Capítulo 5- Dados mutáveis

Até aqui o *Scheme* não nos permitia modificar as entidades criadas, obrigando-nos 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. Como veremos, esta solução não é a mais correcta e nem sempre resulta. A forma correcta para realizar este tipo de operação recorre aos chamados *modificadores* que o *Scheme* disponibiliza para modificar entidades simples, *set!*, e para modificar entidades compostas, *set-car!*, *set-sdr!* e *append!*. Com estes modificadores é possível criar várias abstracções de dados mutáveis, nomeadamente, *filas de espera* e *tabelas*. Os *vectores* e as *cadeias de caracteres* constituem abstracções de dados mutáveis importantes, disponibilizadas pelo *Scheme*. A propósito das cadeias de caracteres e da necessidade de guardar dados de sessão para sessão de um programa, introduzem-se os *ficheiros*.

1- Modificadores

As entidades ou objectos computacionais têm uma história associada que será actualizada sempre que algum acontecimento relevante ocorra. Vamos supor que uma conta bancária é caracterizada pelo respectivo saldo e que este é associado a um dado do *Scheme*. Quando se processam movimentos naquela conta, depósitos ou levantamentos, o respectivo saldo deverá ser actualizado, ou seja, a situação do objecto deverá ser devidamente alterada. Até aqui, o *Scheme* não nos permitia actualizar os valores associados aos seus objectos, limitação que complica casos como o saldo de contas bancárias. Há quem pense que aplicando, por exemplo, um *define* a um objecto anteriormente definido, que o seu valor é alterado. De facto, assim não acontece. Apenas se cria uma nova entidade com o mesmo nome, desaparecendo a anterior.

```
↦(define objecto-a 10)
objecto-a
↦objecto-a
10
↦(define objecto-a (+ 3 objecto-a))
objecto-a
↦objecto-a
13
```

Apesar de *objecto-a* ser uma nova entidade, parece que tudo se comporta como se fosse uma actualização da entidade inicial. No exemplo descrito, a criação de um novo objecto até nem se

torna demasiado inconveniente. Todavia, assim já não acontece quando o objecto em causa é, por exemplo, uma lista de centenas ou milhares de elementos. Neste caso, a simples alteração de um desses elementos pode obrigar à criação de uma nova lista, com o mesmo nome da lista inicial, copiando as centenas ou milhares dos seus elementos e alterando apenas um deles...
Como vimos, a alteração de um objecto não é possível com a funcionalidade que o *Scheme* nos oferece até aqui. Esta impossibilidade acabou por ser camuflada criando uma nova entidade, a partir da entidade inicial, mas com os inconvenientes já referidos. Mas esta saída nem sempre funciona. Por exemplo, uma entidade criada dentro de um procedimento, podendo ter o mesmo nome de uma outra existente no exterior do procedimento, não será acessível fora desse procedimento. Os objectos definidos num procedimento só dentro dele serão reconhecidos e acessíveis, pois ao terminar a execução de um procedimento, também terminam os objectos nele definidos. A propósito, analisar o exemplo que se segue.

```
(define altera-obj-a
     (lambda ()
       (define objecto-a (+ 5 objecto-a))
       (* objecto-a 2)))
```

```
↦objecto-a
13
↦(altera-obj-a)
36
↦objecto-a
13
```

Ao definir *objecto-a* dentro do procedimento *altera-objecto-a*, a expressão `(+ 5 objecto-a)` é calculada procurando *objecto-a* no nível imediatamente acima, neste caso, o Ambiente Global do Scheme. O cálculo de `(* objecto-a 2)` já utiliza *objecto-a* acabado de definir no interior do procedimento. Não esquecer, contudo, que esta variável *objecto-a* desaparece logo que termina a execução do procedimento onde foi criada.
Este tipo de situação poderá ser resolvida através de uma *forma especial* do Scheme, o modificador *set!*, que apresenta a seguinte forma genérica:

(set! nome expressão)

A regra de cálculo de *set!* é:
⇒ A *expressão* é calculada e o valor resultante é ligado ao objecto *nome*.

Analisar com atenção o exemplo que se segue.

```
(define altera-mesmo-obj-a
  (lambda ()
    (set! objecto-a (+ 5 objecto-a))
    (* objecto-a 2)))
```

```
↦objecto-a
13
↦(altera-mesmo-obj-a)
36
↦objecto-a
18
```

Como o *objecto-a* não é reconhecido no procedimento `altera-mesmo-obj-a`, por não ter sido criado no procedimento nem ser seu parâmetro, vai ser procurado no nível acima (neste caso, no *Ambiente Global do Scheme*). O *objecto-a* é, no procedimento referido, um objecto *livre*, e *set!* vai conseguir modificá-lo onde ele se encontrar.

Quando se considerou o tema *Abstracção de dados*, foram referidos um *construtor*, *cons* e dois *selectores*, *car* e *cdr*. A partir de agora, dispõe-se também de um *modificador*, *set!*. Os objectos manipulados pelos *modificadores* são designados por *objectos* ou *dados mutáveis*, pois é possível alterar o valor que lhes está associado.

Exemplo 5.1

O José, o António e a Maria compram e vendem um certo tipo de artigo, do qual apenas pretendemos conhecer a quantidade que cada um tem em armazém. Inicialmente, o armazém de cada um está completamente vazio.
Para resolver este problema, do ponto de vista dos dados, vamos considerar 3 entidades simples, independentes, para representar os 3 armazéns, designando-os por *jose*, *antonio*, e *maria*.
Para completar esta abstracção, vamos considerar os procedimentos *mostra-situacao*, *vende*, e *compra*, cuja funcionalidade pode deduzir-se da interacção que se segue:

```
↦(mostra-situacao)
jose: 0
antonio: 0
maria: 0
↦(compra 'maria 70)
jose: 0
antonio: 0
maria: 70
↦(compra 'jo 50)
engano!!!
jose: 0
antonio: 0
maria: 70
↦(compra 'antonio 60)
jose: 0
antonio: 60
maria: 70
↦(compra 'maria 50)
jose: 0
antonio: 60
maria: 120
↦(vende 'antonio 20)
jose: 0
antonio: 40
maria: 120
↦(vende 'maria 100)
jose: 0
antonio: 40
maria: 20
```

A solução que se apresenta parece ser suficientemente simples, dispensando comentários.

```
(define jose 0)         ; a representação
(define antonio 0)      ; dos
(define maria 0)        ; armazens

(define mostra-situacao     ; visualizar a situação do
  (lambda ()                ; artigo em armazém
    (display "jose: ")
    (display jose)
    (newline)
    (display "antonio: ")
    (display antonio)
    (newline)
    (display "maria: ")
    (display maria)))
```

```
(define compra                    ; compra de artigo
  (lambda (nome quantidade)       ; aumenta a quantidade de artigo em armazém
    (cond
      ((eq? nome 'jose) (set! jose (+ jose quantidade)))
      ((eq? nome 'antonio) (set! antonio (+ antonio quantidade)))
      ((eq? nome 'maria) (set! maria (+ maria quantidade)))
      (else
        (display "engano!!!")
        (newline)))
    (mostra-situacao)))

(define vende                     ; venda de artigo
  (lambda (nome quantidade)       ; diminui a quantidade de artigo em armazém
    (cond
      ((eq? nome 'jose) (set! jose (- jose quantidade)))
      ((eq? nome 'antonio) (set! antonio (- antonio quantidade)))
      ((eq? nome 'maria) (set! maria (- maria quantidade)))
      (else
        (display "engano!!!")
        (newline)))
    (mostra-situacao)))
```

Exercício 5.1

Em relação ao exemplo anterior, prever as tentativas de venda superior à quantidade de artigo armazenado, apresentando a atitude a tomar face a este tipo de hipótese e as alterações correspondentes ao nível do procedimento *venda*.

2- Pares mutáveis

Nem todas as situações em que os *dados mutáveis* são necessários se resumem a dados simples, manipuláveis através de *set!*. Para dados compostos, constituídos em pares, o *Scheme* disponibiliza mais 3 *modificadores*, cujas formas genéricas são:

(set-car! par expressão)

> O modificador *set-car!* calcula *expressão* e liga o valor resultante ao elemento da esquerda de *par*. O valor que devolve é indefinido.

(set-cdr! par expressão)

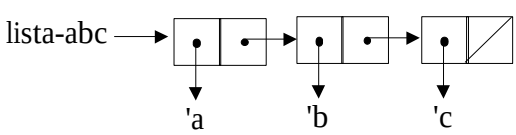
> O modificador *set-cdr!* calcula *expressão* e liga o valor resultante ao elemento da direita de *par*. O valor que devolve é indefinido.

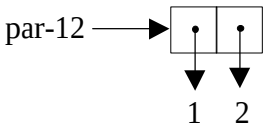
(append! lista-1 lista-2 lista-3 ... lista-n)

> O modificador *append!* rompe o *cdr* do último par da *lista-1* e fá-lo apontar para *lista-2*, faz algo de semelhnate à *lista-2*, cujo último *cdr* é posto a apontar para a *lista-3*, e assim sucessivamente até *lista-n* que não sofre qualquer modificação. O valor que devolve é indefinido.

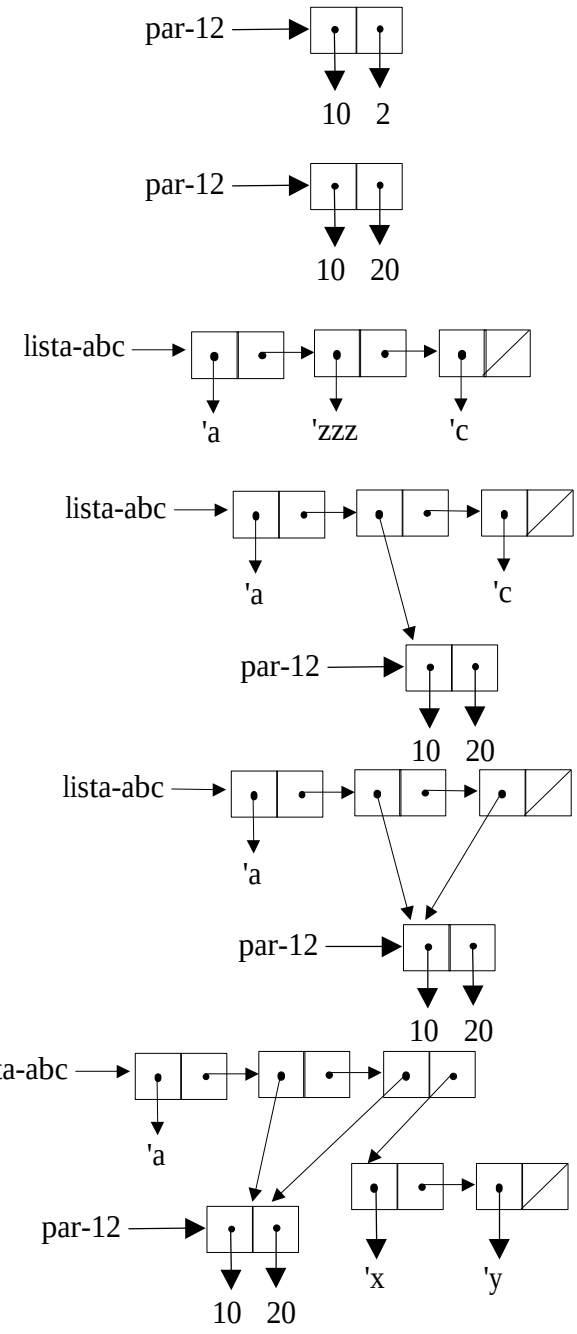
Convém analisar com muita atenção os exemplos que se seguem, utilizando as figuras como suporte.

```
↦(define lista-abc (list 'a 'b 'c))
lista-abc
↦lista-abc
(a b c)

↦(define par-12 (cons 1 2))
par-12
↦par-12
```



