Anexo A - **Principais procedimentos da linguagem Scheme**

Neste anexo resumem-se os principais procedimentos da linguagem Scheme, acompanhados de exemplos, e agrupados em:

- Processamento booleano
- Processamento numérico
- Processamento trigonométrico
- Controlo de sequência
- Entrada/Saída
- Processamento de Pares e Listas
- Processamento de Caracteres e Cadeia de caracteres (*strings*)
- Processamento de Vectores
- Processamento de Ficheiros
- Vários

Uma descrição completa desta linguagem encontra-se em

Revised (5) Report on the Algorithmic Language Scheme
http://www.cs.indiana.edu/classes/c211-mill/home/r5rs-html/r5rs_toc.html#TOC83

Processamento booleano

```
#f                          ; valor booleano falso.
#t                          ; valor booleano verdadeiro.
(boolean? x)                ; se x for booleano, devolve #t, se não devolve #f.
(and x1 x2 x3 ...)          ; calcula x1, x2, x3, ..., até encontrar um
                            ; valor #f, e devolve #f.
                            ; Não encontrando qualquer valor #f, devolve #t.
(or x1 x2 x3 ...)           ; calcula x1, x2, x3, ..., até encontrar um
                            ; valor #t, e devolve #t.
                            ; Não encontrando qualquer valor #t, devolve #f.
(not x)                     ; se x = #t, devolve #f, se não devolve #t.
```

Processamento numérico

```
(number? x)                 ; devolve #t, se x for número, se não #f.
(integer? x)                ; devolve #t, se x inteiro, se não #f.
(real? x)                   ; devolve #t, se x real, se não #f.
(+ x1 x2 x3 ...)            ; devolve soma de x1, x2, x3, ...
(- x1 x2 x3 ...)            ; subtrai a x1, sucessivamente, x2, x3, ...
                            ; Com um só argumento:   (- x) devolve -x.
(* x1 x2 x3 ...)            ; devolve produto de x1, x2, x3, ...
(/ x1 x2 x3 ...)            ; divide x1, sucessivamente, por x2, x3, ...
                            ; Com um só argumento:   (/ x) devolve 1/x.
(< x1 x2 x3 ...)            ; devolve #t, se x1, x2, x3,... em ordem crescente,
                            ; se não #f.
(> x1 x2 x3 ...)            ; devolve #t, se x1, x2, x3,... em ordem decrescente,
                            ; se não #f.
(= x1 x2 x3 ...)            ; devolve #t, se x1, x2, x3, ... iguais, se não #f.
(<= x1 x2 x3 ...)           ; devolve #t, se x1, x2, x3, ... em ordem
                            ; não decrescente, se não #f.
```

```
(>= x1 x2 x3 ...)            ; devolve #t, se x1, x2, x3, ... em ordem
                             ; não crescente, se não #f.
(add1 x)                     ; devolve x+1.
(sub1 x)                     ; devolve x-1.
(sqrt x)                     ; devolve raiz quadrada de x, sendo x >= 0.
(exp x)                      ; devolve e^x.
(expt x y)                   ; devolve x^y.
(log x)                      ; devolve logaritmo de x na base e.
(abs x)                      ; devolve valor absoluto de x.
                             ; (abs 5.1) devolve 5.1 e (abs -5.1) devolve 5.1.
(round x)                    ; devolve inteiro mais próximo de x. Existindo 2
                             ; inteiros igualmente distantes, devolve o que for par.
                             ; (round 2.7) devolve 3.0, (round 2.5) devolve 2
                             ; e (round 3.5) devolve 4.
(ceiling x)                  ; devolve inteiro igual a x ou imediatamente acima.
                             ; (ceiling 5.3) devolve 6.0 e (ceiling -5.3) devolve -5.0.
(floor x)                    ; devolve inteiro igual a x ou imediatamente abaixo.
                             ; (floor 5.3) devolve 5.0, (floor -5.3) devolve -6.0.
(truncate x)                 ; devolve inteiro constituído pela parte inteira de x.
                             ; (truncate 2.7) devolve 2.0.
(max x1 x2 x3 ...)           ; devolve o valor do maior argumento.
(min x1 x2 x3 ...)           ; devolve o valor do menor argumento.
(quotient x d)               ; devolve o quociente da divisão inteira x/d.
                             ; (quotient 7 3) devolve 2.
                             ; (quotient -17 3.0) devolve -5.0.
(remainder x d)              ; devolve o resto (com o mesmo sinal de x)
                             ; da divisão inteira x/d.
                             ; (remainder -7 3) devolve -1, pois -7= 3 * -2 + -1.
                             ; (remainder 7 -3) devolve 1, pois  7= -3 * -2 + 1.
                             ; (remainder 7 3) devolve 1, pois  7= 3 * 2 + 1.
                             ; (remainder -7 -3) devolve -1, pois -7= -3 * 2 + -1.
(modulo x d)                 ; devolve o resto (com o mesmo sinal de d) da
                             ; divisão inteira x/d.
                             ; quando x e d têm o mesmo sinal,
                             ; modulo devolve o mesmo resultado que remainder.
                             ; (modulo -7 3) devolve 2 e (modulo 7 -3) devolve -2.
                             ; (modulo 7 3) devolve 1 e (modulo -7 -3) devolve -1.
(gcd x1 x2 x3 ...)           ; devolve máximo divisor comum dos argumentos.
(lcm x1 x2 x3 ...)           ; devolve menor múltiplo comum dos argumentos.
(negative? x)                ; devolve #t, se x < 0, se não #f.
(positive? x)                ; devolve #t, se x > 0, se não #f.
(zero? x)                    ; devolve #t, se x = 0, se não #f.
(even? x)                    ; devolve #t, se x par, se não #f.
(odd? x)                     ; devolve #t, se x ímpar, se não #f.
(random n)                   ; devolve um inteiro pseudo-aleatório,
                             ; situado entre 0 e n-1.
                             ; Não faz parte da definição do Scheme.
                             ; Em certas implementações, random não tem parâmetros.
```

