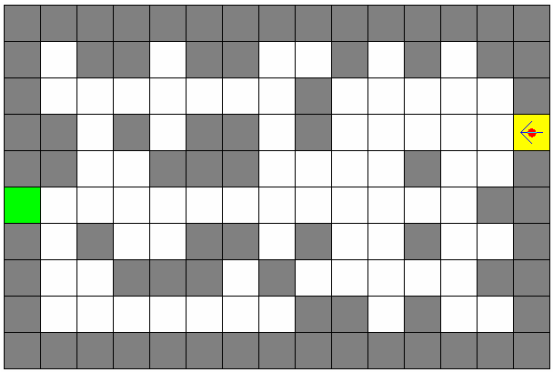
Módulo 1.7 - **Introdução à programação através de abstracções**

Convém entender, desde já, que a programação não se resume à codificação de um programa numa determinada linguagem. A programação é, sobretudo, uma actividade de reflexão, onde a linguagem de implementação nem é o elemento mais importante. Perante um problema, é necessário, em primeiro lugar, encontrar uma ideia que apoie a resolução e traçar uma estratégia para pôr essa ideia em movimento.
Neste Módulo, pretende-se mostrar que a programação é, de facto, uma actividade criativa, que se alimenta de ideias, onde, com um conhecimento mínimo de uma linguagem (neste caso, o Scheme) e através de algumas abstracções orientadas para os problemas, é possível programar situações já com alguma complexidade, nomeadamente, atravessar percursos e labirintos, muitas vezes com o acompanhamento de sons a ilustrar o que vai ocorrendo em cada momento.

Palavras-Chave
programação, produção de sons, abstracção audio, abordagem de-cima-para-baixo (*top-down*), abstracção naves, abstracção tabuleiro, abstracção janela gráfica, labirintos, procedimentos recursivos.

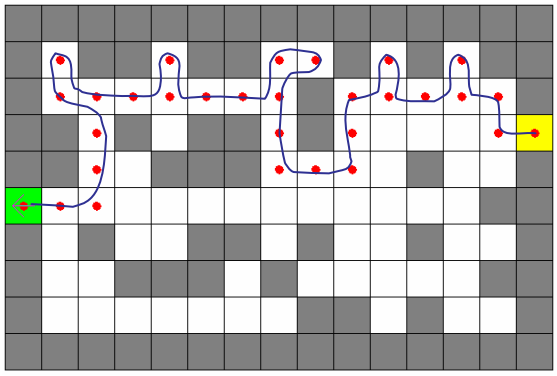
Programação é, em primeiro, a procura de ideias para resolver problemas

O problema que tem agora na sua frente é um labirinto, visualizado num tabuleiro. As células pintadas a cinzento representam obstáculos, paredes que não permitem a passagem. Pelo contrário, as células pintadas a branco mostram os caminhos livres.
A célula verde é o objectivo, pois é necessário conduzir uma nave até esta célula, nave inicialmente colocada na célula amarela e já a apontar para dentro do labirinto.

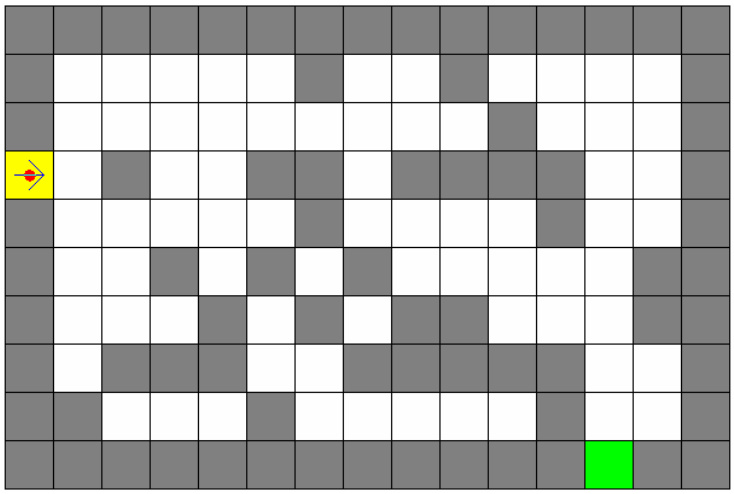


Pode constatar que há mais do que um percurso possível entre as duas células.
Como programar a nave para seguir um deles, mesmo que não seja o melhor?

Parece que alguém já descobriu uma ideia para levar a nave da célula amarela até à célula verde. Terá ainda pegado nessa ideia e, com a linguagem Scheme, programou a nave, tendo obtido o resultado que pode observar na figura. A linha azul foi desenhada para mostrar o percurso seguido pela nave, notando-se a passagem repetida em várias células.



Imagine que a nave só se descola em frente, mas pode rodar (90º) tanto para a direita como para a esquerda, ou seja, pode usar os seguintes três comandos:
frente, roda-dir e roda-esq.
O desafio que agora lhe é colocado é descobrir uma ideia própria, o que seria o ideal, para atravessar o labirinto ou então tentar descobrir a ideia utilizada anteriormente.
Utilize neste exercício a figura que se segue.

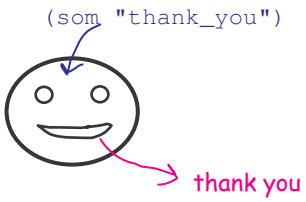


Se não conseguiu encontrar uma ideia para resolver o labirinto, adianta-se uma sugestão. No novo labirinto, "coloque-se em cima da nave", mantenha a sua mão direita sempre encostada à parede e vá fazendo avançar a nave, utilizando os três comandos referidos...

Se conseguiu encontrar uma ideia que lhe permita conduzir a nave até à célula verde, tanto melhor. Mas se não conseguiu, também não é motivo para grande preocupação, pois voltará a encontrar este problema, mais è frente, ainda neste Módulo. O importante é ficar a saber que, com um pequeno conjunto de comandos relacionados com naves, o labirinto pode ser atravessado, atacando o problema no domínio de uma linguagem próxima do problema concreto (controlo de naves: frente, roda, ...) e não no domínio de uma linguagem de programação que, de tão genérica, pouco tem a ver com esse problema. E isto acaba por resumir a importância das abstracções na programação, neste caso particular, a *abstracção naves*, que será, em breve, alvo da nossa atenção.

Produção de sons - *Abstracção audio*

Imagine que pretendia desenvolver um programa que expresse alguns dos seus resultados através de sons. Uma calculadora que diz o resultado em vez de usar o ecrã, uma nave que emite sons quando se desloca ou a personagem de um jogo que exprime emoções de contentamento ou agradecimento. Provavelmente, gostaria de ter acesso a alguma funcionalidade básica, que lhe permitisse este tipo de programação.



Observe a figura e tente perceber o funcionamento do procedimento som...
mas o Scheme não disponibiliza esta funcionalidade e tê-la-ia de desenvolver se alguém ainda não o tivesse feito.

O acesso ao procedimento *som* faz-se através de

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

Por agora, bastar-lhe-á saber que o ficheiro audio.scm, com a definição do procedimento som, se encontra no directório do Scheme designado por PLT\collects\user-feup.

