Capítulo 3- Abstracção de Dados

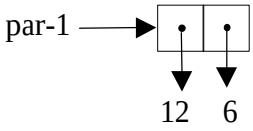
Os tipos de dados *Scheme* considerados até aqui foram os *números*, *booleanos* e *procedimentos* (primitivos e compostos) e também os *símbolos*, estes utilizados para dar nomes a variáveis e a procedimentos. Os *símbolos* podem também ser dados directamente manipuláveis em imensas situações reais que não exigem valores numéricos, mas apenas valores simbólicos. Com os dados referidos é possível e extremamente útil criar *dados compostos*, ou seja, dados constituídos por várias partes, em que estas poderão ser números, booleanos, símbolos e também outros dados compostos, entretanto definidos[1]. Assim, perante as características do problema a resolver, será possível compor os dados mais adequados a cada caso. Por exemplo, num problema de características gráficas a 2 dimensões *(2D)*, a criação do tipo de dado *ponto-2D* é perfeitamente justificável, sendo constituído por dois valores numéricos que representam as suas coordenadas *x* e *y* no plano, que deverão ser agrupados. Ainda no contexto do problema gráfico, poderá ser necessário manipular áreas triangulares, definidas por três vértices. O dado do tipo *triângulo* poderá ser criado, agrupando três dados do tipo *ponto-2D*. Em problemas meramente numéricos, compor dados pode também ser necessário, como acontece com as fracções, em que os dados do tipo *fracção* são modelados agrupando dois inteiros que representam, respectivamente, o numerador e o denominador. A representação dos dados que melhor se ajuste a um certo problema e a identificação das operações básicas respectivas constituem a *abstracção de dados* em que se apoiará a resolução desse problema. As decisões do programador sobre este assunto deverão ser tomadas com o máximo cuidado, não esquecendo que essas decisões *(boas ou más)* terão repercussões notórias na qualidade *(boa ou má)* dos programas desenvolvidos. Os *conjuntos*, como um exemplo de abstracção de dados, serão tratados com alguma profundidade. Com alguns *procedimentos gráficos* simples é possível abrir uma janela gráfica e associar-lhe uma caneta que desenha, pinta ou escreve. A implementação destes procedimentos, para o *EdScheme* e *DrScheme*, encontra-se no *Anexo B*. Para algumas situações, a janela, a caneta e os procedimentos gráficos constituem a abstracção de dados.
A complexidade de certos problemas justifica uma abordagem do tipo *de-cima-para-baixo*[2], segundo a qual se procuram, em primeiro lugar, as grandes tarefas associadas ao programa. Se alguma destas tarefas ainda se mostrar complexa, então também se deverá procurar identificar as suas grandes tarefas. A ideia chave traduz-se em dividir as tarefas em tarefas cada vez mais pequenas e simples, de tal forma que os procedimentos que as implementam também se tornem simples. Como se tentará mostrar, a abordagem *de-cima-para-baixo* e a *abstracção de dados* (esta surgindo como uma abordagem *de-baixo-para-cima*) não são conceitos alternativos, mas

[1] Como se verá no final do próximo capítulo, também os procedimentos poderão fazer parte de dados compostos
[2] Abordagem *top-down*

complementares. Acontece que, em determinadas circunstâncias, ou uma ou outra poderá passar mais despercebida.

1- Pares

Em Scheme, o dado *par*, constituído por duas partes, é criado pelo procedimento primitivo *cons* e está na base da criação dos dados compostos.




```
↦(define par-1 (cons 12 6))
par-1
↦ par-1
(12 . 6)
```

Do dado *par-1* são indicadas duas representações: uma textual, *(12 . 6)*, e outra gráfica, designada por *caixa-e-apontador*. Em ambas é notório o agrupamento de dois valores numéricos, constituindo um dado ou uma entidade única.

O procedimento *cons* é um *construtor*, pois aceita as componentes que vão formar um par e devolve esse par. Por seu turno, os procedimentos primitivos *car* e *cdr* são os *selectores*, pois aceitam um par e seleccionam partes desse par.

```
↦(car par-1)
12
↦(cdr par-1)
6
```

As regras de cálculo dos três procedimentos primitivos apresentados são as seguintes:

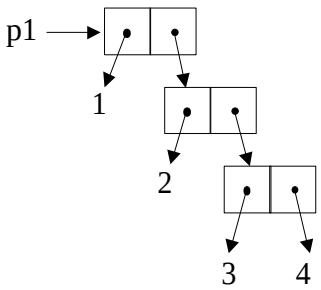
⇒ (cons obj1 obj2)
Aceita 2 dados como argumentos e devolve o par constituído por esses 2 dados.

⇒ (car par)
Aceita um par como argumento e devolve a parte esquerda do par.

⇒ (cdr par)
Aceita um par como argumento e devolve a parte direita do par.

Exemplo 3.1

Se *p1* for definido como sendo *(cons 1 (cons 2 (cons 3 4)))*, a sua representação *caixa-e-apontador* é equivalente à representada na figura. A representação textual deveria ser (1 . (2 . (3 . 4))). Todavia, o *Scheme* elimina o ponto e os parêntesis respectivos, quando o elemento do lado direito de um par é também um par. Por este motivo, a representação textual será, sucessivamente, (1 2 . (3 . 4)), e (1 2 3 . 4). O dado *p1* é constituído por 3 elementos, sendo o primeiro o valor numérico 1, o segundo o valor numérico 2 e o terceiro o (par 3 . 4).



Analisar o valor das expressões que se seguem.

```
↦p1
(1 2 3.4)
↦(car p1)
1
↦(cdr p1)
(2 3.4)
↦(car (cdr p1))
2
↦(cdr (cdr p1))
(3 . 4)
```

```
↦(car (cdr (cdr p1)))
3
↦(cdr (cdr (cdr p1)))
4
```

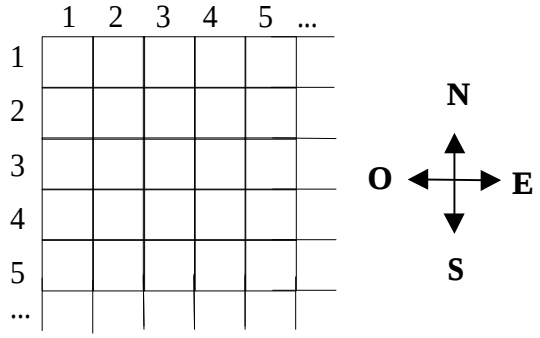
Exercíco 3.1

Se *p2* for definido por *(cons 35 (cons 5 7))*, desenhar a sua representação *caixa-e-apontador*, identificar os seus elementos, e determinar o resultado das expressões que se seguem.

```
↦p2
??
↦(car p2)
??
↦(car (cdr p2))
??
↦(cdr (cdr p2))
??
```

Com os procedimentos primitivos *cons*, *car* e *cdr* é possível criar dados compostos, qualquer que seja a sua complexidade e aceder a qualquer um dos seus elementos.

Vamos considerar, como exemplo, umas tartarugas coloridas que se movimentam num tabuleiro, nas quatro direcções: Norte, Sul, Este e Oeste. Notar que o tabuleiro, sem limites à direita e em baixo, tem à esquerda a coluna 1, e em cima a linha 1.



Pretende-se simular a movimentação de tartarugas no tabuleiro, nas quatro direcções, Norte, Sul, Este e Oeste. Sabe-se que na movimentação para Oeste, a tartaruga não ultrapassa a coluna 1 e, analogamente na movimentação para Norte, não ultrapassa a linha 1. Para além da referida movimentação, pretende-se simular outras operações simples, tais como, a determinação de distâncias entre duas tartarugas. Supõe-se já existentes os seguintes procedimentos:

- *(faz-tartaruga col lin cor)* tem três parâmetros, inteiros positivos, *col*, *lin* e *cor*, e devolve uma tartaruga situada na intersecção da coluna *col* com a linha *lin*, colorida com *cor*. Trata-se de um *constructor*;
- *(linha-tar tar)* tem um parâmetro do tipo *tartaruga* e devolve um inteiro positivo que identifica a linha onde se encontra *tar*. Trata-se de um *selector*;
- *(coluna-tar tar)* tem um parâmetro do tipo *tartaruga* e devolve um inteiro positivo que identifica a coluna onde se encontra *tar*. Trata-se de um *selector*.
- *(cor-tar tartaruga)* tem um parâmetro do tipo *tartaruga* e devolve um símbolo que identifica a cor de *tar*. Trata-se de um *selector*.

Acabámos de definir um certo tipo de dado que designámos por *tartaruga*, e um conjunto de operações associadas. Com base nessas operações vamos avançar para a resolução do problema concreto que tem a ver com a simulação dos movimentos de tartarugas num tabuleiro. E tudo isto sem decidir como será composto ou modelado o objecto *tartaruga*, decisão que poderá ser deixada para mais tarde. Estamos perante um exemplo da aplicação do conceito da *abstracção de dados*, pois não se trabalha directamente com números, mas com entidades que reflectem a natureza do problema a resolver, como sejam, *tartaruga*, *cor*, *linha*, e *coluna*.

Em primeiro lugar é definido o procedimento *visu-tar* que recebe como parâmetro um dado do tipo *tartaruga* e visualiza a respectiva posição e cor.

```
↦(visu-tar (faz-tartaruga 2 5 10))
tartaruga em col: 2, lin: 5, cor: 10
```

```
(define visu-tar
  (lambda (tart)
    (display "tartaruga em col: ")
    (display (coluna-tar tart))
    (display ", lin: ")
    (display (linha-tar tart))
    (display ", cor: ")
    (display (cor-tar tart))
    (newline)))
```

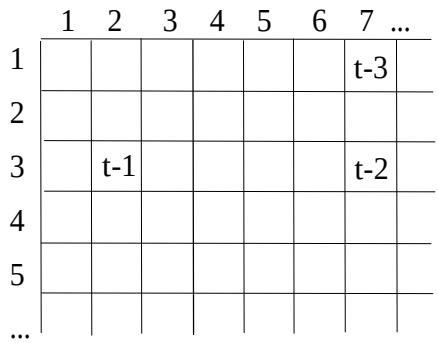
Notar no procedimento *visu-tar*, a utilização dos selectores associados às entidades do tipo *tartaruga*.

Em seguida, são apresentados os procedimentos que movimentam as tartarugas de *n* posições nas quatros direcções, com as limitações impostas pela coluna 1 e linha 1, conforme foi referido.

```
↦(define tar-10 (faz-tartaruga 5 7 10))
tar-10
↦(visu-tar tar-10)
tartaruga em col: 5, lin: 7, cor: 10
↦(define tar-20 (vai-para-norte tar-10 3))
tar-20
↦(visu-tar tar-20)
tartaruga em col: 5, lin: 4, cor: 10
```

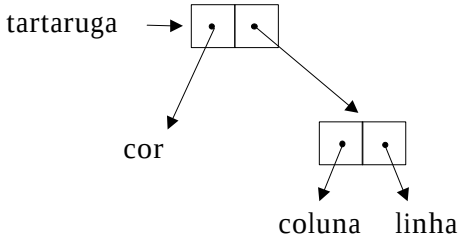
```
(define vai-para-norte                    ; deslocamento para Norte de n
  (lambda (tart n)                        ; posições
    (let ((destino (- (linha-tar tart)
                      n)))                ; de facto, o que acontece é
      (faz-tartaruga (coluna-tar tart)    ; definir uma nova tartaruga
                     (if (< destino 1)    ; com a mesma cor e na mesma
                         1                ; coluna e numa linha mais
                         destino)         ; acima, não ultrapassando
                     (cor-tar tart))))))  ; a linha 1
(define vai-para-sul                      ; deslocamento para Sul de n
  (lambda (tart n)                        ; posições
    (faz-tartaruga (coluna-tar tart)        ; neste caso, não há
                   (+ (linha-tar tart) n)   ; limitações ao deslocamento
                   (cor-tar tart))))
(define vai-para-oeste                    ; ver comentários do procedimento
  (lambda (tart n)                        ; vai-para-norte
    (let ((destino (- (coluna-tar tart)
                      n)))
      (faz-tartaruga (if (< destino 1)
                         1
                         destino)
                     (linha-tar tart)
                     (cor-tar tart))))))
(define vai-para-este                      ; ver comentários do procedimento
  (lambda (tart n )                        ; vai-para-sul
    (faz-tartaruga (+ (coluna-tar tart) n)
                   (linha-tar tart )
                   (cor-tar tart))))
```

São agora apresentadas as definições dos procedimentos que calculam as distâncias entre duas células, distâncias que são referidas em número de posições entre essas tartarugas. Na figura, a *distância-horizontal* entre *t-1* e *t-2* é de 5, a *distância-vertical* é de 2 e a *distância* é de 7, correspondente à soma daquelas duas.




```
↦(define t-1 (faz-tartaruga 2 3 10))
t-1
↦(define t-2 (vai-para-este t-1 5))
t-2
↦(visu-tar t-2)
tartaruga em col: 7, lin: 3, cor: 10
↦(define t-3 (vai-para-norte t-2 6))
t-3
↦(visu-tar t-3)
tartaruga em col: 7, lin: 1, cor: 10
↦(distancia t-1 t-3)
7
```

```
(define distancia-horizontal
  (lambda (tart1 tart2)
    (abs (- (coluna-tar tart1)
            (coluna-tar tart2)))))

(define distancia-vertical
  (lambda (tart1 tart2)
    (abs (- (linha-tar tart1)
            (linha-tar tart2)))))

(define distancia
  (lambda (tart1 tart2)
    (+ (distancia-horizontal tart1 tart2)
       (distancia-vertical tart1 tart2))))
```



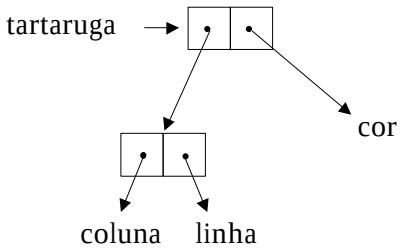
Chegou, finalmente, o momento de se decidir como modelar os dados do tipo *tartaruga*. Entre várias hipóteses, optou-se por utilizar um par, em que a parte da esquerda é a cor da tartaruga e a parte da direita é, ela própria, também um par, correspondendo à posição da tartaruga no tabuleiro (coluna e linha).