Processamento trigonométrico

```
(degrees->radians x)         ; x em graus, devolve valor correspondente em
                             ; radianos (não existe no DrScheme - Full Scheme).
```

```
(radians->degrees x)          ; x em radianos, devolve valor correspondente em
                              ; graus (não existe no DrScheme - Full Scheme).
(sin x)                       ; devolve seno de x, estando x em radianos.
                              ; (sin (degrees->radians 90)) devolve 1.
(cos x)                       ; devolve coseno de x, estando x em radianos.
(tan x)                       ; devolve tangente de x, estando x em radianos.
(acos x)                      ; devolve arco em radianos, cujo coseno é x.
(asin x)                      ; devolve arco em radianos, cujo seno é x.
(atan x)                      ; devolve arco em radianos, cujo tangente é x.
                              ; (radians->degrees (atan 1)) devolve 45.0.
```

Controlo de sequência

```
(if test-exp                                ; (if (< 5 3)
    then-exp                                ;     (- 45 40)
    else-exp)                               ;     (- 40 45)) devolve -5.
(if test-exp                                ; (if (< 5 3)
    then-exp)                               ;     (- 45 40)) nada acontece.
(cond (pred-1 exp1-1 exp1-2 ...)            ; (cond ((< n 3) (display "< 3"))
      (pred-2 exp2-1 exp2-2 ...)            ;       ((> n 4) (display "< 4"))
      ...                                   ;       ...
      (pred-n expn-1 expn-2 ...))           ;       ((> n 10) (display "> 10")))
(cond (pred-1 exp1-1 exp1-2 ...)            ;  (cond ((< n 3) (display "< 3"))
      (pred-2 exp2-1 exp2-2 ...)            ;        ((> n 4) (display "< 4"))
      ...                                   ;       ...
      (else e-else-1 e-else-2 ...))         ;        (else (display "outros")))
(begin exp1 exp2 ...)                       ; calcula exp1, depois exp2, ...
                                            ; e devolve o valor da última expressão.
(case expressão                             ; (define vogal-ou-consoante-case
      ((chave1-1 chave1-2 ...) exp1-1 ...)  ;   (lambda (letra)
      ((chave2-1 chave2-2 ...) exp2-1 ...)  ;     (case letra
      ...                                   ;       ((a e i o u) 'vogal)
      (else exp-else-1 ...) )               ;       (else 'consoante))))
                                            ;
                                            ; > (vogal-ou-consoante-case 'a)
                                            ; vogal
                                            ; > (vogal-ou-consoante-case 'f)
                                            ; consoante
```