Espaço para testar a *abstracção audio.*
O ecrã abre com a *abstracção audio* ou seja com

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

Experimente chamadas ao procedimento *som* como as indicadas

```
> (som "sorry")
> (som "direita")
> (som "anda1")
```

A funcionalidade desta abstracção resume-se a

```
> (som som-pretendido)
```

em que *som-pretendido* é uma cadeia de caracteres que define o som pretendido e pode tomar um dos valores entre:

```
"anda1"          "anda2"          "bomba1"
"chia1"          "chia2"
"curva"          "direita"        "e"           "esquerda"
"poing1"         "ri1"            "ri2"         "ri3"          "rotunda"
"sorry"          "thank_you"      "yaahooo"        "yeehaaa"    "yipee"

"0"   "1"  "2"  "3"  "4"  "5"  "6"  "7"  "8"  "9"
"10"  "11" "12" "13" "14" "15" "16" "17" "18" "19"
"20"  "30" "40" "50" "60" "70" "80" "90"
"100" - cem  "c100" - cento
"200"  "300" "400" "500" "600" "700" "800" "900" "1000"
```

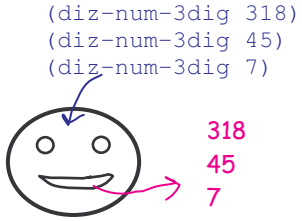
qualquer outro valor produzirá o som equivalente a "oh_no"

No Scheme que está a utilizar pode procurar o directório PLT\collects\user-feup e abrir alguns dos seus ficheiros, todos eles preparados não só por professores da feup, mas também por alunos...
Neste directório, estão colocados outros ficheiros que agrupam conjuntos de procedimentos orientados para a resolução de alguma tarefa. Ou seja, cada um desses ficheiros contém, normalmente, a linguagem própria de um certo problema. No caso dos sons, a linguagem resume-se a som que é o suficiente para pedir a produção de um som. Desta forma, o programador não tem que se preocupar com os pormenores de implementação da produção de sons e dizemos que estamos perante a abstracção som.
Bastará, para tal, escrever nos seus programas
`(require (lib "audio.scm" "user-feup"))`

Teve um primeiro contacto com a abstracção audio, mas mais à frente, vai encontrar a abstracção naves e com ela poderá controlar nave. Através de uma linguagem que terá a ver com naves, vai poder rodar e avançar as naves ou ver quem são os seus vizinhos, etc.

Exemplo 1 - dizer números até 999

O problema que agora tem entre mãos é programar o computador para dizer números inteiros de 1 a 999, como se mostra na figura.



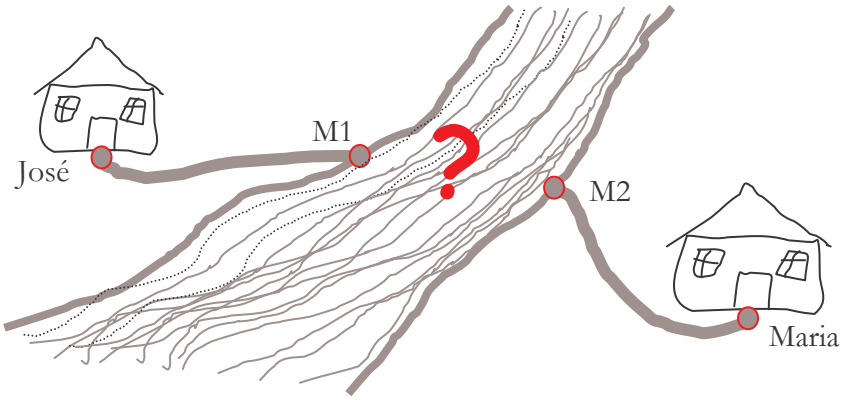
Observe a figura e tente encontrar uma justificação para o nome do procedimento e para o significado dos argumentos que esse procedimento recebe.

Como certamente verificou, a *abstracção audio* não contempla a produção dos necessários 999 sons numéricos, mas apenas: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 30, 40, 50, 60, 70, 80, 90, 100 (cem e cento), 200, 300, 400, 500, 600, 700, 800, 900 e 1000.
O desafio agora colocado é procurar, com estes sons numéricos básicos, adequadamente combinados, produzir os 999 sons diferentes.

Será possível compor os 999 sons numéricos com os sons disponibilizados pela abstracção audio?
Antes de responder, analise a forma como se dizem os números...
e verifique:
- *Uma certa falta de uniformidade ao dizer os números inferiores a 20... cada caso é um caso...*
- *Depois uma certa regularidade... até 99*
- *O número 100 é dito "cem", mas de 101 a 199 é dito "cento" ...*
- *Finalmente, de 200 a 999, verifica-se uma certa regularidade.*

Tem, pela frente, o problema de *dizer os números de 3 dígitos*, números que poderão ir de 1 a 999, situação que apresenta já um certo grau de complexidade. Nestes casos, a estratégia é tentar decompor o problema em problemas mais simples, começando por cima e ir descendo até encontrar problemas de fácil solução. Esta abordagem dá pelo nome *de-cima-para-baixo* (*top-down*).

Mas antes de se atacar o problema de *dizer os números de 3 dígitos*, comece com um exemplo, com o objectivo de ilustrar a abordagem *de-cima-para-baixo*.
Imagine que pretende fazer o percurso da casa do José até à casa da Maria, mas há um rio pelo meio.



A parte do percurso terrestre é para fazer a pé e a parte do rio, entre as margens M1 e M2, é mais complicada e ainda não se sabe muito bem como a fazer. Talvez a nado, talvez de barco... Mas para já, sem descer a pormenores, imagine que tem alguma maneira de atravessar o rio e sem se importar como, já pode escrever:

Problema - Fazer o percurso da casa do José até à casa da Maria

- ir a pé da casa do José até à margem M1
- sub-problema - atravessar o rio de M1 para M2
- ir a pé da casa da margem M2 até à casa da Maria

O problema inicial é assim considerado resolvido e é chegada a altura de focar a atenção apenas no sub-problema da travessia do rio.
Depois de reflectir um pouco, em vez de atravessar a nado, optou-se por fazer a travessia de barco, pois até há barcos para alugar junto às margens M1 e M2.

Atravessar o rio de M1 para M2
- alugar um barco
- entrar no barco em M1
- atravessar o rio de barco de M1 até M2
- sair do barco em M2

Se agora juntar tudo, obtém um algoritmo para o problema inicial.

Problema - Ir da casa do José à casa da Maria
- ir a pé da casa do José até à margem M1
- sub-problema - atravessar o rio de M1 para M2
    - alugar um barco
    - entrar no barco em M1
    - atravessar o rio de barco de M1 até M2
    - sair do barco em M2
- ir a pé da casa da margem M2 até à casa da Maria

Depois de ter entendido este exemplo, conseguirá determinar os sub-problemas do problema dizer números de 3 dígitos?
Dizer as centenas do número, dizer as dezenas do número e dizer as unidades do número serão os sub-problemas do problema dizer número de 3 dígitos?

Sem ver o que se segue, tente compor um algoritmo para este problema, imaginando que os 3 sub-problemas estão já resolvidos.

Apresenta-se uma hipótese de algoritmo para o problema *dizer número de 3 dígitos*, após a identificação dos sub-problemas: *dizer as centenas do número*, *dizer as dezenas do número* e *dizer as unidades do número*.

Problema - dizer número de 3 dígitos

- sub-problema - dizer as centenas do número
- se necessário, ligar centenas com dezenas do número
- sub-problema - dizer as dezenas do número...
    ...incluindo os casos especiais de 10 a 19
- se não for caso de 10 a 19
    - se necessário, ligar dezenas com unidades do número
    - sub-problema - dizer as unidades do número

Foque agora a atenção apenas no sub-problema identificado como sendo *dizer as centenas do número*. Um possível algoritmo será o que se apresenta de seguida.

sub-problema - dizer as centenas do número

- se for o caso especial do número 100
  - diz "cem"
  - se não, casos que concatenam com as dezenas ou as unidades...
    - separa dígito das centenas
    - se o dígito das centenas for 9, 8, ..., 1,
      assim dirá "900", "800", ..., "cento".

```
(define dizer-centenas
  (lambda (n)
    ; se for caso especial 100
    (if (= n 100)
        (som "100")
        ; se não, casos que concatenam com as dezenas ou as unidades
        ; separa digito das centenas
        (let ((c (quotient n 100)))
          ; se o dígito das centenas for 9, 8, ..., 1,
          ; assim dirá "900",  "800", ..., "cento".
          (cond
            ((= c 9)
             (som "900"))
            ((= c 8)
             (som "800"))
            ((= c 7)
             (som "700"))
            ((= c 6)
             (som "600"))
            ((= c 5)
             (som "500"))
            ((= c 4)
             (som "400"))
            ((= c 3)
             (som "300"))
            ((= c 2)
             (som "200"))
            ((= c 1)
             (som "c100")))))))
```

Espaço para testar o procedimento *dizer-centenas*.
O ecrã abre com `(require (lib "audio.scm" "user-feup"))`
e com este procedimento.
Experimente chamadas ao procedimento *dizer-centenas* como as indicadas

```
> (dizer-centenas 987)
> (dizer-centenas 100)
> (dizer-centenas 56)
> (dizer-centenas 3)
```

Em algumas destas chamadas, nada se ouve. Será falha do procedimento?