Com esta decisão tomada, é possível avançar para a definição dos procedimentos associados à abstracção *tartaruga*: O construtor *faz-tartaruga* e os selectores *linha-tar*, *coluna-tar* e *cor-tar*, com os quais foram definidos os procedimentos que simulam os movimentos de tartarugas num tabuleiro.

```
(define faz-tartaruga           ; três parâmetros, inteiros positivos.
  (lambda (col lin cor)         ; Devolve uma tartaruga situada em col
    (cons cor (cons col lin)))) ; e lin, com a cor dada. É um construtor

(define linha-tar               ; parâmetro do tipo tartaruga. Devolve um inteiro
  (lambda (tart)                ; positivo que identifica a linha onde se
    (cdr (cdr tart))))          ; encontra a tartaruga. É um selector
```

```
(define coluna-tar          ; parâmetro do tipo tartaruga. Devolve um inteiro
  (lambda (tart)            ; positivo que identifica a coluna onde se
    (car (cdr tart))))      ; encontra a tartaruga. É um selector

(define cor-tar             ; Um parâmetro do tipo tartaruga. Devolve um
  (lambda (tart)            ; inteiro positivo que identifica a cor da
    (car tart)))            ; tartaruga. É um selector
```

Exercício 3.2

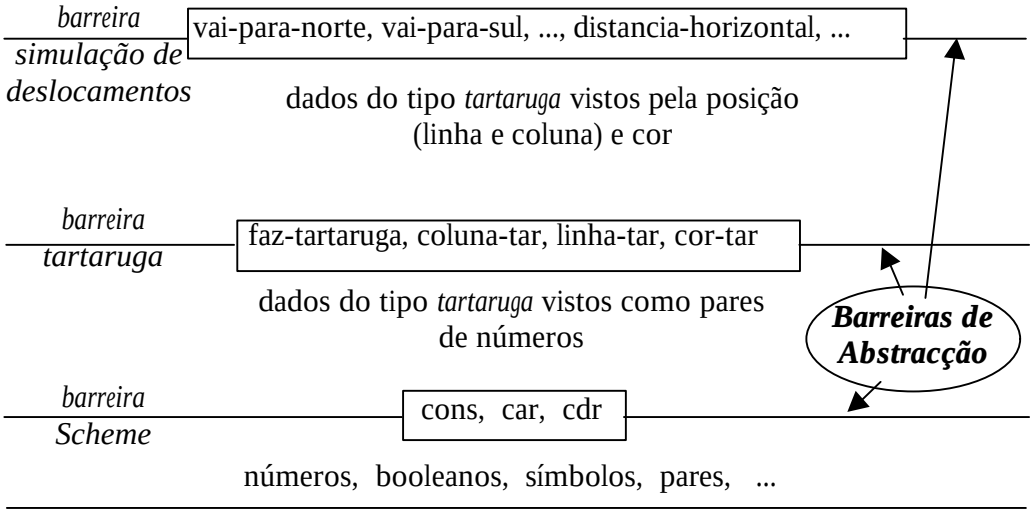


Pretende-se alterar a modelação dos dados do tipo *tartaruga*, como se indica na figura ao lado. Neste contexto, definir as novas versões dos procedimentos básicos para manipulação de objectos deste tipo: *faz-tartaruga*, *linha-tar*, *coluna-tar*, *cor-tart*. Verificar que, após a alteração introduzida nestes procedimentos básicos, os procedimentos que simulam os movimentos de tartarugas num tabuleiro, anteriormente desenvolvidos, mantêm-se válidos.

Exercíco 3.3

No contexto da abstracção *tartaruga*, definir os predicados *(cima? tar1 tar2)*, *(baixo? tar1 tar2)*, *(direita? tar1 tar2)*, e *(esquerda? tar1 tar2)*, todos eles com dois parâmetros do tipo *tartaruga* e que indicam, respectivamente, se *tar1* está acima, abaixo, à direira ou à esquerda de *tar2*.

Uma vantagem da *abstracção de dados* é visível, mesmo num exemplo tão simples como o que acaba de se ser considerado. Com o construtor e os selectores de objectos *tartaruga*, definiram-se alguns procedimentos que simulavam o deslocamento das tartarugas num tabuleiro. Viu-se que a alteração da modelação dos objectos do tipo *tartaruga* apenas teve repercussões sobre os procedimentos construtor e selectores. Todos os outros se mantiveram válidos, graças às *barreiras* que se criaram entre os vários níveis de abstracção.
A barreira colocada ao nível mais baixo corresponde ao que nos é disponibilizado pelo *Scheme*, e sobre ela definiu-se a barreira do tipo *tartaruga*. Sobre esta construiu-se a barreira onde se simulam os deslocamentos das tartarugas, a qual foi completada com os predicados do *exercício 3.3*. Todavia, sobre ela, seria ainda possível definir outras barreiras.



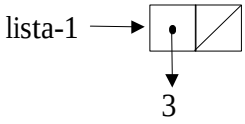
Com as *barreiras de abstracção* ficam muito facilitadas, não só a manutenção e a alteração de programas, mas até o seu desenvolvimento, pois permite deixar para o último instante, decisões importantes, como seja a representação dos dados. Numa primeira fase, o programador poderá decidir-se por uma representação simples de implementar, desenvolver sobre ela uma aplicação e deixar, para uma segunda fase, questões de optimização, tentando encontrar uma representação mais adequada para os dados. As barreiras de abstracção, bem aplicadas, garantirão que, mesmo com outra representação de dados, a aplicação continua válida, sem exigir alterações.

2- Listas

Uma forma de representar sequências de objectos é designada por *Lista*. Sabendo que o Scheme derivada da linguagem *Lisp* e que este nome vem de *List Programming*, poderemos imediatamente antever a relevância das listas em tudo o que se seguirá. De facto, as listas constituem um meio poderoso e flexível para compor dados de qualquer complexidade.

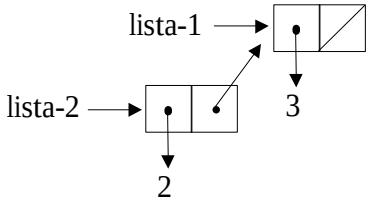
Uma lista de *m+1* elementos é construída com o procedimento primitivo *cons*, juntando um elemento a uma lista de *m* elementos[3].

```
↦(define lista-1 (cons 3 '() ))[4]
lista-1[5]
↦lista-1
(3)
```



Neste caso, juntou-se um elemento, o valor numérico 3, a uma lista vazia.

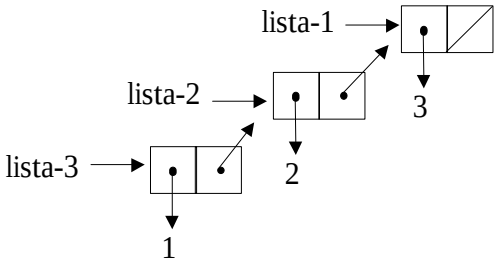
```
↦(define lista-2 (cons 2 lista-1))
lista-2
↦lista-2
(2 3)
```



À lista designada por *lista-1*, juntou-se um elemento, o valor numérico 2, dando origem a *lista-2*.

```
↦(define lista-3 (cons 1 lista-2))
lista-3
↦lista-3
(1 2 3)
↦lista-1        ; a lista-1 mantém-se
(3)             ; intacta
```

As listas que serviram de base à criação de outras listas mantiveram-se intactas, conclusão a que também se chega observando as representações *caixa-e-apontador*.



[3] É por esta razão que lista é considerada como um par

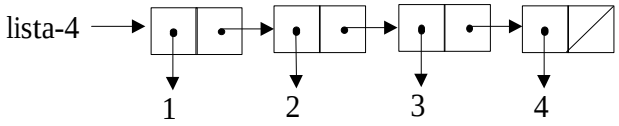
[4] Para expressar lista vazia utiliza-se *'()*, o que será justificado mais à frente

```
↦(define lista-vazia '())
lista-vazia
↦lista-vazia
()
```

[5] A diagonal na parte direita do par significa *fim-de-sequência* de elementos

O procedimento primitivo *list* é uma forma simples para criar listas, a partir dos seus elementos. A lista é criada de uma só vez, o que não acontece com *cons*. No caso de *cons*, acrescenta-se apenas um elemento de cada vez, e a nova lista vai sendo progressivamente constituída, como tivemos oportunidade para verificar.

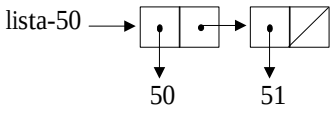
```
↦(define lista-4 (list 1 2 3 4))
lista-4
↦lista-4
(1 2 3 4)
```



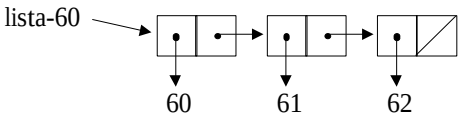
Podemos ver que *(list 1 2 3 4)* equivale a *(cons 1 (cons 2 (cons 3 (cons 4 '()))))*.

Qualquer entidade *Scheme* pode ser elemento de uma lista, o que significa que as próprias listas podem ser elementos de outras listas.

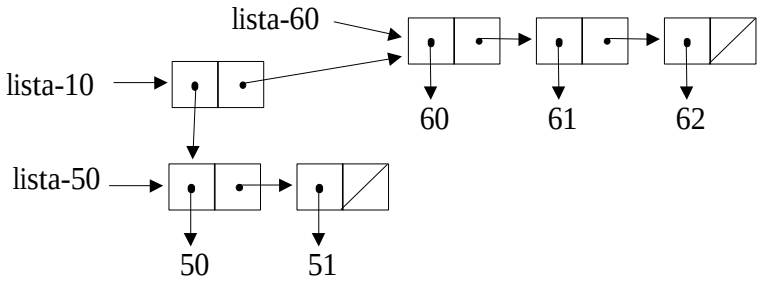
```
↦(define lista-50 (list 50 51))
lista-50
```




```
↦(define lista-60 (list 60 61 62))
lista-60
```



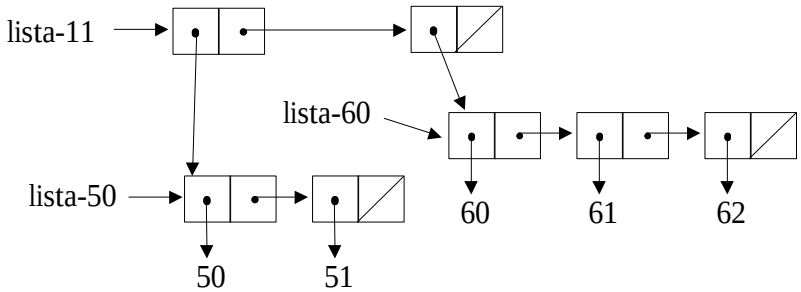

```
↦(define lista-10 (cons lista-50 lista-60))
lista-10
↦ lista-10
((50 51) 60 61 62)
```



Neste exemplo, com *cons*, é criada *lista-10* composta por 4 elementos, uma vez que se junta um elemento, *lista-50*, a uma lista de 3 elementos, *lista-60*. Também se verifica pela representação *caixa-e-apontador*, que *lista-10* é uma sequência de 4 elementos, em que o 1º é *lista-50*, e o 2º, 3º e 4º são, respectivamente, o 1º, 2º e 3º elementos de *lista-60*.

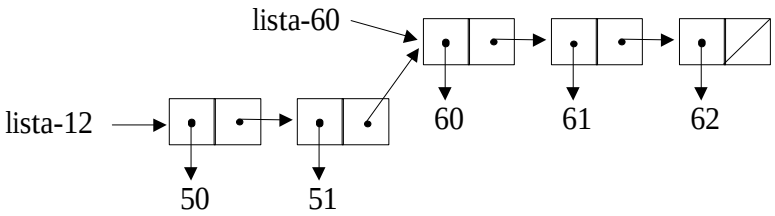
```
↦(define lista-11 (list lista-50 lista-60))
lista-11
↦lista-11
((50 52) (60 61 62))
```

Com *list*, é criada uma nova lista, *lista-11*, composta por 2 elementos. Um é *lista-50* e outro é *lista-60*.



Para além dos procedimentos primitivos *cons* e *list* utilizados para criar listas, o Scheme disponibiliza também o procedimento *append*. Este procedimento aceita listas como argumentos e devolve uma nova lista composta por todos os elementos das listas fornecidas como argumentos.

```
↦(define lista-12 (append lista-50 lista-60))
lista-12
↦lista-12
  (50 51 60 61 62)
```



De facto, o que *append* faz é copiar os elementos das várias listas dadas para uma nova lista, excepto a última lista dada que é partilhada com a lista em criação. A representação *caixa-e-apontador* de *list-12* esclarece o que acaba de se afirmar.
A lista *lista-12* é composta por cinco elementos, dois copiados de lista-50 e os três finais são partilhados com *lista-60*.

Resumem-se as várias formas para criar listas:

⇒ `(cons obj lista)`
Sendo o argumento *lista* uma lista de *m* elementos *(m pode ser zero)*, a lista resultante é de *m+1* elementos, constituída por *obj* e pelos próprios elementos de *lista*. O procedimento *cons* apenas acrescenta um elemento a uma lista existente. Por este motivo, a criação de uma nova lista com *cons* terá que ser feita elemento a elemento.

⇒ `(list obj-1 obj-2 ... obj-m)`
Sendo *m* argumentos, a lista resultante é de *m* elementos, constituída por *obj-1*, *obj-2*, ... *obj-m*. Com *list*, cria-se uma lista de uma só vez, a partir de todos os seus elementos.

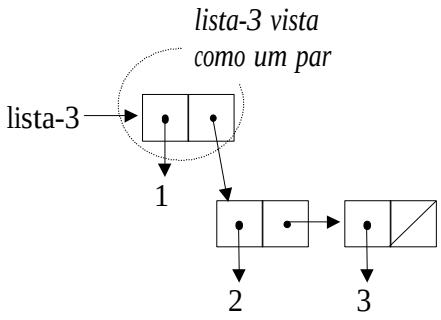
⇒ `(append lista-1 lista-2 ... lista-m)`
A lista resultante tem um número de elementos que é igual à soma do número de elementos das listas que são argumentos, e é constituída por uma cópia dos elementos de *lista-1*, seguida por uma cópia dos elementos de *lista-2*... e partilhando os elementos de *lista-m*. Com *append*, cria-se uma lista de uma só vez a partir dos elementos de várias listas.

Exercício 3.4

Sendo *lista-2* equivalente a *(list 7 (cons 8 9))* e *lista-3* equivalente a (list 1 2 3), apresentar as representações textuais e *caixa-e-apontador* dos objectos a seguir definidos, bem como o número de elementos de cada um deles.

```
(define objecto-1 (cons lista-2 lista-3))
(define objecto-2 (list lista-2 lista-3))
(define objecto-3 (append lista-2 lista-3))
```

Para aceder aos elementos de uma lista utilizam-se os selectores *car* e *cdr*, o que não é para admirar. Basta verificar que uma lista pode ser vista como um par cujo elemento da esquerda é o primeiro elemento da lista e o elemento da direita é toda a lista, excluindo o seu primeiro elemento.




```
↦(car lista-3)
1
↦(define lista-20 (cdr lista-3))[6]
lista-20
↦lista-20
(2 3)

↦lista-3 ;lista-3 mantém-se intacta
(1 2 3)

↦(car (cdr lista-3))          ; equivalente a (cadr lista-3)[7]
2
↦(cdr (cdr lista-3))          ; equivalente a (cddr lista-3)
(3)
↦(car (cdr (cdr lista-3)))    ; equivalente a (caddr lista-3)
3
```

3- Procedimentos para processamento de listas

Com os exemplos que se seguem, para além do treino de programação com listas que se proporciona, pretende-se ainda introduzir mais alguns procedimentos primitivos do Scheme[8].

