Capítulo 1- Breve introdução à linguagem Scheme

Números
Expressões
Símbolos - Forma especial define
Procedimentos Compostos - Exemplo: Aposta do totoloto
Forma especial let
Estruturas de Selecção

Exercícios e Exemplos

A linguagem *Scheme*, apesar de poderosa, caracteriza-se por ter uma sintaxe simples, com poucas regras, que não exige nem muito tempo *(do aluno)* nem muito espaço *(de manuais)*, o que já não se poderá dizer da generalidade das restantes linguagens. Assim, o esforço e atenção podem incidir, sobretudo, sobre a *programação*. Programar significa encontrar ideias para resolver problemas, pô-las em movimento, com implementações elegantes, legíveis e, tanto quanto possível, eficientes, em tempo *(de cálculo)* e em espaço *(de memória*). Incentivar os alunos a criar ideias é o grande objectivo e, neste sentido, o *Scheme* integra-se de uma forma excepcional neste plano. O facto de, contrariamente à generalidade das outras linguagens, não oferecer funcionalidades para certas tarefas, como sejam as estruturas habituais de controlo *(como os ciclos)*, ou a introdução tardia do conceito de afectação *(dados mutáveis)*, o que é visto inicialmente como graves limitações da linguagem, traduz-se num bem precioso, pois este tipo de limitações acaba por retirar ao aluno a hipótese de recorrer à *programação remendada*, induzindo a necessidade de soluções estruturadas, legíveis e propícias a futuras alterações.
Neste capítulo, faz-se uma breve introdução à linguagem *Scheme*, considerando os *números*, *expressões*, *símbolos*, *definição de procedimentos* e *estruturas de selecção*. Neste percurso, são ainda consideradas as formas especiais *define* e *let* do *Scheme*: Para além deste capítulo, a sintaxe do *Scheme* resumir-se-á, fundamentalmente, a notas pontuais ao longo de outros capítulos e a referências a um dos anexo *(Anexo A)*.

1- Números

Os *Números* são dados primitivos disponibilizados pela a linguagem *Scheme*, e representam e devolvem o próprio valor.

Inteiros

```
10  +10  -350 1000
```

Reais

```
10.0  10.   10.350   35E2   -3.57E-3
```

A partir deste momento, já é possível começar a dialogar com o computador, através da linguagem *Scheme*. Para isso, torna-se necessário pôr em funcionamento um programa que entende a sintaxe desta linguagem, e que poderá ser um *interpretador* de *Scheme*. O interpretador coloca-se disponível para responder ao utilizador, pois apresenta-se num ciclo permanente de *leitura-cálculo-escrita*, que facilmente se identifica através de um caracter especial[1], como se indica de seguida.

```
↦
```

Isto significa que o *Scheme* está preparado para ler uma expressão, a fornecer pelo utilizador.

```
↦10
```

A expressão, neste caso o número 10, é calculada e o respectivo valor é escrito ou visualizado no ecrã.

```
↦10
10
```

Vejamos outros exemplos.

```
↦+10
10
↦35E2                    ; representação exponencial: 35E2 = 35 * 10^2
3500.0
↦35E-2                   ; 35E-2 = 35 * 10^-2
0.35
```

2- Expressões

Para processamento dos dados numéricos, a linguagem *Scheme* disponibiliza vários *Procedimentos Primitivos*, como, por exemplo, os operadores aritméticos:

```
+  adição
-  subtracção
*  multiplicação
/  divisão
```

Nesta linguagem, as *Expressões* utilizam uma notação *pré-fixa*, em que o operador aparece antes dos operandos.

```
↦(+ 10 25)
35
↦(+ 10 25 37)
72
↦(- 10 15)
-5
↦(- 10 15 5)
-10
```

Estes exemplos são *Expressões Compostas*[2], e apresentam-se rodeadas por parêntesis; em primeiro aparece o *operador* (*procedimento primitivo*), seguido dos *operandos.* Estes parêntesis indicam ao *Scheme* que se trata da chamada de um procedimento, os quais, nestes exemplos, são todos procedimentos primitivos, uma vez que se integram na definição da linguagem.

---

[1] Sem recorrer à tradução, este caracter será designado por *prompt*, e no livro representar-se-á por ↦

[2] Por sua vez, os números poderão ser considerados como *Expressões Simples*

A regra de cálculo de um procedimento primitivo segue os passos seguintes:

⇒ Calcular os operandos da expressão;

⇒ Aplicar o procedimento primitivo aos operandos calculados.

```
↦(* 2 6 30)
360
```

Do cálculo dos operandos 2, 6, e 30, resultam, respectivamente, os valores 2, 6 e 30. Aplicando-lhes o operador *, o valor resultante será 360.

```
↦(+ 12 (- 45 38))          ; passo intermédio: (+ 12 7)
19
```

Esta expressão apresenta dois operandos que devem ser previamente calculados, antes de se aplicar o procedimento. O primeiro operando devolve imediatamente 12, pois é um número. Mas o segundo, (- 45 48), requer um pouco mais de trabalho. Este operando é, ele próprio, uma expressão composta, exigindo, por seu turno, a aplicação da regra de cálculo de um procedimento primitivo: o cálculo dos seus operandos, 45 e 38, seguido da aplicação do procedimento primitivo -, de onde resulta o valor 7.
Finalmente, estando calculados os operandos da expressão inicial, 12 e 7, será aplicado o procedimento primitivo +.

```
↦(* (+ 34 30) 6)
384
```

Exercício 1.1
Indicar como reagiria o Scheme às expressões

```
↦(+ 23 (* 3 5 3) (- 22 2 34))
?
↦(* (* 34 (- 4 5)) (+ 12 (/ 15 3)))
?
```

Quando uma expressão composta se torna de leitura complicada, o melhor será representá-la de uma forma mais adequada, alinhando verticalmente os operandos de cada um dos operadores da expressão[3]. A última das três expressões poderá apresentar assim uma forma muito mais legível.

```
↦(*
     (* 34
        (- 4 5))
     (+ 12
        (/ 15 3)))
-578.0
```

3- Símbolos

É necessário, frequentemente, definir novos objectos, dando-lhes nomes. Assim é, por exemplo, quando se pretende definir novos procedimentos ou, simplesmente, criar um valor simbólico.

```
↦(define lado-da-casa 16)      ; lado-da-casa definido e ligado ao valor 16
lado-da-casa
↦ lado-da-casa
16
```

[3] Este tipo de formatação é normalmente disponibilizado quando se trabalha no computador com o Scheme

Acabámos de utilizar uma das *Formas Especiais* do Scheme, *define*, que se representa agora na forma genérica.

```
(define símbolo expressão)
```

Cada forma especial, como o próprio nome indica, tem uma regra de cálculo especial. No caso de *define*, a regra é a seguinte:

⇒ Calcular *expressão*;
⇒ Ligar ou associar o valor de *expressão* a *símbolo*.

Quando se associa um nome a um dado numérico, é porque se deseja trabalhar a um nível de abstracção superior. Por exemplo, o número *16* poderá significar muita coisa e, por isso, pouco nos dirá quando aparece numa expressão. Mas se, pelo contrário, em vez de um número surge um nome, *lado-da-casa*, num certo contexto, poder-se-á imaginar do que se trata. Isto é o princípio da *Abstracção de Dados* que, pela sua enorme importância em programação, será retomada noutro capítulo.

