Capítulo 7- Exercícios Finais

Conjunto de exercícios e projectos

O presente capítulo concretiza-se num conjunto de enunciados de exercícios e projectos e, apesar de alguns indicarem pequenas pistas de solução, tentou-se sempre colocar o leitor perante problemas de programação desligados de qualquer contexto ou capítulo tratado. Quando se coloca um exercício no final de um capítulo, uma primeira pista de resolução fica implicitamente estabelecida. Na vida real, os problemas de programação não aparecem rotulados ou associados a capítulos de qualquer livro. O problema surge, torna-se necessário encontrar uma ideia de resolução para ele, vaga no início, mas sucessivamente mais refinada. Só à medida que as várias tarefas e sub-tarefas do problema vão sendo identificadas, é que se clarificam as associações com os capítulos e matérias tratadas em cada um deles. Surge a forma de representar os dados do problema, acompanhada de um certo conjunto de construtores, selectores e até modificadores, ou seja, surge a *abstracção de dados*. E é sobre a abstracção idealizada e com as tarefas e sub-tarefas identificadas que se refina a construção da solução.

Exercício 1

O programa *matriz* estabelece um diálogo com o utilizador e, através desse diálogo, constrói uma matriz. Com a matriz construída o diálogo continua, permitindo escolher uma linha ou uma coluna e, seguidamente, somar ou visualizar os elementos da linha ou da coluna escolhida.

```
↦(matriz)
Numero de linhas: 2
Numero de colunas: 3
proximo elemento da matriz: 1                    ; matriz fornecida
proximo elemento da matriz: 2                    ; 1     2     3
proximo elemento da matriz: 3                    ; 4     5     6
proximo elemento da matriz: 4
proximo elemento da matriz: 5
proximo elemento da matriz: 6
O que deseja? linha/coluna: linha
Numero da linha/coluna: 2
Que pretende? somar/visualizar: somar
15
O que deseja? linha/coluna: coluna
Numero da linha/coluna: 2
Que pretende? somar/visualizar: visualizar
2  5
O que deseja? linha/coluna: coluna
Numero da linha/coluna: 3
Que pretende? somar/visualizar: somar
9
O que deseja? linha/coluna: linha
Numero da linha/coluna: 3
erro...
O que deseja? linha/coluna: linha
Numero da linha/coluna: 1
Que pretende? somar/visualizar: somaaaaaaaaar   ; o programa termina perante
O programa vai terminar...                       ; uma opção desconhecida
```

```
(define matriz
  (lambda ()
    (letrec ((n-lin (le-num 'linhas))  ; leitura nº de linhas
             (n-col (le-num 'colunas)) ; leitura nº de colunas
             (mat (le-matriz (* n-lin n-col)))) ; constrói lista com os
                                                ; elementos da matriz
      (processa-matriz mat n-lin n-col))))
```

1- Fazer uma abordagem *de-cima-para-baixo* ao problema exposto, identificar as suas tarefas e sub-tarefas principais e definir uma abstracção de dados adequada.
2- Escrever em *Scheme* a abstracção de dados definida.
3- Escrever em *Scheme* o programa *matriz*, apoiado na abstracção desenvolvida e nas tarefas e sub-tarefas identificadas.

Exercício 2

Numa língua muito estranha, as palavras são constituídas por letras de um abecedário, como acontece normalmente, mas, neste caso, não se admitem repetições de letras na mesma palavra! Assim, o comprimento das palavras poderá variar entre 1 e *n*, em que *n* representa o número de letras do abecedário. Por exemplo, com um abecedário constituído apenas pelas letras *a* e *b*, as palavras possíveis serão *a*, *b*, *ab* e *ba*, com comprimentos que variam entre 1 e 2. Escrever em *Scheme* o procedimento *comprimento-max-do-dicionario* com o parâmetro *n* e que devolve o número máximo de palavras do dicionário de uma língua com um abecedário de *n* letras.

```
↦(comprimento-max-do-dicionario 1)
1
↦(comprimento-max-do-dicionario 2)
4
```

Exercício 3

Supor que uma aplicação, que utiliza listas, tem necessidade de avaliar frequentemente o comprimento destas. Em vez de usar o procedimento primitivo *length (que se supõe que, ao ser chamado, conta todos os elementos da lista, um a um)*, vai-se tentar outra estratégia que obriga a modificar a forma de representar as listas. Considera-se que o 1º elemento da lista passa a representar o seu comprimento e que o 1º elemento efectivo da lista passa a ser o 2º, o 2º passa a ser o 3º e assim sucessivamente. Deste modo, será necessário escrever novos procedimentos para a manipulação destas *listas*, diferentes dos procedimentos primitivos do *Scheme*.

```
↦(define lis1 (my-null-list))
lis1
↦lis1
(0)
↦(my-car lis1)
Erro... lista vazia1
↦(define lis2 (my-cons 56 lis1))
lis2
↦lis2
(1 56)
↦(define lis3 (my-cons 7 lis2))
lis3
↦lis3
(2 7 56)
↦(my-cdr lis3)
(1 56)
```

```
↦(my-cdr (my-cdr lis3))
(0)
↦(my-length lis3)
2
↦(my-length (my-cdr lis3))
1
```

1- Escrever em Scheme os procedimentos *my-null-list*, *my-car*, *my-cdr*, *my-cons*, *my-length*.
2- Supor agora que se trabalha com listas mutáveis e que se pretende escrever em *Scheme* os procedimentos *my-cons!* e *my-cdr!*. O primeiro recebe como argumentos um elemento e uma lista. Incrementa de uma unidade o comprimento dessa lista e coloca o elemento dado no início dela. O segundo recebe como argumento uma lista. Diminui o comprimento dessa lista de uma unidade e elimina o seu primeiro elemento efectivo. Em ambos os casos a lista dada é alterada e não é criada uma lista nova. O efeito lateral, ou seja, a alteração da lista dada, é o principal objectivo do procedimento, pois o valor que devolve é irrelevante.