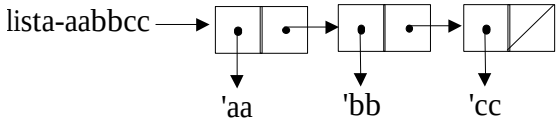


```
(1 . 2)

↦(set-car! par-12 10)
?[1]
↦par-12
(10 . 2)

↦(set-cdr! par-12 20)
?
↦par-12
(10 . 20)

↦(set-car! (cdr lista-abc) 'zzz)
?
↦lista-abc
(a zzz c)

↦(set-car! (cdr lista-abc) par-12)
 ?
↦lista-abc
(a (10 . 20) c)

↦par-12
(10 . 20)

↦(set-car! (cddr lista-abc) par-12)
?
↦lista-abc
(a (10 . 20) (10 . 20))

↦(set-cdr! (cddr lista-abc) '(x y))
?
↦lista-abc
(a (10 . 20) (10 . 20) x y)
```

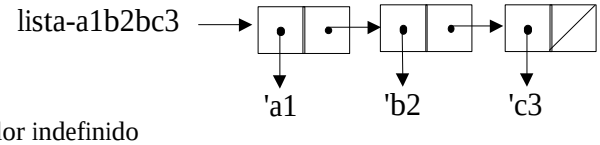


Com o exemplo que se segue procura-se mostrar como funciona o modificador *append!*.

```
↦(define lista-aabbcc '(aa bb cc))
lista-aabbcc
```

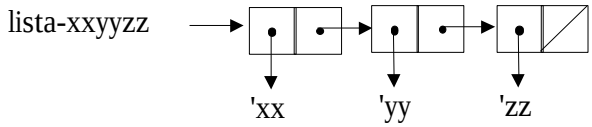



```
↦(define lista-a1b2c3 '(a1 b2 c3))
lista-a1b2c3
```

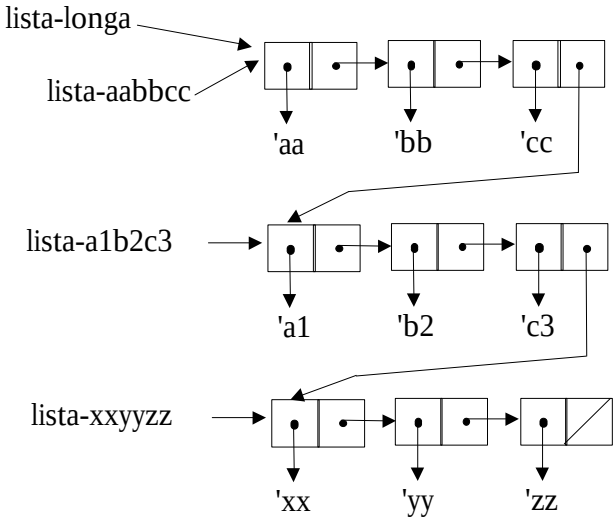


[1] O ponto de interrogação foi aqui utilizado para indicar valor indefinido

```
↦(define lista-xxyyzz '(xx yy zz))
lista-xxyyzz
```

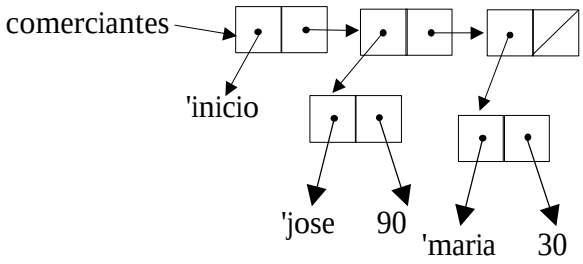



```
↦(define lista-longa (append! lista-aabbcc lista-a1b2c3 lista-xxyyzz))
lista-longa

↦lista-longa
(aa bb cc a1 b2 c3 xx yy zz)

↦lista-aabbcc
(aa bb cc a1 b2 c3 xx yy zz)

↦lista-a1b2c3
(a1 b2 c3 xx yy zz)

↦lista-xxyyzz
(xx yy zz)
```



Exemplo 5.2

Vamos retomar o exemplo dos comerciantes, mas em vez de representar os armazéns como dados, consideram-se constituídos numa lista. Na figura, a lista foi designada por *comerciantes*, e os seus elementos são pares. Em cada um desses pares, o elemento da esquerda corresponde ao nome do comerciante e o da direita à quantidade de artigo armazenado.



Como se pode verificar, existe um elemento no início da lista[2] para garantir que, mesmo na ausência de comerciantes, a lista nunca será nula[3]. Com os modificadores estudados, torna-se relativamente fácil alterar a lista, nomeadamente para lhe acrescentar novos elementos. Vamos utilizar esta possibilidade para introduzir um procedimento que junta novos comerciantes à lista dos comerciantes. Para completar a definição da abstracção, vamos considerar os seguintes procedimentos:

Construtor

(faz-lista-de-comerciantes) - devolve uma lista constituída apenas pelo elemento *cabeça de lista*, neste caso, com o símbolo *inicio*.

Selector

(mostra-situcao-v2) - visualiza a lista *comerciantes*, retirando-lhe a *cabeça de lista*.

Modificadores

(junta-novo-comerciante lista comerciante quantidade) - no início de *lista*, junta *comerciante*, associando-lhe *quantidade*.

---

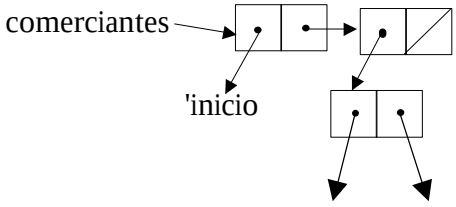
[2] Elemento designado por *cabeça da lista*

[3] Este cuidado é importante, pois os modificadores que vamos utilizar trabalham sobre pares e a lista vazia não é par

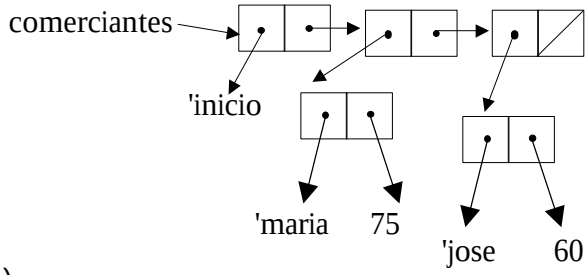
(compra-v2 lista comerciante quantidade) - Em *lista*, adiciona *quantidade* a *comerciante*. Se *comerciante* não existir, visualiza uma mensagem adequada.
(vende-v2 lista comerciante quantidade) - Em *lista*, subtrai *quantidade* a *comerciante*. Se *comerciante* não existir, visualiza uma mensagem adequada.

A funcionalidade destes procedimentos pode clarificar-se com a interacção que se apresenta:

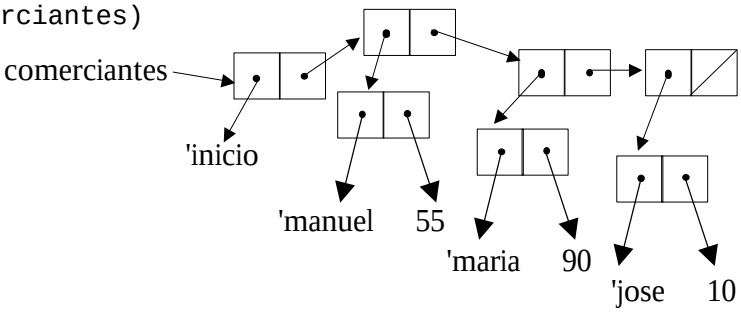
```
↦(define comerciantes (faz-lista-de-comerciantes))
comerciantes                          ; cria uma lta de comerciantes

↦comerciantes                         ; visualiza lista comerciantes como
(inicio)                              ; normalmente é feito pelo Scheme

↦(mostra-situacao-v2 comerciantes)
                                      ; visualiza lista comerciantes, mas
Nao ha' mais comerciantes             ; de uma forma especial

↦(junta-novo-comerciante comerciantes 'jose 60)
ja'-esta'
```




```
↦(mostra-situacao-v2 comerciantes)

jose: 60
Nao ha' mais comerciantes

↦(junta-novo-comerciante comerciantes 'maria 75)
ja'-esta'
↦comerciantes
(inicio (maria . 75) (jose . 60))

↦(mostra-situacao-v2 comerciantes)

maria: 75
jose: 60
Nao ha' mais comerciantes
```




```
↦(compra-v2 comerciantes 'maria 15)
ja'-esta'
↦(mostra-situacao-v2 comerciantes)

maria: 90
jose: 60
Nao ha' mais comerciantes

↦(vende-v2 comerciantes 'jose 50)
ja'-esta'
↦(compra-v2 comerciantes 'jos 45)
jos nao e' comerciante
↦(mostra-situacao-v2 comerciantes)

maria: 90
jose: 10
Nao ha' mais comerciantes

↦(junta-novo-comerciante comerciantes 'manuel 55)
ja'-esta'
↦(mostra-situacao-v2 comerciantes)

manuel: 55
maria: 90
jose: 10
Nao ha' mais comerciantes
```



```
↦comerciantes
(inicio (manuel . 55) (maria . 90) (jose . 10))
```

Recomenda-se uma análise cuidada da solução que se apresenta.

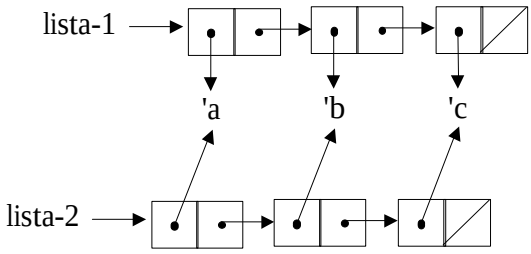
```
(define faz-lista-de-comerciantes
  (lambda ()
    (list 'inicio)))

(define mostra-situacao-v2
  (lambda (lista)
    (letrec ((aux
               (lambda (lis)
                 (cond ((null? lis)
                        (newline)
                        (display "Nao ha' mais comerciantes"))
                       (else
                         (newline)
                         (display (caar lis))
                         (display ": ")
                         (display (cdar lis))
                         (aux (cdr lis)))))))
      (aux (cdr lista)))))

(define junta-novo-comerciante
  (lambda (lista nome quantidade)
    (let ((novo (cons
                  (cons nome quantidade)
                  (cdr lista))))
      (set-cdr! lista novo)
      'ja'-esta')))

(define compra-v2
  (lambda (lista nome quantidade)
    (letrec ((aux                   ; o procedimento interno aux procura na lista
               (lambda (lis)        ; um certo comerciante...
                 (cond ((null? lis)
                        (display nome)                ; se não encontra...
                        (display " nao e' comerciante"))
                       ((eq? nome (caar lis))
                        (set-cdr! (car lis)           ; se encontra...
                                  (+ (cdar lis)
                                     quantidade))
                        'ja'-esta')
                       (else
                         (aux (cdr lis)))))))
      (aux (cdr lista)))))

(define vende-v2
  (lambda (lista nome quantidade)
    (letrec ((aux
               (lambda (lis)
                 (cond ((null? lis)
                        (display nome)
                        (display " nao e' comerciante"))
                       ((eq? nome (caar lis))
                        (set-cdr! (car lis)
                                  (- (cdar lis)
                                     quantidade))
                        'ja'-esta')
                       (else
                         (aux (cdr lis)))))))
      (aux (cdr lista)))))
```

Exercício 5.2

Em relação ao exemplo anterior, prever as tentativas de venda superiores à quantidade de artigo armazenado, bem como as tentativas de juntar novo comerciante com um nome de um comerciante já no activo. Apresentar a atitude do programa face a estas situações e as alterações a introduzir ao nível dos procedimentos *vende-v2* e *junta-novo-comerciante*.