O símbolo *lado-da-casa* ficou *ligado* ou *associado* ao valor 16 e poderá ser incorporado em expressões.

```
↦(+ lado-da-casa 3)
19
↦(* 3 lado-da-casa -1)
-48
```

Quanto aos símbolos, a regra de cálculo é:

⇒ Se um símbolo está ligado a um valor, é devolvido esse valor;
⇒ Caso contrário, é assinalada uma mensagem de erro.

```
↦(* 3 lado-da-moradia)
*: unbound variable: lado-da-moradia      ; pois o símbolo não chegou a ser
                                          ; associado a qualquer valor
```

4- Procedimentos Compostos

Para além dos procedimentos primitivos, o programador poderá criar os seus próprios procedimentos, os *procedimentos compostos*. Em vez de ligar um símbolo a um número, como vimos, agora associa-se um símbolo a uma tarefa específica. Assim, quando o programador pretende executar essa tarefa, não terá mais do que chamá-la através do símbolo respectivo. O programador apenas se preocupa em conhecer esse símbolo e o que faz a tarefa associada; os pormenores podem ficar escondidos. Isto é o princípio da *Abstracção Procedimental*, certamente, uma das principais potencialidades das linguagens de programação.
Antes das formas genéricas utilizadas na definição e chamada de procedimentos compostos, optou-se pela apresentação de alguns exemplos que deverão ser analisados com atenção.

Vamos definir o procedimento *quadrado*, que toma um valor e devolve o quadrado desse valor.

```
(define quadrado                  ; define o nome do procedimento
  (lambda (x)                     ; x é o único parâmetro do procedimento
    (* x x)))
```

O símbolo *quadrado* passou a estar associado a uma tarefa específica: calcular e devolver o quadrado de um número. Na definição deste procedimento, o valor a elevar ao quadrado é representado simbolicamente por *x*. Diz-se, por isso, que *x* é um *parâmetro* do procedimento *quadrado*, neste caso, o único parâmetro.

Um procedimento só realiza a tarefa que lhe está associada quando é chamado.

```
↦(quadrado 5)                    ; chamada do procedimento quadrado
25
↦(quadrado (+ 1 3))              ; outra chamada do procedimento quadrado
16
```

A chamada do procedimento *quadrado* faz-se colocando entre parêntesis o nome do procedimento, seguido do valor que se pretende associar ao parâmetro. Na chamada, o *valor actual* associado ao parâmetro é geralmente designado por *argumento*.

Exercício 1.2
Indicar como reagiria o Scheme à expressão

```
↦(quadrado (quadrado 5))
?
```

Exercício 1.3
Escrever em *Scheme* o procedimento *soma-os-dois-e-tira-5* que, de acordo com o próprio nome, recebe dois valores como argumentos, calcula a soma dos dois e retira-lhe 5.

```
↦(soma-dois-e-tira-5 10 3)       ; o procedimento espera 2 argumentos
8
↦(soma-dois-e-tira-5 10 -3)
2
```

Depois destes exemplos, surge a forma genérica para definir um procedimento:

```
(define nome-do-procedimento
   (lambda (parâmetro-1 parâmetro-2 ...)

      corpo-do-procedimento))
```

Como *define*, o procedimento primitivo *lambda* também é uma forma especial do *Scheme*, cuja regra de cálculo é:

⇒ Os símbolos que se encontram a seguir a *lambda*, entre parêntesis, representam valores genérico e são designados por *parâmetros*.

⇒ Não há lugar a cálculos, apenas é devolvido um procedimento com os parâmetros referidos e cujo corpo corresponde a *corpo-do-procedimento*.

Verificar que *nome-do-procedimento* é um símbolo que fica associado ao valor da expressão, neste caso, a forma especial *lambda*. Ou seja, *nome-do-procedimento* fica associado a um procedimento, definido pelo programador.
Por exemplo, o símbolo *quadrado* fica ligado ao procedimento que tem apenas um parâmetro, *x*.

```
(define quadrado                 ; define o nome do procedimento
  (lambda (x)                    ; x é o único parâmetro do procedimento
    (* x x)))
```

Este procedimento, ao ser chamado, calcula e devolve o quadrado do valor associado a *x*.

```
↦(quadrado 5)          ; (quadrado 5) = (* 5 5) = 25
25
```

Vamos agora analisar a regra de cálculo de uma chamada de um procedimento.

```
↦(nome-do-procedimento argumento-1 argumento-2 ...)
```

⇒ São calculados os argumentos que acompanham *nome-do-procedimento*;
⇒ Os valores dos argumentos substituem os parâmetros respectivos (*argumento-1* substitui *parâmetro-1*, *argumento-2* substitui *parâmetro-2*, ...) no corpo do procedimento;
⇒ É devolvido o valor da última expressão calculada no corpo do procedimento.

```
↦(quadrado (quadrado (+ 3 2)))
625
```

Apresenta-se a sequência de operações motivadas por esta chamada

```
↦(quadrado (quadrado (+ 3 2)))
  (quadrado (quadrado 5))
  (quadrado (* 5 5))
  (quadrado 25)
  (* 25 25)
  625
```

Segue-se a definição do procedimento *soma-dos-quadrados*, que toma dois valores como argumentos e devolve a soma dos quadrados desses valores.

```
(define soma-dos-quadrados        ; procedimento com 2 parâmetros
  (lambda (x y)                   ; parâmetros: x e y
    (+ (quadrado x)
       (quadrado y))))
```

O procedimento *soma-dos-quadrados* apresenta dois parâmetros, *x* e *y*, e recorre duas vezes ao procedimento *quadrado*, previamente definido. Por ter dois parâmetros receberá, em cada chamada, dois argumentos.

```
↦(soma-dos-quadrados 1 2)
5
↦(soma-dos-quadrados 5 (soma-dos-quadrados 2 3))
194
```

Exercício 1.4

Apresentar a sequência de operações motivadas pela chamada:

```
↦(soma-dos-quadrados 5 (soma-dos-quadrados 2 3))
```

Na definição de *soma-quadrados* foi utilizado o procedimento *quadrado*, atendendo à tarefa que lhe está associada. Mas, para isso, não era necessário conhecer *quadrado* por dentro. Este

procedimento surge como se fosse um *bloco* ou uma *caixa-preta*. É a isto que se chama *Abstracção Procedimental*.
Os procedimentos *quadrado* e *soma-dos-quadrados* utilizam ambos um parâmetro designado por *x*. Isto não deverá causar qualquer problema, pois estes nomes têm um significado muito bem *localizado*. Cada um deles apenas é reconhecido no corpo do respectivo procedimento. O *x* de um dos procedimentos não tem nada a ver com o *x* do outro procedimento, pois constituem entidades completamente distintas.

Exemplo 1.1
Pretende-se desenvolver um procedimento em *Scheme* para auxiliar os apostadores de totoloto[4] no preenchimento de boletins. O procedimento gera sequências aleatórias de 6 inteiros, situados entre 1 e 49 e, nesta versão, não é feita a verificação de inteiros repetidos. Assim, se na sequência gerada surgirem números repetidos, o melhor será deitar fora a aposta e pedir outra.

Neste procedimento, uma tarefa fácil de identificar é a geração aleatória de um inteiro entre 1 e 49. Para esta tarefa, vamos definir o procedimento designado por *roleta-1-49* que não tem parâmetros:

```
(define roleta-1-49                    ; Surgindo dúvidas sobre alguns
  (lambda ()                           ; dos procedimentos aqui
    (add1 (remainder (random) 49))))   ; utilizados, consultar o Anexo A
                                       ; com o resumos de procedimentos
```

Uma chamada ao procedimento primitivo *random*[5] devolve, aleatoriamente, um inteiro entre 0 e 32767. Por outro lado, uma chamada ao procedimento primitivo *remainder* devolve o resto da divisão por 49 do inteiro gerado por *random*, ou seja, um inteiro entre 0 e 48, que uma chamada a *add1*[6] coloca entre 1 e 49, como se pretendia.

```
↦(roleta-1-49)
33
↦(roleta-1-49)
8
```

O procedimento *roleta-1-49* não tem parâmetros e por isso é chamado colocando o seu nome entre parêntesis, isolado e sem qualquer argumento que o acompanhe.

