

Módulo 3.5 - **Resumo dos módulos 3.1 a 3.4, exemplos e exercícios**

Resumo dos assuntos abordados nos módulos

Módulo 3.1 - **Abstracção de dados para captar as características dos problemas**

Módulo 3.2 - **Colocar objectos em sequência**

Módulo 3.3 - **Um jogo para testar a memória**

Módulo 3.4 - **Utilizar e construir abstracções para reutilizar código**

Após o resumo são apresentados alguns exemplos e exercícios para treino da matéria relacionada com os módulos acima referidos. Através de um exemplo sobre a colocação de 8 damas num tabuleiro de xadrez é introduzida a técnica de *backtracking*, cuja utilização é incentivada nos dois exercícios (as cenouras e a pintura de mapas).

A utilização do Scheme é fundamental neste módulo.

Resumo dos assuntos abordados nos módulos 3.1 a 3.4

Neste conjunto de módulos é mostrado o que deve ser feito quando os tipos de dados e respectivas operações básicas, oferecidos pelas linguagens de programação, não se adaptam convenientemente aos problemas que se pretendem resolver. Em situações deste tipo, recomenda-se que se componham novos tipos de dados, criando o que se designa por *abstracção de dados* com a qual se pretende captar as características do problema a resolver.

Módulo 3.1

Os tipos de dados fornecidos pelas linguagens de programação estão longe de se adequarem a todas as situações que nos surjam, o que implica a criação de novos tipos de dados de acordo com as características de cada problema. Numa situação que envolva formas gráficas a duas dimensões (2D), é perfeitamente justificável a criação de um novo tipo de dados, o *ponto-2D*, constituído por dois valores numéricos que representam as coordenadas x e y de um ponto no plano.

Um novo tipo de dados que se ajuste às características de um certo problema, acompanhado das operações básicas respectivas, constitui uma espécie de *barreira* sob a qual se encontra o que é essencial para trabalhar com os dados desse problema. É a isto que se chama *abstracção de dados*.

O *par* é a entidade do Scheme que permite criar abstracções de dados. O *par* é acompanhado de operações do tipo *construtor*, para criar objectos novos, e *selector*, para dar acesso às partes de um objecto.

Na abstracção *tartaruga*, os objectos que se podem criar e manipular são tartarugas. Sobre esta abstracção, entretanto disponibilizada num directório do DrScheme, é proposto um projecto de um jogo com alguma complexidade, designado por *A tartaruga escondida*.

Módulo 3.2

Em relação aos dados, nem sempre as situações que encontramos se resolvem com dados simples. Em certos casos é necessário colocar objectos em *sequência*, o que requer dados que são compostos por outros dados. O *par* permite compor dados com qualquer complexidade, mas a *lista* é a forma privilegiada do Scheme para implementar sequências. A lista pode ser vista como um par herdando por isso toda a funcionalidade deste, quer em termos de construção de objectos quer em termos de selecção dos seus componentes. Mas a lista aparece com uma funcionalidade muito mais rica que a do par. Neste módulo, a introdução à funcionalidade da lista é feita ao mesmo tempo que se estuda uma forma de implementar essa mesma funcionalidade e assim se aproveita para ir treinando a programação com listas.

Para além dos *números*, *booleanos*, *pares* e *listas*, aparecem agora os *símbolos*, como sendo um novo tipo de dados muito adequado em determinadas situações. Por exemplo, representar o mês de Janeiro pelo número 1 ou pelo símbolo *janeiro* não é bem a mesma coisa, a legibilidade do símbolo é muito maior.

A sequência de dados ou de objectos pode envolver elementos que são, eles próprios, sequências e assim aparece a lista de listas. Surge, desta forma, a necessidade de processar a *lista em profundidade*, ou seja, a necessidade de percorrer a lista e explorar a estrutura de cada um dos componentes.

Módulo 3.3

O desenvolvimento de um programa que funciona como teste à memória do utilizador surge como um exemplo de aplicação da *abstracção conjunto*. Já depois deste programa desenvolvido, introduziram-se alterações à abstracção conjunto, tendo em vista a obtenção de uma versão com melhor desempenho. A aplicação desta última versão no programa já desenvolvido, sem exigir qualquer alteração neste, demonstra bem que a abstracção funciona como uma barreira que esconde perfeitamente as suas características internas. Isto significa que mantendo a interface de uma abstracção, ou seja, a sua funcionalidade, é possível submetê-la a melhoramentos sem inviabilizar as aplicações que entretanto tenham sido desenvolvidas sobre ela.

Módulo 3.4

A reutilização de código desenvolvido pelo próprio ou por outros é uma prática comum e importante, especialmente se aquele código for digno de confiança. Neste módulo é mostrado como se pode reutilizar código que foi desenvolvido e disponibilizado sob a forma de uma abstracção.

A abstracção *janela gráfica* é apresentada e através dela é possível controlar uma caneta que desenha, pinta e escreve texto, e tudo isto sem exigir qualquer conhecimento dos procedimentos gráficos do *DrScheme*, que assim ficaram completamente escondidos por esta *barreira* da *janela gráfica*. Sobre a abstracção *janela gráfica* estabeleceram-se as abstracções *eixos* e *tabuleiro*, a primeira para visualizar sistemas de eixos ortogonais e a segunda para visualizar e manipular tabuleiros semelhantes aos tabuleiros de damas.