Passe agora para o sub-problema *dizer as dezenas do número*, tendo como primeiro objectivo encontrar um algoritmo adequado.

sub-problema - dizer as dezenas do número

- separa dígitos das dezenas e das unidades
- se casos especiais de 19 a 10...
  - se o dígito das unidades for 9, 8, ... ou 0
    assim dirá "19", "18", ... ou "10"
  - se não, casos que concatenam com as unidades, 20, 30, ... ou 90
    se o dígito das dezenas for 2, 3, ... ou 9

assim dirá "20", "30", ... ou "90".

```
(define dizer-dezenas
  (lambda (n)
    ; separa digitos das dezenas e das unidades
    (let ((d (remainder (quotient n 10) 10))
          (u (remainder n 10)))
      ; se casos especiais de 19 a 10...
      ; se o dígito das unidades for 9, 8, .., ou 0
      ; assim dirá "19", "18", ... ou "10"
      (if (= d 1)
          (cond
            ((= u 9)(som "19"))
            ((= u 8)(som "18"))
            ((= u 7)(som "17"))
            ((= u 6)(som "16"))
            ((= u 5)(som "15"))
            ((= u 4)(som "14"))
            ((= u 3)(som "13"))
            ((= u 2)(som "12"))
            ((= u 1)(som "11"))
            ((= u 0)(som "10")))
          ; se não, casos que concatenam com as unidades: 20, 30, ... ou 90
          ; se o dígito das dezenas for 2, 3, ... ou 9
          ; assim dirá "20", "30", ... ou "90".
          (cond
            ((= d 2)(som "20"))
            ((= d 3)(som "30"))
            ((= d 4)(som "40"))
            ((= d 5)(som "50"))
            ((= d 6)(som "60"))
            ((= d 7)(som "70"))
            ((= d 8)(som "80"))
            ((= d 9)(som "90")))))))
```

Espaço para testar o procedimento *dizer-dezenas*.
O ecrã abre com `(require (lib "audio.scm" "user-feup"))`
e com este procedimento.
Experimente chamadas ao procedimento *dizer-dezenas* como as indicadas

```
> (dizer-dezenas 987)
> (dizer-dezenas 100)
> (dizer-dezenas 16)
> (dizer-dezenas 10)
> (dizer-dezenas 3)
```

E agora tem pela frente o sub-problema *dizer as unidades do número*, e também se começa com um algoritmo para ele.

<u>sub-problema - dizer as unidades do número</u>

- separa o dígito das unidades
- se o dígito das unidades for 9, 8, ... ou 1
  assim dirá "9", "8", ... ou "1"

```
(define dizer-unidades
  (lambda (n)
    ; separa digito das unidades
    ... ... ... para completar
```

Espaço para completar e testar o procedimento *dizer-unidades*.
O ecrã abre com `(require (lib "audio.scm" "user-feup"))`
e com parte deste procedimento.

Alcançada solução para os três sub-problemas do problema inicial *dizer número de 3 dígitos*, recorde o algoritmo anteriormente definido.

Problema - dizer número de 3 dígitos

- sub-problema - dizer as centenas do número
- se necessário, ligar centenas com dezenas do número
- sub-problema - dizer as dezenas do número...
  ...incluindo os casos especiais de 10 a 19
- se não for caso de 10 a 19
  - se necessário, ligar dezenas com unidades do número
  - sub-problema - dizer as unidades do número

No algoritmo correspondente a dizer número de 3 dígitos, faltará apenas encontrar solução para as seguintes situações:
1- ligar centenas com dezenas do número
2- ligar dezenas com unidades do número
Estas duas situações resolvem-se através do som "e"...
Por exemplo, 523 será dito
5 - "quinhentos"
"e"
2 - "vinte"
"e"
3 - "três"
Como desafio, é-lhe agora pedido que especifique melhor estas situações, certamente em função dos valores das centenas, dezenas e unidades de cada número.

```
(define dizer-num-3-digitos
  (lambda (n)
    ; separar os dígitos
    ;    c - das centenas, d - dezenas e u - unidades
    (let ((c (quotient n 100))
          (d (remainder (quotient n 10) 10))
          (u (remainder n 10)))

      ; dizer as centenas do número
      (dizer-centenas n)
      ; se necessário, liga centenas com dezenas
      (if (and (> c 0) (> d 0))
          (som "e"))
      ; dizer as dezenas do número...
      ; incluindo os casos especiais de 10 a 19
      (dizer-dezenas n)

      ; se não for caso de 10 a 19
      ;      se necessário, liga dezenas com unidades do número
      ;      e dizer as unidades do número
      (if  (not (= d 1))
           (begin
             (if (and
                  (or (> c 0) (> d 0))
                  (> u 0))
                 (som "e"))
             (dizer-unidades n))))))
```

Espaço para testar o procedimento *dizer-num-3-digitos*.
O ecrã abre com `(require (lib "audio.scm" "user-feup"))`
com este procedimento e com os procedimentos auxiliares *dizer-centenas*, *dizer-dezenas* e *dizer-unidades*.

Nada se ouve com

```
> (dizer-num-3-digitos 0)
```

Fará sentido ser assim?
Se não for antes, certamente que encontrará resposta no exercício que se segue!

Exercício 1 - dizer números até 999999
Desenvolva um procedimento designado por *dizer-numero*, que diz números de 0 a 999999.
Comece por escrever um algoritmo, tentando identificar os sub-problemas do problema que lhe é colocado.
Pista: Certamente concluirá que dizer números de 3 dígitos é um sub-problema importante no algoritmo e para esse sub-problema tem já a solução apresentada no exemplo anterior.

Espaço para desenvolver e testar o procedimento pedido.
O ecrã abre com o procedimento *dizer-num-3-digitos*.

Exemplo 2 - dizer números em ordem decrescente... uma forma de introduzir a *recursividade*

Dizer os números em ordem decrescente, até atingir zero, é uma acção muito conhecida no lançamento de naves para o espaço: dez, nove, oito, sete, ..., um, *partiiiiida*.

Um algoritmo para este problema é apresentado.

Problema - diz números em ordem decrescente de *n* a zero
- se *n* é zero
  - diz "qualquer coisa que signifique fim de contagem"
  - se não
    - diz o número *n*
    - diz números em ordem decrescente de *n-1* a zero

Repare que, na parte final do algoritmo deste problema,
ele chama-se a si próprio!

Acha isto correcto?
Acha que este algoritmo terá fim?

Tente "executar manualmente" o algoritmo para n = 3
e depois dê a sua opinião.

```
(define diz-numeros-em-ordem-decrescente
  (lambda (n-partida)
    (if (zero? n-partida)
        (som "bomba1")
        (begin
          (dizer-num-3-digitos n-partida)
          (diz-numeros-em-ordem-decrescente (sub1 n-partida))))))
```

Espaço para testar o procedimento *diz-numeros-em-ordem-decrescente*.
O ecrã abre com `(require (lib "audio.scm" "user-feup"))`
com este procedimento e com o procedimento *dizer-num-3-digitos*.