Em primeiro lugar, são definidas três listas, uma delas vazia, pois corresponde a *(list)*[9].

```
↦(define lista-1 (list 1 2 3 4 5 6))
lista-1
↦(define lista-2 (list 10 11 12))
lista-2
↦(define lista-vazia (list))
lista-vazia
↦lista-vazia
()
```

Exemplo 3.2

O procedimento *soma-elementos-da-lista* recebe uma lista como argumento e devolve a soma de todos os seus elementos. Se a lista for vazia, o resultado é conhecido e igual a zero - trata-se do caso base. Os outros casos correspondem a lista não vazia e podem ser tratados recursivamente somando o primeiro elemento da lista, *(car lista)*, com a soma dos restantes, sabendo que os restantes constituem a lista que se obtém da original retirando-lhe o primeiro elemento, *(cdr lista)*. O procedimento *cdr* é, neste caso, a operação de redução, pois vai na direcção do caso base, ou seja, da lista vazia. Temos tudo o que é necessário para escrever o procedimento pedido, uma vez que o Scheme disponibiliza o predicado *null?* que aceita uma lista como argumento e devolve *#t* se a lista for vazia ou, caso contrário, devolve *#f*.

```
↦(null? lista-1)
#f
↦(null? lista-vazia)
#t
```

---

[6] cdr aplicado a uma lista não vazia devolve uma cópia da lista dada depois de lhe retirar o seu primeiro elemento.
[7] Recomenda-se a consulta do Anexo A que indica 28 formas possíveis de combinar car e cdr.
Por exemplo, *(car (cdr lista))* é equivalente a *(cadr lista)*.
[8] Recomenda-se a consulta do Anexo A, no que se refere aos procedimentos com listas.
[9] Equivalente a *(define lista-vazia '())*

```
(define soma-elementos-da-lista
  (lambda (lis)
    (if (null? lis)
        0
        (+ (car lis)
           (soma-elementos-da-lista (cdr lis))))))
```

```
↦(soma-elementos-da-lista lista-2)
33
↦(soma-elementos-da-lista lista-vazia)
0
```

Exercício 3.5

O procedimento *soma-elementos-da-lista* gera um processo recursivo. Em alternativa, escrever uma versão que crie um processo iterativo. Identificar, para cada uma das versões, a ordem de crescimento dos processos gerados, em tempo e em espaço.

Exercício 3.6

O procedimento *comprimento-da-lista* que tem como parâmetro uma lista e devolve o seu comprimento, ou seja, o número dos seus elementos.

```
↦(comprimento-da-lista lista-1)
6
```

Para o procedimento *comprimento-da-lista* identificar o caso base e o caso geral, bem como a operação de redução, antes de analisar a solução que se apresenta de seguida.

```
(define comprimento-da-lista
  (lambda (lis)
    (if (null? lis)
        0
        (add1 (comprimento-da-lista (cdr lis))))))
```

O *Scheme* disponibiliza o procedimento *length* que faz o mesmo que *comprimento-da-lista*.

```
↦(length lista-2)
3
```

Exemplo 3.3

A questão que agora se coloca relaciona-se com a selecção do elemento *n* de uma lista. Para este efeito, considera-se que o primeiro elemento é o elemento 0, o segundo é o elemento 1, e assim sucessivamente. O procedimento que se pretende é designado por *o-elemento-n-da-lista*, aceita uma lista e um inteiro como argumentos, e responde da seguinte maneira:

```
↦(o-elemento-n-da-lista lista-1 2)
3
↦(o-elemento-n-da-lista lista-1 0)
1
```

O caso base corresponde ao argumento inteiro igual a zero, com resposta conhecida, ou seja, *(car lista)*, uma vez que se trata do primeiro elemento da lista.

```
(define o-elemento-n-da-lista
  (lambda (lis n)
    (if (zero? n)
        (car lis)
        (o-elemento-n-da-lista (cdr lis) (sub1 n)))))
```

O *Scheme* disponibiliza o procedimento *list-ref* que faz o mesmo que *o-elemento-n-da-lista*.

```
↦(list-ref lista-1 2)
3
```

Tem-se falado frequentemente no primeiro elemento das listas, depois falou-se no elemento *n*. E se se pretender aceder ao último elemento de uma lista?
A questão não faz sentido se a lista for vazia, pois, neste caso, nem existe o último nem qualquer outro elemento. Se a lista tiver um único elemento, o primeiro e o último são o mesmo elemento e a resposta é conhecida. Trata-se do caso base, que se identifica fazendo *(null? (cdr lista))*[10].

Exercício 3.7

Analisar a solução que se apresenta e justificar a necessidade do procedimento interno *aux*. Seria possível fazer a mesma coisa sem recorrer a um procedimento auxiliar?

```
(define o-ultimo-elemento-da-lista
  (lambda (lis)
    (letrec ((aux
               (lambda(lis-aux)
                 (if (null? (cdr lis-aux))
                     (car lis-aux)
                     (aux (cdr lis-aux))))))
      (if (null? lis)
          (display "lista vazia")
          (aux lis)))))
```

```
↦(o-ultimo-elemento-da-lista '())
lista vazia
```

Exemplo 3.4

Apresenta-se outra solução baseada num procedimento não recursivo para aceder ao último elemento de uma lista, que faz uso dos procedimentos primitivos *length* e *list-ref*.

```
(define o-ultimo-elemento-da-lista-v1
  (lambda(lis)
    (let ((comprimento (length lis)))
      (if (zero? comprimento)
          (display "lista vazia")
          (list-ref lis (sub1 comprimento))))))
```

Segue-se mais um procedimento não recursivo para aceder ao último elemento de uma lista, que utiliza o procedimento primitivo *reverse*. Este procedimento primitivo toma uma lista como argumento e devolve outra lista constituída pelos elementos da lista dada, mas na ordem inversa.

```
↦lista-2
(10 11 12)
↦(reverse lista-2)
(12 11 10)
```

```
(define o-ultimo-elemento-da-lista-v2
  (lambda(lis)
    (if (zero? (length lis))
        (display "lista vazia")
        (car (reverse lis)))))
```

[10] Em vez de (null? (cdr lista)) também poderia ser (= 1 (length lista))

```
↦(o-ultimo-elemento-da-lista-v2 lista-2)
12
↦lista-2
(10 11 12)                ; lista-2 mantém-se intacta
```

Nos exemplos que se seguem, os procedimentos sobre listas devolvem listas, o que não acontecia com os anteriores.

Exemplo 3.5

O procedimento *remove-primeiros-n* tem uma lista e um inteiro positivo, *n*, como parâmetros, e devolve uma lista equivalente à primeira, depois de lhe retirar os primeiros *n* elementos.

```
↦(remove-primeiros-n lista-1 2)
(3 4 5 6)
↦lista-1
(1 2 3 4 5 6)            ; lista-1 mantém-se intacta
```

```
(define remove-primeiros-n
  (lambda(lis n)
    (if (zero? n)
        lis
        (remove-primeiros-n (cdr lis)
                            (sub1 n)))))
```

Também neste caso, o *Scheme* disponibiliza o procedimento *list-tail*, com uma funcionalidade semelhante a *remove-primeiros-n*.

```
↦(list-tail lista-1 2)
(3 4 5 6)
```

Exemplo 3.6

O procedimento *remove-ate-elem* aceita dois argumentos: um potencial elemento de lista e uma lista. Este procedimento devolve uma lista equivalente à lista dada, depois de lhe retirar os seus primeiros elementos até encontrar um elemento que seja igual ao elemento fornecido, o qual já não será retirado.

```
↦(remove-ate-elem 4 lista-1)
(4 5 6)
↦(remove-ate-elem 30 lista-1)
()
```

```
(define remove-ate-elem
  (lambda(elem lis)
    (cond ((null? lis) '())
          ((equal? (car lis)              ; equal?... !!!
                   elem)
           lis)
          (else
            (remove-ate-elem elem
                             (cdr lis))))))
```

A novidade que se verifica neste procedimento é o predicado *equal?*. De facto, poderia ser o operador =, num contexto limitados a valores numéricos. Mas o predicado *equal?* é muito mais genérico, e o espectro dos seus operandos é muito mais largo[11].

```
↦(equal? (list 1 2) (list 1 2))
#t
↦(= (list 1 2) (list 1 2))
=: wrong argument type pair (expected number)
```

Retomando o procedimento *remove-ate-elem*, também neste caso, o Scheme disponibiliza o procedimento *member* com uma funcionalidade semelhante[12].

```
↦(member 4 lista-1)
(4 5 6)
```

Exemplo 3.7

O procedimento *junta-duas-listas* aceita duas listas como argumentos e devolve uma lista com uma cópia de todos os elementos daquelas listas. Trata-se portanto de um caso particular de *append*, pois este procedimento primitivo do Scheme aceita um número não fixo de listas.

```
↦(junta-duas-listas lista-1 lista-2)
(1 2 3 4 5 6 10 11 12)
↦(append lista-1 lista-2)
(1 2 3 4 5 6 10 11 12)
```

```
(define junta-duas-listas
  (lambda (lis-1 lis-2)
    (if (null? lis-1)
        lis-2
        (cons (car lis-1)
              (junta-duas-listas (cdr lis-1)
                                 lis-2)))))
```

3- Símbolos

Imaginemos uma situação em que o António se dirige ao José em voz alta:

Hipótese 1- Diz ‘o teu nome’.

Hipótese 2- Diz o teu nome.

A resposta provável será José, o nome da pessoa a quem António se dirigiu. Todavia, se António em vez de comunicar oralmente o tivesse feito por escrito, as respostas às duas hipóteses teriam sido, muito provavelmente, diferentes.

Hipótese 1- o teu nome.

Hipótese 2- José.

De onde vem a diferença? Na hipótese 1, uma parte da frase vem delimitada por plicas, significando que essa parte não deverá ser processada, mas mantida integralmente. Na hipótese 2, a frase deverá ser completamente processada.

---

[11] Entre *equal? (o menos restritivo, aplicável a tudo)* e *= (o mais restritivo, só aplicável a números)*, o Scheme ainda disponibiliza mais dois predicados, *eq?* e *eqv?*, cujos campos de aplicação não estão muito bem especificados, dependendo, algumas vezes, da implementação do próprio interpretador Scheme. Por este motivo, não se lhes dedica grande atenção, e utiliza-se apenas = em contextos numéricos simples (não compostos) e *equal?* nas restantes situações.

[12] Para além de *member*, que utiliza o predicado *equal?*, o Scheme ainda disponibiliza *memv* e *memq* que utilizam, respectivamente, *eqv?* e *eq?*, e por este motivo, com um campo de aplicação igualmente não muito bem especificado, como se refere na anterior nota de fim-de-página.

Em Scheme, uma *plica* tem um significado idêntico ao que acabámos de apresentar, possibilitando delimitar expressões ou parte de expressões que não deverão ser processadas, mas mantidas integralmente. Vejamos alguns exemplos:

```
↦(define lista-vazia '())
lista-vazia
↦lista-vazia
()
↦(define lista-123 '(1 2 3))
lista-123
↦lista-123
(1 2 3)
↦(define nome 'ana)
nome
↦nome
ana
↦(define lista-nomes '(ana jose manuel))
lista-nomes
↦lista-nomes
(ana jose manuel)
```

Se, como alternativa a este último exemplo, tentássemos

```
↦(define lista-nomes (list ana jose manuel))
```

a expressão `(list ana jose manuel)` seria processada para se determinar o valor que iria ser associado ao símbolo *lista-nomes*, e então os argumentos *ana*, *jose*, e *manuel* seriam entendidos como nomes de variáveis, provavelmente não definidas.

Como alternativa à *plica*, o Scheme disponibiliza o procedimento *quote*, com uma funcionalidade semelhante.

```
↦(quote (1 d 2 f))
(1 d 2 f)
```

Exercício 3.8
Analisar as várias situações que se apresentam e indicar os respectivos resultados:

```
↦(define a 10)
a
↦(car '(a b c d))
???
↦(car (list a))
???
↦(car (list 'a))
???
↦(car '(list a))
???
↦(list (a 'a))
???
```

Exemplo 3.8
Imaginemos que numa certa aplicação computacional são utilizadas cores, as quais são referidas directamente pelos seus nomes e não por códigos numéricos. Os nomes das cores em jogo e os

respectivos códigos são: `'vermelho'` código 1, `'verde'` código 2, `'azul'` código 3, e `'amarelo'` código 4.
O procedimento *num->cor*, que se segue, recebe como argumento um número, possivelmente um código de uma cor, e devolve um símbolo correspondente ao nome dessa cor.

```
↦(num->cor 1)
vermelho
↦(num->cor 6)
codigo nao previsto
```

```
(define num->cor
  (lambda (num)
    (cond ((= num 1) 'vermelho)
          ((= num 2) 'verde)
          ((= num 3) 'azul)
          ((= num 4) 'amarelo)
          (else (display "codigo nao previsto")))))
```

Exercício 3.9

Definir o procedimento *cor->num* que responde como se indica:

```
↦(cor->num 'vermelho)
1
↦(cor->num 'amarelo)
4
↦(cor->num 'rosa)
cor nao prevista
```

4- Abstracção de dados - Conjunto

A definição e a construção da abstracção *Conjunto* são vistas com alguma profundidade, sendo analisadas várias alternativas para representar este tipo de entidades. Podemos definir informalmente *conjuntos* como sendo colecções de objectos distintos, cujos operadores típicos são:

- *(faz-conjunto)*
  Não espera qualquer argumento e devolve um conjunto vazio;
- *(junta-elemento-a-conjunto obj conj)*
  Toma um objecto e um conjunto como argumentos e devolve um conjunto formado pelos elementos do conjunto dado aos quais se junta aquele objecto
- *(uniao-de-conjuntos conj1 conj2)*
  Toma dois conjuntos como argumentos e devolve um conjunto com os elementos que pertencem a pelo menos um dos conjuntos;
- *(interseccao-de-conjuntos conj1 conj2)*
  Toma dois conjuntos como argumentos e devolve um conjunto com os elementos que pertencem necessariamente aos dois conjuntos;
- *(elemento-do-conjunto? obj conj)*
  É um predicado que toma um objecto e um conjunto como argumentos e devolve *#t* ou *#f*, caso o objecto pertença ou não ao conjunto dado.

Por uma breve análise, poderemos concluir que todos os operadores se apresentam como construtores de conjuntos (todos eles devolvem conjuntos), excepto o último, *elemento-do-conjunto?*, que se assume como um selector.
Falta implementar estes operadores para que os conjuntos, como uma abstracção de dados, passem a estar disponíveis para o desenvolvimento de aplicações que envolvam este tipo de dados. Para isso, é necessário decidir como os representar, decisão importante, mas não dramática, se não for a melhor. De facto, se após o desenvolvimento de uma aplicação se vier a concluir pela necessidade de uma representação mais adequada[13], o que há a fazer é escolher outra representação que nos pareça melhor e voltar a implementar os operadores relacionados com os conjuntos. As *barreiras de abstracção* convenientemente aplicadas garantirão que a aplicação não deverá sofrer alterações, independentemente da representação escolhida para os dados[14]. Assim sendo, e sem grandes preocupações de errar, mas sobretudo para tentar rapidamente chegar a uma abstracção disponível, vamos utilizar listas não ordenadas para representar os conjuntos. Isto significa, por exemplo, que o conjunto constituído por 1, 2, 3, e 4, poderá ser representado por (1 2 3 4), ou (3 4 1 2), ou uma outra lista que contenha apenas aquela valores. O conjunto vazio será representado por uma lista vazia.