Exercícios e exemplos

São apresentados alguns exercícios e exemplos para consolidação da matéria tratada nos módulos 3.1 a 3.4.

Para estes exercícios e exemplos recomenda-se a consulta do *Anexo A (principais procedimentos do Scheme)*, em particular, a secção correspondente ao Processamento de listas. Para reutilização de abstracções previamente desenvolvidas ou criação de novas, sugere-se o estudo do *Anexo C (reutilização de código)* e a consulta dos *Anexos B (abstracção janela gráfica)* e *D (abstracção tabuleiro)*.

Exercício 1

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

Escreva em Scheme os seguintes procedimentos:

- 1- O construtor *faz-ponto* recebe dois valores numéricos, respectivamente, as coordenadas x e y de um ponto, e devolve uma entidade do tipo *ponto-2D*. Os selectores *x-coord* e *y-coord* recebem um *ponto-2D* e devolvem, respectivamente, as coordenadas x e y desse ponto.
- 2- Um rectângulo, com os lados paralelos aos eixos de coordenadas, pode ser definido por dois vértices opostos. O construtor *faz-rectangulo* recebe duas entidades do tipo *ponto-2D* que representam dois vértices opostos que definem um rectângulo e devolve uma entidade do tipo *rectangulo*. O procedimento *area-rectangulo* recebe um *rectangulo* e devolve a área desse rectângulo.

- 3- O procedimento *distancia* recebe duas entidades *ponto-2D* como argumentos e devolve a distância entre elas.
- 4- O procedimento *area-de-intersecao* recebe duas entidades *rectângulo* e devolve a área de intersecção desses 2 rectângulos.

Espaço para desenvolver e testar os procedimentos *faz-ponto*, *x-coord*, *y-coord*, *faz-rectangulo*, *area-rectangulo*, *distancia* e *area-de-intersecao*.
O ecrã abre vazio.

Exercício 2

Escreva em Scheme os seguintes procedimentos:

- 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 o teu cao) 'gato 'cao)
(o meu gato e o teu cao)
> (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 o teu cao) 'gato 'cao)
(o meu gato e o teu gato)
> (substitui-todas-ocorrencias '() 'gato 'cao)
()
```

Espaço para desenvolver e testar os procedimentos *substitui-primeira-ocorrencia* e *substitui-todas-ocorrencias*.
O ecrã abre vazio.

Exercício 3

Escreva em Scheme os seguintes procedimentos:

- 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*.

Espaço para desenvolver e testar os procedimentos *elimina-primeira-ocorrencia*, *elimina-todas-ocorrencias* e *elimina-ultima-ocorrencia*.
O ecrã abre vazio.

Exercício 4

Escreva em Scheme os seguintes procedimentos:

- 1- O procedimento *junta-ordenado* tem dois parâmetros, *crescente-1* e *crescente-2*, que são duas listas com elementos numéricas ordenados do menor para o maior. Sabe-se que *junta-ordenado* devolve uma lista cujos elementos são os elementos das listas dadas, em ordem crescente.

```
> (junta-ordenado '(2 3 40) '(1 3 34))
(1 2 3 3 34 40)
> (junta-ordenado '() '(1 3 34))
```

```
(1 3 34)
> (junta-ordenado '(2 3 40) '())
(2 3 40)
```

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

```
> (junta-ordenado-sem-repetidos '(2 3 40) '(1 3 34))
(1 2 3 34 40)
> (junta-ordenado-sem-repetidos '() '(1 3 34))
(1 3 34)
> (junta-ordenado-sem-repetidos '(2 3 40) '())
(2 3 40)
```

Espaço para desenvolver e testar os procedimentos *junta-ordenado* e *junta-ordenado-sem-repetidos*.
O ecrã abre vazio.

Exercício 5

Escreva em Scheme os seguintes procedimentos:

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

```
> (reverse '(a (b c) (d (e f))))
((d (e f)) (b c) a)
> (inverter-posicao-de-todos '(a (b c) (d (e f))))
(((f e) d) (c b) a)
```

- 2- Antes de escrever o procedimento *substituir-todos*, analise as três 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)
()
```

- 3- O procedimento *planificador*, com o parâmetro *lista*, devolve uma lista formada por todos os *não pares* de *lista*.

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

Espaço para desenvolver e testar os procedimentos *inverte-posicao-de-todos*, *substitui-todos* e *planificador*.
O ecrã abre vazio.

Exemplo 1

Projecto - Caminho

Vamos imaginar um caminho traçado sobre um tabuleiro composto por 25 células, organizadas numa matriz 5 x 5. Numa das células do tabuleiro é colocada 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 5, a tartaruga não pode deslocar-se mais na horizontal, para a direita. Também, quando atinge a linha do fundo, a tartaruga não pode deslocar-se mais na vertical, para baixo. 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

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

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

células 2, 9, 12, 14, 15, 17 e 23, o tabuleiro toma o aspecto indicado na figura.

