Anexo A

Resumo dos principais procedimentos do *Scheme*

Para processamento de booleanos

```
#f                  ; falso
#t                  ; verdadeiro
(boolean? x)        ; se x for booleano, devolve #t; se não, devolve #f
                    ;
(and x1 x2 x3 ...)  ; calcula x1, x2, ..., até encontrar um valor #f, e
                    ; devolve #f
                    ; Não encontrando qualquer valor #f, devolve #t
(or x1 x2 x3 ...)   ; calcula x1, x2, ..., 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
                    ;
```

Para processamento de números

```
(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,... 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) devolve 5; (abs -5) devolve 5
(round x)         ; devolve inteiro mais próximo de x. Existindo 2
                  ; inteiros igualmente distantes, devolve o que for par
                  ; (round 2.7) devolve 3.0
(ceiling x)       ; devolve inteiro igual a x ou imediatamente acima
```

```
                ; (ceiling 5.3) devolve 6.0; (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
(inexact->exact z)  ; devolve o número exacto mais próximo de z
                    ; (inexact->exact 2.7) devolve 3
(exact->inexact z)  ; devolve o número não exacto mais próximo de z
                    ; (exact->inexact 2.7) devolve 2.7
                    ; (exact->inexact 3) devolve 3.0
(max x1 x2 x3 ...)  ; devolve o maior argumento
                    ;
(min x1 x2 x3 ...)  ; devolve o 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
                    ; quando x e d têm o mesmo sinal,
                    ; remainder devolve o mesmo resultado que modulo
(modulo x d)        ; devolve o resto (com o mesmo sinal de d) da
                    ; divisão inteira x/d
                    ; (modulo -7 3) devolve 2
                    ; (modulo 7 -3) devolve -2
                    ; (modulo 7 3) devolve 1
                    ; (modulo -7 -3) devolve -1
                    ; quando x e d têm o mesmo sinal,
                    ; modulo devolve o mesmo resultado que remainder
(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)            ; devolve um inteiro pseudo-aleatório,
                    ; situado entre 0 e 32767
                    ; Não faz parte da definição do Scheme. Assim, em certas
                    ; implementações, random tem um parâmetro.
                    ; Nesses casos, (random n) devolve um inteiro entre 0 e n-1
                    ;
```

Para processamento trigonométrico

```
(degrees->radians x) ; estando x em graus,
                     ; devolve valor correspondente em radianos
(radians->degrees x) ; estando x em radianos,
                     ; devolve valor correspondente em graus
(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
```

De controlo

```
(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)) devolve ()
                                      ;
(cond (predicado-1 exp1-1 exp1-2 ...)   ;  (cond ((< n 3) (display "< 3"))
      (predicado-2 exp2-1 exp2-2 ...)   ;        ((> n 4) (display "< 4"))
      ...                               ;       ...
      (predicado-n expn-1 expn-2 ...))  ;        ((> n 10) (display "> 10")))
                                        ;
(cond (predicado-1 exp1-1 exp1-2 ...)   ;  (cond ((< n 3) (display "< 3"))
      (predicado-2 exp2-1 exp2-2 ...)   ;        ((> n 4) (display "< 4"))
      ...                               ;       ...
      (else exp-else-1 exp-else-2 ...)) ;        (else (display "outros")))
                                        ;
(begin exp1 exp2 ...)                 ; calcula, em sequência, exp1, exp2, ...
                                      ; e devolve o valor da última expressão
```

De Entrada/Saída

```
(display arg)                   ; (begin (display "E' ")
                                ;        (display 5)
                                ;        (display " um numero par?"))
                                ;
                                ; visualiza:  E’ 5 um numero par?
                                ;
                                ; (display "Ele disse \"Olá\".")

                                ;
                                ; visualiza: Ele disse "Olá".
(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
                                ; introduzindo abcd pelo teclado, x vale abcd
                                ;
(set! var exp)              ; 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
```