Exercício 4

Um programa designado por *teclado* suporta uma interacção como se indica. O programa começa por interrogar o utilizador sobre a operação aritmética que pretende executar *(so - somar; su - subtrair; mu - multiplicar; di - dividir)*. Caso o utilizador opte por uma operação fora deste elenco, o programa voltará a interrogá-lo sobre a operação a executar. Depois de identificada a operação, o programa interroga o utilizador sobre os números que pretende operar. A série de números a operar deve terminar com zero.

```
↦(teclado)
operacao (so su mu di): so
numero (zero para terminar): 3
numero (zero para terminar): 4
numero (zero para terminar): 0
resultado: 7
Terminou o programa
↦(teclado)
operacao (so su mu di): di
numero (zero para terminar): 5
numero (zero para terminar): 1
numero (zero para terminar): 2
numero (zero para terminar): 0
resultado: 2.5
Terminou o programa
↦(teclado)
operacao (so su mu di): s
operacao (so su mu di): su
numero (zero para terminar): 4
numero (zero para terminar): 56
numero (zero para terminar): 6
numero (zero para terminar): 0
resultado: -58
Terminou o programa
```

1- Fazer uma abordagem *de-cima-para-baixo* ao problema exposto, identificar as suas tarefas e sub-tarefas principais e definir uma abstracção de dados adequada.
2- Escrever em *Scheme* a abstracção de dados definida.
3- Escrever em *Scheme* o programa *teclado*, apoiado na abstracção desenvolvida e nas tarefas e sub-tarefas identificadas.

Exercício 5

O programa *classificacoes* pede as classificações dos alunos de uma turma e, no final, imprime o gráfico das classificações positivas.

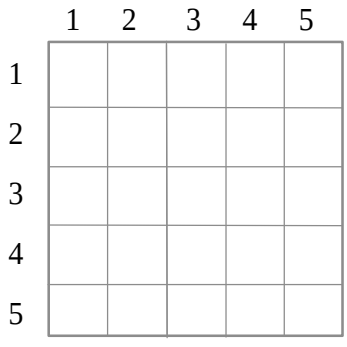
```
↦(classificacoes 'ip1)

classificacao= 15
classificacao= 12
classificacao= 16
classificacao= 11
classificacao= 13
classificacao= 14
classificacao= 12
classificacao= 16
classificacao= 14
classificacao= 13
classificacao= 17
classificacao= 10
classificacao= 7
classificacao= 18
classificacao= 15
classificacao= 14
classificacao= 8
classificacao= 13
classificacao= -3          ; um valor negativo indica que não há mais classificações


Classificacoes de ip1:
10  *
11  *
12  * *
13  * * *
14  * * *
15  * *
16  * *
17  *
18  *
19
20
```

1- Fazer uma abordagem *de-cima-para-baixo* ao problema exposto, identificar as suas tarefas e sub-tarefas principais e definir uma abstracção de dados adequada.
2- Escrever em *Scheme* a abstracção de dados definida.
3- Escrever em *Scheme* o programa *classificacoes*, apoiado na abstracção desenvolvida e nas tarefas e sub-tarefas identificadas.

Exercício 6



Imaginar um alvo constituído por uma matriz de 5 linhas por 5 colunas. Esse alvo é representado computacionalmente por uma lista de 5 sublistas; uma sublista de 5 elementos para cada linha do alvo.
Numa jogada é escolhida uma célula do alvo *(através da linha e coluna correspondentes)* e o número de pontos que se pretende somar aos existentes nessa célula.
O procedimento *jogar* permite escolher um alvo e o número de jogadas consecutivas. No final das jogadas, este procedimento visualiza o estado do alvo. É possível continuar a jogar com o mesmo alvo na situação em que se encontre ou então limpá-lo com *limpar-alvo* e recomeçar.

```
↦(define alvo-1 (criar-alvo))
alvo-1
↦alvo-1
((0 0 0 0 0) (0 0 0 0 0) (0 0 0 0 0) (0 0 0 0 0) (0 0 0 0 0))
↦(tiro-ao-alvo alvo-1 2)       ; jogar com alvo-1 em 2 jogadas
Jogada 2                  ; solicitação para a 1ª jogada
Linha: 3                  ; selecciona a linha 3
Coluna: 5                 ; e a coluna 5
Pontos: 3                 ; a jogada vale 3 pontos, a somar aos existentes
Jogada 1                  ; solicitação para a 2ª jogada
Linha: 2
Coluna: 4
Pontos: 6
(0 0 0 0 0)               ; situação actual de alvo-1
(0 0 0 6 0)
(0 0 0 0 3)
(0 0 0 0 0)
(0 0 0 0 0)
ok

↦(tiro-ao-alvo alvo-1 1)
Jogada 1
Linha: 3
Coluna: 8             ; coluna inexistente
Pontos: 5                ; então, ...
Jogada 1                 ;           repete jogada
Linha: 2
Coluna: 4
Pontos: 1
(0 0 0 0 0)
(0 0 0 7 0)
(0 0 0 0 3)
(0 0 0 0 0)
(0 0 0 0 0)
ok
```

1- Representar *alvo-1*, sob a forma *caixa-e-apontador*, no final do segundo conjunto de jogadas indicadas anteriormente.
2- Fazer uma abordagem *de-cima-para-baixo* ao problema exposto, identificar as suas tarefas e sub-tarefas principais e definir uma abstracção de dados adequada (algumas sugestões para a abstracção: *criar-alvo*, *limpar-alvo!*, *ler-jogada*, *actualizar-alvo!*, e *visualizar-alvo*).
3- Escrever em *Scheme* a abstracção de dados definida.
4- Escrever em *Scheme* o programa *tiro-ao-alvo*, apoiado na abstracção desenvolvida e nas tarefas e sub-tarefas identificadas.