Este procedimento só pode ser utilizado para gerar, aleatoriamente, números entre 1 e 49. Mas se se pretendesse, por exemplo, gerar números aleatórios entre 1 e 6, para simular o lançamento de um dado? Parece que seria necessário escrever um novo procedimento, certamente, com o nome *roleta-1-6*.

```
(define roleta-1-6
  (lambda ()
    (add1 (remainder (random) 6))))
```

Uma solução mais flexível teria sido, em vez de *roleta-1-49*, definir, por exemplo, *roleta-1-n* com um parâmetro *n*, para gerar aleatoriamente um inteiro entre 1 e n.

```
(define roleta-1-n
  (lambda (n)
    (add1 (remainder (random) n))))
```

[4] Uma aposta do totoloto é uma sequência de seis números não repetidos, situados entre 1 e 49
[5] Em certas implementações do *Scheme*, como acontece no *DrScheme*, *random* tem um parâmetro. Nestes casos, *(random n)* devolve um número aleatório entre 0 e n-1
[6] *(add1 n)* é equivalente a *(+ n 1)*

Com este procedimento seria fácil simular lançamentos de um dado.

```
↦(roleta-1-n 6)
5
↦(roleta-1-n 6)
3
```

Mas também não seria complicado simular a geração de números aleatório entre 1 e qualquer outro inteiro positivo. Por exemplo, para gerar um número do totoloto:

```
↦(roleta-1-n 49)
27
```

Definido o procedimento *roleta-1-49*, vamos agora definir o procedimento *aposta* que deverá gerar, por cada chamada, uma aposta de totoloto:

```
↦(aposta)                  ; verificar que a chamada do procedimento aposta não
Numero 1: 42               ; inclui argumentos, pois o procedimento foi definido
Numero 2: 3                ; sem parâmetros
Numero 3: 41
Numero 4: 32
Numero 5: 40
Numero 6: 10
```

Vai-se pedir uma nova aposta.

```
↦(aposta)
Numero 1: 49
Numero 2: 32
Numero 3: 6
Numero 4: 26
Numero 5: 19
Numero 6: 22
```

Verificar que no procedimento *aposta* não se espera a devolução de um valor, mas de um conjunto de mensagens no ecrã. Uma solução possível para o procedimento *aposta* é agora apresentado.

```
(define aposta
  (lambda ()
    (display "Numero 1: ")        ; Por agora, serve esta solução.
    (display (roleta-1-49))       ; Atendendo ao seu carácter repetitivo
    (newline)                     ; deverá ser fácil encontrar uma
    (display "Numero 2: ")        ; solução mais compacta
    (display (roleta-1-49))
    (newline)
    (display "Numero 3: ")
    (display (roleta-1-49))
    (newline)
    (display "Numero 4: ")
    (display (roleta-1-49))
    (newline)
    (display "Numero 5: ")
    (display (roleta-1-49))
    (newline)
    (display "Numero 6: ")
    (display (roleta-1-49))
    (newline)))
```

Uma pequena explicação sobre o procedimento *display*[7], utilizado em *aposta* para visualizar dados, encontra-se no *Anexo A*. Deste procedimento também não se espera um valor para ser posteriormente processado, mas sim um efeito lateral, ou seja, a visualização no ecrã de um

[7] Por seu lado, o procedimento *read* permite a entrada de dados através do teclado (Anexo A)

número, `(display (roleta-1-49))`, ou de uma cadeia de caracteres, indicada entre *aspas*, `(display "Numero 1: ")`. Quanto a *newline*[8], o próprio nome reflete a missão deste procedimento.
No procedimento *aposta*, facilmente se identifica uma tarefa, que se repete seis vezes:

```
Visualizar "Numero i: "
Visualizar um número aleatório entre 1 e 49
Mudar de linha
```

O procedimento *aposta-aux* vai encarregar-se desta tarefa.

```
(define aposta-aux
  (lambda (i)
    (display "Numero ")
    (display i)
    (display ": ")
    (display (roleta-1-49))
    (newline)))
```

O procedimento *aposta* poderá agora tomar outra forma, designada por *aposta-melhorada*, que se baseia na utilização de *aposta-aux*.

```
(define aposta-melhorada
  (lambda ()
    (aposta-aux 1)
    (aposta-aux 2)
    (aposta-aux 3)
    (aposta-aux 4)
    (aposta-aux 5)
    (aposta-aux 6)))
```

Exercício 1.5

Para pavimentar um parque desportivo utilizam-se peças quadradas de dois tipos, umas de lado com dimensão *lado1* e outras com dimensão *lado2*. Escrever o procedimento *area* com os parâmetros *n1*, *lado1*, *n2* e *lado2*, em que *n1* e *n2* representam o número de peças de cada tipo, e devolve a área pavimentada.

```
↦(area 200 5 400 10)
45000
↦(area 1 5 1 10)
125
```

Na escrita de *area*, utilizar o procedimento *quadrado*, anteriormente definido.

Exercício 1.6

Numa variante do exercício anterior, continuam-se a utilizar peças de dois tipos, mas o número delas é igual. Escrever o procedimento area-com-mesmo-numero-de-pecas com os parâmetros n, lado1, e lado2 e que responde da seguinte maneira

```
↦(area-com-mesmo-numero-de-pecas 15 10 40)
25500
↦(area-com-mesmo-numero-de-pecas 1 20 40)
2000
```

Este procedimento deve basear-se em *soma-dos-quadrados*, já definido.

[8] Ver Anexo A

5- Forma especial *let*

Vamos supor um procedimento que recebe os comprimentos dos três lados de um triângulo, determina o seu perímetro e as percentagens de cada lado em relação ao perímetro. Para melhor se entender o que faz o referido procedimento, designado por *percentagem-lado-perimetro*, analisemos o seu comportamento em três situações.

```
↦(percentagem-lado-perimetro 10 15 20)
perimetro do triangulo: 45
22.2222222222222
33.3333333333333
44.4444444444444
↦(percentagem-lado-perimetro 10 15 15)
perimetro do triangulo: 40
25.0
37.5
37.5
↦(percentagem-lado-perimetro 15 15 15)
perimetro do triangulo: 45
33.3333333333333
33.3333333333333
33.3333333333333
```

A solução designada por *percentagem-lado-perimetro-1* visualiza o perímetro do triângulo e as percentagens dos seus lado em relação ao perímetro. Todavia, o cálculo do perímetro, `(+ a b c)`, é repetido quatro vezes!...

```
(define percentagem-lado-perimetro-1
  (lambda (a b c)
    (display "perimetro do triangulo: ")
    (display (+ a b c))
    (newline)
    (display (* 100 (/ a (+ a b c))))
    (newline)
    (display (* 100 (/ b (+ a b c))))
    (newline)
    (display (* 100 (/ c (+ a b c))))))
```

Para evitar repetições deste tipo, bastaria definir uma variável local, com o valor correspondente ao cálculo repetido, pronta a ser utilizada sempre que fosse necessária. Esta hipótese é possível de implementar com *let* uma outra forma especial do *Scheme*. A solução que se segue, *percentagem-lado-perimetro*-2, onde se define localmente a variável *perimetro*, é mais eficiente que a anterior por eliminar o cálculo repetido do perímetro, para além de se tornar mais legível.

```
(define percentagem-lado-perimetro-2
  (lambda (a b c)

    (let ((perimetro (+ a b c)))
      (display "perimetro do triangulo: ")   ; assinalado a tracejado
      (display perimetro)                    ; o corpo de let
      (newline)                              ; único local onde é
      (display (* 100 (/ a perimetro)))      ; reconhecido perimetro
      (newline)
      (display (* 100 (/ b perimetro)))
      (newline)
      (display (* 100 (/ c perimetro))) )))
```

A expressão *let* apresenta-se com a forma:

```
(let ( (nome-1 expressão-1)
       (nome-2 expressão-2)
       ( ...              ) )

   corpo-de-let)
```

Os símbolos *nome-1*, *nome-2*, ..., são apenas reconhecidos no corpo de *let*, e estão ligados aos valores das expressões respectivas, *expressão-1*, *expressão-2*,... .
Retomando o exemplo, apresenta-se uma outra solução que introduz um novo nível de *let*, onde já é possível utilizar a variável local *perimetro* para definir as percentagens *perc-a*, *perc-b* e *perc-c*.