Entrada/Saída

```
(display arg)                 ; (begin (display "E' ")
                              ;        (display 5)
                              ;        (display " um numero par?"))
                              ; visualiza:  E' 5 um numero par?
                              ; (display "Ele disse \"Ola'\".")
                              ; visualiza: Ele disse "Ola'".
(newline)                     ; (begin (display "E' ")
                              ;        (display 5)
                              ;        (newline)
                              ;        (display " um numero par?"))
                              ; E' 5
                              ;  um numero par?
(read)                        ; (let ((x (read)))
```

```
                              ; introduzindo 13 pelo teclado, x vale 13
                              ; mas introduzindo abcd, x vale abcd.
```

Processamento de Pares e Listas

```
(cons x1 x2)                  ; constrói um par.
                              ; (cons 45 2) devolve par (45 . 2).
(car x)                       ; selecciona e devolve o elemento da esquerda de um par.
                              ; (car (cons 45 2)) devolve 45.
(cdr x)                       ; selecciona e devolve o elemento da direita de um par.
                              ; (cdr (cons 45 2)) devolve 2.
(cadr x)                      ; Combina um cdr com um car.
                              ; (cadr (list 1 2 3)) devolve 2.
                              ; O Scheme disponibiliza os seguintes 28
                              ; procedimentos, que são composições de car e cdr:
caar  caaar  cdadr  caadar  cadddr  cddaar  cadr  caadr  cddar
caaddr  cdaaar  cddadr  cdar  cadar  cdddr  cadaar  cdaadr  cdddar
cddr  caddr  caaaar  cadadr  cdadar  cddddr  cdaar  caaadr  caddar  cdaddr
                              ; (caddr x) equivalente a (car (cdr (cdr x))).
(list x1 x2 x3 ...)           ; constrói uma lista.
                              ; (list 1 2 3) devolve lista (1 2 3).
                              ; (list 1 (cons 34 2) 7) devolve (1 (34 . 2) 7).
(quote x)                     ; devolve argumento sem o processar.
(equivalente a 'x)            ; (quote (1 2 3)) devolve lista (1 2 3).
                              ; '(1 2 3) devolve lista (1 2 3).
                              ; (cons elem lista) devolve lista composta por
                              ;                elem e todos elementos de lista.
                              ; (cons 1 '(4 all)) devolve lista (1 4 all).
                              ; (car '((a b) c d)) devolve  (a b).
                              ; (cdr '(a . 1)) devolve 1.
                              ; (cddr '(a ((b c) d) e)) devolve (e).
                              ; (caadr '(a ((b c) d) e)) devolve (b c).
(append list1 list2...)       ; devolve uma lista que inclui os elementos
                              ; de list1, list2, ...
                              ; (append (list 1 2) '(4 outros))
                              ; devolve lista (1 2 4 outros)
(length lista)                ; devolve comprimento de lista.
                              ; (length (list 12 34 1 (list 1 2) 3)) devolve 5.
(list-ref lista x)            ; devolve o elemento de lista de índice x.
                              ; 1º elemento tem índice 0, 2º tem índice 1, ...
                              ; (list-ref (list 1 2 3 4) 2) devolve 3.
(reverse lista)               ; devolve lista com elementos de lista em ordem inversa.