A utilização dos modificadores origina, por vezes, algumas surpresas, como se pretende mostrar nos exemplos que se seguem.

```
↦(define lista-1 '(a b c))
lista-1
↦(define lista-2 '(a b c))
lista-2
```

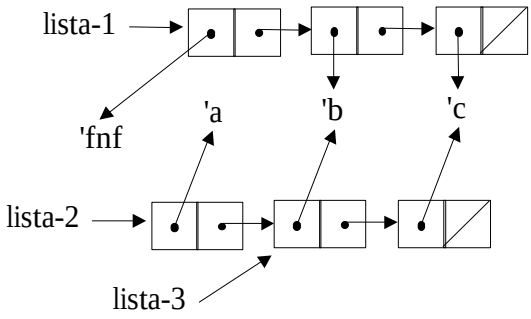


A representação gráfica de *lista-1* e *lista-2* mostra que o *Scheme* não duplica os símbolos que cria, e *'a*, *'b* e *'c* são partilhados pelas duas listas. Apesar do conteúdo destas ser exactamente o mesmo, vejamos a resposta de *eq?* e *equal?* quando as duas listas são comparadas.

```
↦(eq? lista-1 lista-2)
#f
↦(equal? lista-1 lista-2)
#t
```

A resposta *#f* justifica-se pelo facto de *eq?* comparar os apontadores para o início de cada uma das listas, que são diferentes, e não os respectivos conteúdos, que são iguais. A resposta *#t* justifica-se, pois *equal?* compara, não os referidos apontadores como acontecia com *eq?*, mas sim os seus conteúdos. Para *equal?* dois objectos são iguais se, quando visualizados, apresentam o mesmo efeito.

```
↦(set-car! lista-1 'fnf)
?[4]
↦ lista-1
 (fnf b c)
↦ lista-2
(a b c)
```



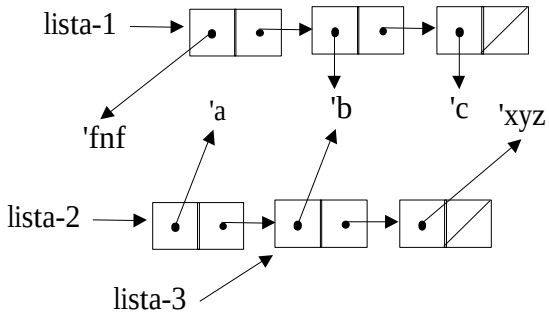
A modificação de elementos de *lista-1* não influenciou *lista-2*, pois, apesar de partilharem elementos, são lista criadas de uma forma independente.

Observar agora o que acontece, quando se cria uma lista a partir de outra existente.

```
↦(define lista-3 (cdr lista-2))
```

[4] Como já atrás se referiu numa situção análoga, o ponto de interrogação também foi aqui utilizado para indicar valor não definido para este procedimento na especificação da linguagem *Scheme*

```
lista-3
↦ lista-3
 (b c)
↦(set-car! (cddr lista-2) 'xyz)
?
↦ lista-1
(fnf b c)
↦ lista-2
(a b xyz)
↦ lista-3
(b xyz)
```



A modificação de elementos de *lista-2* acabou por se repercutir em *lista-3*, pois esta lista foi criada como sendo uma parte de *lista-2*.

Exercício 5.3

Ter em conta as seguintes definições:

```
↦(define lista-a '(a b c))
lista-a
↦(define lista-b lista)
lista-b
```

Indicar e justificar as respostas do *Scheme* representadas por *??*, na interacção que se segue.

```
↦(set-car! lista-a 'fnf)
??
↦ lista-a
??
↦ lista-b
??
```

Exercício 5.4

Na interacção que se segue, indicar as respostas do *Scheme* representadas por *??*.

```
↦(define x (list 'a 'b (cons 'c 5)))
x
↦x
(a b (c . 5))
↦(define y '(e f g h))
y
↦y
(e f g h)
↦(define z (append x y))
z
↦z
??
↦x
??
↦y
(e f g h)
↦(define t (append! x y))
t
↦t
??
↦x
??
↦y
??
```

```
↦(define w (append! x y))
w
↦w
(a b (c . 5) e f g h e f g h e f g h e f g h e f g h e f
g h e f g h e f g h e...)
↦y
??
↦t
??
```

Exercício 5.5

Indicar e justificar o resultado da aplicação de *append!*, representado por *??*.

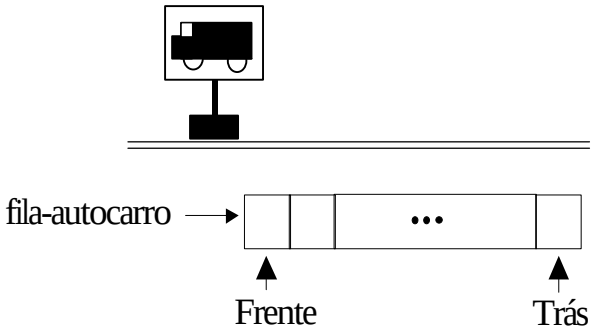
```
↦(define x (cons 1 (cons 7 8)))
x
↦x
(1 7 . 8)
↦(define y '(e f g))
y
↦(define z (append! x y))
z
↦z
??
```

3- Abstracção com dados mutáveis - Filas de espera

Uma abstracção de dados com imensas aplicações é a chamada *Fila de Espera*, reconhecida muitas vezes por *FIFO – First In First Out*.

Facilmente se reconhece a sua utilidade na simulação de uma fila de espera para o autocarro, para a cantina, ou nos veículos que se aproximam de um cruzamento.

Na fila de espera, o primeiro elemento a ser servido é o que se encontra na posição da frente. Quem acaba de chegar ocupará o lugar no fim da fila.



Um conjunto de operações, utilizadas na criação e manipulação de filas de espera, poderia ser:

Construtor

(cria-fila) - devolve uma fila vazia.

Selectores

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

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

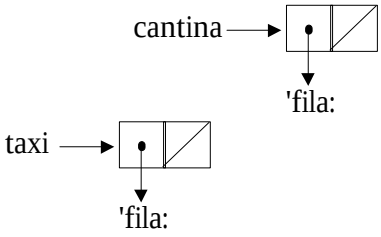
Modificadores

(entra-na-fila! fila item) - insere *item* no fim de *fila* e devolve *fila* alterada.

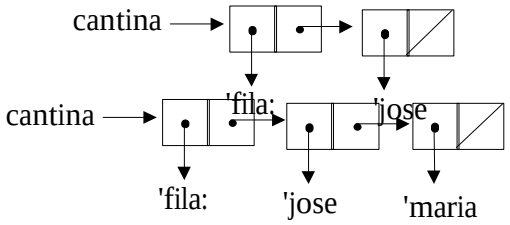
(sai-da-fila! fila) - retira o primeiro elemento de *fila* e devolve fila alterada.

Uma primeira hipótese a explorar para modelar a fila de espera será uma lista, cuja *cabeça* não fará parte da fila. Esta *cabeça* é apenas um símbolo, que no exemplo que se segue é designado por *fila:*, e que garante que uma fila vazia será representada por uma lista não vazia.

```
↦(define cantina (cria-fila))
cantina
↦cantina
(fila:)

↦(define taxi (cria-fila))
taxi
```

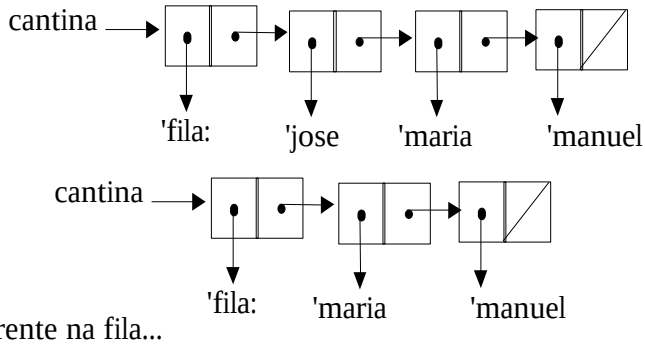



```
↦(entra-na-fila! cantina 'jose)
(fila: jose)

↦(entra-na-fila! cantina
'maria)
(fila: jose maria)
```



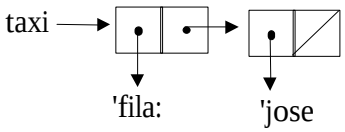
Quem acaba de chegar vai para o fim da fila...

```
↦(entra-na-fila! cantina 'manuel)
(fila: jose maria manuel)

↦(sai-da-fila! cantina)
(fila: maria manuel)
```



Quem sai em primeiro lugar é quem está à frente na fila...

```
↦(entra-na-fila! taxi 'jose)
(fila: jose)
```

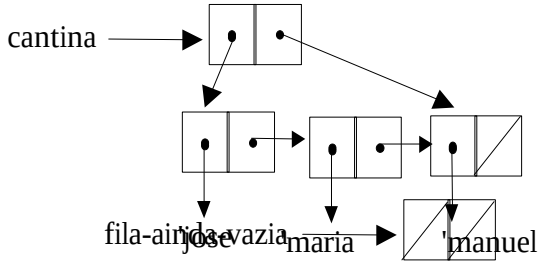


Passemos à implementação dos procedimentos para criação e manipulação de filas de espera.

```
(define cria-fila
  (lambda ()
    (list 'fila:)))

(define fila-vazia?  ; verifica se fila vazia
  (lambda (fila)             ; nesta verificação, a cabeça da lista
    (null? (cdr fila))))     ; não é considerada como fazendo parte da fila

(define frente-da-fila        ; devolve o primeiro elemento da fila
  (lambda (fila)              ; mas não o retira da fila
    (if (null? (cdr fila))
        (display "erro: Fila vazia!...")
        (cadr fila))))

(define entra-na-fila!              ; introdução de um elemento, no final da
  (lambda (fila item)               ; fila, realizada através de append!
    (append! fila (list item))
    fila))

(define sai-da-fila!          ; retira o primeiro elemento da fila
  (lambda (fila)
    (if (null? fila)
```

```
        (display "erro: Fila vazia!...")
        (begin
          (set-cdr! fila (cddr fila))
          fila))))
```

O procedimento `entra-na-fila!` vai percorrer toda a lista que representa a fila de espera para lhe juntar um novo elemento. Trata-se de uma operação *O(n)*. Por outro lado, o acesso ao primeiro elemento da fila, como acontece nos procedimentos `sai-da-fila!` e `frente-da-fila`, é uma operação *O(1)*.



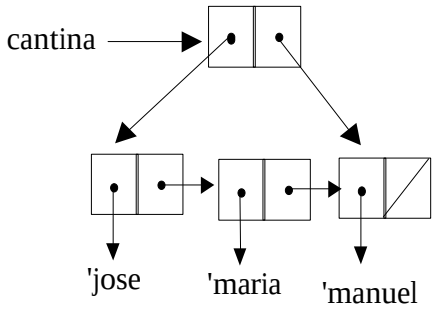
Para se conseguir um acesso *O(1)* também em relação ao último elemento, modela-se a fila de espera como se fosse um par, em que o elemento da esquerda aponta para o primeiro elemento da fila e o elemento da direita para o último elemento da fila. Esta modelação disponibiliza um acesso *O(1)*, tanto para o primeiro como para o último elemento da fila.

Neste caso, a fila é sempre vista como um par, mesmo quando está completamente vazia. Assim, não há necessidade de recorrer ao artifício de colocar um falso primeiro elemento, uma *cabeça de lista*, como se fez anteriormente, para garantir que uma fila vazia não surgisse como uma lista vazia.

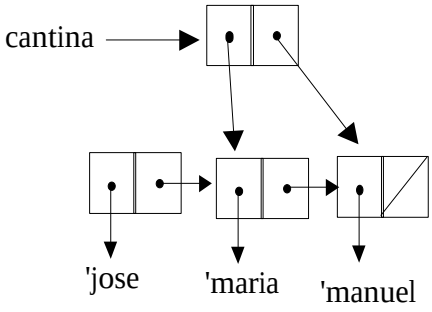
Exercício 5.6

Escrever os procedimentos para criação e manipulação de filas de espera modeladas como se fossem um par. Juntam-se algumas figuras de apoio.