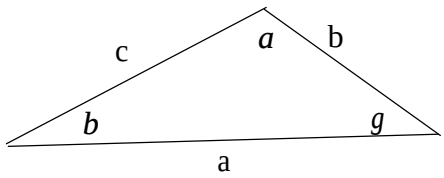
```
(define percentagem-lado-perimetro-3
  (lambda (a b c)

    (let ((perimetro (+ a b c)))
      (let ((perc-a (* 100 (/ a perimetro)))
            (perc-b (* 100 (/ b perimetro)))
            (perc-c (* 100 (/ c perimetro))))
        (display "perimetro do triangulo: ")
        (display perimetro)
        (newline)
        (display perc-a)
        (newline)
        (display perc-b)
        (newline)
        (display perc-c)      )    )))
```

Aproveita-se para relembrar que os parâmetros de um procedimento são entidades locais, apenas reconhecidas no corpo do procedimento. É por isso que os parâmetros de procedimentos diferentes podem ter nomes iguais, o que não provoca qualquer confusão. Com *let*, criam-se também entidades locais, mas com um campo de acção ainda mais restrito que o dos parâmetros do procedimento. Estas entidades criadas por *let* são reconhecidas unicamente no corpo desta forma especial do *Scheme*[9].

Exemplo 1.2

A fórmula de *Briggs*[10] permite, a partir dos comprimentos dos lados *a*, *b* e *c* de um triângulo, determinar os seus três ângulos internos opostos aos lados, respectivamente, *a*, *b*, e *g*. Sendo *sp* o semi-perímetro do triângulo, ou seja *(a + b + c)/2*, então os ângulos são determinados por:



$$\sin\frac{a}{2}=\sqrt{\frac{(sp-b)*(sp-c)}{b*c}} \qquad \sin\frac{b}{2}=\sqrt{\frac{(sp-a)*(sp-c)}{a*c}} \qquad \sin\frac{g}{2}=\sqrt{\frac{(sp-a)*(sp-b)}{a*b}}$$

[9] Este assunto será retomado no próximo capítulo, quando se tratar do tema *Procedimentos como Blocos*.
[10] Henry Briggs foi um matemático inglês (1561-1631) e o primeiro professor de geometria no Gresham House em Londres. Em 1624 publicou *Arithmetica Logarithmica* que continha os logaritmos até 14 casas decimais dos números inteiros de 1 a 20.000 e de 90.000 a 100.000.

Escrever em *Scheme* o procedimento designado por *briggs* que recebe os três lados de um triângulo e visualiza os seus três ângulos internos.

```
↦(briggs 10 10 10)
O triangulo com os lados de comprimento: 10, 10, 10
tem os seguintes angulos: 60.0, 60.0, 60.0

↦(briggs 10 15 15)
O triangulo com os lados de comprimento: 10, 15, 15
tem os seguintes angulos: 38.9424412689814, 70.5287793655093,
70.5287793655093
```

```
(define briggs
  (lambda (a b c)
    (let ((sp (/ (+ a b c)
                 2))
          (ab (* a b))
          (ac (* a c))
          (bc (* b c)))           ; depois de sp definido...
      (let ((sp-a (- sp a))       ; ... poderá ser utilizado num let
            (sp-b (- sp b))       ; mais interno
            (sp-c (- sp c)))
        (let ((ang1 (radians->degrees (* 2
                                         (asin (sqrt (/ (* sp-b sp-c)
                                                        bc))))))
              (ang2 (radians->degrees (* 2
                                         (asin (sqrt (/ (* sp-a sp-c)
                                                        ac))))))
              (ang3 (radians->degrees (* 2
                                        (asin (sqrt (/ (* sp-a sp-b)
                                                       ab)))))))
          (display "O triangulo com os lados de comprimento: ")
          (display a)
          (display ", ")
          (display b)
          (display ", ")
          (display c)
          (newline)
          (display "tem os seguintes angulos: ")
          (display ang1)
          (display ", ")
          (display ang2)
          (display ", ")
          (display ang3)
          (newline)))))))
```

O procedimento *briggs* utiliza a função trigonométrica[11] *asin*, que manipula os ângulos em radianos, e justifica a conversão de radianos para graus, realizada por *radians->degrees*. No primeiro *let*, são definidos o semi-perímetro *sp* e os termos *ab*, *ac* e *bc*. Depois, no corpo do primeiro let, e já com *sp* definido, num segundo *let* são definidos *sp-a*, *sp-b* e *sp-c*. Finalmente, num terceiro *let*, ou seja, no corpo dos dois *let* anteriores, são definidos os três ângulos, segundo a fórmula de Briggs.

[11] Consultar o Anexo A, no que se refere a funções trigonométricas

Exercício 1.7

A área de um triângulo pode determinar-se a partir do comprimento dos seus lados *a*, *b*, e *c*, com a fórmula que se segue, em que *sp* é o semi-perímetro do triângulo.

$$area-tri = \sqrt{sp*(sp-a)*(sp-b)*(sp-c)}$$

Escrever o procedimento *area-triangulo* que tem como parâmetros os três lados de um triângulo e devolve a sua área. Sugere-se a definição de uma variável local *sp* para evitar a repetição do cálculo do semi-perímetro.

Exemplo 1.3

Pretende-se definir o procedimento *area-circulo*, que não tem parâmetros e responde da seguinte maneira.

```
↦(area-circulo)
raio: 10
Area do circulo e': 314.159265358979
```

Como se pode verificar, o raio do círculo é pedido ao utilizador, que o fornecerá a partir do teclado. Assim, na implementação do procedimento *area-circulo* será utilizada a primitiva *read*[12].

```
(define area-circulo
  (lambda ()
    (display "raio: ")
    (let ((raio (read)))
      (newline)
      (display "Area do circulo e': ")
      (display (area-cir-aux raio)))))

(define area-cir-aux
  (lambda (r)
    (* pi r r)))              ; area = pi * r², em que pi é uma
                              ; constante reconhecida pelo Scheme
```

Nesta implementação recorreu-se a um procedimento auxiliar que, recebendo o raio de um círculo, limita-se a devolve a sua área.