                              ; (reverse (list 1 2 3 4)) devolve lista (4 3 2 1).
(list-tail lista num)         ; devolve a cauda de lista.
                              ; (list-tail lista-1 2) devolve (3 4 5) se
                              ; lista-1 for (1 2 3 4 5).
(map op lista)                ; aplica a operação op a cada elemento da lista
                              ; e devolve a lista resultante.
                              ; (map add1 '(1 3 5 7) devolve (2 4 6 8).
                              ; (map (lambda (x) (+ 2 x)) '(1 3 5)) devolve (3 5 7).
(for-each op lista)           ; aplica a operação op a cada elemento da lista
                              ; mas só interessam os efeitos laterais, como
                              ; seja, a visualização.
```

```
                          ; (for-each display '(1 3 5)) visualiza 135
                          ; (for-each display '("pri" " " "seg")) visualiza
                          ; pri seg
(apply op lista)          ; aplica a operação op a cada elemento da lista
                          ; e devolve o valor resultante.
                          ; (apply + '(1 2 3)) devolve 6.
                          ; (map + '(1 2 3)) devolve (1 2 3).
                          ; (apply + '(1 2 3 2 -2 1)) devolve 7.
                          ; (apply max '(1 2 3 2 -2 1)) devolve 3.
                          ; (apply min '(1 2 3 2 -2 1)) devolve -2.
(member x lista)          ; utiliza equal? para comparar x com os
                          ; elementos de lista e devolve #f se lista não
                          ; contém x, se não devolve a sublista que vai
                          ; desde a ocorrência de x até fim de lista.
                          ; (member 'a '(b c d e)) devolve #f.
                          ; (member '(a) '(b (a) (b a))) devolve ((a) (b a)).
(memq x lista)            ; idêntico a member, mas utilizando eq?.
(memv x lista)            ; idêntico a member, mas utilizando eqv?.
(assoc x lista)           ; devolve o 1º elemento de lista que é uma lista e
                          ; cujo 1º elemento é x.
                          ; (assoc 5 '((2 df) (6) (5 r t) (7 a b c)))
                          ; devolve (5 r t).
                          ; (assoc 6 '((2 df) 5 r t (6) (7 a b c)))
                          ; devolve (6).
(null? x)                 ; devolve #t se x é uma lista vazia, se não devolve #f.
                          ; (null? (list 1 2 3)) devolve #f.
                          ; (null? '()) devolve #t.
(pair? x)                 ; devolve #t, se x for um par, se não devolve #f.
                          ; (pair? (cons 1 2 3)) devolve #t.
                          ; (pair? '()) devolve #f.
                          ; (pair? '(a b)) devolve #t.
(set-car! par obj)        ; obj substitui a parte esquerda de par.
                          ; (define lista '(a b c))
                          ; (set-car! lista 'xyz)
                          ; lista passa a ser (xyz b c).
(set-cdr! var obj)        ; obj substitui a parte direita de par.
                          ; (define lista '(a b c))
                          ; (set-cdr! lista '(xyz))
                          ; lista passa a ser (a xyz).
(set! var exp)            ; neste caso, não tem a ver com pares, mas com
                          ; entidades simples.
                          ; É atribuído a var, variável previamente
                          ; definida, o valor de exp.
                          ; (define x 9)
                          ; (set! x (* x x)) a variável x toma o valor 81.
```