Exercício 7

O procedimento *elem-divisor* toma dois argumentos, *num* e *vec*, respectivamente, um número inteiro e um vector, e procura o primeiro elemento de *vec* que divide *num*[1]. Para melhor se entender a funcionalidade de *elem-divisor*, analisar os exemplos que se seguem:

```
↦(define v1 (vector))
v1
↦v1
#()
↦(define v2 (vector 7 6 5 4 3 2))
```

[1] O elemento *elem* divide *num* se o resto desta divisão (inteira) for zero

```
v2
↦v2
#(7 6 5 4 3 2 1)
↦(elem-divisor 10 v1)
#f
↦(elem-divisor 10 v2)
5
↦(elem-divisor 13 v2)
#f
```

Escrever em *Scheme* o procedimento *elem-divisor*, integrando localmente os procedimentos auxiliares necessários.

Exercício 8

1- Escrever um programa em *Scheme*, com a designação *adivinha*, que adivinha um número entre 0 e 63, escolhido pelo utilizador, no menor número possível de tentativas. Vamos imaginar que o utilizador tinha escolhido o número 51. A interacção podia ser a seguinte:

```
↦(adivinha)
O meu palpite e': 31
Acertei? certo/alto/baixo: baixo        ; certo, alto, ou baixo será uma
O meu palpite e': 47                    ; das respostas a dar pelo utilizador
Acertei? certo/alto/baixo: baixo
O meu palpite e': 55
Acertei? certo/alto/baixo: alto
O meu palpite e': 51
Acertei? certo/alto/baixo: certo
adivinhei!!!...
```

2- Escrever em *Scheme* o programa *novo-adivinha* que espera dois argumentos para definir os limites inferior e superior, entre os quais o número a adivinhar se situa. O exemplo anterior seria lançado com a chamada:

```
↦(novo-adivinha 0 63)
O meu palpite e':...
```

Exercício 9

Analisar, nos exemplos que se seguem, o funcionamento do procedimento *actualizar*:

```
↦(actualizar 5 '((3 1) (6 2)))
((3 1) (5 1) (6 2))
↦(actualizar 3 '((3 1) (6 2)))
((3 2) (6 2))
↦(actualizar 3 '((3 2) (6 2)))
((3 3) (6 2))
↦(actualizar 5 '())
((5 1))
```

Escrever em *Scheme* o procedimento *actualizar*.

Exercício 10

Verificar, nos exemplos que se seguem, que o procedimento *soma* toma uma lista como argumento e devolve a soma dos elementos da lista que são números positivos:

```
↦(soma '(1 2 3))
6
```

```
↦(soma '(1 a 3 5))
9
↦(soma '(12.6 -25 hj))
12.6
```

O procedimento *maior-soma* toma um vector como argumento único, vector este cujos elementos são listas.
Este procedimento devolve a maior das somas associadas aos elementos do vector, calculadas de uma forma idêntica à indicada para o procedimento *soma*.

```
↦(define v1 '#((1 2 3) (2 aa 7) (3 ww e)))
v1
↦v1
#((1 2 3) (2 aa 7) (3 ww e))
↦(maior-soma v1)
9
```

Escrever em *Scheme* os procedimentos *soma* e *maior-soma*.

Exercício 11

O procedimento *matriz* toma um número não definido de argumentos, com os quais começa por definir uma matriz quadrada. Quando o número de argumentos não chega para uma matriz quadrada, a matriz é completada com zeros.
Exemplos de chamadas de *matriz*, com as matrizes constituídas em cada caso:

(matriz a1)

a1

(matriz a1 a2)

a1	a2
0	0

(matriz a1 a2 a3)

a1	a2
a3	0

(matriz a1 a2 a3 a4 a5 a6)

a1	a2	a3
a4	a5	a6
0	0	0

O procedimento *matriz* responde da seguinte forma, num exemplo em que são fornecidos 7 argumentos:

```
↦(matriz 1 2 3 4 5 6 7)

O que deseja? linha/coluna/fim: coluna
Número de coluna, 1..3: 1
1  4  7
O que deseja? linha/coluna/fim: linha
Número de linha, 1..3: 2
4  5  6
O que deseja? linha/coluna/fim: linha
Número de linha, 1..3: 3
7  0  0
O que deseja? linha/coluna/fim: coluna
Número de coluna, 1..3: 3
3  6  0
O que deseja? linha/coluna/fim: linha
Número de linha, 1..3: 4
erro...
O que deseja? linha/coluna/fim: fim
fim...
```

1- Fazer uma abordagem *de-cima-para-baixo* ao problema exposto, identificar as suas tarefas e sub-tarefas principais e definir uma abstracção de dados adequada.
2- Escrever em *Scheme* a abstracção de dados definida.

3- Escrever em *Scheme* o programa *matriz*, apoiado na abstracção desenvolvida e nas tarefas e sub-tarefas identificadas.