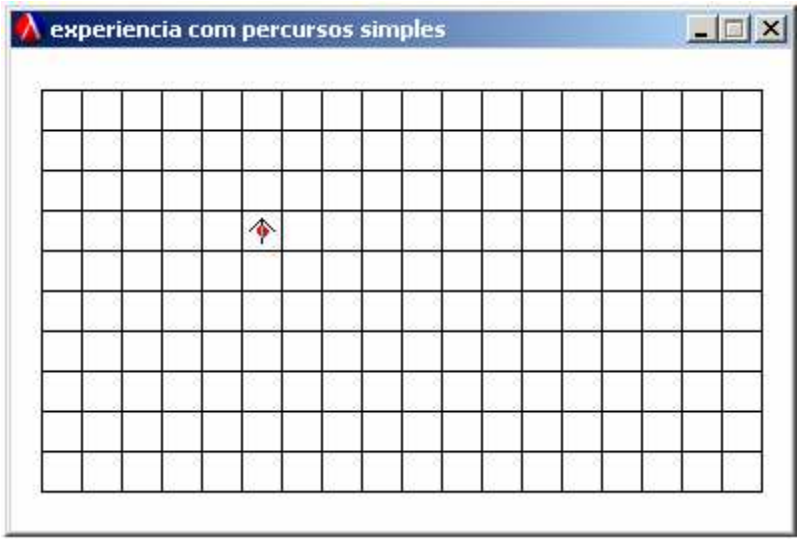
Um procedimento que se chama a si próprio
é um procedimento recursivo.
E diz-numeros-em-ordem-decrescente é um exemplo de um procedimento recursivo.

A recursividade, pela importância que tem em programação, é um tema que vai encontrar mais à frente...

A condução de naves

Vai agora envolver-se com a *abstracção naves* e treinar alguma da sua funcionalidade, usando comandos sobre uma nave.

Numa janela gráfica, com a designação *experiencia com percursos simples*, é apresentado um tabuleiro rectangular de 18 x 10 células, com uma nave na célula (5, 3) orientada para Norte (para cima).



Estando a nave na célula de coordenadas (5, 3),
1- identifique a célula de coordenadas (0, 0).
2- quais são as coordenadas da célula situada no canto inferior-direito

Espaço para um primeiro teste da *abstracção naves*.
O ecrã abre na situação indicada na figura.

Suponha que a nave da figura foi designada por *nave-k* e agora é convidado a conduzi-la, através dos comandos:

```
> (roda-dir nave vezes)
      roda para a direita um ângulo equivalente a vezes×90º
> (roda-esq nave vezes)
      roda para a esquerda um ângulo equivalente a vezes×90º
> (frente nave n)
      avança nave num percurso equivalente a n células.
      o percurso é marcado com a cor associada à nave.
      se tentar passar as fronteiras do tabuleiro, avança até à fronteira
      e devolve -1
      se tiver êxito, devolve o índice de cor da última célula ocupada
      pela nave, cor existente antes de ser alterada pelo rasto e pela
      cor da nave.
```

Se quiser saber como surgiu o tabuleiro e a nave, o que se recomenda, observe o código que agora se apresenta.

Para aceder à *abstracção tabuleiro*.

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

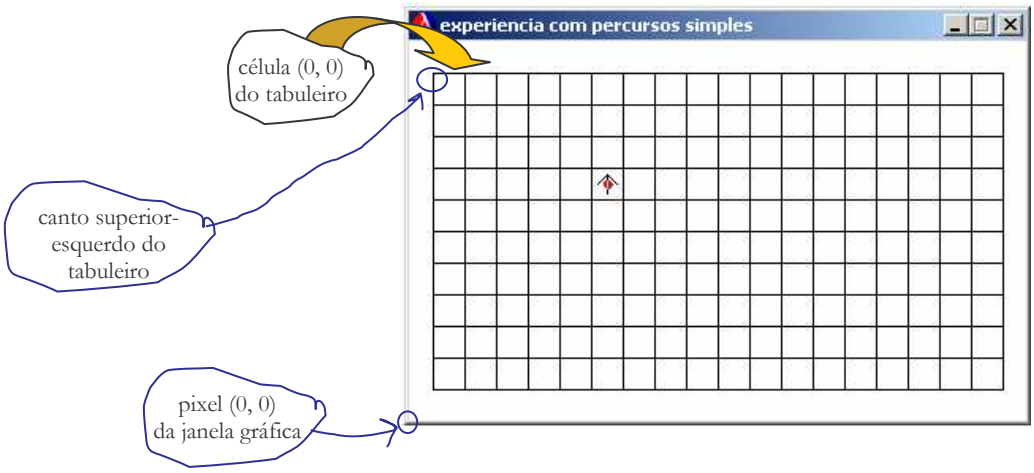
Esta abstracção permite abrir uma janela gráfica, onde será visualizado um tabuleiro.

Não fique com a ideia de que para usar a funcionalidade da janela gráfica precisa da abstracção tabuleiro.
Aliás, por este meio, apenas terá acesso a uma parte reduzida da funcionalidade associada à janela gráfica. Pouco mais do que criar uma janela...
Para ter um acesso completo àquela funcionalidade, o seu programa deverá incluir
`(require (lib "swgr.scm" "user-feup"))`
mas, para já, não tem que se preocupar com este assunto.

```
; -- prepara uma janela gráfica onde será visualizado um tabuleiro -------
(define largura-jan 390)  ; largura da janela em pixels
(define altura-jan 240)   ; altura da janela em pixels
(define titulo-jan "experiencia com percursos simples")  ; titulo da janela
;
; cria uma janela com a designação jan-tab, usando o procedimento janela
(define jan-tab (janela largura-jan altura-jan titulo-jan))
```

Agora, segue-se a preparação do tabuleiro.

```
; -- prepara um tabuleiro onde se deslocará a nave --------------
(define lado-cel 20)     ; número de pixels do lado de cada uma das células quadradas
(define num-cel-x 18)    ; número de células em x
(define num-cel-y 10)    ; número de células em y
(define org-x 15)        ; coordenadas, da janela gráfica, do ponto correspondente
(define org-y 220)       ; ao canto superior-esquerdo do tabuleiro
;
; cria um tabuleiro com a designação tab-teste, usando o procedimento tabuleiro
(define tab-teste (tabuleiro org-x org-y lado-cel num-cel-x  num-cel-y))
```



Finalmente, a preparação de naves com as quais irá continuar o teste desta abstracção.

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

(define cor-rasto 18)        ; vermelho - cor do rasto da nave
```

Tabela de cores
(definida na abstracção janela-gráfica)

índice	R	G	B	cor
0	0.0	0.0	0.0	preto
1	0.0	0.0	0.5	azul escuro
2	0.0	0.0	1.0	azul
3	0.0	0.5	0.0	verde escuro
4	0.0	0.5	0.5	cyan escuro
5	0.0	0.5	1.0	azul cyan
6	0.0	1.0	0.0	verde
7	0.0	1.0	0.5	verde cyan
8	0.0	1.0	1.0	cyan
9	0.5	0.0	0.0	vermelho escuro

10	0.5	0.0	0.5	magenta escuro
11	0.5	0.0	1.0	azul magenta
12	0.5	0.5	0.0	amarelo escuro
13	0.5	0.5	0.5	cinzento
14	0.5	0.5	1.0	azul cinza
15	0.5	1.0	0.0	verde amarelado
16	0.5	1.0	0.5	verde cinza
17	0.5	1.0	1.0	cyan pálido
18	1.0	0.0	0.0	vermelho
19	1.0	0.0	0.5	vermelho magenta
20	1.0	0.0	1.0	magenta
21	1.0	0.5	0.0	vermelho amarelado
22	1.0	0.5	0.5	vermelho cinza
23	1.0	0.5	1.0	magenta pálido
24	1.0	1.0	0.0	amarelo
25	1.0	1.0	0.5	amarelo pálido
26	1.0	1.0	1.0	branco

```
(define x-nave 5)                ; entre 0 e 17 - coordenada x inicial da nave
(define y-nave 3)                ; entre 0 e 9 - coordenada y inicial da nave
(define ori-inicial 'n) ; pode ser 'n, 's, 'e, 'o - orientação inicial da nave

(define t-resposta 200)         ; tempo de resposta da nave, em ms, ao rodar ou
                                ; ao avançar para a próxima célula

; cria uma nave com a designação nave-k, usando o procedimento cria-nave
(define nave-k (cria-nave tab-teste x-nave y-nave ori-inicial cor-rasto t-resposta))
```

Espaço para experimentar controlar a *nave-k*, criar outras naves, controlando-as através da funcionalidade da *abstracção naves*.
O ecrã abre na situação indicada na figura anterior.