A tartaruga segue sempre um caminho de acordo com uma estratégia muito simples: desloca-se prioritariamente na horizontal, para a direita, e só quando fica bloqueada neste movimento (ou encontrou um obstáculo ou alcançou a coluna 5) é que tenta o deslocamento na vertical, para baixo.

Para representar computacionalmente um tabuleiro, escolhemos uma lista contendo os índices das células com obstáculos. O tabuleiro, com obstáculos indicados, terá a seguinte representação: (2 9 12 14 15 17 23).

O procedimento *caminho*, que se pretende desenvolver, tem dois parâmetros, o ponto de partida, onde se coloca a tartaruga, e uma lista que representa o tabuleiro e vai indicando o caminho percorrido por esta, de acordo com a estratégia já indicada.

```
> (define tabuleiro '(2 9 12 14 15 17 23))
> (caminho 1 tabuleiro)
1; 6; 7; 8; 13; 18; 19; 20; 25; consegui
> (caminho 11 tabuleiro)
11; 16; 21; 22; falhei
> (caminho 2 tabuleiro)
partida errada!!!
> (caminho 3 tabuleiro)
3; 4; 5; 10; falhei
```

O procedimento *caminho* verifica se o ponto de partida, fornecido como argumento, coincide com uma célula ocupada por um obstáculo. Se assim for, a mensagem "*partida errada!!!*" é imediatamente visualizada e o processo termina. Tratando-se de uma célula livre, inicia-se a caminhada, ou seja, é chamado o procedimento *andar*, que é o núcleo central da resolução do problema, como verá.

```
(define caminho
  (lambda (partida obstaculos)
    (if (em-obstaculo? partida obstaculos)      ; partida sobre um obstáculo?
        (display "partida errada!!!")           ; sim
        (andar partida obstaculos)))            ; não
```

O procedimento *caminho* utiliza o procedimento *em-obstaculo?* que indica se uma célula contém ou não um obstáculo.

```
(define em-obstaculo?
  (lambda (elem lista)
    (member elem lista)))
```

No procedimento em-obstaculo?, utiliza-se um pequeno artifício que complica um pouco a sua legibilidade. Estamos a falar da utilização de member. Explique a sua utilização nesta situação. Apresente uma solução alternativa para que o procedimento se apresente mais legível.

O procedimento *andar* é recursivo e tem dois casos de terminação:

- Quando *posicao* (variável que modela a posição da tartaruga) atinge a célula 25, que significa objectivo atingido;
- Quando a tartaruga fica bloqueada (não consegue deslocar-se nem na horizontal nem na vertical).

O procedimento *andar* tem também dois casos gerais

- Estando a tartaruga bloqueada na horizontal, avança para a linha seguinte:

```
(andar (+ 5 posicao))      ; avançar para a célula imediatamente
                          ; abaixo (na linha seguinte) corresponde
                          ; a somar 5 à posição actual da tartaruga.
```

- Não estando bloqueada na horizontal, avança sobre essa linha:

```
(andar (add1 posicao))     ; avançar para a célula imediatamente
```

```

; ao lado (na mesma linha) corresponde
; a somar 1 à posição actual da tartaruga.

```

Pelos casos gerais do procedimento *andar*, conclui-se que a operação de redução, utilizada no passo recursivo, traduz-se em adicionar um certo valor a *posicao*, tentando assim chegar a 25. Num dos casos, adicionando-lhe 1 (*movimento na horizontal*), no outro caso, adicionando-lhe 5 (*movimento na vertical*).

Explique o funcionamento do procedimento andar.

```

(define andar
  (lambda (posicao obstaculos)

    (display posicao)          ; visualiza posicao
    (display "; ")
    (cond
      ((= posicao 25)          ; atingiu a célula 25
        (display "consegui")
        (newline))
      ((bloqueado? posicao obstaculos)
        (display "falhei")    ; ficou bloqueado
        (newline))
      ((bloqueado-horiz? posicao obstaculos) ; bloqueado na horizontal?
        (andar (+ 5 posicao) obstaculos)) ; Sim. Avança na vertical
      (else
        (andar (add1 posicao) obstaculos)))) ; Não. Avança na horizontal

```

O procedimento *andar* utiliza o procedimento *bloqueado?* que determina se a tartaruga está ou não bloqueada, ou na horizontal ou na vertical.

```

(define bloqueado?
  (lambda (posi obs)          ; bloqueado significa...
    (and (bloqueado-horiz? posi obs) ; bloqueado na horizontal
          (bloqueado-vert? posi obs)))) ; e bloqueado na vertical

(define bloqueado-horiz?
  (lambda (posi obs)
    (or
      (em-coluna-direita? posi) ; bloqueado na horizontal significa estar...
      (em-obstaculo? (add1 posi) obs)))) ; na coluna 5
                                          ; ou ter um obstáculo à direita

(define bloqueado-vert?
  (lambda (posi obs)
    (or
      (em-linha-fundo? posi) ; bloqueado na vertical significa estar...
      (em-obstaculo? (+ posi 5) obs)))) ; na linha 5, ou ter um
                                          ; obstáculo por baixo

(define em-coluna-direita? ; está encostado à direita?
  (lambda (posicao)
    (zero? (remainder posicao 5))))

(define em-linha-fundo? ; está encostado ao fundo?
  (lambda (posicao)
    (> posicao 20)))

```

Acabámos de apresentar 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. O tabuleiro com os obstáculos foi representado através de uma lista perfeitamente normal. Quanto à tartaruga, um inteiro chegou para representar a célula que ocupa no tabuleiro.