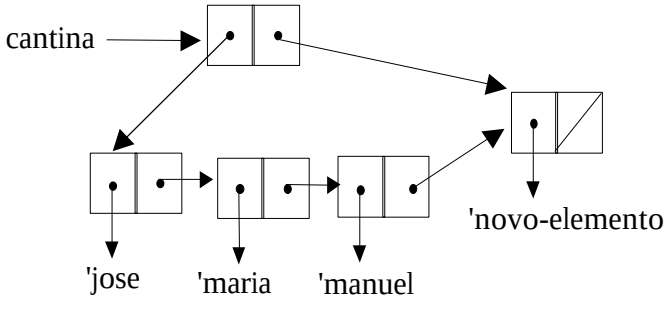
Situação da fila num certo momento



Situação após a saída de um elemento



Situação após a entrada de um elemento



Exercício 5.7

Para além dos procedimentos pedidos no exercício anterior, sugere-se também a definição de um outro, designado por *visualiza-fila*, que recebe um fila de espera como argumento e comporta-se da seguinte maneira:

```
↦(define fila-surpresa (cria-fila))
```

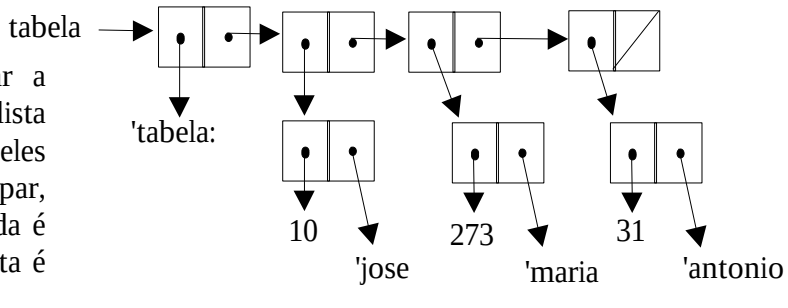
```
fila-surpresa
↦(visualiza-fila fila-surpresa)
(fila: )
↦(entra-na-fila! fila-surpresa 'jose)
ja'-esta'...
↦(entra-na-fila! fila-surpresa 'maria)
ja'-esta'...
↦(visualiza-fila fila-surpresa)
(fila: jose maria)
↦(sai-da-fila! fila-surpresa)
ja'-esta'...
↦(visualiza-fila fila-surpresa)
(fila: maria)
```

4- Abstracção com dados mutáveis - Tabelas

Para além dos conjuntos, filas de espera e outras abstracções de dados, também as tabelas se apresentam como uma abstracção com grande utilidade. Perante várias abstracções, cada uma com as características próprias, caberá ao programador escolher a que melhor se adeqúe ao problema que pretenda resolver ou, caso não encontre nenhuma interessante, criar uma nova abstracção. Nos conjuntos, os seus elementos são pura e simplesmente lançados no seu interior e, quando é necessário aceder a algum deles, a procura faz-se elemento a elemento. Nas filas de espera, os seus elementos já se encontram organizados, normalmente, por ordem de chegada. Nas tabelas, os seus elementos constituem-se em registos, associando cada um deles um valor e uma ou mais chaves, a partir das quais se acede ao valor.

Por exemplo, a tabela que se segue é composta por três elementos ou registos.

Número	*Nome*
10	jose
273	maria
31	antonio



Podemos começar a imaginar a tabela, representada por uma lista de registos, cada um deles implementado através de um par, em que o elemento da esquerda é a chave, e o elemento da direita é o valor associado à chave.

Para evitar que a lista, na ausência de registos, apareça completamente vazia, inclui-se um primeiro elemento fixo, a *cabeça da lista*, o qual não faz parte efectiva da tabela. Para completar a abstracção *tabela*, vamos considerar os seguintes procedimentos básicos:

Construtor

(cria-tabela) - devolve uma tabela vazia.

Selector

(procura-tabela chave tabela) - se *chave* existir em *tabela*, devolve o *valor* associado a *chave*. Caso contrário, devolve *#f*.

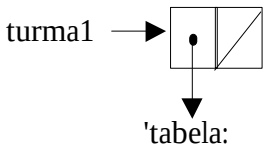
Modificadores

(insere-na-tabela! chave valor tabela) - se *chave* existir em *tabela*, actualiza o valor associado a *chave* com *valor*. Caso contrário, cria um novo registo com *chave e valor* e insere-o no início da tabela. Em ambos os casos, devolve o símbolo *ok*.

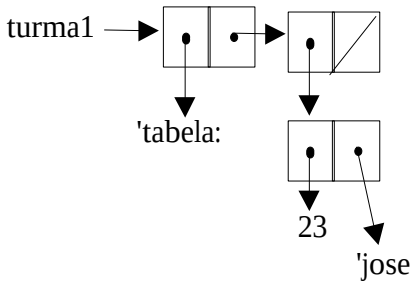
(retira-da-tabela! chave tabela) - se *chave* existir em *tabela*, retira da tabela o registo *chave valor-associado*. Caso contrário, deixa a tabela intacta. Em ambos os casos, devolve o símbolo *ok*.

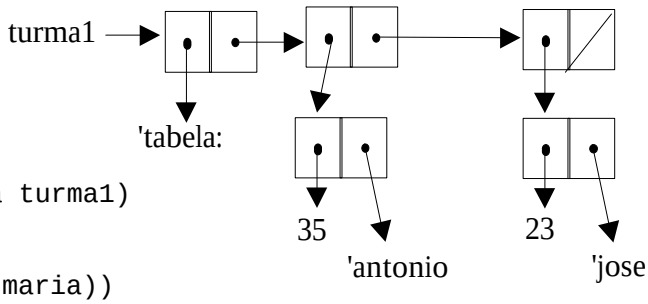
Segue-se uma sequência de utilização dos procedimentos acabados de referir, a fim de clarificar o seu modo de funcionamento.

```
↦(define turma1 (cria-tabela))
turma1
↦turma1
(tabela:)
```

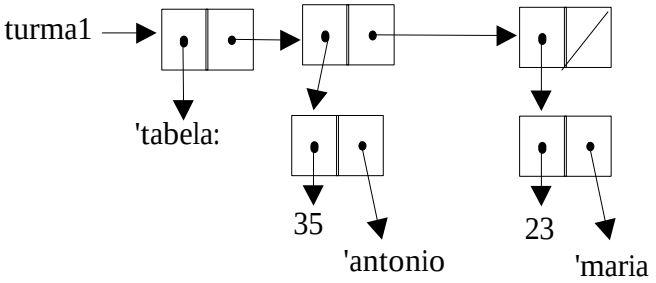



```
↦(insere-na-tabela! 23 'jose turma1)
ok
↦turma1
(tabela: (23 . jose))

↦(insere-na-tabela! 35 'antonio turma1)
ok
↦turma1
(tabela: (35 . antonio) (23 . jose))
```






```
↦(insere-na-tabela! 23 'maria turma1)
ok
↦turma1
(tabela: (35 . antonio) (23 . maria))
```



Passemos agora à implementação dos procedimentos para criação e manipulação de tabelas.

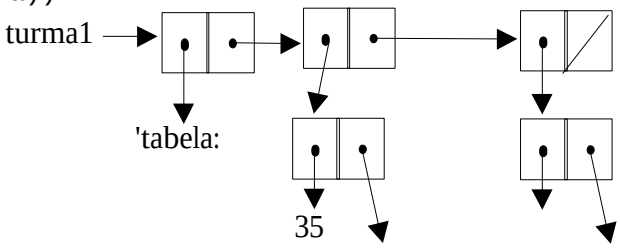
```
(define cria-tabela
  (lambda ()
    (list 'tabela:)))

(define procura-tabela
  (lambda (chave tabela)
    (let ((registo (assoc chave tabela)))
      (if registo
          (cdr registo)
          #f))))
```

```
(define assoc[5]
  (lambda (chave registos)
    (cond ((null? registos) #f)
          ((equal? chave (caar registos))(car registos))
          (else (assoc chave (cdr registos))))))

(define insere-na-tabela!
  (lambda (chave valor tabela)
    (let ((registo (assoc chave (cdr tabela))))
      (if registo
          (set-cdr! registo valor)
          (set-cdr! tabela
                    (cons (cons chave valor)
                          (cdr tabela)))))
    'ok))
```

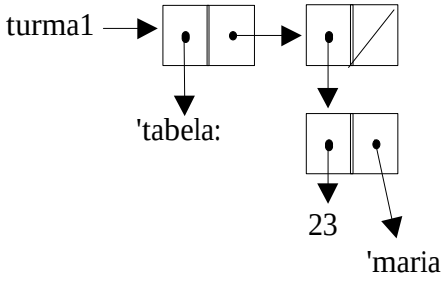
Exercício 5.8

Escrever o procedimento *retira-da-tabela!* conforme especificação já apresentada. Para clarificar o problema, junta-se alguma informação adicional.

```
↦turma1
(tabela: (35 . antonio) (23 . maria))
```



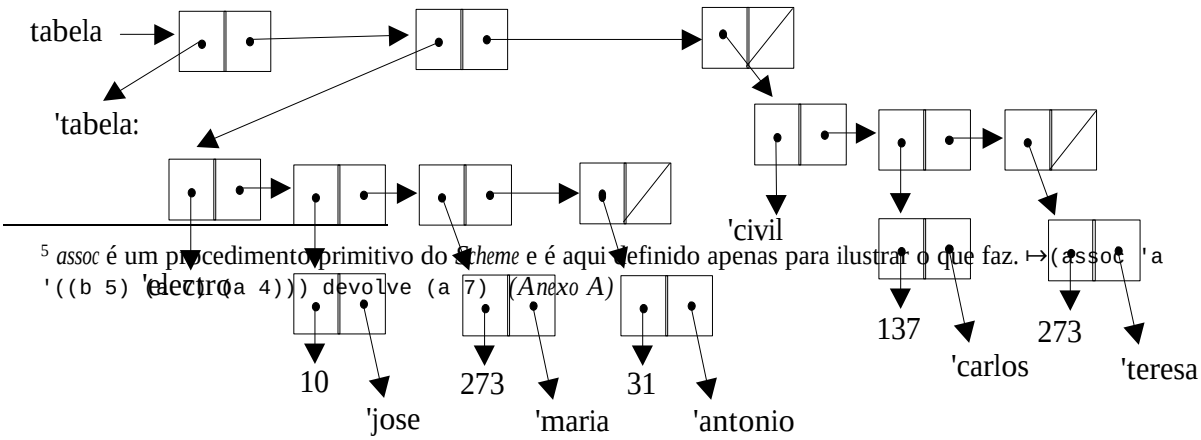

```
↦(retira-da-tabela! 35 turma1)
ok
↦turma1
(tabela: (23 . maria))
```



A abstracção *tabela*, em que os elementos são acedidos através de uma chave, é designada por *tabela unidimensional*. Quando os elementos da tabela são acedidos através de duas chaves, estamos perante a *tabela bidimensional*. No exemplo que se segue, as chaves associadas a cada um dos alunos são o departamento e o número de aluno:

electro		civil	
10	jose	137	carlos
273	maria	273	teresa
31	antonio		

Com as chaves *electro* e *273* acede-se a *maria*, e com *civil* e *273* o aluno alcançado é *teresa*.



[5] *assoc* é um procedimento primitivo do *Stheme* e é aqui definido apenas para ilustrar o que faz. ↦(assoc 'a '((b 5) (a 4))) devolve (a 7) (*Anexo A*)

Verificar, na figura, que a tabela bidimensional pode ser vista como uma lista de tabelas unidimensionais. De facto, quer a subtabela *electro* quer a subtabela *civil* apresentam uma estrutura de tabela unidimensional, ou seja, uma lista cujo primeiro elemento é a chamada *cabeça de lista*, que não faz parte efectiva da tabela, e os restantes elementos são registos, cada um deles implementado através de um par, em que o elemento da esquerda é a chave, e o elemento da direita é o valor associado à chave.