```
> (cria-nave tab cel-x cel-y orient c-rasto t-resposta)
      cria uma nave e devolve uma estrutura equivalente à lista
        (tab cel-x cel-y orient c-rasto t-resposta)
      em que:
            orient pode ser 'n - norte, 's - sul, 'e - este, 'o - oeste

> (roda-dir nave vezes)
      roda para a direita um ângulo de vezes×90°
> (roda-esq nave vezes)
      roda para a esquerda um ângulo de vezes×90°
> (frente nave comp)
      avança nave num percurso equivalente a comp células.
      o percurso é marcado com a cor associada à nave.
      se tentar passar as fronteiras do tabuleiro, avança até à fronteira
      e devolve -1.
      se tiver êxito, devolve o índice de cor da última célula ocupada
      pela nave, cor antes de ser alterada pelo rasto e pela cor da nave.
> (nave-tab nave)
      selector que devolve o tabuleiro em que a nave foi criada
> (nave-pos-x nave)
      selector que devolve a coordenada x da nave, na posição actual
> (nave-pos-y nave)
      selector que devolve a coordenada y da nave, na posição actual
> (nave-ori nave)
      selector que devolve símbolo associado à orientação da nave
> (nave-cor-rasto nave)
      selector que devolve a cor do rasto associada à nave
> (nave-t-espera nave)
      selector que devolve o tempo de resposta da nave
> (ve-frente nave)
      devolve o índice de cor da célula à frente da nave ou
      devolve -1 se tentar ver fora do tabuleiro
> (ve-tras nave)
      devolve o índice de cor da célula atrás da nave ou
      devolve -1 se tentar ver fora do tabuleiro
> (ve-dir nave)
      devolve o índice de cor da célula à direita da nave ou
      devolve -1 se tentar ver fora do tabuleiro
> (ve-esq nave)
      devolve o índice de cor da célula à esquerda da nave ou
      devolve -1 se tentar ver fora do tabuleiro
```

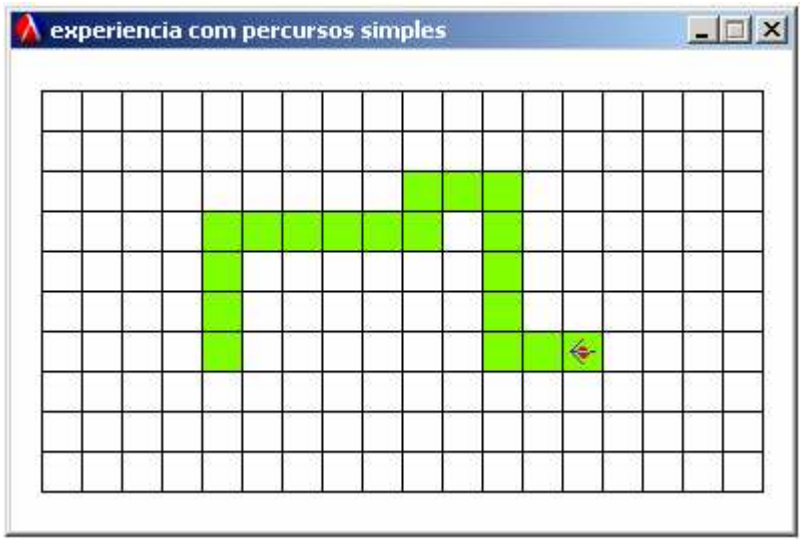
Certamente verificou a presença de sons, quando as naves rodavam ou se deslocavam. Esta funcionalidade está acessível, ao nível da abstracção naves, pela inclusão da abstracção audio, com

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

No entanto, se pretender usar a abstracção audio no seu programa, também deverá incluir nele este último require.

Exemplo 3 - condução de uma nave num percurso conhecido

Numa janela gráfica, é apresentado um tabuleiro com um pequeno percurso pintado a verde. Numa extremidade do percurso já se encontra uma nave.
Desenvolva o procedimento designado por *segue-o-percurso* que faz com que a nave atinja a outra extremidade do percurso, emitindo um som de alegria quando atinge esse objectivo.



A nave colocada numa das extremidades do percurso foi criada com

```
(define n (cria-nave tab-teste 13 6 'o 18 200))
```

Tente uma justificação para os vários argumentos utilizados na chamada cria-nave.

Neste caso, o problema resolve-se com um algoritmo muito simples, pois bastará conduzir a nave com comandos de frente e rodar, através de um percurso fixo, previamente conhecido.

```
; para ter acesso à abstracção audio
(require (lib "audio.scm" "user-feup"))
;
(define segue-o-percurso
  (lambda (nave)
    (frente nave 2)
    (roda-dir nave 1)
    (frente nave 4)
    (roda-esq nave 1)
    (frente nave 2)
    (roda-esq nave 1)
    (frente nave 1)
    (roda-dir nave 1)
    (frente nave 5)
    (roda-esq nave 1)
    (frente nave 3)
    (som "ri3")))
```

Espaço para testar o procedimento *segue-o-percurso*.
O ecrã abre com o tabuleiro já com o percurso pintado a verde e com a nave colocada na extremidade do lado direito e também com o procedimento a testar.
Como sabe, a nave foi criada com

```
(define n (cria-nave tab-teste 13 6 'o 18 200))
```

Experimente o comando:

```
> (segue-o-percurso n)
```

Aproveite para criar outras naves, com outras características, e aplique o mesmo comando...
Não feche, para já, este espaço, pois vão surgir mais algumas actividades sobre ele...

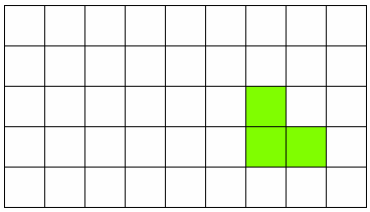
É agora convidado a analisar o código que visualiza, numa janela gráfica, um tabuleiro com um percurso pintado a verde e com uma nave colocada numa das extremidades desse percurso.

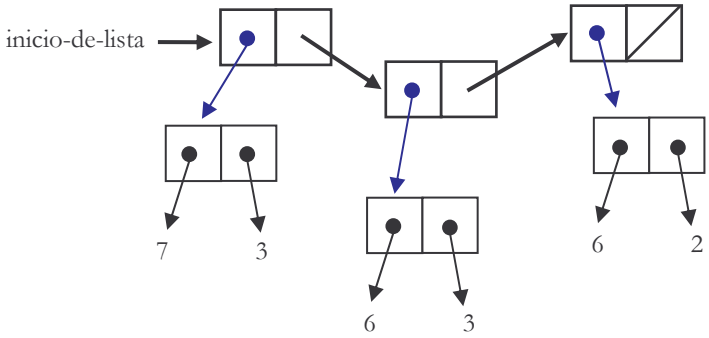
Numa primeira análise deste código, não se preocupe muito com a definição e visualização do percurso, operação que faz uso do procedimento celulas da abstracção tabuleiro. No entanto, pode já observar a relação entre os argumentos que surgem nas várias chamadas do procedimento cons do Scheme com as coordenadas das células que definem o percurso.