Espaço para testar o procedimento *caminho*.
O ecrã abre com o código desenvolvido.

Exercício 6

No contexto do exemplo anterior, introduza as alterações necessárias à solução apresentada, para chegar ao procedimento *caminho-no-tabuleiro*, que passa a responder como se indica.

```
> (define obs (list 2 9 12 14 15 17 23))
> (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  .  .
```

Espaço para desenvolver e testar o procedimento *caminho-no-tabuleiro*.
O ecrã abre com o código desenvolvido no exemplo anterior.

Exercício 7

Ainda no contexto do exemplo anterior, introduza as alterações necessárias à solução apresentada, para chegar ao procedimento *caminho-no-tabuleiro-graf* que, utilizando a janela corrente e a abstracção *tabuleiro*, passa a responder como se indica. Este procedimento tem apenas um parâmetro que define a posição inicial da tartaruga.

Para incluir a abstracção *tabuleiro*, fazer

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

```
> (caminho 1)
Lado do tabuleiro em celulas : 6
Lado da celula em pixels: 25
```

```
1- obstaculo      2- posicao partida
8- partida        9- limpar trajetoria
10- fim
1
Coluna (de 0 a 5): 2
Linha (de 0 a 5): 0
```

```
1- obstaculo      2- posicao partida
8- partida        9- limpar trajetoria
10- fim
1
Coluna (de 0 a 5): 2
Linha (de 0 a 5): 1
```

```
1- obstaculo      2- posicao partida
8- partida        9- limpar trajetoria
10- fim
1
Coluna (de 0 a 5): 2
Linha (de 0 a 5): 2
```



a escolha de célula já
preenchida, elimina
obstáculo

```

1- obstaculo      2- posicao partida
8- partida        9- limpar trajetoria
10- fim

```

```

1
Coluna (de 0 a 5): 2
Linha (de 0 a 5): 1

```

```

1- obstaculo      2- posicao partida
8- partida        9- limpar trajetoria
10- fim

```

```

8

```

```

1- obstaculo      2- posicao partida
8- partida        9- limpar trajetoria
10- fim

```

```

10

```

```

acabou

```



Espaço para desenvolver e testar o procedimento *caminho-no-tabuleiro-graf*.
O ecrã abre com o código desenvolvido no exemplo anterior.

Exemplo 2

Projecto - Adivinha-número

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 computador. Vejamos um exemplo de utilização.

```

> (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)

Neste exemplo, considerou-se não ser necessário gastar muito tempo 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. Quanto aos aspectos funcionais do jogo, optou-se por uma abordagem *de-cima-para-baixo*, para identificar as tarefas principais do programa *adivinba-numero*.

- gerar um número aleatório entre 0 e 31;
- pedir ao utilizador, quatro vezes, uma lista 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 *faça-perguntas*, procedimento recursivo em que o caso base corresponde ao contador *vezes* atingir o valor zero. As tarefas de *faça-perguntas* são:

- visualizar uma mensagem pedindo a indicação de uma lista de números entre 0 e 31;
- verificar se o número gerado 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 ()
    ;
    ; definição local do procedimento faz-perguntas
    ;
    (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-1-n 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."))
            (faz-perguntas 4 num-sel))))))
```

```

(begin
  (display "Falhaste. O numero certo seria: ")
  (display num-sel))
(newline))))
;
; para aceder ao procedimento roleta-1-n contido em varios.scm
; (ver Anexo C)
(require (lib "varios.scm" "user-feup"))

```









Espaço para testar o programa *adivinha-numero*.
O ecrã abre com o código desenvolvido.

*No corpo do programa, identifique as suas tarefas principais.
No corpo do procedimento local faz-perguntas, identifique as suas tarefas principais*

Exemplo 3 - avançar e recuar... a técnica de *backtracking*

Num tabuleiro de xadrez, uma rainha ataca outra rainha se ambas estiverem na mesma linha horizontal, vertical ou a 45°.

O problema que se pretende resolver consiste em posicionar 8 rainhas num tabuleiro de xadrez, de modo que nenhuma delas possa atacar outra. No tabuleiro indicado ao lado é apresentada uma dessas posições, mas existem ainda mais 91... e o desafio é descobri-las com a ajuda do computador.

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

Uma forma de codificar a posição com 8 rainhas, indicada no tabuleiro, é a lista (5 7 2 6 3 1 4 8) de comprimento 8.

Explique como funciona a codificação que levou à lista (5 7 2 6 3 1 4 8).

A lista (5 7 2 6 3 1 4 8) representa uma posição legal, pois nenhuma rainha ataca outra.

Das posições que se seguem, embora não sendo de comprimento 8, descubra manualmente as que são legais e ilegais:
(1 4 8), (8 4 8), (5) e (6 4 8)

Em vez de verificar manualmente se uma posição é ou não legal pode utilizar, como se fosse uma caixa-preta, o predicado *tentativa-legal?*.