Exercício 12

O predicado *tres-juntos?* tem dois argumentos, *simb* e *lis*, sendo o primeiro um símbolo e o segundo uma lista. Este procedimento devolve *#t* quando encontra em *lis*, pelo menos, três símbolos seguidos e iguais a *simb*.

```
↦(define lista1 '(o o x - x o o o x x o - x))
  lista1
↦(tres-juntos? 'x lista1)
  #f
↦(tres-juntos? 'o lista1)
  #t
↦(tres-juntos? 'u lista1)
  #f
```

1- Escrever em *Scheme* o predicado *tres-juntos?*.
2- Escrever em *Scheme* o predicado *n-juntos?* com três argumentos, *num*, *simb* e *lis*, sendo o primeiro o número de símbolos iguais a *simb* que se pretendem encontrar juntos na lista *lis*.

```
↦(define lista1 '(o o x - x o o o x x o - x))
  lista1
↦(n-juntos? 2 'x lista1)
  #t
```

Exercício 13

d1 e *d2* são listas com três elementos, que representam datas: *(dia mes ano)*.
O procedimento *idade-anos* toma dois argumentos, *d1* e *d2*, e calcula a idade, à data *d2*, de uma pessoa que tenha nascido em *d1*. A resposta é dada em número inteiro de anos.

```
↦(define hoje (list 18 01 96))
hoje
↦(idade-anos (list 01 02 93) hoje)
idade = 2 anos
↦(idade-anos (list 01 04 94) hoje)
idade = 1 ano
↦(idade-anos (list 01 11 95) hoje)
idade = 0 anos
↦(idade-anos (list 01 12 96) hoje)          ; quando d1 é posterior a d2
idade = ?                                   ; a resposta é ?
```

1- Apresentar uma abstracção de dados para esta situação *(estruturas de dados e construtores, selectores e, eventualmente, modificadores)*.
2- Sobre a abstracção referida, escrever em *Scheme* o procedimento *idade-anos*.
3- Escrever em *Scheme* o procedimento *idade-anos-meses*, que difere do procedimento *idade-anos*, essencialmente no tipo de resposta, pois considerada também, para além do número de anos, o número de meses.

```
↦(idade-anos-meses (list 01 04 94) (list 16 01 96))
idade = 1 ano e 9 meses
↦(idade-anos-meses (list 15 11 95) (list 16 01 96))
idade = 0 anos e 2 meses
```

```
↦(idade-anos-meses (list 16 11 95) (list 16 01 96))
idade = 0 anos e 1 mes
↦(idade-anos-meses (list 17 11 95) (list 16 01 96))
idade = 0 anos e 1 mes
↦(idade-anos-meses (list 01 12 96) (list 16 01 96))
idade = ?
```

Exercício 14

As figuras mostram duas rodas concêntricas, com igual número de sectores, onde são representados valores inteiros. Na *fig. 1*, as duas rodas apresentam valores coincidentes apenas num caso *(valor 3)*.

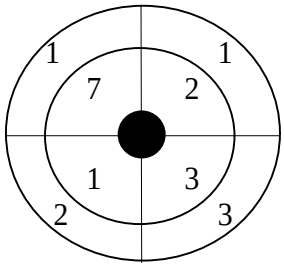


fig. 1

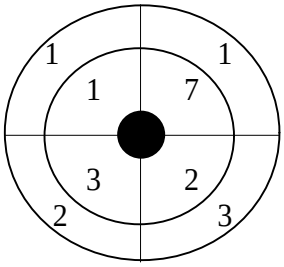


fig. 2

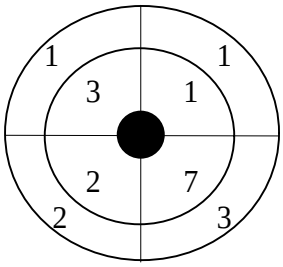


fig. 3

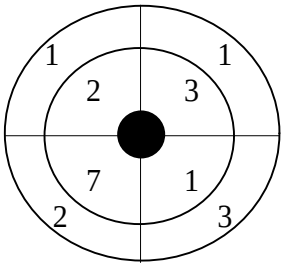


fig. 4

A roda exterior é fixa e a interior pode rodar no sentido dos ponteiros do relógio. Na *fig. 2*, a roda interior avançou um passo, e o número de valores coincidentes continua a ser um *(valor 1)*. Se a roda interior avançar mais um passo, *fig. 3*, o número de valores coincidentes passa a ser 2 *(valores 1 e 2)*. Finalmente, se a roda interior avançar mais um passo, *fig. 4*, o número de coincidências passa a zero. O procedimento *max-coinc* toma duas listas de comprimento igual, que representam duas roda, e determina a situação com o maior número de coincidências. Para as rodas indicadas na *fig. 1*, devolveria a seguinte mensagem:

número de passos= 2 número máximo de coincidências= 2

1- Escrever em *Scheme* o procedimento *mais-um-passo*, que recebe uma lista e devolve uma lista de igual comprimento, rodada de uma posição.

```
↦(mais-um-passo (list 1 2 3 4))
(4 1 2 3)
```

2- Escrever em *Scheme* o procedimento *num-coinc*, que recebe duas listas de igual comprimento e devolve o número de valores coincidentes.

```
↦(num-coinc (list 1 2 3 4) (list 23 4 3 5))
1                                          ; só 3 é valor coincidente
```

3- Escrever em *Scheme* o procedimento *max-coin*, baseado em *mais-um-passo* e em *num-coinc*.

4- Apresentar a *abstracção de dados* implícita na solução *max-coin*.

Exercício 15

Pretende-se escrever a *classe-aluno* para a criação de objectos que permitam armazenar informação sobre os alunos e as respectivas notas de disciplinas. A informação a armazenar deverá ser o *número do bilhete de identidade*, o *nome*, e uma *lista de disciplinas*, em que cada disciplina é representada por uma lista com dois elementos contendo o *nome* e a *nota* da disciplina.