A definição de uma variável local com o valor devolvido por *read* é uma prática usual. Se assim não fosse este valor ou era imediatamente utilizado ou perder-se-i, como acontece na versão *area-circulo-nao-recomendavel*. Neste caso, o valor do raio lido é utilizado uma vez e nunca mais estará acessível[13].

```
(define area-circulo-nao-recomendavel
  (lambda ()
    (display "raio: ")
    (let ((area (area-cir-aux (read))))
      (newline)
      (display "Area do circulo e': ")
      (display area))))
```

[12] Consultar o Anexo A

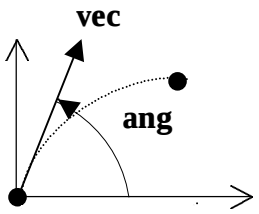
[13] Mais um argumento para se definirem variáveis locais com os valores devolvidos por *read*.
A expressão (* (read) (read)) devolverá um valor correcto, que é o produto de dois números lidos através do teclado. Contudo, não se pode garantir o valor devolvido pela expressão (- (read) (read)). A razão é simples. O *Scheme* não garante a ordem de cálculo dos argumentos na chamada de um procedimento. Não se sabe, portanto, se o primeiro número lido do teclado corresponde ao primeiro ou ao segundo *read*. De facto, no caso da multiplicação, esta ordem não tem importância, mas assim não acontece com a subtracção, em que a ordem dos operandos não é arbitrária. Todos sabemos que (* 5 10) = (* 10 5) = 50, mas que (- 5 10) = -5 e (- 10 5) = 5.

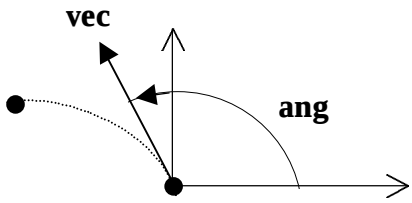
6- Estruturas de Selecção

São frequentes as situações que implicam a tomada de decisões e, perante várias hipóteses, opta-se por alguma delas. Por exemplo, se se pretende simular o sorteio de 6 bolas do Totoloto, em vez de repetir 6 vezes a escrita de um certo conjunto de instruções, aliás como foi feito no Exemplo 1.1, teria sido preferível programar algo do tipo:

- Sortear mais uma bola;
- Se ainda não foram sorteadas 6 bolas, repete-se a acção anterior. Caso contrário, acabou de ser apurada uma aposta, não sendo necessário sortear mais bolas.

Outro exemplo:





Nas figuras, se o vector *vec* representar a velocidade inicial de um projéctil lançado da origem dos eixos e *ang* o ângulo de lançamento, pode pretender-se que o projéctil não seja lançado para "trás". Então, *ang* deverá ser menor que 90º. Para se considerar esta condição, poderiamos escrever:

```
(if (< ang 90)
    (faz-o-lançamento ...)
    (display "projectil mal orientado"))
```

Nesta expressão *if*, surge uma *expressão de relação*, (< ang 90), cujo resultado pode apenas assumir 2 valores:

⇒ *#t* (verdadeiro), se *ang* for menor que 90, ou
⇒ *#f* (falso), se *ang* for maior ou igual que 90.

As entidades com valores *#t* e *#f* são designadas de *booleanas* e as expressões de que resultam booleanos são designadas por *Predicados*. Por exemplo, (< 30 40) é um predicado com valor *#t* e (< 30 30) é um predicado com valor *#f*.

Agora, é fácil identificar mais uma forma especial do Scheme, *if*:

```
(if expressão-predicado
    expressão-consequente
    expressão-alternativa)
```

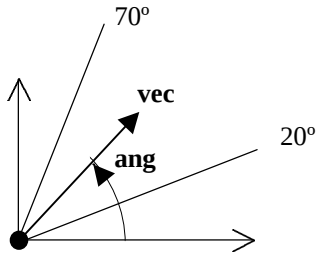
A regra de cálculo da forma especial *if* é a seguinte:

⇒ Calcular a *expressão-predicado*;
⇒ Se resultar *#t (verdadeiro)*, é calculada a *expressão-consequente*;
⇒ Se resultar *#f (falso)*, é calculada a *expressão-alternativa*.

Os *procedimentos de relação* disponibilizadas pelo *Scheme* são[14]: *> (maior)* , *< (menor)*, *= (igual)*, *>= (maior ou igual)*, e <= (menor ou igual).

[14] Consultar o Anexo A

```
↦(if (> 5 12)
      5
      (* 2 7))
14
↦(if (<= 5 12)
      5
      (* 2 7))
5
```



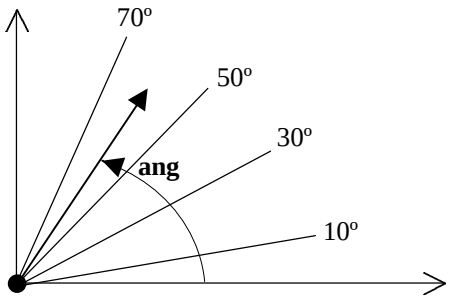
Retomando o exemplo do lançamento do projéctil. Pretende-se agora garantir que o ângulo de lançamento deve ocorrer entre 20 e 70º. Ou seja, o ângulo de lançamento deve ser maior que 20º e menor que 70º.
Para se considerar esta nova condição, podemos escrever:

```
(if (and
       (> ang 20)
       (< ang 70))
    (faz-o-lançamento ...)
    (display "projectil mal orientado"))
```

A expressão-predicado é, neste exemplo, uma *expressão lógica*, que, como acontecia com as expressões de relação, também origina resultados *booleanos*.
Os procedimentos lógicos são especialmente indicados para definir *condições compostas*.

```
(if (and
       (>= altura 160)
       (<= altura 180))
    (display "Altura média")
    (display "Ou muito alto ou muito baixo!!!"))
```

Os *procedimentos lógicos* disponibilizadas pelo *Scheme* são[15]: *and (e)*, *or (ou)*, *not (não)*.
Faz parte da cultura *Scheme* terminar com *?* o nome dos procedimentos que são predicados, ou seja, que devolvem valores booleanos. O próprio Scheme dá o exemplo, com os predicados que disponibiliza: *negative?*, *positive?*, *zero?*, *even?*, *odd?*, *boolean?*, *symbol?*, *procedure?*, *number?*, *integer?*, *real?* e outros. Nesta altura, recomenda-se uma consulta ao Anexo A, com o objectivo de se verificar a funcionalidade de cada um destes predicados.



Retomando o exemplo do lançamento do projéctil, pretende-se agora garantir os seguintes tipos de lançamento:

- Se *ang* maior que 70º ou menor ou igual que 10º, não haverá lançamento;
- Se *ang* menor ou igual que 70º e maior que 50º, haverá um lançamento a grande altitude;
- Se *ang* menor ou igual que 50º e maior que 30º, haverá um lançamento a média altitude;
- Se *ang* maior ou igual que 30º e maior que 10º, haverá um lançamento a baixa altitude.

[15] Consultar o Anexo A

Para ter em conta todas estas condições, define-se o procedimento *projectil*:

```
(define projectil
  (lambda (ang)
    (if (or
          (> ang 70)
          (<= ang 10))
        (display "projectil mal orientado")
        (if (> ang 50)
            (display "lancamento a grande altitude")
            (if (> ang 30)
                (display "lancamento a media altitude")
                (display "lancamento a baixa altitude"))))))
```

```
↦(projectil 10)
projectil mal orientado
↦(projectil 40)
lancamento a media altitude
↦(projectil 70)
lancamento a grande altitude
↦(projectil 120)
projectil mal orientado
```

Não se poderá dizer que o procedimento *projectil*, baseado numa sequência de *if's*, seja muito legível. Em situações como esta, é preferível utilizar a forma especial *cond*:

```
(define projectil-melhorado
  (lambda (ang)
    (cond ((or
             (> ang 70)
             (<= ang 10))
           (display "projectil mal orientado"))
          ((> ang 50)
           (display "lancamento a grande altitude"))
          ((> ang 30)
           (display "lancamento a media altitude"))
          (else
            (display "lancamento a baixa altitude")))))
```

Identificando a forma especial do Scheme *cond*:

```
(cond (predicado-1 exp1-1 exp1-2 ...)              ; cláusula 1

      (predicado-2 exp2-1 exp2-2 ...)              ; cláusula 2

      ...

      (else exp-else-1 exp-else-2 ...) )           ; cláusula else
```

A regra de cálculo da forma especial *cond* é a seguinte:

⇒ Calcular os predicados, começando em *predicado-1*, até encontrar um predicado com valor *#t*;

⇒ Logo que se encontre um predicado *#t*, calcular as expressões correspondentes;

⇒ Se não se encontrar qualquer predicado com valor *#t*, calcular as expressões correspondentes a *else*.