Antes de se apresentar a implementação dos operadores que actuam sobre conjuntos, vejamos como funcionam:

```
↦(define c-1 (faz-conjunto))
c-1
↦ c-1
()
↦(define c-2 (junta-elemento-a-conjunto 5 c-1))
c-2
↦ c-2
(5)
↦(define c-3 (junta-elemento-a-conjunto 'abc c-2))
c-3
↦ c-3
(abc 5)
↦(interseccao-de-conjuntos c-2 c-3)
(5)
↦(define c-10 (faz-conjunto))
c-10
↦(define c-11 (junta-elemento-a-conjunto 4 c-10))
c-11
↦(define c-12 (junta-elemento-a-conjunto 5 c-11))
c-12
↦ c-12
(5 4)
↦(uniao-de-conjuntos c-12 c-3)
(4 abc 5)
```

Segue-se agora a implementação dos operadores escolhidas para esta abstracção.
A construção de um conjunto vazio não é tarefa complicada, pois basta devolver uma lista vazia, de acordo com a representação escolhida para os conjuntos.

---

[13] A inadequação pode resultar, por exemplo, da representação inicialmente escolhida ser muito pouco eficiente do ponto de vista dos tempos de resposta ou então por ser muito gastadora de memória
[14] Como aconteceu no exemplo das tartarugas coloridas

```
(define faz-conjunto
  (lambda()
    '()))[15]
```

Para se determinar se um objecto pertence a um conjunto, comparam-se, um a um, os elementos do conjunto com o objecto dado. Se o conjunto for vazio, a resposta é imediata e igual a *#f*, pois o objecto não pode estar num conjunto vazio. Trata-se do caso base. Não sendo vazio, compara-se o objecto com o primeiro elemento da lista que representa o conjunto. Se forem iguais, então a resposta é *#t* e acaba a comparação. Sendo diferentes, recorre-se a uma estratégia recursiva, baseada na redução da lista que representa o conjunto.

```
(define elemento-do-conjunto?
  (lambda (obj conj)
    (cond ((null? conj) #f)
          ((equal? obj (car conj)) #t)
          (else (elemento-do-conjunto? obj (cdr conj))))))
```

Juntar um elemento a um conjunto resume-se, no tipo de representação escolhida, a juntar um objecto a uma lista, tendo apenas o cuidado de garantir que esse objecto não é ainda elemento do conjunto, para evitar elementos repetidos.

```
(define junta-elemento-a-conjunto
  (lambda (obj conj)
    (if (elemento-do-conjunto? obj conj)
        conj
        (cons obj conj))))
```

A união de dois conjuntos pode resumir-se a percorrer um dos conjuntos e identificar, um a um, todos os seus elementos que não pertençam ao segundo conjunto. O conjunto resultante da união será constituído pelos elementos identificados como não pertencendo ao segundo conjunto, mais os elementos deste segundo conjunto. Garante-se assim que a união dos dois conjuntos continua a ser um conjunto sem elementos repetidos.
Numa óptica de solução recursiva, o caso geral resume-se à redução de um dos conjuntos, tomando-se, um a um, todos os seus elementos, até se atingir um conjunto vazio. Atingido o conjunto vazio, esta situação correspondente ao caso base, e a união será o outro conjunto. Enquanto não se atinge o conjunto vazio, vai-se verificando se cada um dos seus elementos já se encontra no outro conjunto. Se assim for, não será considerado para fazer parte da união, caso contrário, será imediatamente considerado.

```
(define uniao-de-conjuntos
  (lambda (conj1 conj2)
    (cond ((null? conj1) conj2)
          ((elemento-do-conjunto? (car conj1) conj2)
           (uniao-de-conjuntos (cdr conj1) conj2))
          (else
            (cons (car conj1)
                  (uniao-de-conjuntos (cdr conj1) conj2))))))
```

Exercício 3. 10

Explicar o funcionamento do procedimento *interseccao-de-conjuntos*, a seguir indicado.

[15] Em vez de *'()*, também poderia ser *(list)* ou *(quote ())*

```
(define interseccao-de-conjuntos
  (lambda (conj1 conj2)
    (cond ((or
             (null? conj1)
             (null? conj2))
           '())
          ((elemento-do-conjunto? (car conj1) conj2)
           (cons (car conj1)
                 (interseccao-de-conjuntos (cdr conj1) conj2)))
          (else
            (interseccao-de-conjuntos (cdr conj1) conj2)))))
```

Exercício 3.11

Após a implementação dos operadores que actuam sobre conjuntos, baseados em listas não ordenadas e sem elementos repetidos, apresenta-se a Ordem de Crescimento dos processos que cada um deles gera. Comentar a tabela que se segue.

	tempo	espaço
faz-conjunto	*O(1)*	*O(1)*
elemento-do-conjunto?	*O(n)*	*O(1)*
junta-elemento-a-conjunto	*O(n)*	*O(1)*
uniao-de-conjuntos	$O(n^2)$	*O(n)*
interseccao-de-conjuntos	$O(n^2)$	*O(n)*

5- Abordagem *de-cima-para-baixo* e a Abstracção de dados

Vamos atacar este tema com a ajuda de um exemplo. Um programa, que poderá ser utilizado para testar a capacidade de memorização de quem o utiliza, vai ser desenvolvido sobre a abstracção *conjunto*, fazendo uso dos seguintes operadores: *faz-conjunto*, *elemento-do-conjunto?* e *junta-elemento-a-conjunto*. O programa, designado por *testa-memoria*, visualiza números aleatórios, um a um, e pergunta, para cada um deles, se já terá saído. As respostas incorrectas vão sendo contadas, pois, no final, o programa visualiza essa informação. A abstracção *conjunto* será utilizada para memorizar os números que vão sendo visualizados, permitindo assim verificar a correcção das respostas dadas. O programa recebe três argumentos, os dois primeiros definem a gama em que os números aleatórios são gerados, e o terceiro define a quantidade de números que são visualizados em cada sessão. Segue-se um exemplo de uma dessas sessões:

```
↦(testa-memoria  1 10 5)

Serao visualizados 5 numeros aleatorios, um a um, situados entre 1 e 10
e tera' que responder, para cada um deles, se ja' saiu.

Mais um numero: 9
Ja' saiu? s/n:
n
Mais um numero: 1
Ja' saiu? s/n:
b                                   ; resposta diferente de s ou n
Ja' saiu? s/n:                      ; repete a pergunta
n
Mais um numero: 9
Ja' saiu? s/n:
s
Mais um numero: 10
Ja' saiu? s/n:
n
Mais um numero: 7
Ja' saiu? s/n:
n
```

```
Dos 5 numeros, falhou 0
```

O programa *testa-memoria* já apresenta uma certa complexidade, justificando uma abordagem do tipo *de-cima-para-baixo*[16]. Segundo esta abordagem, procuram-se, em primeiro lugar, as grandes tarefas associadas ao programa e imagina-se para cada uma delas um procedimento. Todavia, se alguma destas tarefas ainda se mostrar complexa, então também se deverá procurar identificar as suas grandes tarefas e definir, para cada uma delas, os respectivos procedimentos. A ideia chave é dividir as tarefas em tarefas cada vez mais pequenas e simples, de tal forma que os procedimentos que as implementam também sejam simples.
Retomando o programa *testa-memoria*, começamos por identificar as suas grandes tarefas:

⇒ Visualizar a mensagem inicial

⇒ Repetir ciclicamente as perguntas e contabilizar o número de respostas erradas

⇒ Visualizar a mensagem final.

Assim, podemos começar a escrever o programa *testa-memoria*.

```
(define testa-memoria
  (lambda (lim-inf lim-sup n-vezes)
    ;
    (mensagem-inicial lim-inf lim-sup n-vezes)
    ;
    (let ((respostas-erradas (repete-ciclo n-vezes lim-inf lim-sup)))
      ;
      (mensagem-final n-vezes respostas-erradas))))
```

Por uma breve análise, verifica-se que a primeira e terceira tarefas são relativamente simples, admitindo, de imediato, a definição dos respectivos procedimentos.

```
(define mensagem-inicial
  (lambda (lim-i lim-s n-perguntas)
    (newline)
    (display "Serao visualizados ")
    (display n-perguntas)
    (display " numeros aleatorios, um a um, situados entre ")
    (display lim-i)
    (display " e ")
    (display lim-s)
    (newline)
    (display "e tera' que responder, para cada um deles, se ja' saiu.")
    (newline)))

(define mensagem-final
  (lambda (n-perguntas n-erros)
    (newline)
    (newline)
    (display "Dos ")
    (display n-perguntas)
    (display " numeros, falhou ")
    (display n-erros)))
```

Geralmente, a simplicidade de uma tarefa e do correspondente procedimento mede-se pelo número de linhas necessárias à definição deste. Um procedimento que ultrapasse uma página já se torna de legibilidade duvidosa, dificultando a sua escrita, teste e, sobretudo, a introdução de alterações futuras. Um procedimento com estas características será sempre de manutenção complexa. Seria bom que os procedimentos não ultrapassassem a meia página, o ideal até seria o

[16] Abordagem *top-down*

quarto de página. Foi com estas regras de bom senso em mente que se procurou identificar as grandes tarefas da segunda tarefa, a que repete ciclicamente as perguntas e contabiliza o número de respostas erradas.

⇒ Enquanto *numero de perguntas* for diferente de zero, executar as seguintes tarefas:

- Gerar um novo número aleatório
- Visualizar o número gerado
- Perguntar se o número já saiu (o que implica a memorização de todos os números já saídos) e determinar se a resposta dada é incorrecta
- Juntar o número gerado ao conjunto dos números gerados, se for necessário actualizar o número de respostas erradas, e descontar um ao *número de perguntas*

⇒ Devolver o número de respostas erradas

A repetição do conjunto de tarefas não foi obtida chamando recursivamente o procedimento *repete-ciclo*, com a redução de *n-perguntas*, pois não seria possível contabilizar o número de respostas erradas e memorizar os números aleatórios gerados. Por este motivo, optou-se por introduzir o procedimento recursivo *aux*, local ao procedimento *repete-ciclo*, com os parâmetros *n-perg*, *n-erros*, e *saidos*, através dos quais será possível, em cada chamada recursiva, reduzir o número de perguntas, contar as respostas erradas e actualizar o conjunto dos números já saídos. Verificar que a tarefa que *pergunta se o número já saiu* exigiu uma decomposição adicional em tarefas mais simples.

```
(define repete-ciclo
  (lambda (n-perguntas lim-i lim-s)
    ;
    (letrec ((aux
               (lambda (n-perg n-erros saidos)
                 (if (zero? n-perg)
                     n-erros      ; o caso base ...
                     ;              o caso geral:    gera novo número aleatório
                     (let ((novo-num (roleta-n-m lim-i lim-s))) ; que
                       (visu-novo-numero novo-num)      ; será visualizado
                       ;
                       (let ((resposta-errada
                               (pergunta-se-ja-saiu novo-num saidos)))
                         ;
                         (aux (sub1 n-perg)
                              (if resposta-errada
                                  (add1 n-erros)
                                  n-erros)
                              (junta-elemento-a-conjunto
                                novo-num
                                saidos))))))))
      ;     chama aux com n-perguntas, zero erros, e conjunto vazio...
      (aux n-perguntas 0 (faz-conjunto)))))

(define roleta-n-m       ; gera um número aleatório
  (lambda (inf sup)      ; entre inf e sup
    (+ inf
       (remainder (random)
                  (- sup (sub1 inf))))))

(define visu-novo-numero
  (lambda (num-aleat)
    (newline)
    (display "Mais um numero: ")
    (display num-aleat)
    (newline)))
```

```
(define pergunta-se-ja-saiu
  (lambda (num saidos)
    (let ((ja-saiu (elemento-do-conjunto?   ; verifica se já saiu
                     num
                     saidos))
          (sim (resposta-sim?)))  ; determina a resposta do utilizador
      (or
        (and ja-saiu           ; a resposta está
             (not sim))              ; errada?
        (and (not ja-saiu)
             sim)))))

(define resposta-sim?              ; devolve #t se resposta s
  (lambda ()                       ; devolve #f se resposta n
    (display "Ja' saiu? s/n: ")    ; nos outros casos, repete a pergunta
    (let ((resposta (read)))
      (cond ((equal? resposta 's)
             #t)
            ((equal? resposta 'n)
             #f)
            (else
              (newline)
              (resposta-sim?))))))
```

Grande parte do capítulo tem sido dedicada ao conceito da *abstracção de dados*, mas também se introduziu a abordagem *de-cima-para-baixo*. Estes conceitos não parecem constituir vias alternativas, mas sim, muitas vezes, vias complementares. A criação da abstracção de dados que se adeqúe a um problema segue, geralmente, uma via *de-baixo-para-cima*. Define-se a melhor estrutura de dados, define-se ainda as operações básicas, e com elas estabelece-se uma barreira de abstracção. Sobre a barreira estabelecida definem-se outras abstracções e as respectivas barreiras, e assim sucessivamente, e se necessário for, até à solução final. Com o programa *testa-memoria*, deduziu-se rapidamente, que a melhor abstracção seria o *conjunto*, sem com isto significar que seja esta a única solução possível. Mas depois de estabelecida esta base, tentou-se uma abordagem *de-cima-para-baixo*, identificando as tarefas principais do problema e, destas, as mais complexas, foram ainda alvo de uma abordagem semelhante. Há quem defenda que se utilize um único tipo de abordagem ou *de-cima-para-baixo* ou *de-baixo-para-cima*. A inexistência de uma opinião única sobre o assunto e pela experiência pessoal, sugere-se uma abordagem mista, atacando a base, com a *abstracção de dados*, seguindo-se uma abordagem *de-cima-para-baixo* na identificação de tarefas e subtarefas. Caberá a cada um, com muito bom senso e também com a sua experiência, dosear o esforço em cada uma das vias, estando conscientes que, em certas situações, não se justifica um grande esforço na abordagem *de-cima-para-baixo*, porque a tarefa a resolver se apresenta muito simples, ou na *abstracção de dados*, se os dados do problema e as suas operações básicas forem facilmente resolvidas com as abstracções disponibilizadas pela própria linguagem de programação.

Exercício 3.12

Uma outra representação possível para os conjuntos seria a lista com os elementos não ordenados, mas permitindo elementos repetidos. Por exemplo, o conjunto {1 d 4 3} poderia ser representado pela lista (1 d 4 3), mas também pela lista (d 1 4 1 d 3) e por muitas outras. Para este tipo de representação, implementar os operadores e identificar a respectiva ordem de crescimento.

Exercício 3.13

No contexto do exercício anterior, verificar que o programa *testa-memoria* também funciona com os operadores relativos a esta nova representação dos conjuntos e nem requer alterações se esses operadores mantiverem a mesma *interface*, ou seja, o mesmo nome e os mesmos parâmetros. Explicar este facto à luz das *barreiras de abstracção*.