Processamento de Caracteres e Cadeia de caracteres *(strings)*

```
#\B                       ; representa o caracter B.
#\b                       ; representa o caracter b.
#\7                       ; representa o caracter 7.
#\space                   ; representa o caracter "espaço".
#\newline                 ; representa o caracter "nova linha".
```

```
(char?  cx)                    ; devolve #t se cx é caracter, e #f se não for.
                               ; (char? #\5) devolve #t.
                               ; (char? 5) devolve #f.
(char->integer  cx)            ; devolve o código (inteiro) do caracter cx.
                               ; (char->integer #\5) devolve 53.
                               ; (char->integer #\space) devolve 32.
(integer->char  cod)           ; devolve o caracter cujo código é cod.
                               ; (integer->char 37) devolve #\%.
(char=? c1 c2)                 ; devolve #t se os caracteres c1 e c2 forem iguais.
                               ; (char=? #\a #\A) devolve #f.
(char-ci=? c1 c2)              ; devolve #t se os caracteres c1 e c2
                               ; forem iguais, sem ter em conta se são
                               ; letras maiúsculas ou minúsculas.
                               ; (char-ci=? #\a #\A) devolve #t.
char e char-ci também          ; (char>? #\a #\A) devolve #t.
com >?, <?, >=? e <=?          ; (char-ci>? #\a #\A) devolve #f.
(char-upper-case? c)           ; devolve #t se c é letra maiúscula.
(char-down-case? c)            ; devolve #t se c é letra minúscula.
(char-upper-case c)            ; se c for letra, devolve c maiúscula, se não devolve c.
(char-down-case c)             ; se c for letra, devolve c minúscula, se não devolve c.
(char-alphabetic? c)           ; devolve #t se c é uma das letras.
(char-numeric? c)              ; devolve #t se c é um dos dígitos decimais.
(char-whitespace? c)           ; devolve #t se c for "espaço" ou "nova linha".
(string ch1 ...)               ; os argumentos são caracteres e devolve
                               ; uma cadeia de caractares.
                               ; (string #\a #\b #\c) devolve "abc".
(string? arg)                  ; devolve #t, se arg for cadeia de caracteres.
                               ; (string? "abc") devolve #t.
(string-length cadeia)         ; devolve comprimento de cadeia.
                               ; (string-length "This is a string") devolve 16.
                               ; (string-length "") devolve 0.
(string-copy cad)              ; devolve uma cadeia igual ao argumento cad.
(make-string num c)            ; devolve uma cadeia com caracteres c, de comprimento num.
(make-string num)              ; (make-string 3 #\a) devolve "aaa".
                               ; (make-string 3) devolve "   ".
(string-append cad1 cad2 ...)  ; devolve uma cadeia de caracteres,
                               ; concatenando as cadeias cad1, cad2 ...
                               ; (string-append "This is " "a string")
                               ; devolve "This is a string".
(string-ref cadeia k)          ; devolve o elemento de ordem k de cadeia.
                               ; (string-ref "abcd 1234" 2) devolve #\c.
                               ; (string-ref "abcd 1234" 0) devolve #\a.
(substring cadeia inicio fim)  ; devolve uma sub-cadeia.
                               ; (substring "This is a string" 0 4) devolve "This".
                               ; (substring "This is a string" 5 6) devolve "i".
(symbol->string simbolo)       ; constrói uma cadeia a partir de simbolo.
                               ; (symbol->string 'hello) devolve "hello".
(string->symbol cadeia)        ; constrói um símbolo a partir de cadeia.
                               ; (string->symbol "abc") devolve abc.
(list->string lista)           ; constrói uma cadeia a partir de uma lista de caracteres.
                               ; (list->string (list #\a #\b)) devolve "ab".
(string->list cadeia)          ; constrói uma lista a partir de cadeia
                               ; (string->list "ab") devolve (#\a #\b).
(number->string numero)        ; constrói uma cadeia a partir de numero.
```

```
                              ; (number->string 123) devolve "123".
(string->number cadeia)       ; constrói um número a partir de cadeia.
                              ; (string->number "abc") devolve #f.
                              ; (string->number "12345") devolve 12345.
(string=? str1 str2 ...)      ; (string=? "abc" "abc") devolve #t.
                              ; (string=? "abc" "ABC") devolve #f.
(string-ci=? str1 str2 ...)   ; (string-ci? "abc" "ABC") devolve #t.
string e string-ci também     ; (string>? "abc" "ABC") devolve #t.
com >?, <?, >=? e <=?         ; (string-ci>? "abc" "ABC") devolve #f.
(string-fill! cad c)          ; a cadeia cad já existente, é preenchida com caracteres c.
(string-set! cad ind c)       ; o caracter da cadeia cad com o indice
                              ; ind é substituído pelo caracter c.
```