⇒ Nota: A cláusula *else* é opcional. Se não existir e se do cálculo de todos os predicados resultar *#f*, nenhuma das expressões de *cond* será calculada.

Sobre as formas especiais *if* e *cond*, poder-se-á dizer que *if* é de evitar em situações com mais de duas opções, por se tornar de mais difícil leitura. E mesmo com duas opções, é também de evitar *if* quando a *expressão-consequente* ou a *expressão-alternativa* envolvem mais do que uma expressão, situação que obriga à utilização de *begin*[16].

```
(if expressão-predicado
    (begin
       expressão-consequente-1
       expressão-consequente-2
                ...
       expressão-consequente-m)

    (begin
       expressão-alternativa-1
       expressão-alternativa-2
                ...
       expressão-alternativa-n) )
```

```
(cond (expressão-predicado

       expressão-consequente-1
       expressão-consequente-2
                ...
       expressão-consequente-m)

      (else
       expressão-alternativa-1
       expressão-alternativa-2
                ...
       expressão-alternativa-n) )
```

Exemplo 1.4

Apresentam-se duas versões de um procedimento que recebe dois valores e visualiza a relação de grandeza entre eles. Na versão *comparador-com-if* recorre-se a *if* e, na versão *comparador-com-cond*, a *cond*. Ambos respondem da mesma maneira e diferem apenas no grau de legibilidade.

```
(define comparador-com-if
  (lambda (x y)
    (if (> x y)
        (begin
          (display x)
          (display " e' maior que ")
          (display y))
        (if (> y x)
            (begin
              (display y)
              (display " e' maior que ")
              (display x))
            (begin
              (display "ambos iguais a ")
              (display x))))))
```

```
↦(comparador-com-if 2 3)
3 e' maior que 2
↦(comparador-com-if 3 2)
3 e' maior que 2
↦(comparador-com-if 3 3)
ambos iguais a 3
```

[16] *begin* é mais uma das formas especiais do *Scheme*. A sua forma genérica é *(begin expressão-1 expressão-2 ... expressão-n)* e garante que as expressões são calculadas em sequência, desde a primeira até à última. Esta forma especial devolve o valor da última expressão. Este tipo de sequenciamento está implícito no corpo dos procedimentos quando contém mais do que uma expressão e também nas cláusulas de *cond*

```
(define comparador-com-cond
  (lambda (x y)
    (cond ((> x y)
           (display x)
           (display " e' maior que ")
           (display y))
          ((> y x)
           (display y)
           (display " e' maior que ")
           (display x))
          (else
            (display "ambos iguais a ")
            (display x)))))
```

Exercício 1.8

O procedimento *rectangulo-maior* tem como parâmetros lado-a1, lado-a2, lado-b1, e lado-b2, em que os dois primeiros correspondem ao comprimentos dos lados do rectângulo A e os dois últimos aos lados do rectângulo B. Escrever este procedimento em Scheme que calcula a área de cada um dos rectângulos, compara-as e responde da seguinte maneira:

```
↦(rectangulo-maior 10 20 15 5)
Reactangulo A: 200
Reactangulo B: 75
O rectangulo A e' maior 125 unidades

↦(rectangulo-maior 10 20 15 18)
Reactangulo A: 200
Reactangulo B: 270
O rectangulo B e' maior 70 unidades

↦(rectangulo-maior 10 20 40 5)
Reactangulo A: 200
Reactangulo B: 200
Os rectangulos apresentam igual area.
```

Exercícios e Exemplos de final de capítulo

Nesta Secção são apresentados exercícios e exemplos para consolidação da matéria dada. Recomenda-se o estudo e a resolução de todos eles, em frente do computador, atitude que é fundamental tomar desde o início do processo de aprendizagem da programação. Apesar de ainda muito limitados em termos do conhecimento das potencialidades do *Scheme*, é já possível resolver alguns problemas interessantes.

Exemplo 1.5

Os números decimais ímpares terminam em 1, 3, 5, 7 e 9. Vamos agora inventar uma nova classe de números, os ímpares-curvos (!!!) como sendo os números ímpares que terminam em 3, 5 e 9 (dígitos que, no respectivo desenho, usam segmentos curvos!).
Considere o procedimento *impar-curvo?* que determina se um número é ou não ímpar-curvo.

```
↦(impar-curvo? 2345)
#t
↦(impar-curvo? 2346)
#f
```

```
↦(impar-curvo? 2347)
#f
```

Escrever em *Scheme* o procedimento *impar-curvo?*

Resolução:
A solução que se apresenta começa por definir localmente a variável *digito-menos-significativo*, que se obtém calculando o resto da divisão inteira do número dado por 10.
Deduz-se se o número em questão é ou não *impar-curvo*, testando, sucessivamente, se *digito-menos-significativo* é 3, 5 ou 9.

```
(define impar-curvo?
  (lambda (num)
    (let ((digito-menos-signif (remainder num 10)))
      (cond ((= digito-menos-signif 3) #t)
            ((= digito-menos-signif 5) #t)
            ((= digito-menos-signif 9) #t)
            (else #f)))))
```

Uma outra solução é conseguida através de uma condição composta, que utiliza o operador lógico *or*.

```
(define impar-curvo?
  (lambda (num)
    (let ((digito-menos-signif (remainder num 10)))
      (or
         (= digito-menos-signif 3)
         (= digito-menos-signif 5)
         (= digito-menos-signif 9)))))
```

Exemplo 1.6

Supor que existe um procedimento *quantos-positivos*, que compara os dois argumentos *x* e *y* e fornece, como resultado 2, 1 ou 0, conforme o número de argumentos positivos.

```
↦(quantos-positivos 1 2)
2
↦(quantos-positivos 1 -2)
1
↦(quantos-positivos -1 -1)
0
```

Por seu turno, o procedimento *soma-positivos*, também compara os dois argumentos *a* e *b*, e fornece:

- A soma dos dois, se ambos forem positivos;
- O valor positivo, se apenas um dos valores for positivo;
- O valor 0, se ambos forem negativos.

Escrever em Scheme o procedimento *soma-positivos*, utilizando o procedimento *quantos-positivos*, suposto já existente.

Resolução:
Optou-se por criar localmente a variável *numero-de-posit*, que vai conter o número de positivos em jogo, através de uma chamada ao procedimento *quantos-positivos*.

```
(define soma-positivos
  (lambda (a b)
    (let ((numero-de-posit (quantos-positivos a b)))
      (cond ((= numero-de-posit 2) (+ a b))
            ((zero? numero-de-posit) 0)
            (else
              (if (positive? a)
                  a
                  b))))))
```

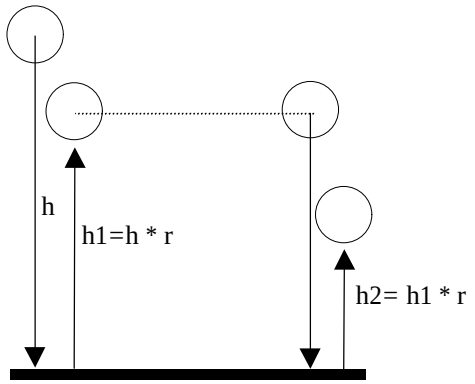
Quanto ao procedimento *quantos-positivos*. A solução baseia-se nos seguintes testes: *x* e *y* são ambos positivos; *x* e *y* são ambos negativos; se não for nenhum destes casos, então um dos argumentos será positivo.

```
(define quantos-positivos
  (lambda (x y)
    (cond ((and
             (positive? x)
             (positive? y))
           2)
          ((and
             (negative? x)
             (negative? y))
           0)
          (else 1))))
```

```
↦(soma-positivos 3 2)
5
↦(soma-positivos 3 -2)
3
↦(soma-positivos -3 2)
2
↦(soma-positivos -3 -2)
0
```

Exemplo 1.7

Uma bola é largada de uma altura *h* sobre uma superfície lisa, ficando a saltar durante algum tempo. Supor que ao saltar a bola toca a superfície sempre no mesmo ponto. A distância percorrida pela bola é a soma dos movimentos descendentes e ascendentes.