Ao criar um objecto *aluno*, através da *classe-aluno*, apenas serão fornecidos o *nome* e o *número do bilhete de identidade*, devendo a lista das disciplinas estar inicialmente vazia. Os métodos

associados a esta classe são *juntar-disciplina!*, *alterar-nota!*, *ler-nota*, *ler-nome*, *ler-numero* e *visualizar*, cujo funcionamento se deduz nos exemplos que se seguem:

```
↦(define adao (classe-aluno "adao manuel" 1056))
adao
↦(adao 'juntar-disciplina! 'ip1 '18)
(ip1 18)
↦(adao 'juntar-disciplina! 'ip2 '17)
(ip2 17)
↦(adao 'ler-nota 'ip2)
17
↦(adao 'ler-nome)
adao manuel
↦(adao 'ler-numero)
1056
↦(adao 'alterar-nota! 'ip2 '19)
19
↦(adao 'visualizar)
1056 "adao manuel" ((ipc1 18)(ip2 19))
```

Escrever em *Scheme* a *classe-aluno,* integrando os métodos *juntar-disciplina!*, *alterar-nota!*, *ler-nota, ler-nome, ler-numero* e *visualizar*.

Exercício 16

Pretende-se escrever a *classe-conta-bancaria* para a criação de objectos que permitam armazenar informação sobre contas bancárias e respectivos movimentos. A informação a armazenar deverá ser o *nome do dono da conta*, o *número*, a *data de criação*, e uma lista com os *movimentos* efectuados. Cada *movimento* é representado por uma lista com três elementos: o seu *número*, que será atribuído sequencial e automaticamente, a *data*, e o respectivo *valor*.
Ao criar um objecto *conta-bancaria*, através da *classe-conta-bancaria*, será fornecido o *nome*, o *número* e *a data de criação*, devendo a lista dos movimentos estar inicialmente vazia. Os métodos associados a esta classe são *movimento!*, *saldo*, *ler-nome*, *ler-numero*, *ler-data-de-criacao* e *ultimos-3-movimentos*, cujo funcionamento se deduz nos exemplos que se seguem:

```
↦(define conta-da-ana (classe-conta-bancaria "ana" 12345 '23-7-97))
conta-da-ana
↦(conta-da-ana 'movimento! '23-7-97 100)
(1 '23-7-97 100)
↦(conta-da-ana 'movimento! '24-7-97 200)
(2 '24-7-97 200)
↦(conta-da-ana 'movimento! '25-7-97 -150)
(3 '25-7-97 -150)
↦(conta-da-ana 'ler-nome)
ana
↦(conta-da-ana 'ler-numero)
12345
↦(conta-da-ana 'ler-data-criacao)
23-7-97
↦(conta-da-ana 'movimento! '25-7-97 -50)
(4 '25-7-97 -50)
↦(conta-da-ana 'saldo)
100
↦(conta-da-ana 'ultimos-3-movimentos)
```

```
(2 '24-7-97 200)
(3 '25-7-97 -150)
(4 '25-7-97 -50)
```

Escrever em *Scheme* a *classe-conta-bancaria*, integrando os métodos *movimento!*, *saldo*, *ler-nome*, *ler-data-de-criacao*, *ler-numero* e *ultimos-3-movimentos*.

Exercício 17

Segundo a conjectura *Goldback*, os inteiros pares, maiores que 2, são iguais à soma de 2 números primos. Exemplos: 4 = 2 + 2; 6 = 3 + 3; 8 = 5 + 3; 10 = 7 + 3; 12 = 7 + 5; 14 = 11 + 3; ...
Considerar agora um programa em *Scheme* que suporta diálogos do seguinte tipo:

```
↦(goldback)
Apresentar um inteiro par e positivo, maior que 2: 3
3 não é par!!!

Apresentar um inteiro par e positivo, maior que 2: 4
4 = 2 + 2

Apresentar um inteiro par e positivo, maior que 2: -1
-1 não é positivo!!!

Apresentar um inteiro par e positivo, maior que 2: 20
20 = 17 + 3

Apresentar um inteiro par e positivo, maior que 2: 0
FIM!
```

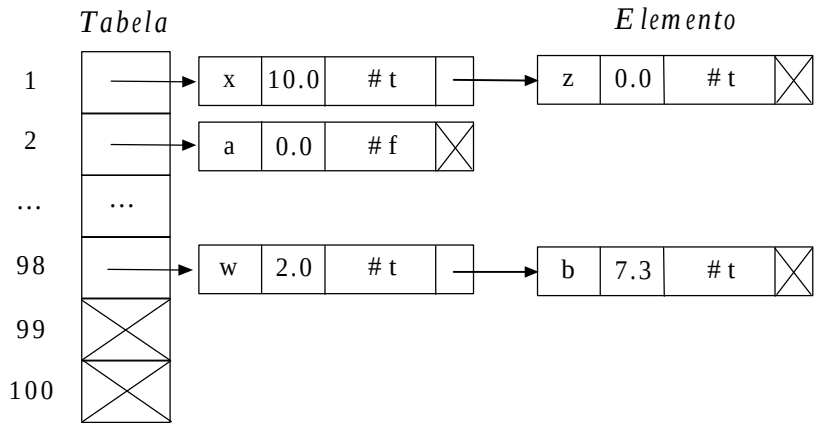
1- Escrever em *Scheme* o predicado *e-primo?*, que recebe um numero positivo e determina se é ou não primo, devolvendo, respectivamente, *#t* ou *#f*.
Pista: Procurar um divisor do *número*, de 2 até *número* dividido por 2.
2- Escrever em *Scheme* o procedimento *gold* que recebe um número par e positivo, maior que 2, e visualiza os primos correspondentes, de acordo com a conjectura de *goldback*. Por exemplo, a chamada (gold 20) visualiza 17 + 3.
Pista:

a) De *número - 1* a 2 procurar um primo *p1*.
b) De *p1* a 2 procurar um primo *p2*, tal que *número = p1 + p2*.
c) Se não encontrar *p1* e *p2* naquela situação, repetir a) de *número-p1* a 2 ...