Exercício 5.9
Tomando por base os procedimentos apresentados para a *tabela unidimensional*, escrever os procedimentos respectivos para a *tabela bidimensional*. Testar cuidadosamente a solução encontrada e ter em conta que estamos perante um problema com um grau de dificuldade relativamente elevado.

5- Abstracção com dados mutáveis - Vectores

Em primeiro lugar, convém perceber o que é que a abstracção *vector* traz de novo, relativamente às abstracções já consideradas. Nos *conjuntos*, os seus elementos são pura e simplesmente lançados no seu interior e, quando é necessário aceder a algum deles, a procura faz-se elemento a elemento, pois não há, normalmente, uma ordem estabelecida entre eles. Nas *filas de espera*, os seus elementos já se encontram organizados, normalmente, por ordem de chegada. Nas *tabelas*, os seus elementos constituem-se em registos, associando cada um deles um valor e uma ou mais chaves, a partir das quais se acede ao valor. Relembrando o que foi feito, todas estas abstracções se basearam em listas. Os conjuntos utilizaram listas não mutáveis, mas agora poderiam ser implementados com listas mutáveis, como aconteceu com as filas de espera e com as tabelas. Em todos estas abstracções, o acesso aos seus elementos é sequencial, começando no primeiro e avançando na direcção do último. No caso especial da fila de espera em que, fundamentalmente, só interessa aceder ao primeiro e ao último elementos, foi utilizada uma solução que minimizava o problema do acesso sequencial, modelando a fila como se fosse um par, em que o elemento da esquerda apontava para a parte da frente da fila e o elemento da direita para a parte de trás.
A abstracção *vector* apresenta uma característica nova que é a possibilidade de acesso directo[6] a qualquer um dos seus elementos. A cada elemento é associado um índice e, através desse índice, o vector dá acesso directo a esse elemento sem obrigar a passar por qualquer um dos outros.
Por exemplo, se a lista *ls* contiver 1000 elementos, o acesso ao elemento que se encontra na posição 900[7], poderá ser conseguido fazendo

```
↦(list-ref ls 900)
```

Podemos imaginar que esta operação implica 900 *cdr*'s seguidos de um *car*. Se os mesmos 1000 elementos constituírem o vector *vec*, o elemento colocado na posição 900 é acedido directamente, fazendo

[6] É o chamado acesso aleatório *(random access)*, pois as condições de acesso são independentes da posição do elemento a aceder.
[7] Na posição zero está o primeiro elemento.

```
↦(vector-ref vec 900)
```

Estamos perante uma abstracção que se recomenda em situações em que ocorre com grande frequência o acesso aos seus elementos, para uma simples leitura ou até mesmo para modificação. Os procedimentos para criação e manipulação de vectores podem ser consultados no *Anexo A*. Os exemplos que se seguem ilustram o modo de utilização desses procedimentos.

```
↦(define v1 (vector 'a 6 'ab 90)) ; cria um vector, enumerando os seus elementos
v1
↦(vector? v1)                     ; predicado que verifica se argumento é vector
#t
↦v1
#(a 6 ab 90)                      ; visualização de um vector feita pelo Scheme
↦(define v2 (vector-copy v1))     ; faz cópia de um vector
v2
↦v2
#(a 6 ab 90)
↦(vector-fill! v2 'abc)           ; preenche um vector existente
valor não definido
↦v2                               ; efeito de vector-fill! sobre v2
#(abc abc abc abc)
↦(vector-length v1)               ; devolve o comprimento de um vector
4
↦(vector-ref v1 1)                ; acesso a um elemento - leitura
6
↦(vector-ref v1 2)
ab
↦(make-vector 3)                  ; outra forma de criar um vector
#(() () ())
↦(make-vector 3 'elem)            ; ainda outra forma de criar um vector
#(elem elem elem)
↦(list->vector '(1 2 (6 a) 7))    ; criar um vector a partir de uma lista
#(1 2 (6 a) 7)
↦(vector->list '#(1 2 3 4))       ; criar uma lista a partir de um vector
(1 2 3 4)
↦(vector->list (vector 1 2 3 4))
(1 2 3 4)
↦(define v1 (vector 0 2 4 6 8))
v1
↦v1
#(0 2 4 6 8)
↦(vector-set! v1 2 5)             ; modificar um elemento de um vector
valor não definido
↦v1                               ; efeito de vector-set! svector v1
#(0 2 5 6 8)
```

Exemplo 5.3

Um programa faz a estatística sobre os lançamentos de um dado. O utilizador do programa é interrogado sobre o número de lançamentos que pretende simular. Se este número for superior a zero, passa-se à simulação de um número equivalente de lançamentos de um dado, sendo anotadas as ocorrência de cada uma das suas 6 faces. No final, são visualizadas as frequências de ocorrência dessas faces.

```
↦(estatistica-dos-dados)
Quantos lançamentos (< 1, termina)?
200
Resultados:
1- 40
```

```
2- 35
3- 33
4- 33
5- 35
6- 24

Quantos lançamentos (< 1, termina)?
2000
Resultados:
1- 346
2- 339
3- 327
4- 329
5- 339
6- 320

Quantos lançamentos (< 1, termina)?
0
Acabou...
```

A primeira decisão que se tomou refere-se à forma de representar computacionalmente as ocorrências das faces do dado, durante a simulação dos vários lançamentos. Optou-se por um vector de 6 posições, inicialmente todas elas com o valor zero, e à medida que a simulação progride, as ocorrências das faces vão sendo contadas e registadas no citado vector. As ocorrências da face 1 são contabilizadas na posição 0[8], as da face 2 na posição 1, ..., e as da face 6 na posição 5.

O programa *estatistica-dos-dados* foi decomposto nas seguintes tarefas:

⇒ Pergunta inicial sobre o número de lançamentos, incluindo a condição de finalização;

⇒ Leitura do número de lançamentos;

⇒ Sobre o número de lançamentos:

- Se se verificar a condição de finalização, visualização da mensagem respectiva;
- Caso contrário, simulação dos lançamentos, visualização do resultado, e novo lançamento do programa *estatistica-dos-dados*.

```
(define estatistica-dos-dados
  (lambda()
    (display "Quantos lançamentos (< 1, termina)? ")
    (let ((numero-lancamentos (read)))
      (cond ((<= numero-lancamentos 0)
             (newline)
             (display "Acabou..."))
            (else
              (let ((contador-faces (lancar-dado numero-lancamentos)))
                (newline)
                (display "Resultados:")
                (newline)
                (visu-vector contador-faces)
                (estatistica-dos-dados)))))))
```

Das tarefas do programa *estatistica-dos-dados,* a mais complexa e que exige um tratamento mais pormenorizado tem a ver com a simulação dos lançamentos do dado. É realizada pelo procedimento *lancar-dados*, onde é definida a variável local *conta-faces*, um vector de seis posições. Este procedimento apoia-se no procedimento local *aux* que executa os lançamentos necessários e actualiza o vector, que será devolvido no final.

```
(define lancar-dado
  (lambda (num-vezes)
    (let ((conta-faces (make-vector 6 0)))
      (letrec ((aux
```

[8] Não esquecer que as posições dos vectores começam a partir do índice zero

```
                  (lambda (n-vezes)
                    (if (zero? n-vezes)
                        conta-faces
                        (let ((face-menos-1 (sub1 (roleta-1-6))))
                          (vector-set!
                            conta-faces
                            face-menos-1
                            (add1 (vector-ref conta-faces
                                              face-menos-1)))
                          (aux (sub1 n-vezes)))))))
        ;
        (aux num-vezes)))))

(define roleta-1-6              ; gera e devolve um número aleatório entre 1 e 6
  (lambda()
    (add1 (remainder (random) 6))))
```

Das tarefas do programa *estatistica-dos-dados*, ainda se distingue a visualização do resultado, mais especificamente, a visualização o vector que representa a ocorrência das faces. Esta visualização baseia-se numa solução recursiva, em que a redução do problema no passo recursivo, assume uma forma diferente da utilizada para as listas e que é característica dos vectores. Inicialmente determina-se o comprimento do vector a processar e define-se um procedimento local, neste caso designado por *aux*, com o parâmetro *indice*. A chamada inicial de *aux* apresenta-se com o argumento zero, que vai sendo incrementado em cada nova chamada. A condição de terminação corresponde a verificar-se a igualdade entre este argumento e o comprimento do vector, pois sendo este comprimento igual a 6, os índices válidos localizam-se entre 0 e 5.

```
(define visu-vector
  (lambda (vec)
    (let ((comprim (vector-length vec)))
      (letrec ((aux
                  (lambda (indice)
                    (if (= indice comprim)
                        (newline)
                        (begin
                          (display (add1 indice))
                          (display "- ")
                          (display (vector-ref vec indice))
                          (newline)
                          (aux (add1 indice)))))))
        (aux 0)))))
```

Exercício 5.10

Em relação ao exemplo anterior, como o índice da primeira posição dos vectores é 0, associou-se a posição genérica *i* à face *i+1*. Há quem prefira, em situações análogas, usar um vector com uma posição a mais do que o necessário, desprezar a posição de índice 0 e fazer a associação da posição genérica *i* à face *i*. Para seguir esta via, introduzir as alterações necessárias à solução apresentada no exemplo anterior.

Para criar um vector de comprimento 5, com os 5 primeiros pares, podemos escrever:

```
↦(vector 0 2 4 6 8)
#(0 2 4 6 8)
```

Para criar um vector de comprimento 40, com os primeiros 40 pares, a tarefa já se complica:

```
↦(vector 0 2 4 6 8 10 12 14 16 18 20 22 24 26 28 30 32 34 36 38 40 42
44 46 48 50 52 54 56 58 60 62 64 66 68 70 72 74 76 78)
#(0 2 4 6 8 10 12 14 16 18 20 22 24 26 28 30 32 34 36 38 40 42 44 46 48
50 52 54 56 58 60 62 64 66 68 70 72 74 76 78)
```

Devemos reconhecer que, em situações como esta, será de tentar uma forma diferente para criar vectores. Assim, a ideia a explorar passa pela definição de um procedimento, designado por *gera-vector-com-proc*, que recebe como argumentos um procedimento, *proc*, e um valor inteiro,

comp, e devolve um vector de comprimento *comp*, cujo elemento genérico *i* é igual ao valor de *proc* em *i*.

```
↦(gera-vector-com-proc (lambda (i) (* 2 i)) 5)
#(0 2 4 6 8)
↦(gera-vector-com-proc (lambda (i) (* 2 i)) 40)
#(0 2 4 6 8 10 12 14 16 18 20 22 24 26 28 30 32 34 36 38 40 42 44 46 48
50 52 54 56 58 60 62 64 66 68 70 72 74 76 78)
↦(gera-vector-com-proc (lambda (i) (* i i)) 15)   ; vector com as potências
#(0 1 4 9 16 25 36 49 64 81 100 121 144 169 196)   ; de 2 ...
↦(gera-vector-com-proc add1 40)
#(1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26
27 28 29 30 31 32 33 34 35 36 37 38 39 40)
```

Analisar a definição do procedimento *gera-vector-com-proc*, sobretudo no que respeita à forma característica como se implementa a recursividade com vectores, como já foi referido no exemplo anterior.

```
(define gera-vector-com-proc
  (lambda (proc comprimento)
    (letrec((vec (make-vector comprimento))
            (aux
              (lambda (indice)
                (if (= indice comprimento)
                    vec
                    (begin
                      (vector-set! vec indice (proc indice))
                      (aux (add1 indice)))))))
     ;
      (aux 0))))
```

Exercício 5.11

Definir o procedimento *estica-vector* que toma um vector de comprimento *comp* e devolve um vector de comprimento *novo-comp*, com *novo-comp* > *comp*, em que os *comp* primeiros elementos são os elementos do vector dado e os restantes elementos são iguais a 0.

```
↦(estica-vector '#(a 1 b 2 c 3 d 4) 13)
#(a 1 b 2 c 3 d 4 0 0 0 0 0)
```

Pista: Utilizar o procedimento *gera-vector-com-proc*.