Em cada salto, a bola sobe a uma altura que é calculada multiplicando *h* altura do salto anterior por um factor *r* (0 < r < 1), designado por *coeficiente de amortecimento*.

Vamos supor que o procedimento *salto*, com os parâmetros *h* e *r*, devolve a altura do salto da bola quando esta é largada da altura *h*, sabendo que *r* é o coeficiente de amortecimento.

Escrever em *Scheme* o procedimento *distancia-1*, que toma os valores *h*, e *r*, e devolve a distância percorrida pela bola desde o momento que é largada da altura *h* até ao final do primeiro salto, ou

seja, desce e sobe uma vez. Este procedimento deverá utilizar o procedimento *salto*, que também é necessário escrever.

Resolução:

```
(define distancia-1
  (lambda (h r)
    (+ h (salto h r))))

(define salto
  (lambda (h r)
    (* h r)))
```

```
↦(distancia-1 2 .5)
3.0
↦(distancia-1 3 .2)
3.6
```

Exercício 1.9

Tomando como referência o exemplo anterior, escrever agora o procedimento *distancia-2*, que toma os valores *h*, e *r*, e devolve a distância percorrida pela bola desde o momento que é largada da altura *h* até ao final do segundo salto, ou seja, desce, sobe, torna a descer e torna a subir[17].

```
↦(distancia-2 2 .5)
4.5
↦(distancia-2 3 .2)
4.32
```

Exercício 1.10

Um percurso rodoviário é composto por uma parte em piso horizontal, mas também apresenta uma parte em subida e outra em descida. De uma viatura é conhecido o consumo médio, aos 100 Km, quando se desloca em percurso horizontal. Em relação a este consumo, a mesma viatura, em subida, gasta mais 30% e, em descida, menos 10%.
Escrever em *Scheme* o procedimento *consumo-total* que tem como parâmetros *consumo*, *horiz*, *sub* e *desc*, que representam, respectivamente, o consumo médio da viatura em percurso horizontal, o número de Km de percurso horizontal, o número de Km em subida e, finalmente, o número de Km em descida. Este procedimento deve devolver a quantidade em litros gasta pela viatura no percurso definido pelos parâmetros respectivos.

Exercício 1.11

Tomando como base o exercício anterior, pretende-se agora escrever o procedimento *gasolina-suficiente* que tem como parâmetros *gas-no-tanque* (que representa a quantidade de combustível que a viatura tem no tanque), e ainda os parâmetros do procedimento *consumo-total*. O procedimento pretendido deve responder, conforme o caso, com uma das seguintes mensagens:

- Nao é suficiente. Faltam x litros
- É suficiente. Sobram x litros
- Gasolina à justa

[17] O procedimento *distancia-n*, que devolve a distância percorrida pela bola até ao final do n-ésimo salto, fica para o próximo capítulo.

Exemplo 1.8

Para este exercício vamos supor que a Terra é uma esfera perfeita, com um raio de 6378 Km. Sendo assim, o perímetro da Terra, quando se percorre o Equador (latitude 0º) será de *2 x pi x 6378000* metros. Pretende-se conhecer o perímetro da Terra, percorrendo um paralelo ao Equador, identificado pela respectiva latitude (entre 0 e 90º, só no Hemisfério Norte). Se para a latitude 0º o perímetro coincide com o comprimento do Equador, para a latitude 90º, que coincide com o pólo Norte, o perímetro será 0 metros. Mas qual será o perímetro do trópico de Câncer, sabendo que se encontra à latitude 23.5º N? E o perímetro do Círculo Polar Árctico a 66.5º N?

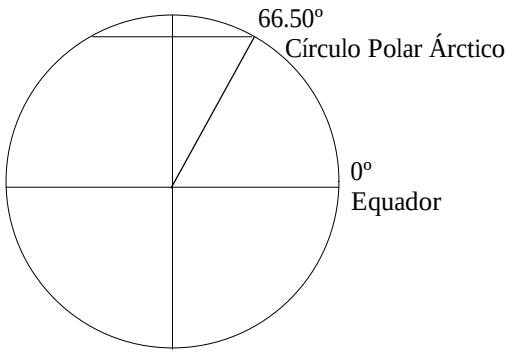
Pretende-se definir o procedimento *perimetro-paralelo* que responde da seguinte forma:

```
↦(perimetro-paralelo 0)
40074155.8891914 metros

↦(perimetro-paralelo 66.5)
15979532.3487802 metros

↦(perimetro-paralelo 90)
2.45375329629861e-009 metros

↦(perimetro-paralelo 89.5)
349708.543934398 metros
```



Resolução:

```
(define pi 3.141592653589793)        ; valor de pi
(define raio-terra 6378000)          ; raio da Terra em metros

(define perimetro-paralelo               ; visualiza o resultado
  (lambda (latitude)                     ; no formato indicado
    (display (peri-paralelo latitude))
    (display " metros")
    (newline)))

(define peri-paralelo              ; calcula o perímetro do paralelo
  (lambda (latitude)               ; cuja latitude é o argumento
    (* 2
       pi
       (raio-paralelo latitude))))

(define raio-paralelo               ; calcula o raio do paralelo
  (lambda (lat)                     ; cuja latitude é o argumento
    (* raio-terra
       (cos (degrees->radians lat))))) ; Atenção: O Scheme quer
                                       ; os ângulos em radianos[18]
```

Nesta resolução, principiámos pelo mais geral, com o procedimento *perimetro-paralelo* que trata do diálogo. Este procedimento recorre ao procedimento *peri-paralelo* que calcula o perímetro de um paralelo para uma certa latitude. Com este objectivo, *peri-paralelo* recorre ao procedimento *raio-paralelo* que calcula o raio do paralelo cuja latitude é fornecida.

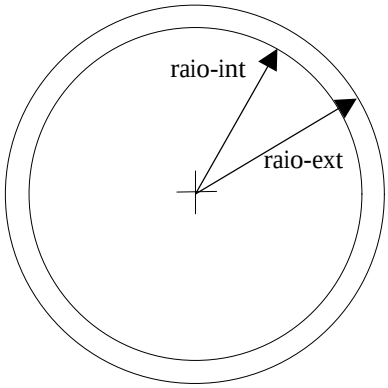
[18] Consultar o Anexo A

Exercício 1.12

No contexto do exemplo anterior, supor que se pegava num arame de 40074155.8891914 metros e com ele se fazia um anel para colocar à volta da Terra, sobre o Equador. O anel ficava justo à Terra, portanto à distância zero, em todos os seus pontos.
Imagine agora que cortava o anel e que lhe acrescentava um metro. O anel era novamente colocado à volta da Terra, no Equador, mas agora já não ficava completamente justo. Se o anel fosse colocado concêntrico com o Equador, a que distância ficaria o anel da Terra? E se a mesma experiência fosse feita num paralelo, muito junto ao Pólo Norte. A distância seria muito diferente?
Pretende-se definir o procedimento *distancia-terra* com dois parâmetros, um associado à latitude e outro ao comprimento de arame a acrescentar ao anel. Este procedimento determina a distância a que fica o anel da Terra, quando se lhe acrescenta aquele comprimento de arame.