Processamento de Vectores

```
(vector obj1 obj2 ...)        ; constrói um vector com obj1, obj2, ...
                              ; (vector 'a 3 '(a b)) devolve #(a 3 (a b)).
(vector? obj)                 ; devolve #t, se obj é vector.
                              ; (define v1 (vector 'a 6 'abc 90))
                              ; (vector? v1) devolve #t.
(make-vector comp)            ; constrói um vector de comprimento comp,
                              ; em que os seus elementos são todos ().
                              ; (make-vector 3) devolve #(() () ()).
(make-vector comp elem)       ; constrói um vector de comprimento comp,
                              ; em que os seus elementos são todos elem.
                              ; (make-vector 3 'a) devolve #(a a a)
(list->vector lis)            ; constrói um vector a partir da lista lis.
                              ; (list->vector '(1 6 a 7)) devolve #(1 6 a 7).
(vector->list vec)            ; constrói uma lista a partir do vector vec.
                              ; (vector->list (vector 'ab 4)) devolve (ab 4).
(vector-length vec)           ; devolve comprimento do vector vec.
                              ; (define v1 (vector 'a 6 'abc 90))
                              ; (vector-length v1) devolve 4.
(vector-ref vec k)            ; devolve o elemento de índice k do vector vec.
                              ; (define v1 (vector 'a 6 'abc 90))
                              ; (vector-ref v1 1) devolve 6.
                              ; (vector-ref v1 2) devolve abc.
(vector-fill! vec elem)       ; o vector vec existente, é preenchido com elem.
                              ; (define v-3-elementos (vector 1 2 3))
                              ; (vector-fill! vec-3-elementos "ac")
                              ; vec-3-elementos passa a ser #("ac" "ac" "ac").
(vector-set! vec k elem)      ; Modifica o vector vec, trocando o elemento
                              ; de ordem k por elem.
                              ; (define v1 (vector 0 2 4 6 8))
                              ; v1 devolve #(0 2 4 6 8).
                              ; (vector-set! v1 2 5) devolve valor não
                              ; definido e v1 devolve #(0 2 5 6 8).
```

Processamento de Ficheiros

```
(load nome-fich)              ; carrega o ficheiro designado por nome-fich
                              ; e calcula as expressões nele contidas.
                              ; Se se tratar do ficheiro em c:\expfnf\lixo111.txt
                              ; (load c:\\expfnf\\lixo111.txt)
```

```
                              ; atenção à duplicação do carácter \
                              ; pois o Scheme elimina um deles
(open-output-file nome-fich)  ; devolve uma porta de saída que fica
                              ; associada ao ficheiro de saída nome-fich.
                              ; (define f1 (open-output-file "c:\\expfnf\\lixo111.txt"))
                              ; todas as chamadas de newline e display que
                              ; refiram f1 vão para o ficheiro associado.
                              ; (display "isto vai para o ficheiro" f1)
                              ; (newline f1)
                              ; (display 456 f1)
(open-output-file nome-fich 'append)   ; uma variante de open-output-file, que permite
                                       ; acrescentar mais elementos a partir do fim
                                       ; do ficheiro
(open-output-file nome-fich 'replace)  ; uma variante de open-output-file, que permite
                                       ; refazer o conteúdo de um ficheiro existente,
                                       ; limpando-o previamente
(close-output-port porta-s)   ; (close-output-port porta-s) fecha
                              ; o ficheiro de saída associado a porta-s.
(open-input-file nome-fich)   ; devolve uma porta de entrada que fica
                              ; associada ao ficheiro de entrada nome-fich.
                              ; (define porta-e (open-input-file nome-ficheiro)
                              ; todas as chamadas de read que refiram porta-e
                              ; vão buscar dados ao ficheiro associado.
                              ; (read porta-e)
(eof-object? ultimo-elem-lido) ; sempre que um ficheiro é acedido em
                              ; leitura, através de read, o elemento
                              ; lido deverá ser testado a fim de se
                              ; verificar se já se atingiu o fim do ficheiro.
(close-input-port porta-e)    ; (close-output-port porta-e) fecha o
                              ; ficheiro de entrada associado a porta-e.
(file-exists? nome-fich)      ; verifica se o ficheiro nome-fich existe.
(delete-file nome-fich)       ; elimina o ficheiro nome-fich.
                              ; (file-exists? "c:\\expfnf\\lixo111.txt")
                              ; devolve #t se o ficheiro existir.
                              ; (delete-file "c:\\expfnf\\lixo111.txt")
                              ; (file-exists? "c:\\expfnf\\lixo111.txt")
                              ; devolve #f
```