Agora vamos tentar uma representação dos conjuntos baseada em listas com elementos em ordem crescente e sem elementos repetidos. Para facilitar o problema, supõe-se que os conjuntos são constituídos por elementos numéricos. O conjunto vazio continua a ser representado por uma lista vazia, pelo que se mantém *faz-conjunto*.
Quanto a *elemento-do-conjunto?*, há uma diferença a considerar, pois a pesquisa deve terminar logo que o objecto se apresente menor que o primeiro elemento da lista que representa o conjunto. Não esquecer que os elementos estão ordenados do menor para o maior, e se o objecto é menor que o primeiro da lista, então seria pura perda de tempo continuar a pesquisa.

```
(define elemento-do-conjunto?
  (lambda (obj conj)
    (cond ((or
            (null? conj)
            (< obj ( car conj)))
           #f)
          ((= obj (car conj)) #t)
          (else (elemento-do-conjunto? obj (cdr conj))))))
```

Em termos de espaço, o processo gerado por *elemento-do-conjunto?* apresenta um comportamento *O(1)*. O tempo gasto na pesquisa é, em média, *n/2*, em que *n* representa o número de elementos do conjunto. Sendo assim, retirando a parte constante, o processo em termos de tempo tem um comportamento *O(n)*.
O procedimento *junta-elemento-a-conjunto* não deverá basear-se, como na representação anterior, em *elemento-do-conjunto?*. Naquela representação, o elemento a acrescentar é colocado em qualquer posição, depois de se ter determinado que ainda não fazia parte do conjunto. Na representação actual, o elemento a acrescentar deverá ser colocado ordenado, por isso é necessário encontrar a sua posição correcta, percorrendo a lista que representa o conjunto. Ao percorrer a lista com este objectivo, aproveita-se também para determinar se o objecto já faz parte ou não do conjunto. Evita-se assim percorrer a lista duas vezes.

```
(define junta-elemento-a-conjunto
  (lambda (obj conj)
    (cond ((null? conj)
           (list obj))
          ((> obj (car conj))
           (cons (car conj)
                 (junta-elemento-a-conjunto obj (cdr conj))))
          (else
            (cons obj conj)))))
```

Em termos de espaço, o processo gerado por *junta-elemento-a-conjunto* apresenta um comportamento *O(n)*. O tempo gasto na pesquisa é, em média, *n/2*, em que *n* representa o número de elementos do conjunto. Sendo assim, retirando a parte constante, o processo em termos de tempo tem um comportamento *O(n)*.
Na representação de conjuntos como listas ordenadas, a intersecção de dois conjuntos realiza-se percorrendo-os simultaneamente. Verifica-se se o primeiro elemento de um conjunto é igual ao primeiro elemento do outro conjunto. Se assim for, esse elemento pertence à intersecção e

avança-se para o elemento seguinte, nos dois conjuntos. Sendo diferentes, avança-se apenas para o elemento seguinte no conjunto em que o primeiro elemento é menor. Ou seja, em cada passo acontece sempre um avanço: Apenas num dos conjuntos, quando os primeiros elementos são diferentes, ou nos conjuntos, quando os primeiros elementos são iguais.

```
(define interseccao-de-conjuntos
  (lambda (conj1 conj2)
    (if (or
          (null? conj1)
          (null? conj2))
        '()
        (let ((elem-1 (car conj1))
              (elem-2 (car conj2)))
          (cond
            ((= elem-1 elem-2)
             (cons elem-1
                   (interseccao-de-conjuntos (cdr conj1)
                                             (cdr conj2))))
            ((< elem-1 elem-2)
             (interseccao-de-conjuntos (cdr conj1)
                                       conj2))
            (else
              (interseccao-de-conjuntos conj1
                                        (cdr conj2)))))))))
```

Em termos de espaço, o processo gerado por *interseccao-de-conjuntos* apresenta um comportamento *O(n)*. O tempo gasto na pesquisa é, em média, *n*, quando praticamente todos os elementos pertencem à intersecção (situação em que se avança quase sempre nos dois conjuntos) e *2xn*, quando poucos elementos pertencem à intersecção (situação em que se avança apenas num dos conjuntos). Retirando a parte constante, o processo em termos de tempo tem um comportamento *O(n)*, muito melhor que o comportamento $O(n^2)$ da representação não ordenada.

Exercício 3.14
Definir o procedimento *uniao-de-conjuntos* na representação de conjuntos com listas ordenadas e identificar a respectiva Ordem de Crescimento.

6- Procedimentos gráficos
Os procedimentos gráficos disponibilizados pelas várias implementações do *Scheme* são, em geral, sofisticados e requerem alguma experiência para uma utilização correcta. Para os exercícios propostos que requeiram saída gráfica, optou-se por um conjunto de procedimentos simples que funcionam numa janela gráfica, à qual se associa uma caneta que desenha, pinta[17] e escreve, com cores à escolha. No *Anexo B*, os procedimentos gráficos são apresentados, bem como alguns exemplos de utilização. No mesmo anexo, encontra-se a definição dos procedimentos gráficos para o *EdScheme* e para o *DrScheme*. Para outras implementações de *Scheme* o conjunto de procedimentos deverá ser re-escrito, tarefa que poderá ser considerada no contexto do capítulo sobre *Dados Mutáveis*.

Exemplo 3.9
No capítulo anterior, surgia como exercício, a escrita em *Scheme* do procedimento *caminho-errante-2d* que esperava, como argumentos, dois valores inteiros, considerados as coordenadas *x* e *y* de

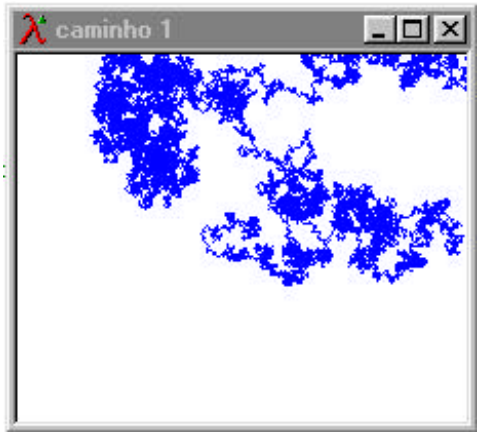
[17] Pintar significa preencher polígonos com a cor corrente

um ponto 2D, tomado como ponto de partida de um caminho. O procedimento gera, aleatoriamente, um número inteiro entre 1 e 3 e quando o número gerado é 1 subtrai 1 à coordenada *x*, quando é 2 soma 1 a essa mesma coordenada, e quando é 3 nada altera. Depois de gerar mais um número aleatório entre 1 e 3, e faz a mesma coisa para a coordenada *y*. Este procedimento não tinha fim.

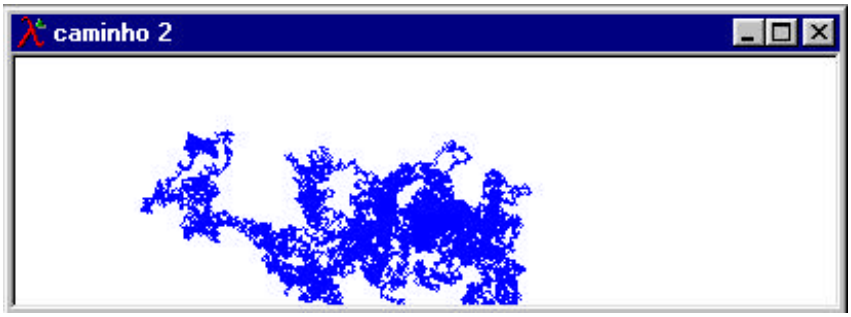
```
↦(caminho-errante-2d 20 20)
20:20  21:21  22:20  21:21  21:21  21:20  21:20  22:20  22:19
22:18  21:18  21:19  20:19  21:20  22:20  22:19  21:19  22:20
21:19  20:20  21:21  20:21  19:21  19:22  18:21  19:22  18:22
19:23  19:24  18:24  19:24  18:24  17:25  18:25  18:24
let: interrupt!
```

Como exemplo, vai-se desenvolver o procedimento *caminho-errante-2d-graf* que tem como parâmetros *x* e *y* que representam as coordenadas do ponto de partida e *n-pontos* que define o número de pontos do caminho. A evolução do percurso segue as mesmas regras da versão anterior, mas o resultado vem sob a forma gráfica, pois o procedimento vai unindo os pontos gerados através de segmentos de recta.

```
↦(janela 220 180 "caminho 1")
#[graphics window 7351396]
↦(caminho-errante-2d-graf 110 90 50000)
```




```
↦(janela 400 120 "caminho 2")
#[graphics window 7321968]
↦(caminho-errante-2d-graf 200 60 100000)
```



O procedimento *caminho-errante-2d-graf* apresenta uma definição recursiva, em que o caso base corresponde a *n-pontos* igual a zero, significando que não há mais pontos para visualizar. O caso geral contempla o cálculo do ponto seguinte, de acordo com as regras anteriormente indicadas. É por este facto que são determinados, previamente, dois números aleatórios entre 1 e 3, designados localmente por rol-x e rol-y. A utilização de um procedimento interno auxiliar, com parâmetros equivalentes aos de *caminho-errante-2d-graf*, justifica-se para evitar a repetição da selecção da cor, que é sempre a mesma, em todos os ciclos do processo.

```
(define caminho-errante-2d-graf
  (lambda (ori-x ori-y n-pontos)
    (cor 9)
    (letrec ((aux
              (lambda (x y n-pontos)
                (if (zero? n-pontos)
                    ;
```

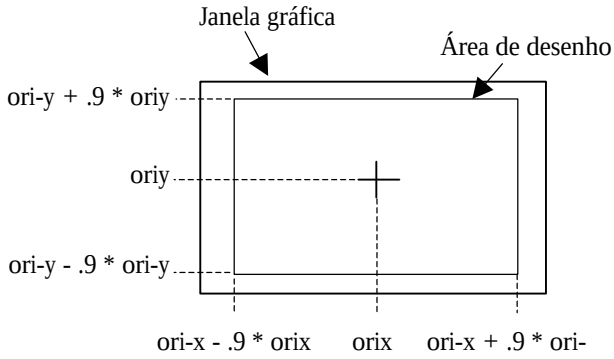
```
                    (display "terminou...")
                    ;
                    (let ((rol-x (roleta 3))
                          (rol-y (roleta 3)))
                      (let ((x-seg (cond ((= rol-x 1)(+ x 1))
                                         ((= rol-x 2)(- x 1))
                                         (else x)))
                            (y-seg (cond ((= rol-y 1)(+ y 1))
                                         ((= rol-y 2)(- y 1))
                                         (else y))))
                        (desenha (list (list x y)
                                       (list x-seg y-seg)))

                        (aux x-seg y-seg (sub1 n-pontos)))))))
      (aux ori-x ori-y n-pontos))))

(define roleta
  (lambda (limite)
    (add1 (remainder (random) limite))))
```

Exercício 3.15

No contexto do exemplo anterior, inidicar as alterações necessárias ao procedimento *caminho-errante-2d-graf* para utilizar *desenha-rel* em vez de *desenha*.

Exercício 3.16

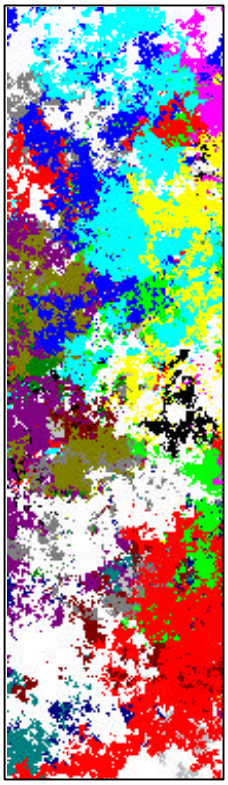
O procedimento *caminho-errante-2d-graf* apenas utiliza a cor 9 *(azul)*. Pretende-se agora uma nova versão do procedimento, designada por *caminho-errante-2d-graf-colorido*, que utiliza várias cores. Normalmente, a origem do traçado, de coordenadas *ori-x* e *ori-y*, coincide com o centro da janela gráfica. Agora, ao desenhar, é verificado se o ponto seguinte ultrapassa os limites de uma área de desenho rectangular, centrada na origem do traçado, como se indica na figura. Sempre que é detectado que o ponto seguinte ultrapassa os limites da área:

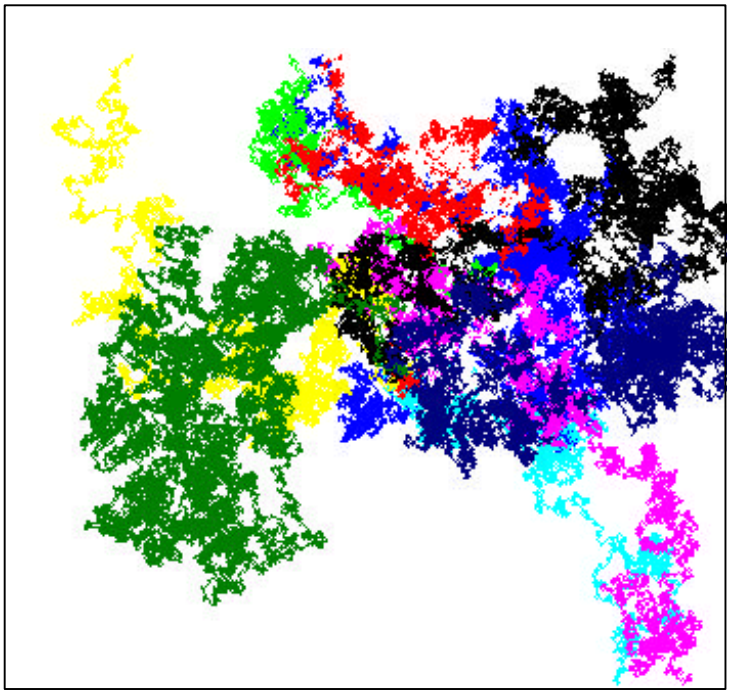


⇒ A cor corrente, de indice *i*, passa a ser a cor de índice *i + 1*, excepto quando *i = 15*, situação em que a cor seguinte passa a ser *0*;

⇒ O ponto seguinte retoma a origem do traçado.

Escrever em *Scheme* o procedimentto *caminho-errante-2d-graf-colorido* que mediante o formato da janela gráfica corrente, a origem do traçado, o número de pontos e a sorte e paciência do momento, permite saídas gráficas como as que se indicam.