Sugere-se a seguinte pista:

- Para a latitude dada, determinar o raio do respectivo paralelo, seja *raio-int*;
- Para a latitude dada, determinar o perímetro do respectivo paralelo, conhecido *raio-int*. Acrescentar a este perímetro o comprimento dado;
- Determinar o raio correspondente ao perímetro aumentado, seja *raio-ext*;
- Achar a diferença entre *raio-ext* e *raio-int*.



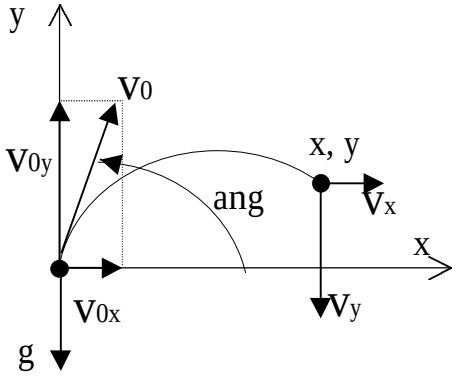
Escrever em Scheme o procedimento *distancia-terra* e experimentar as seguintes situações:

```
↦(distancia-terra 0 1)          ; acrescentar 1 metro no anel do Equador
???
↦(distancia-terra 23.5 1)       ; acrescentar 1 metro no anel
???                             ; do Trópico de Câncer
↦(distancia-terra 66.5 1)       ; acrescentar 1 metro no anel
???                             ; do Círculo Polar Árctico
↦(distancia-terra 90 1)         ; acrescentar 1 metro no anel
???                             ; do Pólo Norte
```

A análise dos resultados obtidos vai pôr em evidência uma conclusão, que poderá ser considerada muito estranha...

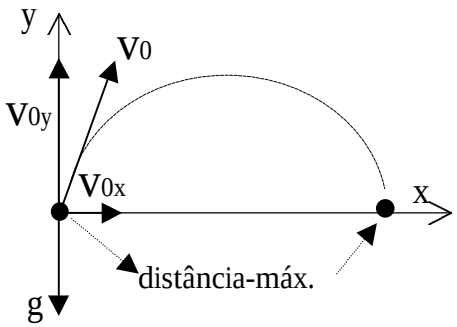
Exercício 1.13

Para este exercício vai ser necessário relembrar, embora de uma forma um pouco simplificada, algumas fórmulas do movimento de projécteis no espaço.



$x = x_0 + v_{0x} \cdot t$
$y = y_0 + v_{0y} \cdot t - 0.5 \cdot g \cdot t^2$
$v_x = v_0 \cdot \cos ang$
$v_y = v_0 \cdot \sin ang - g \cdot t$
$g = 9.75\ m/s^2$ (aceleração devida à gravidade)

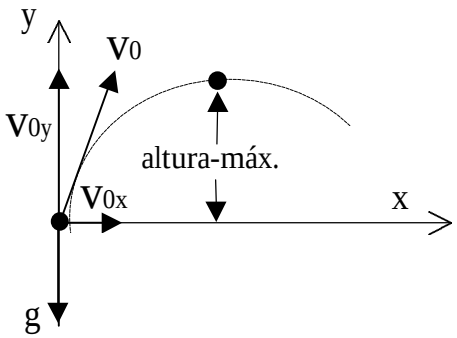
Um projéctil é lançado de um ponto x_0, y_0, com a velocidade inicial v_0. O ângulo de lançamento, *ang*, é definido pelo eixo OX e pelo vector v_0.
As equações *x* e *y* definem a posição do projéctil, em função do ponto inicial, velocidade inicial, tempo e gravidade.
As equações v_x e v_y definem a velocidade do projéctil, em função da velocidade inicial, velocidade inicial, tempo e gravidade. Estas equações para t = 0, correspondem a v_{0x} e v_{0y}.
No conjunto de exercícios que se seguem, considerar que o ponto de lançamento está na origem do sistema de eixos, *0*, *0*.



1- Escrever em *Scheme* o procedimento *lançamento-distancia-max*, com dois parâmetros, *vel-inicial* e *ang-inicial*, indicando, respectivamente, a velocidade e ângulo iniciais do lançamento de um projéctil. O procedimento devolve a distância máxima alcançada pelo projéctil.
Pista sugerida:

⇒ Com a expressão *y* determinar *t* para y=0;

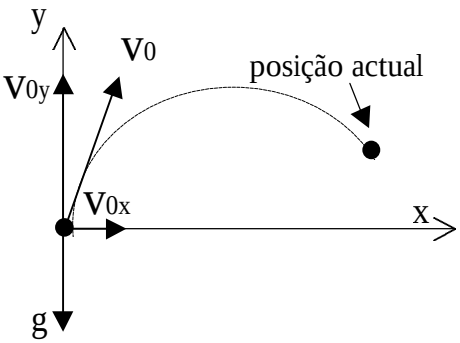
⇒ Determinar *x* com a respectiva expressão, para *t* determinado.



2- Escrever em *Scheme* o procedimento *lançamento-altura-max*, com dois parâmetros, *vel-inicial* e *ang-inicial*, indicando, respectivamente, a velocidade e ângulo iniciais do lançamento de um projéctil. O procedimento devolve a altura máxima alcançada pelo projéctil.
Pista sugerida:

⇒ Com v_y determinar *t* para v_y =0;

⇒ Determinar *y*, com a respectiva expressão, para *t* determinado.



3- Escrever em *Scheme* o procedimento *lançamento-posicao*, com três parâmetros, *vel-inicial*, *ang-inicial* e *tempo*, indicando, respectivamente, a velocidade e ângulo iniciais do lançamento de um projéctil, e o intervalo de tempo contado a partir do lançamento.
O procedimento devolve as coordenadas x e y da posição onde se encontra o projéctil, no final daquele intervalo de tempo.
Pista sugerida:

⇒ Calcular v_{ox} e v_{oy}, através de v_x e v_y, para t=0;

⇒ Determinar *x* e *y*, para *t* fornecido.

4- Com os três procedimentos a funcionar, tentar algumas experiências:

⇒ Com *lancamento-distancia-max* e com uma certa velocidade inicial, tentar descobrir qual o ângulo que permite o lançamento mais longo.

⇒ Com *lancamento-posicao* e com a velocidade inicial e o ângulo que permitiram o lançamento mais longo, tentar descobrir o intervalo de tempo necessário para alcançar o ponto mais distante.

⇒ Com as várias tentativas da experiência anterior, desenhar a trajectória do projéctil. Considerar o desenho da trajectória para além do ponto do solo mais distante.

⇒ Com *lancamento-altura-max* e com uma certa velocidade inicial, tentar descobrir qual o ângulo que permite o lançamento mais alto.

⇒ Com *lancamento-posicao* e com a velocidade inicial e o ângulo que permitiram o lançamento mais alto, tentar descobrir o intervalo de tempo necessário para alcançar o ponto mais alto.

⇒ E, para terminar, pedir a um colega que defina um ponto qualquer no solo (x = ? e y = 0). Agora, utilizando um ou mais dos três procedimentos, contar o número de tentativas necessárias para atingir aquele ponto com o projéctil. A prova final deverá ser feita com o procedimento *lancamento-posicao*, ou seja, será necessário determinar a velocidade e o ângulo iniciais, bem como o tempo necessário para atingir o alvo.
Nota: Cada utilização de um procedimento é considerada como uma tentativa.

Este exercício será retomado noutro capítulo, no projecto de um jogo gráfico interactivo.