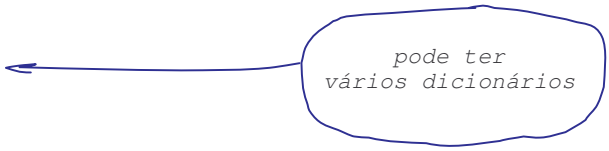
Espaço para desenvolver e testar o procedimento pedido. O ecrã abre vazio.

Exercício 14

Projecto - Jogo *adivinhar-palavras*

O programa *adivinhar-palavras* implementa um jogo que desafia o utilizador a tentar adivinhar palavras. O único parâmetro do programa corresponde a um dicionário de palavras, como se indica no exemplo.

```
(define dicionario-1
  (list "curso"
        "nota"
        "livro"
        "jantar"
        "erro"
        "elevador"
        "letra"))
```



Analise com muita atenção a interacção que se segue.

```
> (adivinhar-palavras dicionario-1)


palavra a adivinhar: (* * * *)    (palavra com 4 letras)
letras erradas: ()                (para já, não há letras erradas...)
Proxima letra:
n

palavra a adivinhar: (n * * *)    (acertou a 1ª letra)
letras erradas: ()                (e ainda não há letras erradas)
Proxima letra:
y
errou...


palavra a adivinhar: (n * * *)
letras erradas: (y)                (já há uma letra errada)
Proxima letra:
r
errou...


palavra a adivinhar: (n * * *)
letras erradas: (y r)
Proxima letra:
t

palavra a adivinhar: (n * t *)
letras erradas: (y r)
Proxima letra:
o

palavra a adivinhar: (n o t *)
letras erradas: (y r)
Proxima letra:
a

palavra a adivinhar: (n o t a)
letras erradas: (y r)
Acertou...
```

1. Faça uma abordagem *de-cima-para-baixo* ao programa *adivinhar-palavras*, começando por identificar as suas tarefas principais e, se for necessário, defina uma abstracção de dados adequada.
2. Escreva em Scheme o programa *adivinhar-palavras*.

Espaço para desenvolver e testar o programa pedido. O ecrã abre vazio.

No mesmo espaço, escreva a versão *adivinhar-palavra-v2* que também tem um único parâmetro, o qual, em vez de ser um dicionário de palavras como acontecia na versão *adivinhar-palavras*, é uma cadeia de caracteres que representa o nome de um ficheiro onde se encontram as palavras do dicionário. Preveja um programa auxiliar para preparar os ficheiros de palavras.

Pista: Formato que se sugere para este tipo de ficheiro:

```
"nota"
"jantar"
"computador"
```

```
> (adivinhar-palavras-v2 "dicionario-1.txt")

palavra a adivinhar: (* * * * *)
...
```

Exercício 15

Projecto - Helicóptero digital

Um helicóptero imaginário é controlado através do teclado, utilizando-se para tal um conjunto de comandos. O helicóptero, quando no ar, não deixa rasto. Para deixar rasto, ao deslocar-se, terá que estar em contacto com o terreno. Os comandos disponíveis são os seguintes:

Comando	Efeito
1	Põe o helicóptero no ar
2	Põe o helicóptero em contacto com o terreno
3	Faz o helicóptero rodar 90º para a direita
4	Faz o helicóptero rodar 90º para a esquerda
5 d	Desloca o helicóptero (pelo ar ou em contacto com o terreno) *d* posições para a frente
6	Visualiza o terreno (com os rastos feitos pelo helicóptero)
7	Limpa os rastos feitos pelo helicóptero no terreno
8	Código não utilizado
9	Guarda o estado do terreno e do helicóptero num ficheiro e termina o programa

O terreno é suposto ser um quadrado de dimensão 20 x 20, a que corresponde 20 x 20 células, cada uma identificada pela linha e coluna respectivas. O helicóptero é caracterizado pela sua *posicao*, dada pela linha e coluna onde se encontra (não necessariamente uma linha e uma coluna do terreno, pois o helicóptero pode deslocar-se para fora dos limites daquele), e é ainda caracterizado pela sua *orientacao* (um dos pontos cardeais: N, S, E ou O) e *situacao* (no ar ou no plano do terreno).

```
linha 20  --------------------
          --------------------
          --------------------
          --------------------
```

```
          ----------H---------
            ...
          --------------------
          --------------------
Linha 1   --------------------

     Helicóptero: L16, C11; no terreno; N
```