```
; acesso à abstracção tabuleiro
(require (lib "tabuleiro.scm" "user-feup"))

; -- preparação de uma janela gráfica onde será visualizado um tabuleiro -------
(define largura-jan 390)  ; largura da janela em pixels
(define altura-jan 240)   ; altura da janela em pixels
(define jan-tab (janela largura-jan altura-jan "experiencia com percursos simples"))

; -- preparação de um tabuleiro onde será visualizado um percurso --------------
(define lado-cel 20)    ; número de pixels do lado da célula quadrada
(define num-cel-x 18)   ; número de células em x
(define num-cel-y 10)   ; número de células em y
(define org-x 15)       ; coordenadas do canto inferior-esquerdo
(define org-y 220)      ; do tabuleiro
;
(define tab-teste (tabuleiro org-x org-y lado-cel num-cel-x  num-cel-y))

; ---- definição e visualização do percurso ------------
(define cor-perc 15)         ; verde amarelado

(celulas tab-teste
         (list (cons 13 6) (cons 12 6) (cons 11 6) (cons 11 5)
               (cons 11 4) (cons 11 3)
               (cons 11 2) (cons 10 2) (cons 9 2)
               (cons 9 3) (cons 8 3)(cons 7 3) (cons 6 3)
               (cons 5 3) (cons 4 3)(cons 4 3) (cons 4 4)
               (cons 4 5) (cons 4 6))
         'l 300 cor-perc)                ; 'l- limpa com cor-perc
;
; --- fim da definição do percurso

; acesso à abstracção naves e criação de uma nave
(require (lib "naves.scm" "user-feup"))
(define n (cria-nave tab-teste 13 6 'o 18 200))
```

Certamente que, em todo este código, apenas terá tido alguma dificuldade, aliás previsível, na parte relativa ao percurso. Mais propriamente, na parte em que se definem as células que constituem o percurso.

Um percurso é definido por uma sequência de células, pintadas com uma certa cor. A representação dessa sequência de células é feita através de uma lista (criada pelo procedimento list do Scheme) de elementos.
No exemplo que se segue, o percurso de três células apenas seria definido por `(list (cons 7 3) (cons 6 3) (cons 6 2))`.





Cada um destes elementos representa uma célula do percurso através das suas coordenadas reunidas num par (devolvido pelo procedimento cons do Scheme).
Não sendo para já muito necessário, este tema é tratado com alguma profundidade na Parte Abstracção de Dados desta obra.
Para uma consulta rápida pode utilizar o Anexo A.

Espaço para testar a funcionalidade da *abstracção tabuleiro*.
O ecrã abre com um tabuleiro já com um percurso pintado a verde e com uma nave colocada na extremidade do lado direito.
A nave foi criada com

```
(define n (cria-nave tab-teste 13 6 'o 18 200))
```

Sugere-se a alteração não só da cor do percurso, mas também da sua estrutura.

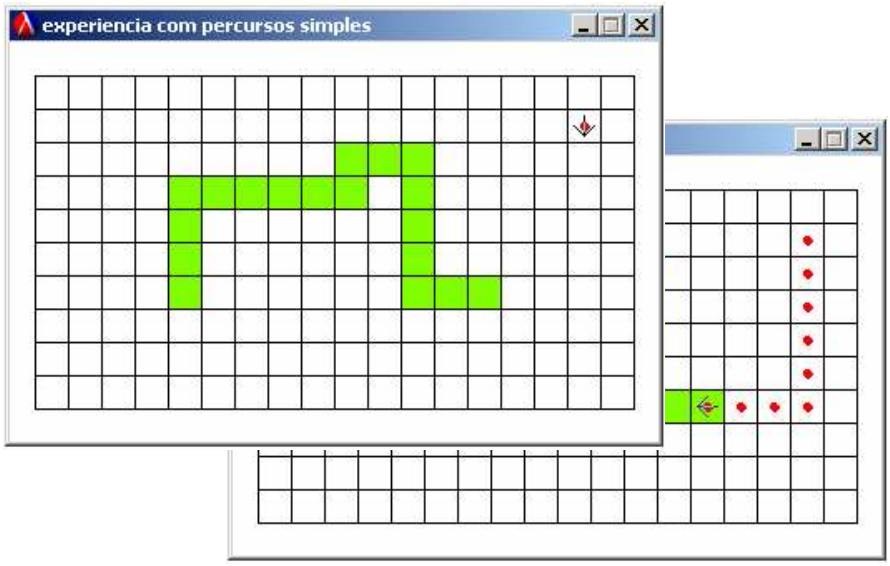
Apresenta-se agora um resumo da funcionalidade da *abstracção tabuleiro*.

```
> (tabuleiro org-x org-y lado num-cel-x num-cel-y)
      cria um tabuleiro e devolve uma estrutura equivalente à lista
        (org-x org-y lado num-cel-x num-cel-y)
> (celula tab x y simbolo cor-actual)
      altera o aspecto de uma célula, de acordo com cor-actual, em que
      simbolo pode ser:
simbolo    acção
  +          desenha uma cruz +
  x          desenha uma crux x
  c          desenha círculo O
  p          desenha ponto .
  n          desenha seta para norte
  s          desenha seta para sul
  e          desenha seta para este
  o          desenha seta para oeste
  l          limpa célula
> (indice-celula tab x y)
      fornece o índice de cor do ponto central da célula
> (limpa-tab tab cor-actual)
```

```
      limpa o tabuleiro com cor-actual
> (celulas tab lista simb t-espera cor-actual)
      aplica celula (indicada anteriormente) à lista de células
     (especificadas por pares de coordenadas x e y)
> (cel-x tabul)
      devolve num-cel-x
> (cel-y tabul)
      devolve num-cel-y

; limpa e janela são importados da abstracção janela gráfica
; e também disponibilizadas através da abstracção tabuleiro
> (limpa)
      limpa a janela gráfica onde se encontrará o tabuleiro
> (janela larg alt titulo)
      cria uma janela gráfica e devolve uma estrutura equivalente à
      lista (larg alt titulo)
```

Exercício 2 - deslocação de uma nave para outra célula

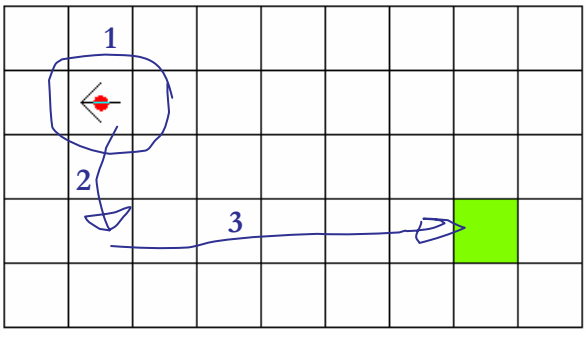
No contexto do exemplo anterior, não é garantido que a nave surja numa das extremidades do percurso.



Assim, pretende-se que desenvolva o procedimento *posiciona-nave* que tem como parâmetros: *nave*, *x-nova-posi* e *y-nova-posi*. A nave deverá ser deslocada para uma nova célula, cujas coordenadas são representadas pelos dois últimos parâmetros do procedimento.

Para este exercício, tem agora algumas pistas.
A figura mostra:
- as três operações que deverão ser programadas para levar a nave da posição actual à nova posição.
Identifique essas três operações.

- uma das quatro possíveis posições relativas da nave e a nova posição, ou seja, neste caso a nave está para cima e para a esquerda da nova posição
Identifique as restantes três posições relativas.



Estas pistas são uma ajuda para poder concretizar uma ideia de solução para o problema exposto. Depois precisa de uma algoritmo para essa ideia.

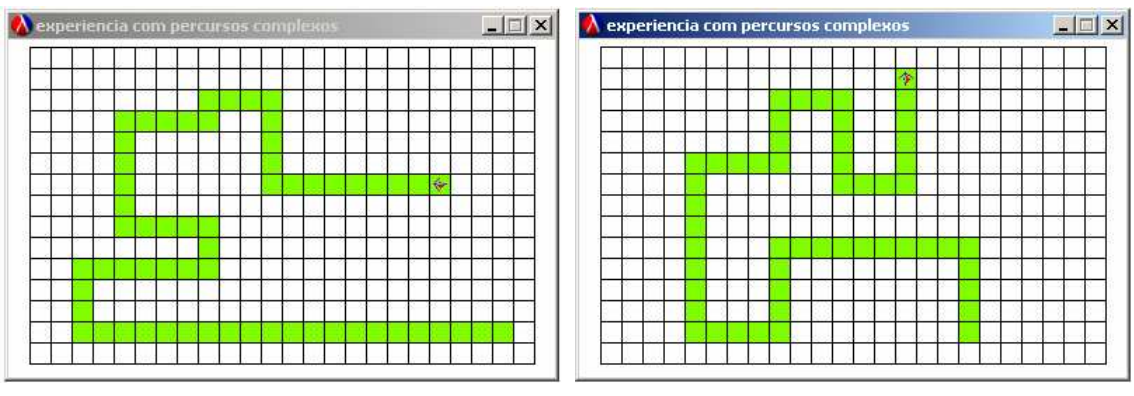
Espaço para desenvolver e testar o procedimento *posiciona-nave*.
O ecrã abre com o tabuleiro já com o percurso pintado a verde e com uma nave, designada por *n*, colocada, por exemplo, na célula (16, 1), a apontar para sul.
Experimente o procedimento desenvolvido, posicionando a nave no início do percurso ou noutras células à escolha.
Crie naves noutras posições e experimente o mesmo procedimento a partir delas.

Se tiver encontrado dificuldades inultrapassáveis no exercício anterior, sugere-se que analise uma possível solução no código que se segue:

```
(define posiciona-nave
  (lambda (nave x-nova-posicao y-nova-posicao)
    (let ((desloca-x (- x-nova-posicao (nave-pos-x nave)))
          (desloca-y (- y-nova-posicao (nave-pos-y nave))))
      (cond ((zero? desloca-y)
             ; não desloca na vertical
             )
            ((positive? desloca-y)
             (orienta-nave nave 's)
             (frente nave desloca-y))
            (else
             (orienta-nave nave 'n)
             (frente nave (abs desloca-y))))
      (cond ((zero? desloca-x)
             ; não desloca na horizontal
             )
            ((positive? desloca-x)
             (orienta-nave nave 'e)
             (frente nave desloca-x))
            (else
             (orienta-nave nave 'o)
             (frente nave (abs desloca-x)))))))

(define orienta-nave
  (lambda (nave nova-orientacao)
    (if (not (equal? (nave-ori nave) nova-orientacao))
        (begin
          (roda-dir nave 1)
          (orienta-nave nave nova-orientacao)))))
```

Exemplo 4 - condução de uma nave em percursos desconhecidos

O problema que agora se coloca apresenta-se com um nível de dificuldade bastante superior à dos anteriores, que apareciam sempre com o mesmo percurso. Na situação actual, o percurso é desconhecido, e a nave tem que ser programada para o atravessar. Só se sabe que a nave é colocada numa das extremidades do tal percurso, orientada ou não para avançar.
Imaginando que o percurso está pintado com uma cor fixa, a ideia é programar a nave para seguir esse percurso analisando a cor das células vizinhas, avançando para a célula que for verde...
A figura mostra dois percursos possíveis entre tantos outros.



Que funcionalidade das naves utilizaria para fazer a análise das células vizinhas?
Como faria para detectar o fim do percurso?

Sabe-se que a nave está colocada no início do percurso. Então, um algoritmo possível poderá ser:

Problema - segue percurso desconhecido

- sub-problema - determina nova direcção da nave
- se chegou ao fim
  - expressa o seu contentamento, através de um som adequado
  - se não
    - orienta a nave de acordo com a nova direcção
    - avança a nave para a próxima célula
    - segue percurso desconhecido

Tem aqui um algoritmo recursivo. Porquê?

sub-problema - determina nova direcção da nave

- se a célula em frente é da cor cor-percurso, devolve 0
- se não,
  se a célula da direita é da cor cor-percurso, devolve 1
- se não,
  se a célula de trás é da cor cor-percurso, devolve 2
- se não,
  se a célula da esquerda é da cor cor-percurso, devolve 3
- se não, devolve -1

Tente explicar a escolha dos valores devolvidos no algoritmo
"determina nova direcção da nave",
sabendo que a nave pode rodar para a direita ou para a esquerda de ângulos de 90º.

Finalmente, a codificação do algoritmo do sub-problema, seguida pela codificação do algoritmo do problema designado por *segue percurso desconhecido.*

```
; para produção de sons...
(require (lib "audio.scm" "user-feup"))

; define nova direcção para a nave em função do percurso
; atenção aos valores devolvidos por este procedimento...
(define nova-direccao
  (lambda (nave cor-percurso)
    (cond ((= (ve-frente nave) cor-percurso) 0)
          ((= (ve-dir nave) cor-percurso) 1)
          ((= (ve-tras nave) cor-percurso) 2)
          ((= (ve-esq nave) cor-percurso) 3)
          (else -1))))

(define segue-percurso-desconhecido
  (lambda (nave cor-percurso)
    (let ((direccao (nova-direccao nave cor-percurso)))
      (if (= direccao -1)
          (som "ri1")
          (begin
            (cond
              ((= direccao 1)(roda-dir nave 1))
              ((= direccao 2)(roda-dir nave 2))
              ((= direccao 3)(roda-esq nave 1)))
            (frente nave 1)
            (segue-percurso-desconhecido nave cor-percurso))))))
```

Espaço para testar o procedimento *segue-percurso-desconhecido*.
O ecrã abre com o tabuleiro já com um percurso pintado e com a nave *n* colocada no início e com o procedimento a testar.
Verifique que o procedimento conduz a nave até ao fim do percurso, com a chamada

```
> (segue-percurso-desconhecido n cor-per)
```

Pode experimentar alterar a posição da nave, deslocando-a com os comandos adequados, ou então tentar alterar o percurso, fazendo modificações na lista que o define, e verificar se o procedimento continua a funcionar correctamente.

Exercício 3 - condução de uma nave em percursos desconhecidos e com cruzamentos

A nave está colocada no início do percurso pintado a verde (cor de índice 15) e deve ser programada para atingir a célula final, pintada a cinzento (cor de índice 13).
Esta nave segue normalmente em frente, excepto quando encontra uma célula azul (cor de índice 14), sinal colocado no percurso para indicar que a nave deve rodar à direita, ou uma célula vermelha (cor de índice 22), sinal para rodar à esquerda.

Coloque-se "em cima da nave" e faça o percurso, obedecendo aos sinais indicados.
Chega ou não à célula pintada a cinzento?

Agora, pretende-se que desenvolva o procedimento *segue-percurso-com-sinais* que tem *nave* como parâmetro único.

O procedimento agora pedido, segue-percurso-com-sinais, não tem como parâmetro a cor do percurso que a nave deverá seguir. Como obterá este procedimento essa informação?

Espaço para desenvolver e testar o procedimento *segue-percurso-com-sinais*.
O ecrã abre com o tabuleiro já com o percurso pintado como se indica na figura e com a nave *n* colocada no início.
Verifique se o procedimento que desenvolveu conduz a nave até ao fim do percurso.

Pode também experimentar alterar o percurso, por exemplo, colocando os sinais noutras células e verificar se o procedimento continua a funcionar correctamente.

Projectos com labirintos

Para esta etapa são-lhe apresentadas duas propostas de projectos, ambas já com um nível de dificuldade bastante elevado.

Projecto 1 - Labirinto

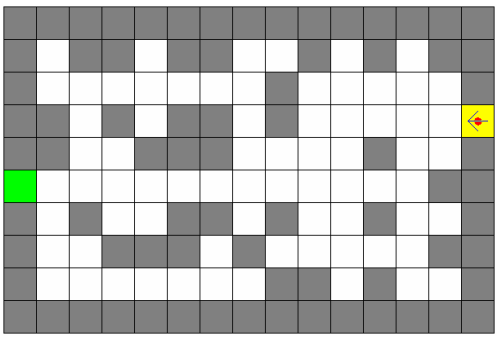
Num tabuleiro é gerado, aleatoriamente, um labirinto. Repare que a fronteira do tabuleiro é constituída por um muro, no qual há apenas duas aberturas devidamente identificadas por cores distintas. A entrada, pintada a amarelo, onde é colocada uma nave a apontar para dentro do labirinto, e a saída, pintada a verde.
O número de obstáculos que formam o interior do labirinto é definido percentualmente em relação à ocupação. Por exemplo, para uma ocupação de 100%, o labirinto ficaria completamente bloqueado, sem qualquer célula por onde a nave pudesse transitar.

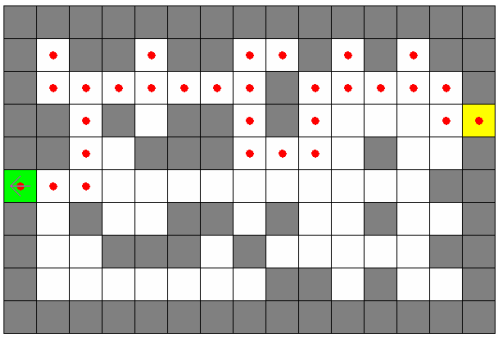
Para preenche aleatoriamente o interior do labirinto é aceitável determinar um x e um y aleatoriamente e verificar se a célula (x, y) ainda não está ocupada.
Se já estiver ocupada, o melhor será gerar outro x e outro y aleatoriamente...

Reflicta um pouco e diga, em que momento, esta forma de preenchimento poderá apresentar algum problema de funcionamento.
Consegue indicar outra forma de preenchimento que evite o referido problema, mesmo que não saiba, para já, como a programar em Scheme?

Também se aceita que alguns labirintos propostos não sejam muito interessantes, por não darem qualquer hipótese à nave. Por exemplo, a célula imediatamente após a entrada ou imediatamente antes da saída estar bloqueada. Nestes casos, para os quais a nave também deverá estar preparada, podem ser postos de lado, recomeçando e fazendo aparecer novo labirinto.

Analise com atenção as figuras que seguem.

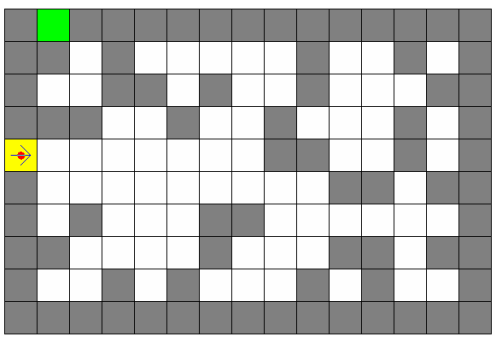


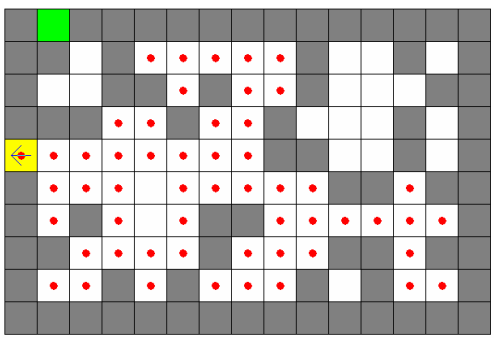


Neste caso, a nave encontra um caminho entre a entrada e a saída.
Observe o "ar" de satisfação com que a nave acaba...