Por exemplo, para verificar se (6 4 8) é ou não legal bastará fazer executar (tentativa-legal? 6 (list 4 8)).

; verifica se, ao juntar um novo elemento a uma posição legal, resulta
; uma posição legal com o comprimento incrementado de 1

```

(define tentativa-legal?
  (lambda (novo posi-legal)
    (letrec ((tentativa-aux?
      ;
      (lambda (p-legal distancia)
        (if (null? p-legal)
          #t
          (let ((prox (car p-legal)))
            (and
              (not (= prox novo))
              (not (= prox (+ novo distancia)))
              (not (= prox (- novo distancia)))
              (tentativa-aux? (cdr p-legal)
                             (add1 distancia)))))))
      (tentativa-aux? posi-legal 1))))

```

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

Espaço para testar o procedimento *tentativa-legal?*.
O ecrã abre com este procedimento.

Apresente uma explicação para o modo de funcionamento do predicado tentativa-legal?

Imagine que se encontra na posição legal correspondente a (4 1 7 5 3 8), indicada no tabuleiro, imediatamente antes de colocar a rainha na coluna 2 e na linha 6.

No que respeita à coluna 2, nas linhas 8 e 7 não seria possível colocar uma rainha, foi por isso que se chegou à linha 6.

A linha 6 é a primeira hipótese possível. No entanto, a posição obtida, (6 4 1 7 5 3 8), apesar de legal, impede que se alcance uma posição legal de comprimento 8.



*Verifique, a começar pela linha 8 e tentando até à linha 1, a impossibilidade de colocar uma rainha na coluna 1 a partir da posição legal (6 4 1 7 5 3 8).
Então que solução propõe?*

Se na coluna 1 chegou à linha 1 e não encontrou qualquer hipótese de completar uma posição legal, então alguma coisa está mal para trás e a solução é *recuar* (*backtracking*) para a coluna 2 e, em vez de 6, tentar 5, 4, 3, 1... e se de nenhuma resultar uma posição possível, recuar para a coluna 3...

Se continuasse a recuar, em que situação poderia concluir que não haveria solução?

A ideia é partir de uma posição vazia e fazer (vai-construindo-solucao ' ())

```
(define tentativa-inicial 8)
(define comprimento-solucao 8)

(define vai-construindo-solucao
  (lambda (posi-legal)
    (if (solucao? posi-legal)
        posi-legal
        (avanca tentativa-inicial posi-legal))))
```

e com avanços e recuos

```
(define avanca
  (lambda (novo posi-legal)
    (cond
      ((zero? novo) (recua posi-legal))
      ((tentativa-legal? novo posi-legal)
       (vai-construindo-solucao (cons novo posi-legal)))
      (else (avanca (sub1 novo) posi-legal)))))

(define recua
  (lambda (posi-legal)
    (if (null? posi-legal)
        (list)
        (avanca (sub1 (car posi-legal))
                  (cdr posi-legal)))))
```

chegar certamente a uma solução como, por exemplo, (5 7 2 6 3 1 4 8).

```
(define solucao?
  (lambda (posi-legal)
    (= comprimento-solucao (length posi-legal))))
```

Espaço para testar o procedimento *vai-construindo-solucao* e os procedimentos auxiliares.
O ecrã abre com os procedimentos a testar.

A partir de uma lista vazia chegou-se à solução (5 7 2 6 3 1 4 8). Para encontrar outra solução, a ideia é recuar sobre a solução obtida, imaginando que é uma falha, e tentar construir outra solução a partir dessa situação de "falha". Por exemplo, para obter 3 soluções,

```
> (let*
  ((sol1 (vai-construindo-solucao '()))
   (sol2 (vai-construindo-solucao (backtrack sol1)))
   (sol3 (vai-construindo-solucao (backtrack sol2))))
  (list sol1 sol2 sol3))

((5 7 2 6 3 1 4 8) (4 7 5 2 6 1 3 8) (6 4 7 1 3 5 2 8))
>
```

Verifique manualmente que estas 3 soluções são, de facto, posições legais de 8 damas num tabuleiro de xadrez.

E agora para encontrar todas as soluções possíveis.

```
(define faz-todas-as-solucoes
  (lambda ()
    (letrec
      ((ciclo
        (lambda (solucao)
          (if (null? solucao)
              '()
              (cons solucao
                    (ciclo (recua solucao)))))))
      (ciclo (faz-solucao (list))))))

> (length (faz-todas-as-solucoes))
92
> (faz-todas-as-solucoes)
((5 7 2 6 3 1 4 8)
 (4 7 5 2 6 1 3 8)
 (6 4 7 1 3 5 2 8)
 ...
 (3 5 2 8 6 4 7 1)
 (5 2 4 7 3 8 6 1)
 (4 2 7 3 6 8 5 1))
>
```

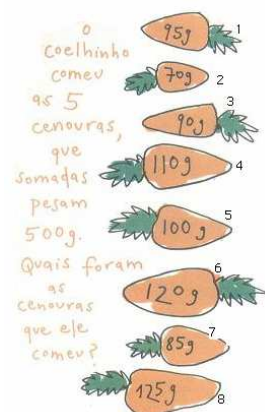
Apresente uma justificação para o caso de terminação do procedimento auxiliar designado por ciclo em faz-todas-as-solucoes.