Vários

```
(symbol? x)                   ; devolve #t, se x for símbolo, se não devolve #f.
(procedure? x)                ; devolve #t, se x procedimento, se não devolve #f.
(define (proc arg1 ...)       ; equivalente a (define proc
   exp1  exp2 ...)            ;                   (lambda (arg1 ...)
                              ;                       exp1 exp2 ...))
                              ; (define (frac x)
                              ;    (- x (floor x)))
                              ; (frac -4.25) devolve 0.75.
(let ((var1 init-exp1)        ; define as variáveis var com o valor das
      (var2 init-exp2)        ; expressões init-exp. Depois, no corpo de
         ...          )       ; let, calcula as expressões exp
  exp1 exp2 ...)              ; e devolve valor da última.
                              ; (let ((a 2)         ; a toma o valor 2
                              ;       (b 3))        ; e b o valor 3.
```

```
                              ;   (let ((a 4)       ; c toma o valor 2-3
                              ;         (c (- a b)))   ; e a o valor 4.
                              ;     (* c a)))              ; devolve -4.
(let* ((var1 init-exp1)       ; define, em sequência, as variáveis var com
       (var2 init-exp2)       ; o valor das expressões init-exp. Ou seja,
          ...          )      ; é possível utilizar numa expressão init-exp
    exp1 exp2 ...)            ; uma variável já definada no próprio let*.
                              ; Depois, no corpo de let, calcula as
                              ; expressões exp e devolve valor da última.
                              ; (let* ((a 2)
                              ;        (b (* 2 a)))
                              ;   (+ a b))
                              ; a toma o valor 2 e b 4. É devolvido 4.
(letrec ((var1 init-exp1)     ; define as variáveis var com o valor das
         (var2 init-exp2)     ; expressões init-exp. Estas definições
           ...          )     ; também podem ser procedimentos recursivos
  exp1 exp2 ...)              ; ou mutuamente recursivos.
                              ; Depois, no corpo de letrec, calcula as
                              ; expressões exp e devolve valor da última.
                              ;
                              ; (let ((a 2)          ; a toma o valor 2
                              ;       (b 3))         ; e b o valor 3
                              ;   (letrec ((soma-todos
                              ;              (lambda (x)
                              ;                (if (zero? x)
                              ;                    0
                              ;                    (+ x
                              ;                       (soma-todos
                              ;                           (sub1 x)))))))
                              ;      (soma-todos (* a b))))
(time exp)                    ; calcula exp, visualiza em ms o tempo de cpu
                              ; gasto no cálculo, o tempo real e o tempo
                              ; de recolha de lixo (garbage collection) e,
                              ; finalmente, devolve resultado do cálculo.
                              ;  (define fib
                              ;    (lambda (n)
                              ;      (if (< n 2)
                              ;          n
                              ;          (+ (fib (- n 1))
                              ;             (fib (- n 2))))))
                              ; > (time (fib 31))
                              ; cpu time: 3922 real time: 3953 gc time: 0
                              ; 1346269
                              ; >
(current-inexact-milliseconds) ; devolve um positivo (não obrigatoriamente um inteiro)
                              ; correspondente ao número de milisegundos que passaram
                              ; desde uma certa data (normalmente desde que a máquina foi
                              ; ligada). Este número nunca diminui enquanto a máquina
                              ; permanecer ligada.
                              ; exemplo de procedimento que implica uma certa espera,
                              ;  especificada em milisegundos...
                              ; (define espera
                              ;   (lambda (t-ms)
                              ;     (let ((limite (+ (current-inexact-milliseconds)
```

```
;                       t-ms)))
;       (letrec ((ciclo
;                  (lambda ()
;                    (if (< (current-inexact-milliseconds)
;                           limite)
;                        (ciclo)))))
;           (ciclo)))))
```