```
> (nave-no-labirinto n)
Desta vez consegui atingir o objectivo
Sou o Maior... ou quase...
>
```

No labirinto que se segue, o insucesso era perfeitamente previsível...





e a expressão final surge adequada.

Espaço para desenvolver e testar o Projecto apresentado. O ecrã abre vazio. Analise com muita atenção o enunciado e se detectar alguma falha na especificação, sugira uma especificação adicional. Tente resolver o problema seguindo uma abordagem *de-cima-para-baixo*. Vá testando as soluções para os sub-problemas que identificar, até chegar à solução do problema proposto...

Problema 2 - Perseguição

```
> (nave-no-labirinto n)
Voltei ao início!!!...
Não consegui sair!!!
Bem me esforcei (só às vezes), mas sem êxito!!!
E não vale rir! Será que conseguiriam melhor que eu?
>
```

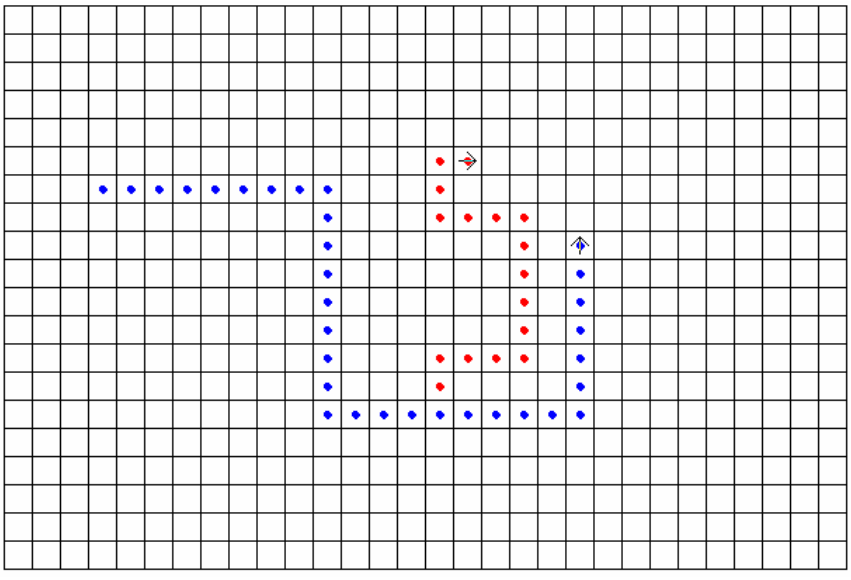
Num tabuleiro são colocadas 2 naves! Ambas dão passos para a frente de comprimento 1. Uma das naves é muito mais lenta do que outra, ou seja, enquanto que a mais lenta dá um passo a outra consegue dar dois (mas sempre um de cada vez...). As duas naves são colocadas num tabuleiro, em células calculadas aleatoriamente e também com orientações aleatórias.
A nave lenta faz um percurso aleatório, como se disse, de passos unitários. Isto significa que a sua orientação, antes de cada passo, é recalculada...

A nave mais rápida tem por objectivo apanhar a nave lenta, tendo por base as seguintes regras:

1- de acordo com uma estratégia a imaginar, começa por tentar encontrar o rasto da nave lenta
2- encontrado o rasto, segue-o numa direcção tentando apanhar a nave lenta.
3- se verificar, em algum momento, que a direcção escolhida no passo anterior não foi a melhor, inverte o seguinte de perseguição...

Como acaba a perseguição?

1- Com a vitória da nave lenta, se ao fim de *n* passos (*n* poderá ser um parâmetro, mas um valor razoável será 100) não tiver sido apanhada. E o programa termina com a mensagem:
   Ganhou a nave lenta...
2- Com a vitória da nave rápida, se conseguir apanhar a nave lenta ainda em andamento. E o programa termina com a mensagem:
   Ganhou a nave rápida...



Na figura, a nave lenta apresenta o rasto vermelho e a nave rápida o rasto azul.

Espaço para desenvolver e testar o Projecto apresentado.
O ecrã abre vazio.
Analise com muita atenção o enunciado apresentado e se detectar alguma falha na especificação, sugira uma especificação adicional.
Tente resolver o problema seguindo uma abordagem *de-cima-para-baixo*.
Vá testando as soluções para os sub-problemas que identificar, até chegar à solução do problema proposto...

Dois pontos prévios:
1-
Levanta-se aqui um pequeno problema que convém resolver de imediato. No tabuleiro onde decorre a perseguição, as células, em cada momento, poderão apresentar uma das seguintes situações:
- não preenchida (índice 26 - branco)
- preenchida com a cor do rasto da nave lenta (por exemplo, índice 18 - vermelho)
- preenchida com a cor do rasto da nave rápida (por exemplo, índice 2 - azul)
- preenchida com a nave lenta, o que corresponde à sobreposição do rasto desta nave com a cor 'temp. Em situações como esta, vá à tabela de cores e tome nota dos coeficientes RGB da cor do rasto. Depois, os coeficientes 0.0 troca por 1.0, e 1.0 troca por 0.0 e mantém os coeficientes 0.5. No caso do vermelho, índice 18, os coeficientes são 1.0, 0.0, 0.0. Trocando como indicado, aparece, 0.0, 1.0, 1.0 que corresponde ao índice 8. Ou seja, o índice 8 corresponde à célula onde se encontra a nave com o rasto de índice 18.

Já agora, como exercício, qual será o índice da célula onde se encontra a nave rápida de cor correspondente ao índice 2?
Será a cor de índice 24?

Que índice de cor não deve ser usada como rasto de nave, por não permitir distinguir entre
uma célula com rasto
e
uma célula com rasto e nave?

2-
Pense numa estratégia para permitir a nave rápida encontrar um ponto da trajectória da nave lenta.

E encontrado esse ponto, poderá ajudar a nave rápida a tomar a melhor decisão, ou seja, a decisão que vá na direcção da nave lenta?