Exemplo 3.10

No capítulo anterior, surgia, como exemplo, a escrita em *Scheme* do procedimento *impar-cima-par-baixo* com um único parâmetro, correspondente a um inteiro positivo. Se esse inteiro for par, dividi-o por 2. Se for ímpar, multiplica-o por 3 e depois soma 1. O resultado assim obtido é sujeito a um tratamento idêntico ao indicado e só pára quando o resultado for 1.

```
↦(impar-cima-par-baixo 6)
6  3  10  5  16  8  4  2  1  OK

↦(impar-cima-par-baixo 31)
31  94  47  142  71  214  107  322  161  484  242  121  364  182  91
274  137  412  206  103  310  155  466  233  700  350  175  526  263
790  395  1186  593  1780  890  445  1336  668  334  167  502  251  754
377  1132  566  283  850 425  1276  638  319  958  479  1438  719  2158
1079  3238  1619  4858  2429  7288  3644  1822  911  2734  1367  4102
2051  6154  3077  9232  4616  2308  1154  577  1732  866  433  1300  650
325  976  488  244  122  61  184  92  46  23  70  35  106  53  160  80
40  20  10  5  16  8  4  2  1  OK
```

Agora interessa-nos uma saída gráfica para este procedimento, já que em certos casos, como aconteceu com o valor 31, a saída alfanumérica não é muito esclarecedora da forma como os números gerados convergem até ao valor 1. A ideia base reside em criar uma função y = f(x), em que *y* corresponde aos valores gerados e *x* corresponde à ordem de ocorrência. Assim, por exemplo, para a sequência correspondente ao número 6, a x=0 corresponde o valor 6, a x=1 o valor 3, a x=2 o valor 10, ..., a x=8 o valor 1). Com este fim em vista, principia-se pela apresentação do procedimento *eixos*, com os parâmetros *origem*, *eixo-xx*, e *eixo-yy* e que desenha um sistema de eixos na janela gráfica corrente. O valor que devolve não é relevante. Segue uma breve descrição dos parâmetros do procedimento *eixos*:

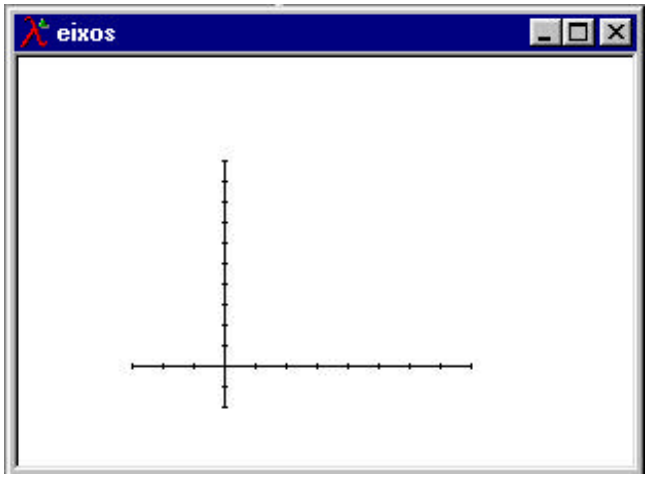
⇒ *origem* - lista de 2 elementos que representam, respectivamente, as coordenadas *x* e *y* da origem dos eixos na janela corrente;

⇒ *escala* - lista de 2 elementos que representam, respectivamente, o número de unidades do ecrã que separam, entre si, as divisões dos eixos do *xx* e do eixo dos *yy*;

⇒ *eixo-xx* - lista de 2 elementos que representam, respectivamente, o número de divisões na parte negativa do eixo dos *xx*, e o número de divisões na parte positiva do mesmo eixo;

⇒ *eixo-yy* - lista de 2 elementos com um significado idêntico ao parâmetro *eixo-xx*, mas agora para o eixo dos *yy*.

```
↦(janela 300 200 "eixos")
#[graphics window 7368308]
↦(eixos '(100 50)
          '(15 10)
          '(3 8)
          '(2 10))
()
```



Neste exemplo, os eixos têm 3 divisões na parte negativa do eixo dos *xx*, 8 divisões na parte positiva do mesmo eixo, e 2 e 10 divisões nas partes negativa e positiva do eixo dos *yy*, as divisões nos eixos apresentam, entre si, um afastamento de 15 unidades do ecrã no eixo dos *xx* e um afastamento de 10 no eixo dos *yy*. A origem dos eixos encontra-se no ponto de coordenadas (100 50).

Por sua vez, *impar-cima-par-baixo-graf*, o procedimento que nos interessa na versão gráfica, tem os parâmetros *numero*, que representa o valor a processar, *origem*, com o mesmo significado do indicado para o procedimento *eixos*, e o parâmetro *escala*, lista de 2 elementos, que funcionam como factores de escala em relação a *x* e *y*.

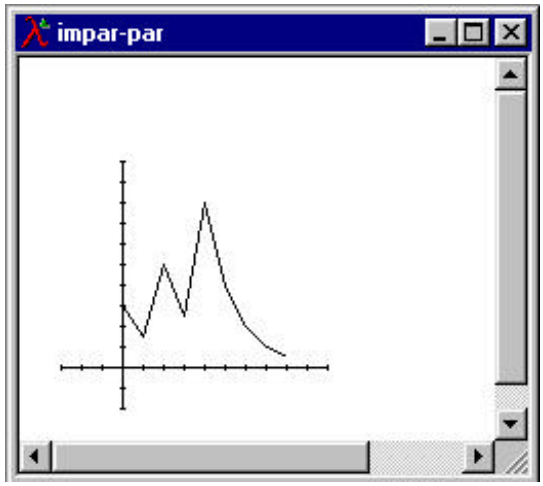
O procedimento *impar-cima-par-baixo-graf* realiza as seguintes tarefas:

⇒ calcula a lista dos números

⇒ prepara esta lista para visualizar

⇒ visualiza a lista preparada

⇒ devolve a lista dos números



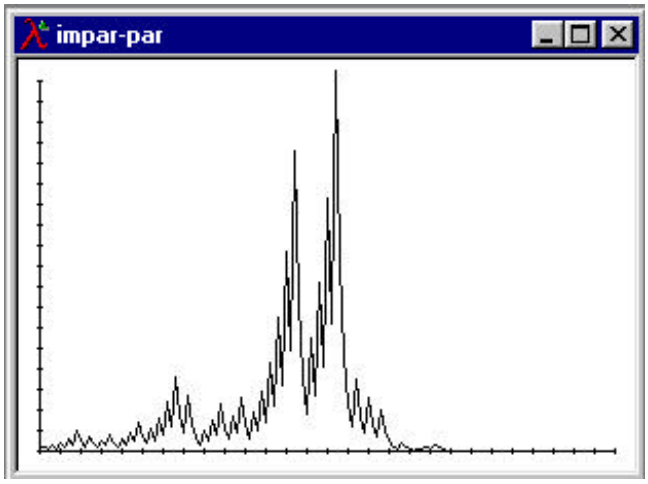

```
↦(janela 300 200 "impar-par")
#[graphics window 7353908]
↦(eixos '(50 50) '(10 10) '(3 10) '(2 10))
()
↦ (impar-cima-par-baixo-graf 6 '(50 50) '(10 5))
(6 3 10 5 16 8 4 2 1)
```

Notar, neste exemplo, que o factor de escala em relação ao eixo dos *yy* é 5, significando que os valores de *y*, no ecrã, são reduzidos a metade, uma vez que a escala correspondente na chamada

do procedimento *eixos* é de 10. Já em relação ao eixo dos *xx*, o factor de escal é de 10, coincidindo com a correspondente escala na chamada de *eixos*, pelo que os valores de *x* são visualizados em verdadeira grandeza.

```
↦(limpa)
()
↦(eixos '(10 10) '(10 10) '(0 28) '(0 18))
()
↦(impar-cima-par-baixo-graf 31 '(10 10) '(2 .02))
(31 94 47 142 71 214 107 322 161 484 242 121 364 182 91 274 137 412 206
103 310 155 466 233 700 350 175 526 ... 53 160 80 40 20 10 5 16 8 4 2 1)
```

Para a quantidade e gama de números gerados para o exemplo relativo a 31, escolheu-se uma *escala* com factores pequenos em *x (2)* e muito pequenos em *y (0.02)*, o que diminuiu o significado visual do gráfico obtido. No entanto, este gráfico permite-nos ver a forma como evoluem os números gerados.



Resolução:

No procedimento *eixos*, são definidos localmente outros procedimentos para desenhar os eixos dos *xx* e dos *yy* e as divisões respectivas. Pode ver-se, através do corpo principal do procedimento, que o desenho do eixo *xx* e das suas divisões iniciam-se no ponto mais à esquerda deste eixo. Por seu turno, o eixo dos *yy* e respectivas divisões tem origem na parte inferior deste eixo.
As divisões são representadas por segmentos verticais para o eixo dos *xx* e horizontais para o eixo dos *yy*, em ambos os casos de comprimento 2.

```
(define eixos
  (lambda (origem escala eixo-xx eixo-yy)
    (let ((ori-x (car origem))   ; coordenadas da origem dos eixos
          (ori-y (cadr origem))
          ;
          (escala-x (car escala)) ; número de unidades do ecrã por
          (escala-y (cadr escala)) ; divisão dos eixos
          ;                              eixo dos xx
          (n-divisoes-x<0 (car eixo-xx))   ; número divisões na parte negativa
          (n-divisoes-x>0 (cadr eixo-xx))  ; número divisões na parte positiva
                    ;                 eixo dos yy
          (n-divisoes-y<0 (car eixo-yy))   ; número divisões na parte negativa
          (n-divisoes-y>0 (cadr eixo-yy))) ; número divisões na parte positiva

      (letrec ((eixo-x
                 (lambda ()
                   (desenha-rel (list (list 0 0)
                                      (list (* (+ n-divisoes-x<0
                                                  n-divisoes-x>0)
                                               escala-x)
```

```
                                        0)))))
                (eixo-y
                  (lambda ()
                    (desenha-rel (list (list 0 0)
                                       (list 0 (* (+ n-divisoes-y<0
                                                     n-divisoes-y>0)
                                                  escala-y))))))
                (divisoes-x
                  (lambda (num)
                    (if (not (zero? num))
                        (begin
                          (desenha-rel (list (list 0 1)
                                             (list 0 -2)
                                             (list 0 1)))
                          (move-rel (list escala-x 0))
                          (divisoes-x (sub1 num))))))
                (divisoes-y
                  (lambda (num)
                    (if (not (zero? num))
                        (begin
                          (desenha-rel (list (list -1 0)
                                             (list 2 0)
                                             (list -1 0)))
                          (move-rel (list 0 escala-y))
                          (divisoes-y (sub1 num)))))))

          ;          o corpo principal do procedimento eixos
          (move (list (- ori-x
                         (* escala-x n-divisoes-x<0))
                      ori-y))
          (eixo-x)
          (move (list (- ori-x
                         (* escala-x n-divisoes-x<0))
                      ori-y))
          (divisoes-x (+ n-divisoes-x<0 n-divisoes-x>0
                         1))
          (move (list ori-x
                      (- ori-y
                         (* escala-y n-divisoes-y<0))))
          (eixo-y)
          (move (list ori-x
                      (- ori-y
                         (* escala-y n-divisoes-y<0))))
          (divisoes-y (+ n-divisoes-y<0 n-divisoes-y>0 1))))))
```

As tarefas associadas a *impar-cima-par-baixo-graf* já foram referidas e são claramente visíveis no corpo deste procedimento.

```
(define impar-cima-par-baixo-graf
  (lambda (numero origem escala)
    (let ((lista-dos-numeros
           (lista-impar-cima-par-baixo numero)))
      (desenha
        (prepara-lista-para-visualizar
          lista-dos-numeros origem escala))
      lista-dos-numeros)))
```

O procedimento *lista-impar-cima-par-baixo* toma um valor inteiro positivo e devolve a lista com a sequência de números gerados a partir desse valor.

```
(define lista-impar-cima-par-baixo
  (lambda (num)
    (cond ((= num 1) '(1))
          ((even? num)
           (cons num
                 (lista-impar-cima-par-baixo (quotient num 2))))
          (else
            (cons num
                  (lista-impar-cima-par-baixo (add1 (* num 3))))))))
```

A preparação de uma lista de pontos 2D, pronta para ser visualizada, a partir da lista dos números gerados, é a tarefa principal do procedimento *prepara-lista-para-visualizar*. Assim, por exemplo, se o primeiro número da lista é *numero1*, o primeiro elemento da lista a visualizar deverá ser a sublista *(ori-x + 0 * escala-x ori-y + numero1 * escala-y)* e se o segundo é *numero2*, o segundo elemento da lista a visualizar deverá ser a sublista *(ori-x + 1 * escala-x ori-y + numero2 * escala-y)*, e assim sucessivamente.

```
(define prepara-lista-para-visualizar
  (lambda (lis origem escala)
    (let ((ori-x (car origem))      ; coordenadas da origem dos eixos
          (ori-y (cadr origem))
          (escala-x (car escala))   ; factores de escala
          (escala-y (cadr escala)))
      (letrec ((aux
                (lambda (lis num)
                  (if (null? lis)
                      '()
                      (cons (list (+ ori-x              ; coordenada x
                                     (* num escala-x))
                                  (+ ori-y              ; coordenada y
                                     (* (car lis) escala-y)))

                            (aux (cdr lis) (add1 num)))))))
        ;
        (aux lis 0)))))
```

Exercícios e exemplos de final de capítulo

Nesta Secção, para além de alguns exercícios para consolidação e assimilação da matéria dada neste capítulo, são ainda apresentados outros que abordam o processamento de *listas em profundidade*.

Exercício 3.17 - Abstracção ponto-2D

Pretende-se criar uma abstracção de dados relacionada com entidades do tipo *ponto-2D*, compostas por dois inteiros, que representam as coordenadas de pontos num plano.

Escrever em *Scheme* os procedimentos que são apresentados nas próximas quatro alíneas.

1- O construtor *faz-ponto* recebe dois inteiros e devolve uma entidade do tipo *ponto-2D*. Os selectores *x-coord* e *y-coord* recebem um ponto-2D e devolvem, respectivamente, a coordenada *x* e *y* desse ponto.

2- Um rectângulo, com os lados paralelos aos eixos de coordenadas, pode ser definido por dois vértices opostos. O procedimento *area-rectangulo* recebe duas entidades *ponto-2D*, supõe que representam dois vértices opostos de um rectângulo e devolve a área desse rectângulo.

3- O procedimento *distancia* recebe duas entidades *ponto-2D* e devolve a distância entre elas.

4- O procedimento *area-de-interseccao* recebe quatro entidades *ponto-2D*, supõe que estas representam 2 rectângulos, e devolve a área de intersecção desses 2 rectângulos.
5- O procedimento *area-de-interseccao-graf* tem os mesmos parâmetros que o procedimento *area-de-interseccao*, devolve o mesmo resultado, mas, para além disto, desenha na janela gráfica corrente os dois rectângulos numa cor e pinta a intersecção noutra cor.