Exercício 8 - um desafio com alguma dificuldade...

Uma solução para este problema poderá ser procurada no exemplo anterior sobre a colocação de 8 rainhas num tabuleiro de xadrez, com a técnica de *backtracking*.

Como pode verificar, a base do enunciado é um problema que normalmente se coloca no ensino básico! No entanto, agora, o problema surge com uma formulação mais geral e pretendendo-se uma solução automática e não manual com acontece no ensino básico.

- 1- Comece por analisar o problema proposto e procure, manualmente, a solução pedida.
- 2- Seguidamente, imagine que uma lista é composta pelo peso de cada uma das cenouras.



- 3- pretende-se que desenvolva um procedimento em Scheme que recebe uma "lista de cenouras" e um peso e devolve uma lista com as cenouras cujos pesos somados equivalem ao peso dado.

Veja uma hipótese de diálogo, com o procedimento pretendido.

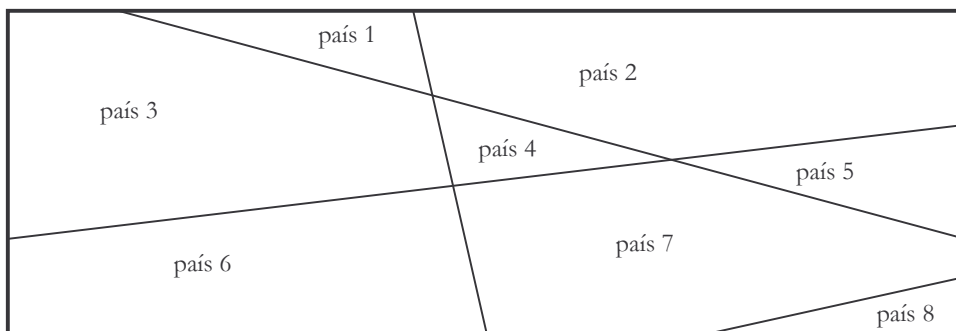
```
> (define lista-cenouras-1 (list 5 7 6 2))  
> (as-cenouras lista-cenouras-1 9)  
(7 2)  
> (as-cenouras lista-cenouras-1 10)  
()
```

Espaço para desenvolver e testar o procedimento *as-cenouras*.
O ecrã abre vazio.

Exercício 9 - mais um desafio com alguma dificuldade...

Projecto - 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*). 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 e 3.



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

A modelação das cores a utilizar 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 chamadas a *colorir-mapa*, procedimento que recebe dois argumentos, mais especificamente, duas listas, uma que representa o mapa e outra as 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 mapal paleta-cores)  
(c1 c2 c2 c1 c1 c1 c2 c1)
```

*Daqui se concluiria que, para colorir o mapa, bastariam apenas duas cores.
Concorda?*

De uma tentativa para colorir o mesmo mapa, com um conjunto de apenas uma cor, não resultaria solução possível.