O helicóptero é visualizado pela letra H, as células ainda não visitadas são visualizadas com -, enquanto que as já visitadas serão representadas por O. Por exemplo, após o comando 5 com d=2, obtém-se:

```
linha 20  --------------------
          --------------------
          ----------H---------
          ----------O---------
          ----------O---------
            ...
          --------------------
          --------------------
Linha 1   --------------------

     Helicóptero: L18, C11; no terreno; N
```

Exemplo de uma sessão com o programa *helicoptero-digital*, em que as condições iniciais correspondem a terreno limpo e o helicóptero com *posicao: 16 11*; *situacao: no-terreno*; *orientacao: N*.

```
> (helicoptero-digital)
Comeca de novo? (s/n): s
```

Com a resposta n, é visualizada a mensagem Nome do ficheiro:
Esta mensagem será seguida do nome de um ficheiro a abrir em leitura e a simulação retomada a partir do seu conteúdo.

```
Comando: 3                    (roda para a direita 90°)
Comando: 5                    (avança
Posicoes em frente: 3          3 posições)
Comando: 4                    (roda para a esquerda 90°)
Comando: 5                     ...
Posicoes em frente: 2
Comando: 4
Comando: 5
Posicoes em frente: 6
Comando: 6                    (visualiza o terreno)

          --------------------
          --------------------
          -------HOOOOOO------
          -------------O------
          ----------OOOO------
            ...
          --------------------
          --------------------
          --------------------

Comando: 9
E' para guardar em ficheiro? (s/n): s
Nome do ficheiro: "heli.txt"
Terminou a viagem...
```

Com a resposta n, o programa termina de imediato com a mensagem
Terminou a viagem...

1. Defina e implemente, se necessário, uma abstracção de dados compatível com o problema exposto.
2. Defina a estrutura dos ficheiros onde guardará o resultado de uma sessão para ser retomado mais tarde.
3. Faça uma abordagem *de-cima-para-baixo* ao programa *helicoptero-digital*, para identificar as suas tarefas e sub-tarefas principais.
4. Escreva em Scheme o programa *helicoptero-digital*, tomando por base os resultados da abordagem anterior.

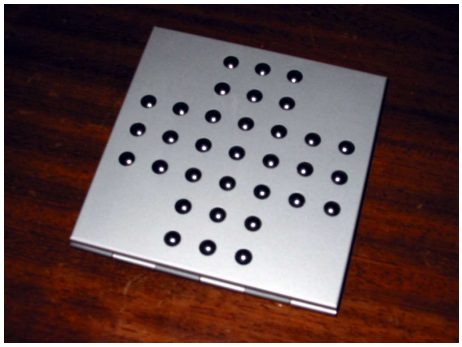
Nota:
O helicóptero na sua deslocação pode ultrapassar os limites do terreno, situação em que não deixa rasto. O programa continuará a actualizar as suas características, de acordo com os comandos que vai recebendo.

Espaço para desenvolver e testar o programa pedido. O ecrã abre vazio.

Pista para uma abstracção de dados
O terreno pode ser representado por um vector de 21 posições, associando a cada uma delas uma cadeia de 21 caracteres. Sugerem-se 21 e não 20, em ambos os casos, para ser possível desprezar a posição de índice 0 e associar, por exemplo, a posição de índice *i* à linha genérica *i*. A sugestão da cadeia de caracteres como elemento do vector tem em vista facilitar a implementação do comando que visualiza o terreno.
Para representar o helicóptero sugere-se uma lista de dois elementos, *posicao* e *caracteristicas*.
O elemento *posicao* poderá ser uma lista de dois inteiros, que representarão a linha e coluna onde se encontra o helicóptero. Deslocar o helicóptero será fácil de implementar, pois traduzir-se-á em somar ou subtrair o valor correspondente à deslocação em termos de linha ou coluna.
Para o elemento *caracteristicas*, propõe-se uma lista de dois elementos, em que o primeiro, designado por *no-terreno*, é um booleano (*#f* significa helicóptero no ar e *#t* sobre o terreno) e o segundo elemento, *direccao*, é um inteiro que pode assumir valores entre 0 e 3, com a codificação seguinte: *0- Norte, 1- Este, 2- Sul, e 3- Oeste*. Com esta codificação de *direccao* pretendeu-se simplificar a implementação dos comandos *rodar 90º para a direita* `(modulo (add1 direccao) 4)` e rodar 90º para a esquerda `(modulo (sub1 direccao) 4)`.