Exercício 3.18 - Substituição de elementos de listas
Escrever em *Scheme* os procedimentos que são apresentados nas próximas duas alíneas.
1- O procedimento *substitui-primeira-ocorrencia* tem 3 parâmetros, *lista*, *novo*, e *velho*, e devolve uma lista equivalente a *lista* depois de substituir a primeira ocorrência de *velho* por *novo*.

```
↦(substitui-primeira-ocorrencia '(o meu cao e' esperto) 'gato 'cao)
(o meu gato e' esperto)
↦(substitui-primeira-ocorrencia '() 'gato 'cao)
()
```

2- O procedimento *substitui-todas-ocorrencias* tem 3 parâmetros, *lista*, *novo*, e *velho*, e devolve uma lista equivalente a *lista* depois de substituir todas as ocorrências de *velho* por *novo*.

```
↦(substitui-todas-ocorrencias '(o meu cao e' esperto) 'gato 'cao)
(o meu gato e' esperto)
↦(substitui-todas-ocorrencias '() 'gato 'cao)
()
```

Exercício 3.19 - Eliminação de elementos de listas
Escrever em *Scheme* os procedimentos que são apresentados nas próximas três alíneas.
1- O procedimento *elimina-primeira-ocorrencia* tem 2 parâmetros, *lista*, e *elemento*, e devolve uma lista equivalente a *lista* depois de eliminar a primeira ocorrência de *elemento*.

```
↦(elimina-primeira-ocorrencia '(o meu cao e' esperto) 'cao)
(o meu e' esperto)
↦(elimina-primeira-ocorrencia '() 'cao)
()
```

2- O procedimento *elimina-todas-ocorrencias* tem 2 parâmetros, *lista*, e *elemento*, e devolve uma lista equivalente a *lista* depois de eliminar todas as ocorrências de *elemento*.
3- O procedimento *elimina-ultima-ocorrencia* tem 2 parâmetros, *lista*, e *elemento*, e devolve uma lista equivalente a *lista* depois de eliminar a última ocorrência de *elemento*.

Exercício 3.20 - *Merge* de listas
Escrever em *Scheme* os procedimentos que são apresentados nas próximas duas alíneas.
1- O procedimento *merge* tem dois parâmetros, *crescente-1* e *crescente-2*, que são duas listas com elementos numéricas ordenados do menor para o maior. Sabe-se que *merge* devolve uma lista cujos elementos são os elementos das listas dadas, em ordem crescente.

```
↦(merge '(2 3 40) '(1 3 34))
(1 2 3 3 34 40)
↦(merge '()'(1 3 34))
(1 3 34)
↦(merge '(2 3 40) '())
(2 3 40)
```

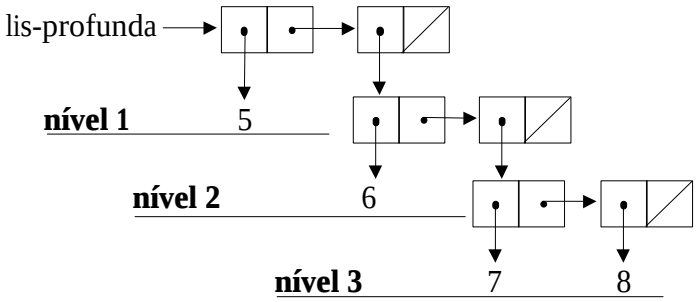
2- O procedimento *merge-sem-repetidos*, com os mesmos parâmetros de *merge*, não inclui elementos repetidos na lista que devolve.

```
↦(merge-sem-repetidos '(2 3 40) '(1 3 34))
(1 2 3 34 40)
↦(merge-sem-repetidos '()'(1 3 34))
```

```
(1 3 34)
↦(merge-sem-repetidos '(2 3 40) '())
(2 3 40)
```

Processamento de listas em profundidade

Em primeiro lugar convém saber em que *nível de profundidade* se encontra um elemento de uma lista. Segue-se um exemplo, *lis-profunda*, `(5 (6 (7 8)))`, que apresenta 3 níveis de profundidade.



No *nível 1*, também designado por *nível de topo*, a lista tem 2 elementos: `5` e `(6 (7 8))`. No *nível 2*, continua a ter 2 elementos: `6` e `(7 8)`. Finalmente, no *nível 3*, os elementos da lista são `7` e `8`.

Outros exemplos:

⇒ `50` 50 está no *nível zero*

⇒ `(50 51 52)` Qualquer um dos elementos da lista está no *nível 1*

⇒ `'((a b) c () ((d (e))))` No *nível de topo*, a lista tem 4 elementos.

No exemplo da figura, a lista tem, no *nível de topo*, apenas um elemento não par[18]: *5*. Todavia, se se considerar a lista em todos os seus níveis, ou seja, se se considerar a lista em profundidade, encontram-se 4 elementos não pares: *5*, *6*, *7* e *8*.

Processar uma *lista em profundidade* significa que, sempre que se encontre um elemento que seja par, então também ele deverá ser tratado em profundidade. A lista *`((a b) c () ((d (e))))`* tem 2 elementos não pares no nível de topo, mas quando considerada em profundidade, encontram-se 6 elementos não pares: *a*, *b*, *c*, *()*, *d* e *e*.

Exercício 3.21

O procedimento *aprofunda-1* tem um parâmetro, *lista*, e *afunda um nível* cada um dos seus elemento. Escrever em *Scheme aprofunda-1* que devolve uma lista equivalente a *lista* depois de a processar, o que corresponde, neste caso, a colocar um parêntesis à volta de cada elemento de *lista*.

```
↦(aprofunda-1 '(a b c))
((a) (b) (c))
```

Exemplo 3.11

Escrever em *Scheme* o procedimento *conta-todos* que tem um parâmetro, *lista*, e devolve o número de elementos que não são pares, contados em todos os níveis de *lista*.

```
↦(conta-todos '((a b) c () ((d (e)))))
6
↦(conta-todos '(() () ()))
3
↦(conta-todos '())
0
```

[18] Uma lista vazia não é par. O *Scheme* disponibiliza o predicado *pair?* que indica se o respectivo argumento é ou não um par (Anexo A).

Resolução
A redução da lista, para além do termo já amplamente utilizado *(procedimento (cdr lista))*, que sugere uma redução ao nível de topo, aparece ainda um termo, quando o primeiro elemento da lista é par, e que representa o padrão de computação típico do processamento em profundidade, ou seja, *(procedimento (cdr lista))* e *(procedimento (car lista))*.
Entende-se que se se pretende contabilizar os elementos que não são pares em todos os níveis de uma lista, encontrando um elemento par, então o passo de recursividade deverá ser

```
(define conta-todos
  (lambda (lis)
    (cond ((null? lis)         ; caso base
           0)

          ;    caso geral em que o elemento é um par... processamento em profundidade
          ((pair? (car lis))
           (+ (conta-todos (cdr lis))
              (conta-todos (car lis))))

          ;    caso geral em que o elemento não é um par...
          (else
            (add1 (conta-todos (cdr lis)))))))
```

Exercício 3.22

Escrever em *Scheme* os procedimentos que são apresentados nas próximas quatro alíneas.
1- O procedimento *soma-todos* tem um único parâmetro, *lista*, e devolve a soma de todos os seus elementos, procurando-os em todos os níveis.

```
↦(soma-todos '((8 5) 12 () ((-8 (-10)))))
7
↦(soma-todos '(() (71) ()))
71
↦(conta-todos '())
0
```

2- Antes de escrever o procedimento *inverter-posicao-de-todos*, analisar as duas chamadas que se seguem.

```
↦(reverse '(a (b c) (d (e f))))[19]
((d (e f)) (b c) a)
↦(inverter-posicao-de-todos '(a (b c) (d (e f))))
(((f e) d) (c b) a)
```

3- Antes de escrever o procedimento *substituir-todos*, analisar as duas chamadas que se seguem.

```
↦(substitui-todos '(a (b (a c)) (a (d a))) 'z 'a)
(z (b (z c) (z (d z)))
↦(substitui-todos '(((1) (0)) 0 '(1)))
((0 (0)))
↦(substitui-todos '() 'cao 'gato)
()
```

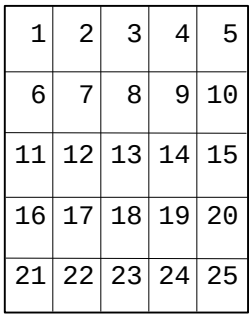
4- O procedimento *planificador*, com o parâmetro *lista*, devolve uma lista formada por todos os átomos[20] de *lista*.

```
↦(planificador '(a (b c d) ((e f) g)))
(a b c d e f g)
```

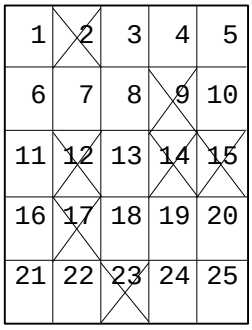
[19] A primitiva *reverse* funciona apenas sobre os elementos do nível de topo
[20] Um átomo é um número, símbolo, booleano, caracter, ou cadeia de caracteres e é identificado pelo predicado *atom?* (Anexo A).

```
↦(planificador '(a (b c d) (a a (g h))))
(a b c d a a g h)
↦(planificador '())
()
```

Exemplo 3.12



Vamos retomar o projecto apresentado no capítulo anterior, relativo ao tabuleiro com 25 células, organizadas numa matriz 5 x 5. Relembrando o problema: Numa das células do tabuleiro é colocado uma tartaruga que apenas se desloca na horizontal para a célula imediatamente ao lado *(nosso lado direito)* ou na vertical para a célula imediatamente abaixo.
Chegando à coluna do *(nosso)* lado direito, a tartaruga não pode deslocar-se mais na horizontal. Também, quando atinge a linha do fundo, a tartaruga não pode deslocar-se mais na vertical.
O objectivo da tartaruga é atingir a célula 25.



O caminho pode complicar-se, pois é possível colocar obstáculos intransponíveis em várias células. Por exemplo, colocando obstáculos nas células 2, 9, 12, 14, 15, 17 e 23, o tabuleiro toma o aspecto indicado na figura ao lado.

A tartaruga segue um caminho de acordo com uma estratégia muito simples: Desloca-se prioritariamente na horizontal e, só quando fica bloqueada neste movimento (ou encontrou um obstáculo ou alcançou a coluna da *(nossa)* direita) é que tenta na vertical.

O procedimento *caminho-versao2* espera dois argumentos: o ponto de partida, onde se coloca a tartaruga, e uma lista que representa os obstáculos colocados no tabuleiro[21].

```
↦(define obs '(2 9 12 14 15 17 23))
obs

↦(caminho-texto 1 obs)
1; 6; 7; 8; 13; 18; 19; 20; 25; consegui
↦(caminho-texto 11 obs)
11; 16; 21; 22; falhei
↦(caminho-texto 2 obs)
partida errada!!!
↦(caminho-texto 3 obs)
3; 4; 5; 10; falhei
```

Na solução que se apresenta, para *caminho-versao2*, é de realçar que este procedimento começa por verificar se há uma "falsa partida" e, se não for esse o caso, chama o procedimento *andar*, que é o núcleo central da resolução do problema. O procedimento *andar* visualiza o ponto de partida e, seguidamente, verifica se já atingiu a célula 25, que corresponde ao caso base. Se não, verifica se é possível deslocar-se na horizontal, em caso de impossibilidade, verifica se pode deslocar-se na vertical, e, finalmente, em caso de nova impossibilidade, conclui que está

[21] No capítulo anterior, o tabuleiro foi modelado através de 25 variáveis independentes. Neste caso, optou-se por modelar, através de uma lista, com a identificação das células com obstáculos.

bloqueado e que falhou a trajectória. Se nalgum dos casos determinou que pode deslocar-se, então chama-se recursivamente, actualizando o ponto de partida.

```
(define caminho-versao2
  (lambda (partida obstaculos)
    (if (e'-obstaculo? partida obstaculos)      ; partida é obstáculo?
        (display "partida errada!!!")            ; sim
        (andar partida obstaculos))))            ; não

(define andar
  (lambda (partida obstaculos)
    (display partida)                          ; visualiza partida
    (display "; ")
    (cond                                      ; verifica se...
      ((= partida 25) (display "consegui"))    ; já atingiu a célula 25
      ((avanca-hor? partida obstaculos)        ; pode avançar na horizontal
       (andar (+ partida 1) obstaculos))
      ((avanca-ver? partida obstaculos)        ; pode avançar na vertical
       (andar (+ partida 5) obstaculos))
      (else (display "falhei")))))       ; a hipótese de bloqueio...

(define avanca-hor?
  (lambda (partida obstaculos)
    (and
      (not (= (remainder partida 5) 0))    ; não está na 5a coluna
      (not (e'-obstaculo? (+ partida 1)
                          obstaculos)))))         ; e na horizontal não
                                                  ; encontra obstáculo

(define avanca-ver?
  (lambda (partida obstaculos)
    (and (< partida 21)                       ; não está na última linha
         (not (e'-obstaculo? (+ partida 5)
                             obstaculos)))))   ; e na vertical não
                                               ; encontra obstáculo

(define e'-obstaculo?
  (lambda (elem lista)    ; se elem faz parte da lista, é devolvida uma
    (member elem lista))) ; sublista de lista que é entendida como #t
                          ; se não fizer parte, é devolvido '(), entendida como #f
```

Temos aqui um exemplo em que a tarefa a tratar é relativamente complexa e exigiu uma abordagem *de-cima-para-baixo*, enquanto que a *abstracção de dados* passou quase despercebida. Os obstáculos foram representados através de uma lista perfeitamete normal e o procedimento *e'-obstaculo?* funcionou como um selector, não tendo sido necessários construtores ou modificadores. Quanto à tartaruga, um inteiro chegou para representar a sua posição no tabuleiro e também não requereu a escrita de construtores ou modificadores.

Exercício 3.23

No contexto do exemplo anterior, introduzir as alterações necessárias à solução apresentada, para chegar ao procedimento *caminho-no-tabuleiro*, que passa a responder da seguinte maneira:

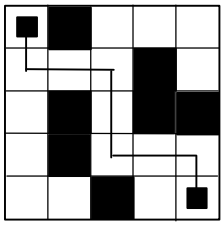
```
↦(caminho-no-tabuleiro 1 obs)

  1  x  .  .  .
  6  7  8  x  .
  .  x 13  x  x
  .  x 18 19 20
  .  .  x  . 25
```

```
↦(caminho-no-tabuleiro 2 obs)
partida errada!!!
↦(caminho-no-tabuleiro 3 obs)

  .  x  3  4  5
  .  .  .  x 10
  .  x  .  x  x
  .  x  .  .  .
  .  .  x  .  .
```

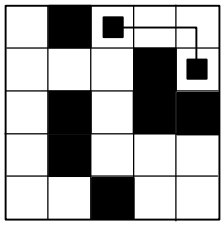
Exercício 3.24

Ainda no contexto do exemplo anterior, introduzir as alterações necessárias à solução apresentada, para chegar ao procedimento *caminho-no-tabuleiro-graf*, que, utilizando a janela corrente, passa a responder da seguinte maneira:

```
↦(caminho-no-tabuleiro 1 obs)
consegui
```



```
↦(caminho-no-tabuleiro 2 obs)
partida errada!!!
```



```
↦(caminho-no-tabuleiro 3 obs)
falhei
```

Exemplo 3.13

Um programa designado por *adivinha-numero* não tem parâmetros, escolhe aleatoriamente um número entre 0 e 31 e convida o utilizador a adivinhar esse número, permitindo-lhe, previamente, fazer 4 enumerações, as quais serão "comentadas" pelo programa. Vejamos um exemplo de utilização e o diálogo estabelecido.

```
↦(adivinha-numero)

Estou a escolher um numero entre 0 e 31
... e tu vais tentar adivinha'-lo com 4 sugestoes...

Indicar lista de numeros entre 0 e 31:
(1 3 5 6 7 10 12 14 16 23 31 24)
O numero ESTA' nos indicados
Indicar lista de numeros entre 0 e 31:
(1 3 5 6 7 10)
O numero NAO ESTA' nos indicados
Indicar lista de numeros entre 0 e 31:
(12 14 16)
O numero NAO ESTA' nos indicados
Indicar lista de numeros entre 0 e 31:
(23 31)
O numero ESTA' nos indicados
A tua vez de adivinhar o numero escolhido: 23
```

```
Falhaste. O numero certo seria: 31

↦(adivinha-numero)

Estou a escolher um numero entre 0 e 31
... e tu vais tentar adivinha'-lo com 4 sugestoes...

Indicar lista de numeros entre 0 e 31:
(1 2 3 4 5 6 7 8 9 10)
O numero ESTA' nos indicados
Indicar lista de numeros entre 0 e 31:
(1 2 3 4 5)
O numero NAO ESTA' nos indicados
Indicar lista de numeros entre 0 e 31:
(6 7 8)
O numero NAO ESTA' nos indicados
Indicar lista de numeros entre 0 e 31:
(9)
O numero NAO ESTA' nos indicados
A tua vez de adivinhar o numero escolhido:
10Parabens... Acertaste.
```

Neste exemplo, considerou-se não ser necessário dispender grande esforço com a definição de uma abstracção de dados específica, pois manifestou-se perfeitamente razoável a utilização de uma lista de números e alguns procedimentos do *Scheme* para manipulação de listas. Assim, de acordo com uma abordagem *de-cima-para*-baixo, identificam-se as seguintes tarefas prinipais do programa *adivinha-numero*:

⇒ seleccionar um número aleatório entre 0 e 31;

⇒ pedir ao utilizador listas de números entre 0 e 31 e comentar as respectivas respostas;

⇒ convidar o utilizador a adivinhar o número seleccionado aleatoriamente;

⇒ apresentar a mensagem de parabéns ou de lamentação por não ter acertado no número.