```
> (colorir-mapa mapal '(azul))          (paleta apenas com a cor azul)
Poucas cores. Nao ha' solucao...
```

- 1- Faça uma abordagem *de-cima-para-baixo* ao problema proposto.
- 2- Escreva em Scheme o procedimento *colorir-mapa*, com base nos resultados da abordagem anterior.

Espaço para desenvolver e testar o programa *colorir-mapa*.
O ecrã abre vazio.

Convém ter em consideração que está perante um problema de média/elevada complexidade e que, por isso, não deverá desanimar se uma solução não surgir de imediato. Este é o tipo de problema que o pode acompanhar durante dias, levando-o a exercitar a sua capacidade criativa e os seus conhecimentos de programação.

Pista

provavelmente a dispensar se já trabalhou o exemplo das 8 rainhas ou o exercício das cenouras, indicados imediatamente antes.

Apresenta-se uma pista para atacar o problema, que deverá ser utilizada apenas em caso de necessidade. A tarefa colorir-mapa deve procurar uma solução para colorir os países de um mapa, a partir de uma paleta de cores. No final, a solução será uma lista com as cores escolhidas, lista que começa vazia e vai crescendo à medida que se encontra uma boa cor para cada um dos países.

A ideia que se pretende explorar consiste em atribuir a primeira cor da paleta ao primeiro país da lista. Assim, a lista de cores, com a solução a procurar, inicia-se com a cor atribuída ao primeiro país. Vamos agora passar ao país seguinte e procurar uma cor para ele. Por tentativas, começaremos pela primeira cor da paleta. Mas será que esta é uma boa cor para colorir o segundo país? Só o saberemos comparando-a com as cores de todos os países vizinhos, entretanto já coloridos. De facto, nesta altura, só o primeiro país está colorido e, ainda por cima, com uma cor igual, o que significa que, se o primeiro país faz parte da lista dos vizinhos do segundo país, esta cor não é boa. Por ordem, tenta-se uma nova cor da paleta, até encontrar uma boa cor para o segundo país. E, de forma idêntica, passáremos ao terceiro país, ao quarto, até não haver mais países para colorir. Nessa altura, existirá uma solução para colorir todos os países.

A parte mais delicada do problema corresponde à situação em que, ao tentar encontrar uma boa cor para um país, começando, como sempre, pela primeira cor da paleta, é alcançada e experimentada a última cor da paleta sem conseguir esse objectivo. Isto poderá ocorrer por uma de duas razões. Ou a paleta tem poucas cores e não existe uma solução para colorir os países sem que a mesma cor se encontre em dois países vizinhos. Ou ocorreu alguma má escolha de cores para países anteriores. Este caso, obriga a recuar ao país anterior, para, a seguir à cor que lhe foi anteriormente atribuída, procurar outra boa cor. Se esta for encontrada, avançamos novamente para o país seguinte. Se não for encontrada, recuamos ao país anterior. Se os recuos forem de tal ordem que se alcance o primeiro país, então não vale a pena fazer outras tentativas, pois já se pode concluir que as cores da paleta não chegam para colorir o mapa. Facilmente se entende que, recuando até ao primeiro país, que havia sido preenchido com a primeira cor da paleta, é inútil

procurar outra boa cor como se fez nos outros países. De facto, seria pura perda de tempo, uma vez que para este tipo de problema, qualquer cor da paleta é igualmente boa para o primeiro país.

Reflectindo um pouco mais sobre a ideia exposta, podemos então considerar que a tarefa colorir-mapa se desenvolve sobre uma tarefa a designar por procurar-solucao, cujos parâmetros estariam associados ao país que, naquele momento, se vai colorir (inteiro com valores de 1 a p , sendo este o número de países do mapa), a cor por onde se inicia a procura de uma boa cor (inteiro com valores de 1 a c , sendo este o número de cores na paleta), e a solução encontrada até ao momento (lista de inteiros com valores de 1 a c , sendo este o número de cores na paleta, que representam os índices das cores atribuídas aos países entretanto coloridos). Agora, procuremos identificar as sub-tarefas de procurar-solucao.

- Enquanto houver países no mapa para colorir, tomar a primeira cor da paleta e verificar se é uma boa cor para o país a colorir. Uma cor será boa para atribuir a um país se nenhum dos países vizinhos, já com cor atribuída, a utiliza. Sendo uma boa cor, retoma-se novamente a tarefa procurar-solucao com o país seguinte, a primeira cor da paleta e a solução a que se acrescentou a cor encontrada. Sendo uma má cor, também se retoma procurar-solucao, mas com o mesmo país, a cor seguinte da paleta, e a solução sem qualquer alteração.

- No entanto, pode atingir-se o fim da paleta, sem encontrar uma boa cor. Ou, de facto, não há solução possível, ou alguma das escolhas não foi bem feita. Assim, recua-se para a solução anterior, ou seja, retoma-se procurar-solucao com o país anterior, com a cor a partir da cor na altura escolhida, e com a solução a que se retirou a última cor. É claro que se tiver de recuar tanto que a alcance o primeiro país, então é porque não há mesmo solução possível.

- Quando não há mais países para colorir, a tarefa procura-solucao devolve a solução encontrada, convertendo os índices das cores nas designações indicadas na paleta.

Desta descrição ainda se identificam algumas sub-tarefas:

- cor-bou? para um país, que pode necessitar de cores-dos-vizinhos-ja-coloridos*
- cores-do-mapa para converter uma lista de índices de cores numa lista com as designações da paleta.*

Exercício 10

Projecto - Jogo das minas

O jogo das minas baseia-se no mapa de um terreno, onde existem 64 minas. No terreno há minas em variadas situações, desde as muito ricas, até às muito pobres. Esta situação é representada por um inteiro que se situa entre +99, para as muito ricas e -99 para as muito pobres. 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 programa.

Durante n percursos, um mineiro visita n minas, e vai somando 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 cada uma das n visitas/jogadas escolhendo a orientação da próxima viagem (Norte/Sul/Este/Oeste) e o programa escolhe aleatoriamente a distância, ou então, escolhe a distância e o programa escolhe aleatoriamente a orientação.

Análise parte de uma sessão do jogo que seria composta por 10 visitas/jogadas, de acordo com o primeiro argumento. O segundo e terceiro argumentos representam, respectivamente, a coluna e

```
> (minas 10 5 3)
```

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
5	1	45	2	-7	67	44	-90	34
6	80	-87	45	-38	58	52	-95	-73
7	77	91	-36	62	-5	9	-23	74
8	-35	74	95	-82	5	-7	9	44

$$\begin{array}{c} \text{N} \\ \text{O} - | - \text{E} \\ \text{S} \end{array}$$

Distancia calculada pelo computador: 2

Col. Lin.	1	2	3	4	5	6	7	8
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
5	1	45	2	-7	67	44	-90	34
6	80	-87	45	-38	58	52	-95	-73
7	77	91	-36	62	-5	9	-23	74
8	-35	74	95	-82	5	-7	9	44

$$\begin{array}{c} \text{N} \\ \text{O} - | - \text{E} \\ \text{S} \end{array}$$

Orientacao calculada pelo computador: N

Col. Lin.	1	2	3	4	5	6	7	8
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
5	1	45	2	-7	67	44	-90	34
6	80	-87	45	-38	58	52	-95	-73
7	77	91	-36	62	-5	9	-23	74
8	-35	74	95	-82	5	-7	9	44

$$\begin{array}{c} \text{N} \\ \text{O} - | - \text{E} \\ \text{S} \end{array}$$

Distancia calculada pelo computador: 6