Exercício 16
Projecto - Jogo *comer-esferas*
O jogo ***comer-esferas*** funciona, normalmente, sobre um suporte como o indicado na fotografia.
Imagine que, na situação inicial, a esfera central foi retirada, criando um buraco (não é o caso da fotografia, onde todas as esferas estão presentes).
Agora, uma esfera, ao passar por cima de UMA (e UMA só) esfera para tomar o lugar do buraco central, "come" essa esfera.

Vamos tentar explicar melhor, supondo um tabuleiro de jogo em que o símbolo ***o*** representa um buraco e o símbolo ***+*** representa uma esfera.
O jogo é activado com ***(joga-comer-esferas)*** e começa por mostrar o tabuleiro na situação inicial, ou seja, com um buraco na posição central.



```
> (joga-comer-esferas)
---------------------------
---    1 2 3 4 5 6 7    ---
- 1        + + +        1 -
- 2        + + +        2 -
```

```
- 3    + + + + + + +    3 -
- 4    + + + o + + +    4 -
- 5    + + + + + + +    5 -
- 6        + + +        6 -
- 7        + + +        7 -
---    1 2 3 4 5 6 7    ---
---------------------------
```

Agora, vai indicar que a esfera situada na linha 4 e coluna 6 vai saltar para o buraco situado na linha 4 e coluna 4. O buraco central desaparece, pois vai lá ser colocada uma esfera, mas, em contrapartida, surgem dois buracos: um correspondente ao local onde estava a esfera que "come" e outro onde estava a esfera "comida".

```
out, linha/coluna: 4 6
in,  linha/coluna: 4 4

---------------------------
---    1 2 3 4 5 6 7    ---
- 1        + + +        1 -
- 2        + + +        2 -
- 3    + + + + + + +    3 -
- 4    + + + o o + +    4 -
- 5    + + + + + + +    5 -
- 6        + + +        6 -
- 7        + + +        7 -
---    1 2 3 4 5 6 7    ---
---------------------------
```

Agora, é a esfera situada na linha 2 e coluna 5 que vai saltar para o buraco situado na linha 4 e coluna 5, passando por cima da esfera situada na linha 3 e coluna 5. Observe os buracos criados...

```
out, linha/coluna: 2 5
in,  linha/coluna: 4 5

---------------------------
---    1 2 3 4 5 6 7    ---
- 1        + + +        1 -
- 2        + + o        2 -
- 3    + + + + o + +    3 -
- 4    + + + + + o +    4 -
- 5    + + + + + + +    5 -
- 6        + + +        6 -
- 7        + + +        7 -
---    1 2 3 4 5 6 7    ---
---------------------------
out, linha/coluna:
...
```

O jogo termina quando não é possível fazer mais jogadas, situação que é detectada pelo programa.
O objectivo maior é terminar apenas com uma esfera. Todavia, ganha, quem ficar com o menor número de esferas por comer...
Tenha em atenção que, na indicação de uma posição de saída ou de entrada, o programa rejeita situações de deslocação impossível. Como situações de deslocação impossível temos:

- Ao indicar uma posição de saída, 1- esta não é uma esfera ou 2- na sua linha ou coluna, não existe um percurso de comprimento que termina num buraco e com uma única esfera na passagem.
- Ao indicar uma posição de entrada, 1- esta não é um buraco ou 2- não se encontra na linha ou na coluna da posição indicada como saída, a uma distância de 2 e com uma única esfera entre elas.

Pretende-se que desenvolva o projecto relativo ao jogo *comer-esferas*, tendo em conta os seguintes aspectos:

1. Defina uma abstracção para apoio à implementação do jogo, indicando uma estrutura de dados adequada e os respectivos operadores (construtores, selectores e modificadores).

2. Faça uma abordagem *de-cima-para-baixo* ao programa *comer-esferas*, para identificar as suas tarefas e sub-tarefas principais.
3. Escreva em Scheme a abstracção definida em 1. e o programa *comer-esferas*, tomando por base os resultados da abordagem anterior.

Espaço para desenvolver e testar o programa pedido.
O ecrã abre vazio.

Depois de verificar a correcção do programa pedido, desenvolva uma interface gráfica para este jogo, baseada na *abstracção tabuleiro*. Esta abstracção, apresentada no *Anexo D*, fica disponível através de

```
(require (lib "tabuleiro.scm" "user-feup"))
```

Atrevo-me a adivinhar que, caso tenha definido uma abstracção adequada para apoio à implementação do jogo, como se pedia em 1., é muito natural que o desenvolvimento da interface gráfica não exija um grande esforço da sua parte... Será que adivinhei?