A segunda tarefa, devido ao seu carácter repetitivo, justificou o procedimento auxiliar *faz-perguntas*, procedimento recursivo, definido localmente, em que o caso base corresponde ao contador *vezes* atingir o valor zero. As tarefas de *faz-perguntas* são:

⇒ visualizar mensagem pedindo a indicação de uma lista de números entre 0 e 31;

⇒ verificar se o número seleccionado aleatoriamente se encontra ou não na lista indicada e informar o utilizador desse facto;

⇒ repetir as tarefas anteriores até atingir o caso base.

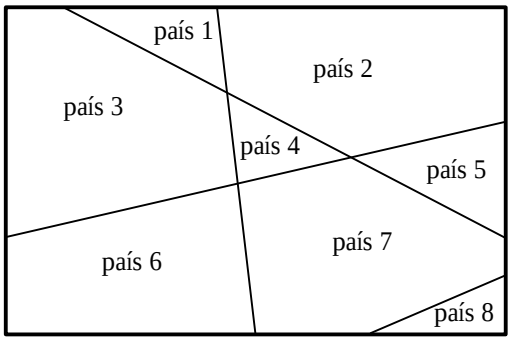
```
(define adivinha-numero
  (lambda ()
    (letrec
      ((faz-perguntas
         (lambda (vezes num-selec)
           (if (positive? vezes)
               (begin
                 (display
                   "Indicar lista de numeros entre 0 e 31:")
                 (newline)
                 (if (member num-selec (read))
                     (display
                       "O numero ESTA' nos indicados")
                     (display
                       "O numero NAO ESTA' nos indicados"))
                 (newline)
```

```
                    (faz-perguntas (sub1 vezes) num-selec))))))

        ;    o corpo principal do programa adivinha-numero
        (newline)
        (newline)
        (display "Estou a escolher um numero entre 0 e 31")
        (newline)
        (display "... e tu vais tentar adivinha'-lo com 4 sugestoes...")
        (newline)
        (newline)
        (let ((num-sel (sub1 (roleta 32))))
          (faz-perguntas 4 num-sel)
          (display "A tua vez de adivinhar o numero escolhido: ")
          (newline)
          (if (= num-sel (read))
              (begin
                (newline)
                (display "Parabens... Acertaste."))
              (begin
                (display "Falhaste. O numero certo seria: ")
                (display num-sel)))
          (newline)))))

(define roleta
  (lambda (num)
    (add1 (remainder (random) num))))
```

Projecto 3.1 - Colorir mapas

Pretende-se colorir mapas de países, com um número mínimo de cores e sem que cores iguais sejam utilizadas em países vizinhos. São considerados países vizinhos os que tiverem, pelo menos, uma linha de fronteira comum *(caso dos países 1 e 2)* ou até um ponto de fronteira comum *(caso dos países 2 e 3)*. O mapa é modelado através de uma lista, tendo um elemento por cada país do mapa. O elemento correspondente a um país representa os países vizinhos desse país. Por exemplo, o *país 1*, sendo o primeiro elemento da lista, tem como vizinhos os países 2, 3 e 4.

```
(define mapa1
  '((2 3 4)        ; país 1
    (1 3 4 5)      ; país 2
    (1 6 7 2 4)    ; país 3
    (1 7 6 2 3 5)  ; país 4
    (7 2 4)        ; país 5
    (3 4 7)        ; país 6
    (3 4 5 6 8)    ; país 7
    (7)))          ; país 8
```



A modelação das cores pode também ser feita através de uma lista.

```
(define paleta-cores
  '(c1 c2 c3 c4))          ; neste caso, utilizam-se 4 cores
```

A atribuição de cores aos vários países é conseguida com uma chamada a *colorir-mapa*, procedimento que espera dois argumentos, ou seja, duas listas, uma que representa o mapa e outra o conjunto de cores utilizadas para colorir o mapa. Este procedimento devolve uma lista com a indicação das cores a atribuir a cada país, a começar pelo *país 1*.

```
↦(colorir-mapa mapa1 paleta-cores)
(c1 c2 c3 c4 c1 c1 c2 c1)
```

Uma tentativa de colorir o mapa com um conjunto de apenas três cores:

```
↦(colorir-mapa mapa1 '(c1 c2 c3))
nao ha' solucao...
```

1- Fazer uma abordagem *de-cima-para-baixo* ao programa *colorir-mapa*.
2- Escrever em *Scheme* o procedimento *colorir-mapa*, com base nos resultados da abordagem anterior.

Projecto 3.2 - Jogo das minas

O jogo das minas baseia-se no mapa de um terreno, onde existem 64 minas. No terreno há minas em variadíssimas situações, desde as muito ricas, até às muito falidas ou endividadas. Esta situação é representada por um inteiro que se situa entre +99 (muito ricas) e -99 (muito endividadas). O mapa, por seu turno, é representado visualmente por uma matriz 8 x 8, em que cada célula representa a situação de uma mina. Este mapa é determinado aleatoriamente pelo computador.

Durante *n* percursos, um mineiro visita *n* minas, e vai somando ou perdendo um valor equivalente ao inteiro que representa a situação da mina visitada. Se visitar uma mina com a situação 80, ganhará 80, todavia, se a mina visitada apresentar uma situação -50, perderá 50. Se visitar duas ou mais vezes a mesma mina, o valor que ganha ou perde será sempre o mesmo.

O mineiro/jogador observa a sua localização no mapa e faz uma das *n* visitas/jogadas escolhendo a orientação da próxima viagem (Norte/Sul/Este/Oeste) e o computador escolhe aleatoriamente a distância, ou então, escolhe a distância e o computador escolhe aleatoriamente a orientação.

Analisar parte de uma sessão que seria composta por 10 visitas/jogadas, de acordo com o primeiro argumento do programa *minas*. O segundo e terceiro argumentos representam, respectivamente, a coluna e linha em que o mineiro se encontra no início da sessão.

```
↦(minas 10 5 3)
------------------------------
Numero restante de jogadas: 10

Terreno de minas:
Col.  1   2   3   4   5   6   7   8
Lin.
 1   45  34 -55   2 -33  80 -78   4
 2   37   5  34 -45   6 -67 -28  -7
 3   -6 -54   2  34  99  90  78 -89
 4   23  45 -78  95  -4  35 -67  49       N
 5    1  45   2  -7  67  44 -90  34    O -|- E
 6   80 -87  45 -38  58  52 -95 -73       S
 7   77  91 -36  62  -5   9 -23  74
 8  -35  74  95 -82   5  -7   9  44

Localizacao do mineiro: col. 5, lin. 3
Situacao do mineiro: 99

Indicar pista para a proxima mina
N/S/E/O/numero? E
Distancia calculada pelo computador: 2

------------------------------
Numero restante de jogadas: 9
```

```
Terreno de minas:
Col.  1   2   3   4   5   6   7   8
Lin.
 1   45  34 -55   2 -33  80 -78   4
 2   37   5  34 -45   6 -67 -28  -7
 3   -6 -54   2  34  99  90  78 -89
 4   23  45 -78  95  -4  35 -67  49       N
 5    1  45   2  -7  67  44 -90  34    O -|- E
 6   80 -87  45 -38  58  52 -95 -73       S
 7   77  91 -36  62  -5   9 -23  74
 8  -35  74  95 -82   5  -7   9  44

Localizacao do mineiro: col. 7, lin. 3
Situacao do mineiro: 177

Indicar pista para a proxima mina
N/S/E/O/numero? 1
Orientacao calculada pelo computador: N

------------------------------
Numero restante de jogadas: 8

Terreno de minas:
Col.  1   2   3   4   5   6   7   8
Lin.
 1   45  34 -55   2 -33  80 -78   4
 2   37   5  34 -45   6 -67 -28  -7
 3   -6 -54   2  34  99  90  78 -89
 4   23  45 -78  95  -4  35 -67  49       N
 5    1  45   2  -7  67  44 -90  34    O -|- E
 6   80 -87  45 -38  58  52 -95 -73       S
 7   77  91 -36  62  -5   9 -23  74
 8  -35  74  95 -82   5  -7   9  44

Localizacao do mineiro: col. 7, lin. 2
Situacao do mineiro: 110

Indicar pista para a proxima mina
N/S/E/O/numero? N
Distancia calculada pelo computador: 6  ; Quando "bate" numa parede, reflete
                                        ; para trás com a distância restante...
------------------------------
Numero restante de jogadas: 7

Terreno de minas:
Col.  1   2   3   4   5   6   7   8
Lin.
 1   45  34 -55   2 -33  80 -78   4
 2   37   5  34 -45   6 -67 -28  -7
 3   -6 -54   2  34  99  90  78 -89
 4   23  45 -78  95  -4  35 -67  49       N
 5    1  45   2  -7  67  44 -90  34    O -|- E
 6   80 -87  45 -38  58  52 -95 -73       S
 7   77  91 -36  62  -5   9 -23  74
 8  -35  74  95 -82   5  -7   9  44

Localizacao do mineiro: col. 7, lin. 6
Situacao do mineiro: 15

Indicar pista para a proxima mina
```

```
N/S/E/O/numero?
...
```

1- Definir e implementar uma abstracção de dados que se adeqúe ao problema exposto.
2- Fazer uma abordagem *de-cima-para-baixo* ao programa *minas*.
3- Escrever em *Scheme* o procedimento *minas*, com base nos resultados da abordagem anterior.

Projecto 3.3 - Lançamento de projécteis

A simulação do lançamento de projécteis retoma um exercício do capítulo 1, mas agora num ambiente de interacção com saída gráfica. A simulação do lançamento de projécteis inicia-se chamando o procedimento *projectil*, em que os dois primeiros argumentos constituem, respectivamente, a largura e a altura da janela onde se visualizará o movimento do projéctil. O terceiro argumento, uma cadeia de caracteres, corresponde ao título da janela.
Uma janela gráfica é imediatamente criada, seguindo-se uma pergunta sobre se se pretende ou não lançar um projéctil. À resposta positiva corresponderá, na janela criada, a visualização de um objecto-alvo numa posição determinada aleatoriamente. O utilizador do simulador é então interrogado sobre os valores iniciais do ângulo e velocidade de um projéctil, colocado no canto inferior esquerdo da janela, para tentar alcançar o objecto-alvo. A trajectória do projéctil começa a ser visualizada na janela até atingir o objecto-alvo ou então até atingir o solo, sem lhe acertar. Para melhor se entender a especificação do simulador, analisar o diálogo e as figuras que se seguem.

```
↦(projectil 150 150 "Antonio Silva")

pretende lançar projectil (n = não; outro- sim):
s
Angulo de partida (0 < ang <= 90):
60
Velocidade inicial (> 0)):
70
Boa-pontaria...

pretende lançar projectil (n = não; outro- sim):
s
Angulo de partida (0 < ang <= 90):
50
Velocidade inicial (> 0)):
60

Ma'-pontaria...
pretende lançar projectil (n = não; outro- sim):
n
A simulação vai terminar
```

1- Desenvolver em *Scheme* o simulador descrito, tendo em conta que as suas principais tarefas poderão ser:

⇒ criar uma janela gráfica de dimensão L x A *(L e A são argumentos)*;
⇒ perguntar se quer lançar o projéctil - Se não quer, termina o programa;
⇒ limpar a janela gráfica corrente;
⇒ visualizar o objecto-alvo, num ponto da janela determinado aleatoriamente;
⇒ pedir o ângulo (0 < ang <90) e a velocidade iniciais (> 0);

⇒ lançar e visualizar o projéctil a partir do canto inferior-esquerdo e, ao mesmo tempo, verificar se o objecto-alvo é atingido. Esta tarefa termina se o alvo é atingido ou se o projéctil cai no solo sem o atingir;

⇒ visualizar a mensagem de acordo com o facto de se ter atingido ou não o objecto-alvo;

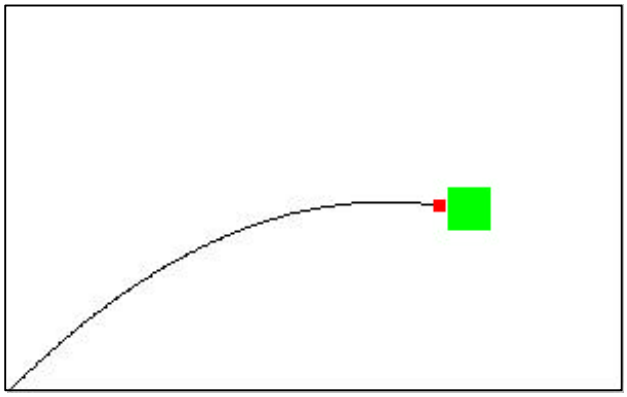
⇒ retomar o ponto em que pergunta se quer ou não lançar o projéctil.

2- Melhorar o simulador apresentado, introduzindo as seguintes alterações:

⇒ A trajectória termina se o projéctil ultrapassar o lado direito da janela;

⇒ A deslocação do projéctil não se faz por saltos, mas de forma contínua, como se indica na figura. Assim, repetidamente:

- O projéctil é visualizado numa nova posição e, passado um curto intervalo de tempo, é apagado;
- Na nova posição, é visualizado mais um ponto que define o rasto do projéctil;
- O projéctil é visualizado numa nova posição da trajectória, muitíssimo perto da posição anterior;



Pista: Visualizar o projéctil com a cor *'temp* (cor temporária- ver Anexo B)