Exemplo 5.4

Vamos supor uma situação em que se pretende processar simultaneamente dois vectores de igual comprimento. O primeiro elemento de cada um dos vectores é processado e registado o resultado, depois é processado o segundo elemento de cada um dos vectores e registado o resultado, e assim sucessivamente até ao último elemento. No final, é devolvido o vector composto pelos resultados referidos.

Na interacção que se segue podemos ver três exemplos do processamento referido.

```
↦(multipa-2-vectores (vector 1 2 3) '#(4 5 6))
'#(4 10 18)
↦(soma-2-vectores (vector 1 2 3) '#(4 5 6))
'#(5 7  9)
↦(cons-2-vectores (vector 1 2 3) '#(4 5 6))
'#((1 . 4) (2 . 5) (3 . 6))
```

Estes três procedimentos apresentam um padrão semelhante, permitindo antever uma solução do seguinte tipo:

```
(define soma-2-vectores
  (lambda (vec1 vec2)
    ((processa-2-vectores +) vec1 vec2)))

(define multiplica-2-vectores
  (lambda (vec1 vec2)
    ((processa-2-vectores *) vec1 vec2)))

(define cons-2-vectores
  (lambda (vec1 vec2)
    ((processa-2-vectores cons) vec1 vec2)))
```

O procedimento *processa-2-vectores* toma um operador como argumento e devolve um procedimento com dois parâmetros, que são dois vectores, sobre os quais aquele operador deve actuar.

```
(define processa-2-vectores
  (lambda (proc)

    (lambda (vec1 vec2)
      (let ((aux
              (lambda (indice)
                (proc (vector-ref vec1 indice)
                      (vector-ref vec2 indice)))))
        (gera-vector-com-proc aux
                              (vector-length vec1))))))
```

Exercício 5.12
Depois de estudar o exemplo anterior, escrever uma solução alternativa para o procedimento *processa-2-vectores*, que não utilize o procedimento *gera-vector-com-proc*.

6- Abstracção com dados mutáveis - Cadeia de caracteres

Os *caracteres* representam mais um tipo de dados que o *Scheme* disponibiliza, para além dos outros já considerados, como sejam, os *números*, *booleanos* e *símbolos*. É um *caracter* que se gera ao actuar uma tecla de um teclado, corresponda essa tecla a uma letra ou a um dígito numérico, a um caracter de pontuação ou de controlo. A *Cadeia de caracteres* é, numa primeira aproximação, um *vector de caracteres*, uma vez que os procedimentos primitivos associados aos vectores e às cadeias apresentam grandes semelhanças. Com o procedimento *display*, as cadeias de caracteres têm sido largamente utilizadas na visualização de mensagens no ecrã.
A cada *caracter* está associado um valor numérico, que nem sempre foi o mesmo em todos os computadores, situação que originava enormes problemas quando se transferiam dados entre computadores que não adoptavam a mesma codificação. Para evitar tal confusão, a partir da década de 60, os fabricantes de computadores começaram a adoptar o *Código ASCII - American Standard Code for Information Interchange* - que estabelece a codificação para todas as letras, dígitos numéricos, caracteres de pontuação e de controlo, num total de 128 caracteres. Uma tabela com o código ASCII é seguidamente apresentado. A 1ª e 2ª colunas dessa tabela, ou sejam as duas colunas mais à esquerda, com os códigos de 00 a 31, estão associadas aos caracteres de controlo, como, por exemplo, *LineFeed* ou *Newline* (código 10), *Carriage Return* (código 13), enquanto que as restantes colunas, com algumas excepções, estão essencialmente associadas aos caracteres de pontuação (3ª coluna), aos dígitos decimais (4ª coluna), às letra maiúsculas (5ª e 6ª colunas), e às letras minúsculas (7ª e 8ª colunas).

Tabela do Código ASCII

00 a 15	16 a 31	32 a 47	48 a 63	64 a 79	80 a 95	96 a 111	112 a 127
NUL	DLE	Space	0	@	P	‘	p

00 a 15	16 a 31	32 a 47	48 a 63	64 a 79	80 a 95	96 a 111	112 a 127
SOH	DC1	!	1	A	Q	a	q
STX	DC2	"	2	B	R	b	r
ETX	DC3	#	3	C	S	c	s
EOT	DC4	$	4	D	T	d	t
ENQ	NAK	%	5	E	U	e	u
ACK	SYN	&	6	F	V	f	v
BEL	ETB	’	7	G	W	g	w
BS	CAN	(	8	H	X	h	x
HT	EM	)	9	I	Y	i	y
LF	SUB	*	:	J	Z	j	z
VT	ESC	+	;	K	[	k	{
FF	FS	,	<	L	\	l	|
CR	GS	-	=	M	]	m	}
SO	RS	.	>	N	^	n	~
SI	US	/	?	O	_	o	DEL

Em *Scheme*, cada caracter é representado pelo seu símbolo antecedido por `#\`, como, por exemplo, *#\a* para o caracter *a*, ou *#\5* para o caracter *5*. O predicado *char?* reconhece os caracteres.

```
↦(char? #\5)
#t
↦(char? 5)
#f
```

O *Scheme* disponibiliza o procedimento *char->integer*, o qual devolve o código associado ao caracter que recebe como argumento.

```
↦(char->integer #\5)
53
↦(char->integer #\a)
97
↦(char->integer #\A)
65
↦(char->integer #\newline)     ; o Scheme aceita representações especiais
10                             ; para alguns caracteres de controlo
↦(char->integer #\space)       ; como, por exemplo, #\space e #\newline
32
```

Por seu lado, *integer->char* realiza a operação inversa de *char->integer*.

```
↦(integer->char 37)
#\%
↦(integer->char 10)
#\newline
```

Para comparação do código dos caracteres, o *Scheme* oferece uma gama completa de predicados: *char=?*, *char>?*, *char<?*, *char>=?*, *char<=?*.

```
↦(char=? #\a #\A)
#f
↦(char>? #\a #\A)
#t
↦(char<? #\9 #\1)
#f
↦(char<? #\9 #\a)
#t
```

Quando não se pretende distinguir entre letras maiúsculas e minúsculas, os predicados deverão ser insensíveis[9] a esta característica, como acontece com: *char-ci=?*, *char-ci>?*, *char-ci<?*, *char-ci>=?*, *char-ci<=?*.

```
↦(char-ci=? #\a #\A)
#t
↦(char-ci>? #\a #\A)
#f
↦(char-ci>=? #\a #\A)
#t
```

No contexto das letras, ainda é de salientar os predicados *char-upper-case?* e *char-down-case?* e os procedimentos *char-upcase* e *char-downcase*, cuja funcionalidade se revela pelos seus nomes e nos exemplos que se seguem.

```
↦(char-upper-case? #\a)
#f
↦(char-upper-case? #\A)
#t
↦(char-upcase #\a)
#\A
↦(char-upcase #\A)
#\A
↦(char-upcase (char-downcase #\A))
#\A
```

Exercício 5.13

O *Scheme* disponibiliza ainda os predicados *char-alphabetic?*, *char-numeric?* e *char-whitespace?*. O primeiro verifica se o argumento é um caracter alfabético ou seja se é uma das letras, o segundo se é um dos dígitos decimais e o terceiro se é um caracter *whitespace*[10]. Escrever em *Scheme* versões pessoais para estes predicados, que deverão responder como se indica.

```
↦(pessoal-char-alpha? #\a)
#t
↦(pessoal-char-alpha? #\1)
#f
↦(pessoal-char-alpha? #\newline)
#f
↦(pessoal-char-num? #\5)
#t
↦(pessoal-char-num? #\newline)
#f
↦(pessoal-char-whitesp? #\a)
#f
↦(pessoal-char-whitesp? #\newline)
#t
```

Como já se referiu, a *cadeia de caracteres* é um *vector de caracteres*, o que se evidencia fazendo o paralelismo entre os procedimentos para manipulação de vectores e para cadeias de caracteres. Encontrar os procedimentos dos vectores equivalentes aos seguintes procedimentos para as cadeias de caracteres e, a partir daqueles, comprovar as respostas dadas pelo Scheme na interacção seguinte.

```
↦(define c1 (string #\a #\6 #\b))
```

[9] *ci*, ou seja, *case-insensitive*

[10] *whitespace* deverá incluir o *space* e o *newline*, mas algumas implementações do *Scheme* poderão considerar ainda outros caracteres

```
c1
↦c1
"a6b"
↦(string? c1)
#t
↦(define c2 (string-copy c1))
c2
↦c2
"a6b"
↦(string-fill! c2 #\z)
valor não definido
↦c2
"zzz"
↦(string-length c1)
3
↦(string-ref c1 1)
#\6
↦(make-string 3 #\a)
"aaa"                                          ; ↦(make-string 3) devolve "   "
↦(list->string '(#\1 #\2 #\a #\7))
"12a7"
↦(string->list "abcd")
(#\a #\b #\c #\d)
↦(string->list (string #\1 #\2 #\3 #\4))
(#\1 #\2 #\3 #\4)
↦(define c51 (string #\0 #\2 #\4 #\6 #\8))
c51
↦51
"02468"
↦(string-set! c51 2 #\a)
valor não definido
↦c51
"02a68"
```

Com estes procedimentos[11], a implementação de soluções recursivas com cadeias de caracteres é semelhante à que foi utilizada com os vectores, e também aqui assume uma forma diferente da utilizada para as listas. Inicialmente determina-se o comprimento da cadeia a processar e define-se um procedimento auxiliar, com um parâmetro que represente o índice ou a posição de um caracter na cadeia. A chamada inicial do procedimento auxiliar apresenta-se com o argumento zero, que vai sendo incrementado em cada nova chamada. A condição de terminação corresponde a verificar-se a igualdade entre este argumento e o comprimento da cadeia.

Exemplo 5.5

Escrever em *Scheme* o procedimento *cadeia-de-maiusculas!*, que recebe uma cadeia de caracteres como argumento e altera a cadeia recebida, transformando as letras minúsculas em letras maiúsculas. Como se pode verificar, o procedimento pedido é um modificador e não um construtor, pois limita-se a alterar um entidade existente.

Solução

```
(define cadeia-de-maiusculas!
  (lambda (cadeia)
    (let ((comprim (string-length cadeia)))
      (letrec ((aux
                 (lambda (indice)
```

[11] Outros procedimentos para processamento de cadeias de caracteres encontram-se no *Anexo A*

```
                    (if (= indice comprim)
                        cadeia
                        (begin
                          (string-set! cadeia
                                       indice
                                       (char-upcase
                                         (string-ref cadeia indice)))
                          (aux (add1 indice)))))))
          ;
          (aux 0)))))
```

Exercício 5.14

Como veremos mais à frente, o *Scheme* disponibiliza o predicado *string=?* que aceita dois argumentos do tipo cadeia de caracteres e devolve *#t* se e só se os argumentos forem iguais, comparando caracter a caracter.

```
↦(string=? "abc" "ab")
#f
↦(string=? "abc" "aBc")
#f
↦(string=? "abc" (string #\a #\b #\c))
#t
```

Escrever em *Scheme* o procedimento *pessoal-cadeia=?*, uma versão pessoal o procedimento *string=?*.

Para além dos procedimentos com paralelismo nos vectores, a abstracção cadeia de caracteres admite ainda outros procedimentos, próprios deste tipo de dados.

```
↦(string-append "123" "abc" "!!!")
"123abc!!!"
↦(substring "012345678" 2 5)
"234"
↦(symbol->string 'bom-dia)
"bom-dia"
↦(string->symbol "bom dia")
bom dia
↦(symbol->string 'bom dia)
symbol->string: wrong number of arguments
↦(string->number "123")
123
↦(string->number "1ab")
#f
↦(number->string 123)
"123"
↦(string=? "abc" "ab3")
#f
↦(string-ci=? "abc" "aBc")
#t
↦(string<? "abc" "123")
#f
↦(string<? "ABC" "abc")
#t
```

Exemplo 5.6

Desenvolver o procedimento *string-cdr* que recebe uma cadeia de caracteres como argumento e devolve uma cadeia equivalente à recebida, mas sem o primeiro caracter.

```
↦(string-cdr "123456")
```

```
"23456"
↦(string-cdr (string-cdr "123456"))
"3456"
```

Solução

```
(define string-cdr
  (lambda (cadeia)
    (substring cadeia
               1
               (string-length cadeia))))
```

Encontrou-se uma solução relativamente simples com a ajuda do procedimento primitivo *substring*. Agora, com *string-cdr*, a recursividade nas cadeias de caracteres poderá também assumir uma forma idêntica à praticada com as listas, para as quais existe o *cdr*.