3- Escrever em *Scheme* o programa *golback*, que suporta o diálogo acima indicado.

Exercício 18

Para a compilação de programas é muitas vezes necessária uma estrutura de dados para armazenar temporariamente os símbolos relativos às constantes, variáveis e tipos que vão sendo definidos e utilizados ao longo do programa. A esta estrutura de dados dá-se normalmente o nome de *tabela de símbolos*. Na figura, apresenta-se uma das suas configurações possíveis: um vector de *n* apontadores para listas de elementos, em que cada *elemento* contém o *símbolo*, o seu valor, e um campo indicativo do seu estado, activo *(#t)* ou inactivo *(#f)*.



Em torno desta estrutura de dados criou-se a abstracção que vamos designar por *tabela de símbolos*, definindo alguns procedimentos, entre os quais de distinguem:
Construtor
O procedimento *criar-tabela-de-simbolos* recebe um inteiro, *num*, e devolve uma *tabela de simbolos* vazia, ou seja, um vector de *num* posições todas elas preenchidas com *'()*.
Selector
O procedimento *ler-simbolo* recebe uma tabela de símbolos, *tab*, um inteiro, *int*, e um símbolo, *simb*, e procura *simb* a partir da posição *int* de *tab*. Devolve *#f* se não encontrar *simb* ou, se o encontrar, devolve uma lista em que o primeiro elemento é o valor associado a *simb* e o segundo é um booleano que representa o seu estado de activação.
Modificador
O procedimento *inserir-simbolo!* aceita cinco argumentos, uma *tabela de símbolos*, um *símbolo* a inserir na tabela, um inteiro que indica a posição do vector onde inserir o símbolo, um *valor associado ao símbolo* e um *booleano* que representa o seu estado de activação, e devolve a estrutura de dados com o elemento inserido. O elemento é inserido no princípio da lista apontada por essa posição, mesmo que outros elementos já lá se encontrem. Caso já exista um elemento na posição dada com o símbolo a inserir, então, simplesmente, ocorrerá a actualização dos campos do valor associado ao símbolo e do seu estado de activação.
O exemplo de tabela apresentado na figura resulta da seguinte sequência de invocações de *inserir-simbolo!*: o símbolo *b* na posição *98*, o símbolo *z* na posição *1*, o símbolo *x* na posição *1*, o símbolo *a* na posição *2*, e o símbolo *w* na posição *98*.

1- Apresentar a estrutura indicada numa representação gráfica *caixa-e-apontador*.
2- Escrever em *Scheme* o procedimento *criar-tabela-de-simbolos*.
3- Escrever em *Scheme* o procedimento *inserir-simbolo!*.
4- Escrever em *Scheme* o procedimento *ler-simbolo*.

Exercício 19 - Projecto da Lista telefónica

Supor que uma lista telefónica é representada por um vector, como se indica no exemplo que se segue:

```
#((joao 1835) (maria 1823) (joaquim 1845) (mafalda 1805)
(carlos 1856) (joana 1830))
```

Sobre uma lista telefónica, o programa *operar-lista* permite interacções do tipo:

```
↦(operar-lista lista-telefonica)
```

```
O que deseja? telefone/alterar/acrescentar/retirar/fim: telefone
indicar o nome: joana
joana tem o telefone 1830

O que deseja? telefone/alterar/acrescentar/retirar/fim: telefone
indicar o nome: pedro
pedro não tem telefone!!!

O que deseja? telefone/alterar/acrescentar/retirar/fim: qualquer-coisa
comando não previsto!!!

O que deseja? telefone/alterar/acrescentar/retirar/fim: alterar
indicar o nome: joana
indicar o novo número: 1900
joana com número alterado

O que deseja? telefone/alterar/acrescentar/retirar/fim: telefone
indicar o nome: joana
joana tem o telefone 1900

O que deseja? telefone/alterar/acrescentar/retirar/fim: alterar
indicar o nome: pedro
indicar o novo número: 2000
pedro não tem telefone

O que deseja? telefone/alterar/acrescentar/retirar/fim: fim
Trabalho finalizado...
```

1- Tendo em conta a interacção anterior, definir rigorosamente como se comportará o programa face a todos os comandos, incluindo os que não foram considerados naquela interacção, e também quando recebe respostas inadequadas.
2- Fazer uma abordagem *de-cima-para-baixo* ao problema exposto, identificar as suas tarefas e sub-tarefas principais e definir uma abstracção de dados adequada.
3- Escrever em *Scheme* a abstracção de dados definida.
4- Escrever em *Scheme* o programa *operar-lista*, apoiado na abstracção desenvolvida e nas tarefas e sub-tarefas identificadas.
Sugestão: As listas telefónicas poderão ser guardadas em ficheiro para utilização posterior.