```
(Quando "bate" numa parede,
  reflecte 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	
5	1	45	2	-7	67	44	-90	34	N
6	80	-87	45	-38	58	52	-95	-73	O - E
7	77	91	-36	62	-5	9	-23	74	S
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- Defina e implemente uma abstracção de dados que se adequê ao problema exposto.
- 2- Faça uma abordagem *de-cima-para-baixo* ao problema.
- 3- Escreva em Scheme o programa *minas*, com base nos resultados da abordagem anterior.

Espaço para desenvolver e testar o projecto proposto.
O ecrã abre vazio.

Exercício 11

Projecto - Lançamento de projecteis

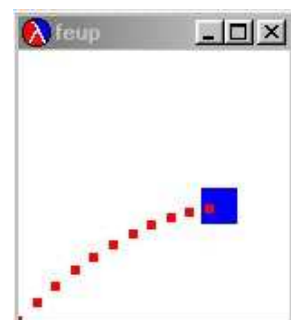
A simulação do lançamento de projecteis retoma um exercício já considerado num outro módulo, mas agora num ambiente de interacção com saída gráfica. A simulação do lançamento de projecteis inicia-se chamando o procedimento *projectil*, em que os dois primeiros parâmetros constituem, respectivamente, a largura e a altura da janela onde se visualizará o movimento do projectil. 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 projectil. 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 projectil, colocado no canto inferior esquerdo da janela, para tentar alcançar o *objecto-alvo*. A trajectória do projectil 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 "feup")

pretende lançar projectil (n = não; outro- sim):
s
Angulo de partida (0 < ang <= 90):
45
Velocidade inicial (> 0):
50
Boa-pontaria...
```

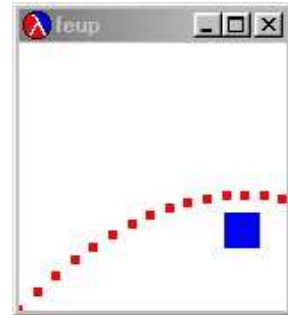
```
pretende lançar projectil (n = não; outro- sim):
s
Angulo de partida (0 < ang <= 90):
45
```



```
Velocidade inicial (> 0)):
50
```

```
Ma'-pontaria...
```

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

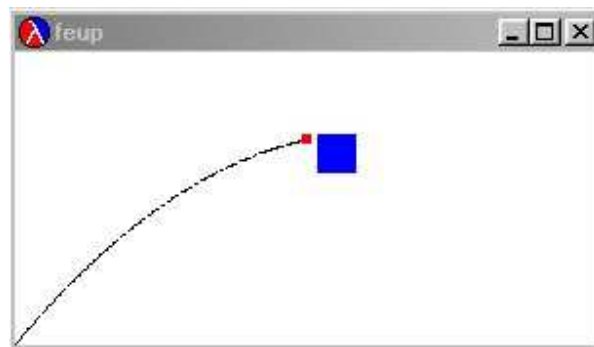


1- Desenvolva em Scheme o simulador descrito, tendo em conta que as suas principais tarefas poderão ser:

- criar uma janela gráfica de dimensão $L \times A$ (L e A são representados pelos dois primeiros parâmetros);
- perguntar se quer ou não lançar o projectil (*se não quer, termina o programa*);
- limpar a janela gráfica corrente (*não seria necessária da primeira vez, mas é mais fácil considerá-la sempre*);
- visualizar o *objecto-alvo*, num ponto da janela determinado aleatoriamente;
- pedir o ângulo ($0 < \text{ang} \leq 90$) e a velocidade ($\text{vel} > 0$) iniciais;
- lançar e visualizar o projectil a partir do canto inferior-esquerdo e, ao mesmo tempo, verificar se o *objecto-alvo* é atingido. Esta tarefa termina se o alvo for atingido ou se o projectil cair no solo sem o atingir;
- visualizar a mensagem de acordo com o facto de ter atingido ou não o *objecto-alvo*;
- retomar o ponto em que pergunta se quer ou não lançar o projectil.

Espaço para desenvolver e testar o projecto proposto.
Siga uma aproximação *de-cima-para-baixo*.
O ecrã abre vazio.

2- Numa segunda versão do programa, melhorar o simulador apresentado, introduzindo as seguintes alterações:



- A trajectória termina se o projectil ultrapassar o lado direito da janela;
- A deslocação do projectil não se faz por saltos, mas de uma forma mais contínua, como se indica na figura. Assim, repetidamente:
 - O projectil é visualizado numa posição e, passado um curto intervalo de tempo, é apagado;
 - Nesta posição, é visualizado um ponto que define o rasto do projectil;

- O projectil avança para uma nova posição da trajectória, muitíssimo perto da posição anterior;

Pista

Para obter o efeito de dinamismo, visualize o projectil com a cor '*temp*' (cor temporária- ver *Anexo B*)