Exercício 5.15
Tomando por base os procedimentos apresentados nos dois últimos exemplos, desenvolver o procedimento *cadeia-de-maiusculas-com-string-cdr*, que não sendo um modificador como *cadeia-de-maiusculas!*, cria uma nova cadeia de caracteres, a partir da cadeia que recebe como argumento, transformando as letras minúsculas em maiúsculas. Sugere-se uma solução recursiva, em que esta assuma a forma de recursividade característica das listas, recorrendo ao procedimento *string-cdr* [12].

Exercício 5.16
Desenvolver o procedimento *cadeia-maius-minus-e-vice-versa*, que transformas as letras maiúsculas em minúsculas e vice-versa, da cadeia que recebe como argumento. Pretende-se duas versões deste procedimento, uma como modificador da cadeia recebida e outra como construtor de uma nova cadeia.

7- Ficheiros

Até ao momento, os programas desenvolvidos procuravam dados vindos do teclado com o procedimento *read* e visualizavam os resultados no ecrã com o procedimento *display*. Esta forma de interactuar com o utilizador nem sempre é aceitável, sobretudo quando os dados a fornecer são em grande quantidade, o que inviabiliza o fornecimento repetido de dados através do teclado, ou então quando se pretende guardar resultados de uma sessão para outra. Tome-se, como exemplo, um programa de análise estatística sobre o aproveitamento escolar dos alunos de um curso. Se se considerar que são 200 os alunos desse curso, e que durante um ano cada deles está inscrito em 10 disciplinas, a média geral do aproveitamento escolar de todo o curso envolve 2000 dados. Não é razoável aceitar que, sempre que o direcor desse curso quer conhecer a média geral, tenha de fornecer 2000 dados pelo teclado. Mas é óbvio que outras análises se podem fazer sobre os resultados obtidos pelos alunos do curso, para além da média geral, é necessário, por exemplo, conhecer a média final ou a média anual de cada aluno. Estes e outros estudos referem-se aos mesmos dados que deverão estar organizados e disponíveis sempre que sejam necessários. Isto é possível, através dos *ficheiros*, implementados sobre memória não voláctil, memória que mantem os dados registados mesmo quando é dada por encerrada uma sessão do programa e o computador é desligado completamente. Como suporte

[12] No contexto da *cadeia de caracteres*, pode classificar-se esta forma de recursividade como *recursividade de substring* e a outra forma, que também foi utilizada com os vectores, como *recursividade de string*

de memória não voláctil temos, como exemplo, os discos e diskettes que equipam praticamente todos os computadores.
Em *Scheme*, para abrir um ficheiro para escrita é necessário:

⇒ Associar uma *porta de saída*[13] com o ficheiro a abrir
```
(define porta-sai (open-output-file nome-ficheiro))
```
[14]

⇒ Todas as chamadas de *display e newline*[15], que especifiquem a *porta de saída* devolvida na operação anterior, não influenciarão o ecrã, mas sim o conteúdo do ficheiro
```
(display dado-a-escrever-no-ficheiro porta-sai)
```

⇒ Fechar o ficheiro, quando não há mais dados para guardar nele
```
(close-output-port porta-sai)
```

Exemplo 5.7

Um programa designado por *cria-ficheiro-ano-de-nascimento* não tem parâmetros e começa por interrogar o utilizador sobre o nome do ficheiro que pretende criar para registar o nome e o ano de nascimento de vários amigos. O programa mantem um diálogo com o utilizador, para que este lhe transmita os dados a registar.

```
↦(cria-ficheiro-ano-de-nascimento)
nome do ficheiro:
anoNasc.txt
Nome (-1 = fim):
joao
Ano de nascimento (-1 = fim):
1949
Nome (-1 = fim):
manuel
Ano de nascimento (-1 = fim):
1952
Nome (-1 = fim):
antonio
Ano de nascimento (-1 = fim):
1950
Nome (-1 = fim):
maria
Ano de nascimento (-1 = fim):
1954
Nome (-1 = fim):
joaquim
Ano de nascimento (-1 = fim):
1943
Nome (-1 = fim):
-1
Ano de nascimento (-1 = fim):
11
```

Após este diálogo, o conteúdo do ficheiro *anoNasc.txt* apresenta o seguinte conteúdo:

```
(joao . 1949)
(manuel . 1952)
(antonio . 1950)
(maria . 1954)
```

---

[13] *output port*

[14] *open-output-file* devolve uma *porta de saída* que fica associada ao ficheiro cujo nome, uma cadeia de caracteres, é recebido como argumento. Se já existe um ficheiro com este nome, o que acontece depende da implementação do *Scheme*, mas o mais usual é limpar o conteúdo do ficheiro. Se o ficheiro não existe, é criado um com o nome especificado.

[15] Até aqui, sempre utilizámos *display* e *newline* sem especificar a porta de saída, o que significa que se prentendia aceder ao *standard-output* (o ecrã)

```
(joaquim . 1943)
```

Como se pode verificar, os dados respeitantes a cada uma das pessoas são registados como pares, em que o elemento da esquerda é o nome e o da direita é o ano de nascimento.

Solução
O programa *cria-ficheiro-ano-de-nascimento* mostra claramente as tarefas identificadas para abrir um ficheiro.

```
(define cria-ficheiro-ano-de-nascimento
  (lambda ()
    (display "nome do ficheiro: ")
    (let ((nome-fich (read)))
      (let ((porta-out (open-output-file nome-fich)))
        (pedir-dados porta-out)
        (close-output-port porta-out)))))
```

De cada vez que o procedimento *pedir-dados* chama *ler-nome-ano*, estabelece-se um diálogo com o utilizador que conduz ao fornecimento dos dados relativos a uma pessoa ou então à indicação de que não há mais dados a fornecer.

```
(define pedir-dados
  (lambda (porta-fich)
    (let ((nome-ano (ler-nome-ano)))
      (if (equal? nome-ano 'fim)
          'fim
          (begin
            (display nome-ano porta-fich)
            (newline porta-fich)
            (pedir-dados porta-fich))))))

(define ler-nome-ano
  (lambda ()
    (newline)
    (display "Nome (-1 = fim): ")
    (let ((nome (read)))
      (newline)
      (display "Ano de nascimento (-1 = fim): ")
      (let ((ano (read)))
        (if (or (equal? nome -1)
                (equal? ano -1))
            'fim
            (cons nome ano))))))
```

Em *Scheme*, para abrir um ficheiro para leitura é necessário:

⇒ Associar uma *porta de entrada*[16] com o ficheiro a abrir

```
(define porta-entr (open-input-file nome-ficheiro))
```
[17]

⇒ Todas as chamadas de *read*[18], que especifiquem a *porta de entrada* devolvida na operação anterior, não ficam à espera do teclado, pois acedem ao conteúdo do ficheiro que foi aberto

```
(read porta-entr)
```

[16] *input port*

[17] *open-input-file* devolve uma *porta de entrada* que fica associada ao ficheiro cujo nome, uma cadeia de caracteres, é recebido como argumento. Se o ficheiro não existe, é gerada uma mensagem de erro

[18] Até aqui, sempre utilizámos *read* sem especificar a porta de entrada, o que significa que se prentendia aceder ao *standard-input* (o teclado)

⇒ Sempre que um ficheiro é acedido em leitura, o elemento lido deverá ser testado a fim de se verificar se já se atingiu o fim do ficheiro[19].

```
(eof-object? ultimo-elemento-resultante-da-operacao-de-read)
```

⇒ Fechar o ficheiro, logo que a operação anterior devolva *#t*

```
(close-input-port porta-entr)
```

Exemplo 5.8

Um programa designado por *le-ficheiro-e-faz-relatorio* não tem parâmetros e começa por interrogar o utilizador sobre o nome do ficheiro onde foram registados o nome e o ano de nascimento de vários amigos, com a ajuda do programa *cria-ficheiro-ano-de-nascimento*, do exemplo anterior. Em seguida, o programa abre o ficheiro indicado, pede ao utilizador a indicação de um ano de referência, e apresenta um relatório com o nome de todas as pessoas registadas no ficheiro e a respectiva idade, no ano de referência.

```
↦(le-ficheiro-e-faz-relatorio)
nome do ficheiro:
anoNasc.txt
ano de referencia:
1999

Relatorio de idades relativo a 1999
joao      50
manuel      47
antonio      49
maria      45
joaquim      56

Fim do relatorio


↦(le-ficheiro-e-faz-relatorio)
nome do ficheiro:
anoNasc.txt
ano de referencia:
1950

Relatorio de idades relativo a 1950
joao      1
manuel      -2
antonio      0
maria      -4
joaquim      7

Fim do relatorio
```

Solução

A implementação do programa *le-ficheiro-e-faz-relatorio* mostra as tarefas identificadas no enunciado.

⇒ pedido do nome do ficheiro a processar;

⇒ pedido do ano em relação ao qual são calculadas as idades;

⇒ Visualização do relatório respectivo

- Cabeçalho do relatório
- Corpo do relatório (através do procedimento *relatorio)*

⇒ Fecho do ficheiro

---

[19] *eof - end of file*

```
(define le-ficheiro-e-faz-relatorio
  (lambda ()
    (display "nome do ficheiro: ")
    (let ((nome-fich (read)))
      (let ((porta-in (open-input-file nome-fich)))
        (newline)
        (display "ano de referencia: ")
        (let ((ano-ref (read)))
          (newline)
          (newline)
          (display "Relatorio de idades relativo a ")
          (display ano-ref)
          (newline)
          (relatorio porta-in ano-ref)
          (close-input-port porta-in))))))
```

O procedimento *relatorio* utiliza o procedimento local, *percorre-ficheiro*, para ler os dados associados a cada pessoa e para verificar quando se atinge o fim do ficheiro. Por seu lado, *percorre-ficheiro*, chama o procedimento auxiliar, *visu-registo*, que toma como argumentos os dados associados a uma pessoa e o ano de referência e visualiza a linha respectiva.

```
(define relatorio
  (lambda (porta-fich ano-refer)

    (letrec ((percorre-ficheiro
               (lambda ()
                 (let ((registo-pessoa (read porta-fich)))
                   (if (eof-object? registo-pessoa)
                       (begin
                         (newline)
                         (display "Fim do relatorio"))
                       (begin
                         (visu-registo registo-pessoa ano-refer)
                         (percorre-ficheiro)))))))
      ;
      (percorre-ficheiro))))

(define visu-registo
  (lambda (reg-pessoa ano-rf)
    (display (car reg-pessoa))
    (display "      ")
    (display (- ano-rf
                (cdr reg-pessoa)))
    (newline)))
```

Exercício 5.17

Relativamente ao exemplo anterior, desenvolver uma versão melhorada do programa *le-ficheiro-e-faz-relatorio*, em que o relatório apresenta os nomes por ordem alfabética e as idades alinhadas, a partir da coluna 20.

```
↦(le-ficheiro-e-faz-relatorio-v2)
nome do ficheiro:
anoNasc.txt
ano de referencia:
1999

Relatorio de idades relativo a 1999
antonio            49
joao               50
joaquim            56
manuel             47
maria              45
```

```
Fim do relatorio
```

Exercícios e exemplos de final de capítulo

Segue-se um conjunto de exercícios e exemplos para consolidação da matéria tratada no capítulo.