Exercício 20 - Projecto do Jogo dos 14 fósforos

Vamos considerar que o *jogo-dos-14-fosforos* vai ter apenas dois jogadores em cada sessão. Inicialmente, são colocados 14 fósforos em cima de uma mesa e cada jogador pode, alternadamente, retirar 1 ou 2 fósforos. Ganha o jogador que retire o último dos 14 fósforos.
1- Imaginar o funcionamento de um programa que permita que dois humanos joguem o *jogo-dos-14-fosforos*.
2- Identificar as principais tarefas e sub-tarefas do problema exposto e, apoiado nelas, escrever em *Scheme* o programa imaginado, e que será designado por *jogo-dos-14-hum-hum*.
3- Escrever uma nova versão do programa que permita o *jogo-dos-14-fosforos* entre um jogador humano e o computador, que será designado por *jogo-dos-14-hum-comp*. Pretende-se que o computador tenha alguma estratégia nas suas jogadas, de tal forma que perante as 2 hipóteses, ou retirar 1 ou retirar 2 fósforos, escolha a que lhe proporcione mais possibilidades de vitória.

Exercício 21 - Projecto do Jogo do Galo

Vamos começar por imaginar uma matriz vazia, *4 x 4*, que representa o tabuleiro onde decorre o *jogo do galo*. Participam dois jogadores que serão identificados por *x* e *o*, fazendo cada um deles, alternadamente, uma jogada. Eis a matriz vazia:

```
1  2  3  4
-  -  -  -
-  -  -  -
-  -  -  -
-  -  -  -
```

Neste jogo, ganha o primeiro jogador que consegue colocar três dos seus símbolos em posições adjacentes quer em linha, em coluna ou em diagonal. Observar três exemplos em que o jogador *x* ganha.

em linha

```
1  2  3  4
-  -  -  -
-  o  -  -
-  o  -  -
x  x  x  -
```

em coluna

```
1  2  3  4
-  -  -  -
-  -  -  x
-  o  -  x
x  o  o  x
```

em diagonal

```
1  2  3  4
-  -  -  -
-  -  x  -
-  x  o  -
x  o  o  x
```

Cada jogador, na sua vez, limita-se a indicar em que coluna quer colocar o seu símbolo, e o preenchimento faz-se na posição mais baixa da coluna escolhida. Exemplo:
jogador 1 *(o)* escolhe a coluna 2: jogador 2 *(x)* também escolhe a coluna 2:

```
1  2  3  4
-  -  -  -
-  -  -  -
-  -  -  -
-  o  -  -
```

```
1  2  3  4
-  -  -  -
-  -  -  -
-  x  -  -
-  o  -  -
```

Tentando uma representação da matriz associada ao tabuleiro do *jogo do galo*: Cada uma das 4 colunas deste jogo é representada por uma lista de 4 elementos, em que cada um destes elementos poderá um dos símbolos *o*, *x*, e -.
Exemplo:

```
1  2  3  4
o  -  -  -
x  -  -  x    coluna 1:  (o  o  x  o)    coluna 2:  (o  x  -  -)
o  x  -  o    coluna 3:  (-  -  -  -)    coluna 4:  (x  o  x  -)
o  o  -  x
```

Por seu turno, o *tabuleiro de jogo* é representada por uma lista de listas. Assim, para o exemplo indicado, a lista será:

```
'((o o x o)(o x - -)(- - - -)(x o x -))
```

1- Um *tabuleiro de jogo* sem células livres é considerada *jogo empatado* e pode ser detectada por:

```
(define empatado?
  (lambda (tabul-jogo)
    (and (coluna-cheia? tabul-jogo 1)
         (coluna-cheia? tabul-jogo 2)
         (coluna-cheia? tabul-jogo 3)
         (coluna-cheia? tabul-jogo 4))))
```

Sendo *tabul-jogo* uma lista que representa um *tabuleiro de jogo*, escrever em *Scheme* o predicado *coluna-cheia?* com 2 parâmetros, *tabul-jogo* e *n*, em que *n* é um inteiro entre 1 e 4, representando uma coluna, e que devolve *#t* se essa coluna não contém células livres.
2- Escrever em *Scheme* o procedimento *joga* tem 3 parâmetros: *tabul-jogo* é um *tabuleiro de jogo*, *simb* é um símbolo que representa um dos jogadores (*o* ou *x*), e *col* é um inteiro de 1 a 4 que representa a coluna onde se vai jogar. Este procedimento devolve um novo *tabuleiro de jogo* que resulta de *tabul-jogo* depois de colocado *simb* na coluna *col*.

Sendo *tabul-jogo1* um tabuleiro de jogo na situação inicial, ou seja, completamente vazio, analisar o que se segue para clarificar o exposto.

```
↦(define tabul-jogo1 (joga tabul-jogo1 'x 2))
tabul-jogo1
↦tabul-jogo1
((- - - -) (x - - -) (- - - -) (- - - -))
↦(define tabul-jogo1 (joga tabul-jogo1 'o 2))
tabul-jogo1
↦tabul-jogo1
((- - - -) (x o - -) (- - - -) (- - - -))
```

3- Supondo que agora se trabalha com *listas mutáveis*, escrever em *Scheme* o procedimento *joga* que passará a designar-se por *joga!*.

4- Escrever em *Scheme* o procedimento *visu-pos-de-jogo* que aceita um *tabuleiro de jogo* como argumento único e visualiza-o como se indica:

```
↦tabul-jogo-50
((- - - -) (x o o -) (x - - -) (o x - -))
↦(visu-pos-de-jogo tabul-jogo-50)

    1  2  3  4
    -  -  -  -
    -  o  -  -
    -  o  -  x
    -  x  x  o
```

5- Imaginar um programa que permita que dois humanos joguem o *jogo-do-galo*. Definir uma abstracção de dados para apoiar o desenvolvimento desse programa. Certamente que a abstracção integrará procedimentos que já foram considerados em alíneas anteriores, mas deverá também considerar outros, como, por exemplo, um procedimento que detecte, após uma jogada, se o jogador atingiu uma das três situações que lhe permite vencer o jogo. Escrever em *Scheme* a abstracção definida.