Exercício 5.17

Utilizando a abstracção *fila de espera*, escrever 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:

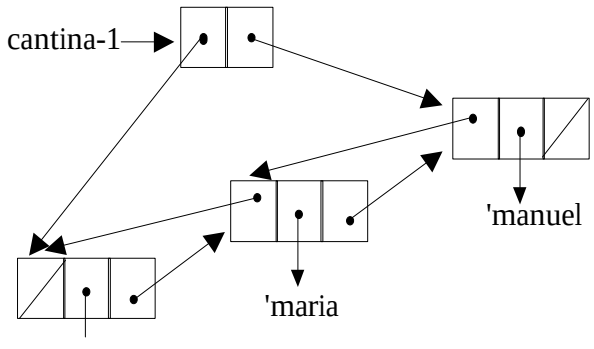
E aluno de Electrotecnia
I aluno de Informática
C aluno de Civil
Q aluno de Química
M aluno de Minas

```
↦(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...
```

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?*.

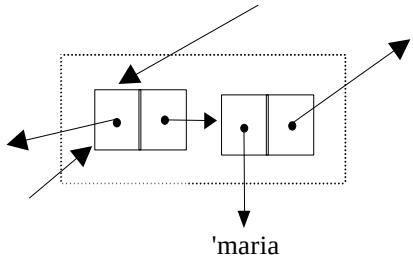
Exercício 5.19

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)*. Este modelo não responde bem se fossem necessárias



as operações de entrada no início da fila e de saída no fim dessa fila.
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.

Considerar que cada elemento da lista duplamente ligada é representado com dois pares, como se mostra na figura.



1- Desenvolver 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- Desenvolver, para as filas duplamente ligadas, os seguintes procedimentos:

Construtor

(cria-fila-dupla) - devolve uma fila dupla 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 *fila* alterada.

(entra-tras-da-fila-dupla! fila item) - insere *item* no fim de *fila* e devolve *fila* alterada.

(sai-frente-da-fila-dupla! fila) - retira o primeiro elemento de *fila* e devolve fila alterada.

(sai-tras-da-fila-dupla! fila) - retira o último elemento de *fila* e devolve fila alterada.

Exercício 5.20

Escrever o procedimento *procura-no-vector* que tem como parâmetros o vector *vec* e o objecto *obj* e devolve o índice da primeira ocorrência de *obj* em *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
```

Exercício 5.21

Os preços unitários dos produtos adquiridos por um cliente foram registados no vector *precos*, enquanto que as quantidades adquiridas foram registadas no vector *quantidades*. Escrever 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))
precos
↦(define quantidades (vector 2 5 3))
quantidades
```

```
↦(gasto-total precos quantidades)
111.75
```

Exercício 5.22

Escrever 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
```

Exercício 5.23

Definir 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
```

Exercício 5.24

Os procedimentos *soma-elementos-de-vector* e *multiplica-elementos-de-vector*, dos exercícios anteriores, pela estrutura semelhante que apresentam, sugerem a definição de uma abstracção procedimental, 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)))
```

Escrever 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. Procurar outras aplicações para *acumulador-de-vector*. Indicar o que resulta de `(acumulador-de-vector cons '())`.

Exercício 5.25

Escrever o procedimento *vector-map* que 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))
```

Exemplo 5.9

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

Construtor

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

Selector

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

Modificador

(entra! equipa nome-jogador) - Em *equipa* entra o jogador designado por *nome-jogador*. Se já fizer parte da equipa, esta não sofre qualquer alteração.

(sai! equipa nome-jogador) - De *equipa* sai o jogador designado por *nome-jogador*. Se não fizer parte da equipa, esta não sofre qualquer alteração.

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

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

```
(define cria-equipa
  (lambda ()
    (list 'equipa)))
(define entra!
  (lambda (equipa jogador)
    (cond
      ((member jogador (cdr equipa))
       'ok)                          ; já faz parte da equipa. Nada a fazer.
      (else
        (set-cdr! equipa                       ; ainda não faz parte da equipa...
                  (cons jogador (cdr equipa)))))   ; então entra novo jogador
    ;
    (cdr equipa)))                  ; em qualquer dos casos, visualiza equipa
(define sai!
  (lambda (equipa jogador)
    (letrec ((aux
               (lambda (eq)
                 (cond
                   ((null? (cdr eq))  ; não faz parte da equipa. Nada a fazer.
                    (cdr equipa))
                   ((equal? (cadr eq) jogador)
                    (set-cdr! eq                ; faz parte da equipa...
                              (cddr eq))        ; então o jogador é retirado
                    (cdr equipa))
                   (else                    ; não foi encontrada neste ciclo...
```

```
                    (aux (cdr eq)))))))   ; nova tentativa
      (aux equipa))))

(define visu
  (lambda (equipa)
    (cdr equipa)))
```

Exercício 5.26

Estudar o exemplo anterior, e desenvolver uma nova versão em que a equipa é modelada por uma lista de 3 elementos modela, inicializada da seguinte maneira:

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

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*. Se *lugar* já estiver ocupado, será desocupado, para entrar o novo jogador.

(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*. Se *lugar* já estiver ocupado, será desocupado, para entrar o novo jogador.

(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.

(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. Se não for encontrado, a equipa não sofre qualquer alteração.

```
↦(define fcp (cria-equi-fute))
fcp
↦fcp
((? ? ? ? ? ? ? ? ? ? ?) (? ? ? ? ? ?) ?)
↦(entra-equi! fcp 1 'correia)
((correia ? ? ? ? ? ? ? ? ? ?) (? ? ? ? ? ?) ?)
↦(entra-equi! fcp 11 'artur)
((correia ? ? ? ? ? ? ? ? ? artur) (? ? ? ? ? ?) ?)
↦(entra-equi! fcp 10 'jardel)
((correia ? ? ? ? ? ? ? ? jardel artur) (? ? ? ? ? ?) ?)
↦(entra-sup! fcp 2 'folha)
((correia ? ? ? ? ? ? ? ? jardel artur) (? folha ? ? ? ?) ?)
↦(entra-treina! fcp 'oliveira)
((correia ? ? ? ? ? ? ? ? jardel artur) (? folha ? ? ? ?) oliveira)
↦(sai-equi! fcp 'folha)
((correia ? ? ? ? ? ? ? ? jardel artur) (? ? ? ? ? ?) oliveira)
↦(entra-equi! fcp 15 's-conceicao)
lugar estranho!...
((correia ? ? ? ? ? ? ? ? jardel artur) (? ? ? ? ? ?) oliveira)
↦(entra-equi! fcp 5 's-conceicao)
((correia ? ? ? s-conceicao ? ? ? ? jardel artur) (? ? ? ? ? ?)
oliveira)
↦(sai-equi! fcp 'artur)
((correia ? ? ? s-conceicao ? ? ? ? jardel ?) (? ? ? ? ? ?) oliveira)
↦(sai-equi! fcp 'oliveira)
((correia ? ? ? s-conceicao ? ? ? ? jardel ?) (? ? ? ? ? ?) ?)
```

```
↦(sai-equi! fcp 'folha)
folha nao esta' na equipa
((correia ? ? ? s-conceicao ? ? ? ? jardel ?) (? ? ? ? ? ?) ?)
```

Exercício 5.27
No contexto do exercício anterior, introduzir as alterações necessárias para evitar que o mesmo jogador ocupe, ao mesmo tempo, mais do que um lugar na equipa.

Exercício 5.28
Ainda no contexto dos exercícios anteriores, considerar agora a modelação de uma equipa como uma lista de 3 elementos, sendo vectores os 2 primeiros. A equipa é inicializada da seguinte maneira: `(#(? ? ? ? ? ? ? ? ? ?) #(? ? ? ? ? ?) ?)`

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

Exercício 5.29
Em relação ao exercício anterior, introduzir as alterações necessárias para evitar que o mesmo jogador ocupe, ao mesmo tempo, mais do que um lugar na equipa.
.

Exercício 5.30
Analisar e comparar as soluções a que chegou para os exercícios 5.26 e 5.28. Indicar as principais diferenças entre as duas soluções.

Exercício 5.31
Escrever em *Scheme* o programa *ocorrencias-de-digitos* que lê um número inteiro e determina 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
```

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

Exercício 5.32
Num capítulo anterior 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)   ; com este argumento serão colocadas 10 perguntas

3 x 8 = 24               ; na primeira pergunta, o programa visualizou 3 x 8 = e o
Resposta certa           ; utilizador escreveu 24. Portanto, resposta certa

2 x 7 = 15
Resposta errada

...
            ; no final das perguntas,
Muito bem                ; esta será a mensagem se o número de erros for inferior a 2
Deve estudar melhor a tabuada      ; ou esta, se o número de erros for 2 ou mais
```

Pretende-se agora o programa *teste-da-tabuada-melhorado* que, relativamente à versão anterior, 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*.
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 5.33
Escrever 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
```

Pista: Procurar uma solução recursiva.

Exercício 5.34
Escrever 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
```

Pista: Procurar uma solução recursiva.

Projecto 5.1- 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 *palavras*
  '((p a l a v r a)
    (n o t a)
    (l i v r o)
    (j a n t a r)
    (e r r o)
    (a d i v i n h a r)
    (l e t r a)))
```

Analisar com muita atenção a interacção que se segue.

```
↦(adivinha-palavras *palavras*)


palavra a adivinhar: (* * * *)   ; neste caso, a palavra a adivinhar tem 4 letras
letras erradas: ()               ; para já, não há letras erradas...
Proxima letra:
n

palavra a adivinhar: (n * * *)   ; já se acertou na 1ª letra
letras erradas: ()               ; não há ainda 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:
o

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

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- Fazer uma abordagem *de-cima-para-baixo* ao programa *adivinhar-palavras*, começando por identificar as suas tarefas e sub-tarefas principais e, se for considerada necessária, definir uma abstracção de dados adequada.
2- Escrever em *Scheme* o programa *adivinhar-palavras*.
3- Escrever 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. Prever um programa auxiliar para preparar os ficheiros de palavras.

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

```
(n o t a)
(j a n t a r)
(c o m p u t a d o r)
(n o i t e)
(s e g u n d a)
(n o m e)
(a d i v i n h a)

↦(adivinha-palavras-v2 "cap5dici1.txt")
```

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

Projecto 5.2- 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 o estado 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---------
            ...
          --------------------
          --------------------       Caractarísticas iniciais do helicóptero:
Linha 1   --------------------        posicao: 16 11; situacao: no-terreno;
                                      orientacao: N
```

O helicóptero é visualizado pela letra H, as células ainda não visistadas por ele 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---------
            ...
          --------------------
          --------------------       Caractarísticas actuais do helicóptero:
Linha 1   --------------------        posicao: 18 11; situacao: no-terreno;
                                      orientacao: 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[20]
Comando: 3                    ; roda para a direita 90º
Comando: 5                    ; avança
Posicoes em frente: 3         ; 3 posições
Comando: 4                    ; roda para a esqueda 90º
Comando: 5                    ; ...
Posicoes em frente: 2
Comando: 4
Comando: 5
Posicoes em frente: 6
Comando: 6

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


Comando: 9
E' para guardar em ficheiro? (s/n): s[21]
Nome do ficheiro: heli.txt
Terminou a viagem...
```

1- Definir e implementar, se necessário, uma abstracção de dados compatível com o problema exposto.
2- Definir a estrutura dos ficheiros onde se guardará o resultado de uma sessão para ser retomado mais tarde.
3- Fazer uma abordagem *de-cima-para-baixo* ao programa *helicoptero-digital*, para identificar as suas tarefas e sub-tarefas principais.
4- Escrever 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.

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 caractere 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

[20] Com a resposta *n* seria visualizada a mensagem *Nome do ficheiro:* à qual seria respondido com o nome de um ficheiro. Então o ficheiro seria aberto em leitura e a simulação retomada a partir do seu conteúdo
[21] Com a resposta *n* o programa terminaria de imediato com a mensagem *Terminou a viagem...*

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 de rodar 90º para a direita `(modulo (add1 direccao))` e rodar 90º para a esquerda `(modulo (sub1 direccao))`.