6- Através de uma abordagem *de-cima-para-baixo*, identificar as principais tarefas do problema e escrever em *Scheme*, apoiado nelas, o programa imaginado, designado por *jogo-do-galo-hum-hum*,

7- Com um grande de dificuldade bastante maior, escrever uma outra versão do programa que permite o *jogo-do-galo* entre um jogador humano e o computador, designada por *jogo-do-galo-hum-comp*. Pretende-se que o computador tenha alguma estratégia nas suas jogadas e, perante as 4 colunas, escolha uma de acordo com a seguinte prioridade: 1ª- a coluna que lhe permita ganhar; 2ª- a coluna que não permite que o adversário ganhe na jogada seguinte; 3ª- a primeira coluna livre a contar da esquerda.

Exercício 22 - Projecto do Caminho mínimo

(projecto de elevado grau de dificuldade)

Um tabuleiro de 5 x 5 tem as suas células identificadas de 1 a 25.

1	2	3	4	5
6	7	8	9	10
11	12	13	14	15
16	17	18	19	20
21	22	23	24	25

A cada uma das células do tabuleiro é associado aleatoriamente um valor inteiro situado entre 0 e 9.

2	5	7	3	0
1	2	8	2	1
0	6	1	0	2
3	7	3	9	4
5	2	5	2	5

Dada uma célula de partida, pretende-se determinar o *caminho mais curto* entre a célula dada e a célula de chegada, que é a 25. Entende-se por *caminho mais curto* como sendo o que corresponde à soma mínima dos valores associados às células por onde passa. Por exemplo, sendo 18 a célula de partida, o caminho mais curto passará pelas células: 18, 23, 24 e 25, a que corresponde um *comprimento* de 3 + 5 + 2 + 5 = 15.

1- Sabendo que os deslocamentos possíveis, a partir de uma posição *pos*, são *pos+1* (se *pos* não estiver na coluna 5) e *pos+5* (se *pos* não estiver na linha 5), escrever em *Scheme* o procedimento *procura-caminho-min-v1* que começa por gerar um tabuleiro com os pesos associados às 25, visualizando-o, de seguida. Pede a célula de partida, determina o caminho mais curto a partir dela, visualiza o respectivo comprimento e as células por onde passa.

```
↦(procura-caminho-min-v1)
4   4   1   9   3
2   6   3   7   5
3   4   8   3   0
1   6   4   2   3
6   1   0   0   6
partida: 8
O comprimento do percurso minimo e': 21
O percurso minimo passa por:
8   9   14   19   24
25
ok
↦(procura-caminho-min-v1)
5   4   9   2   6
5   8   0   6   0
5   8   6   8   8
1   1   2   1   7
4   2   5   8   6
partida: 1
O comprimento do percurso minimo e': 33
O percurso minimo passa por:
1   6   11   16   17
18   19   20   25
ok
```

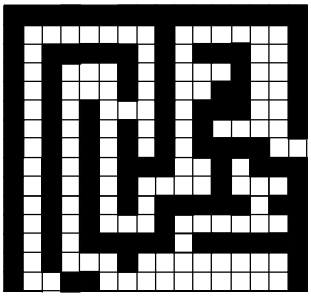
2- Escrever em *Scheme* o procedimento *procura-caminho-v2* que responde como o procedimento *procura-caminho-v1*, diferindo apenas nos deslocamentos possíveis a partir de uma posição *pos*: *pos+1* (se não se encontrar na coluna 5), *pos-1* (se não se encontrar na coluna 1), *pos+5* (se não se encontrar na linha 5) e *pos-5* (se não se encontrar na linha 1).

3- Escrever em *Scheme* o procedimento *procura-caminho-v3* que responde como a versão anterior, mas tem dois parâmetros *n* e *m*. O primeiro define a dimensão do tabuleiro que passa a ser *n x n*. O segundo, *m*, define a gama dos valores inteiros que preenchem aleatoriamente o tabuleiro. Em vez dos inteiros estarem entre 0 e 9, passam a estar entre 0 e m-1.

Exercício 23 - Projecto do Labirinto

(projecto de elevado grau de dificuldade)

Vamos trabalhar com labirintos que têm uma entrada e uma saída, e são representados por tabuleiros quadrados, cujas



células ocupadas e desocupadas são preenchidas, respectivamente, com *#t* e *#f*. Para já, os labirintos são construídos manualmente, ou seja, o utilizador terá que, de alguma maneira, preencher a estrutura de dados que representa cada um dos labirintos a explorar.

Método de orientação no labirinto
Avançar no labirinto sempre com a mão direita em contacto com a parede.
Este método falha quando os labirintos têm duas entradas ligadas por um caminho que não passa pelo ponto de chegada. O método também falha quando os labirintos têm caminhos fechados que retornam ao ponto de partida, rodeando o ponto de chegada.

Pretende-se desenvolver o programa *labirinto* que aceita um labirinto, visualiza-o graficamente, pede a identificação da célula de entrada e visualiza o caminho entre a entrada e a saída.
1- Imaginar o tipo de diálogo que o programa *labirinto* suporta com o utilizador e, através de uma abordagem *de-cima-para-baixo*, identificar as suas tarefas principais e definir uma abstracção de dados que suporte o seu desenvolvimento.
2- Escrever em *Scheme* a abstracção definida.
3- Escrever em *Scheme* o programa *labirinto*.
4- Imaginar e implementar uma ferramenta gráfica para apoiar o utilizador do programa *labirinto* a construir labirintos.
Sugestão: Os labirintos poderão ser guardados em ficheiro para utilização posterior.