Para processamento de Pares e Listas

```
(cons x1 x2)            ; (cons 45 2) devolve par (45 . 2)
                        ;
(car x)                 ; (car (cons 45 2)) devolve 45
                        ;
```

```
(cdr x)                 ; (cdr (cons 45 2)) devolve 2
                        ; Todas as implementações de Scheme disponibilizam
                        ; 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
                        ; ex: (caddr x) equivalente a (car (cdr (cdr x)))
(list x1 x2 x3 ...)     ; (list 1 2 3) devolve lista (1 2 3)
                        ; (list 1 4 (cons 34 2) 7) devolve (1 4 (34 . 2) 7)
(quote xyz) ou 'xyz     ; devolve xyz (símbolo)
                        ; (quote (1 2 3)) devolve lista (1 2 3)
                        ; '(1 2 3) devolve lista (1 2 3)
                        ; (cons elem list) devolve lista composta por
                        ;                  elem e todos elementos de list
                        ; (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 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 list x)       ; devolve elemento de list de índice x
                        ; 1º elemento tem 'indice 0, 2º tem 'indice 1, ...
                        ; (list-ref (list 1 2 3 4) 2) devolve 3
(reverse list)          ; devolve lista com elementos de list em ordem inversa
                        ; (reverse (list 1 2 3 4)) devolve lista (4 3 2 1)
(list-tail lista num)   ; (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 uma lista
                      ; (map add1 '(1 3 5 7) devolve (2 4 6 8)
                      ; (map (lambda (x) (+ 2 x)) '(1 3 5 7)) devolve (3 5 7 9)
(for-each op lista)   ; aplica a operação op a cada elemento da lista
                      ; mas só interessam os efeitos laterais, como seja,
                      ; a visualização.
                      ; O valor devolvido não é normalizado, sendo no EdScheme ()
                      ; (for-each display '(1 3 5)) devolve 135()
                      ; (for-each add1 '(1 3 5)) devolve ()
                      ; (for-each display '("pri" " " "seg")) devolve pri seg()
(apply op lista)      ; aplica a operação op a cada elemento da lista
                      ; e devolve um valor único
                      ; (apply + '(1 2 3)) devolve 6
                      ; (map + '(1 2 3)) devolve (1 2 3)
                      ; (for-each + '(1 2 3)) devolve ()
                      ; (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
                      ; 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)        ; idem member, mas utilizando eq?
                      ;
(memv x lista)        ; idem 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)))
                    ; (6)
                    ;
(null? x)           ; devolve #t se x é uma lista vazia; se não, devolve #f
                    ; (null? 4) devolve #f
                    ; (null? '()) devolve #t
(atom? x)           ; devolve #t se x é símbolo, número, booleano, carácter,
                    ; ou cadeia de caracteres; se não, devolve #f
                    ; (atom? 'bomb) devolve #t         ; símbolo
                    ; (atom? 88) devolve #t            ; número
                    ; (atom? #f) devolve #t            ; booleano
                    ; (atom? #\space) devolve #t       ; carácter
                    ; (atom? "bit") devolve #t         ; cadeia de caracteres
                    ; (atom? '()) devolve #f           ; lista vazia
                    ; (atom? '(1 2 3)) devolve #f      ; lista
                    ; (atom? '#(1 2)) devolve #f       ; vector
                    ; (atom? car) devolve #f           ; procedimento
(pair? x)           ; devolve #t, se x for um par, ou seja, lista não vazia
                    ; se não, devolve #f
                    ; (pair? '()) devolve #f
                    ; (pair? '(a b)) devolve #t
                    ;
(set-car! par obj       ; obj é atribuído à componente-car de par
                        ; (define lista '(a b c))
                        ; (set-car! lista 'xyz)
                       ; lista passa a ser (xyz b c)
(set-cdr! var obj)      ; obj é atribuído à componente-cdr de par
                        ; (define lista '(a b c))
                        ; (set-cdr! lista '(xyz))
                        ; lista passa a ser (a xyz)
```

Para 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?  c)                      ; devolve #t se c é caracter, e #f se não for
                                ; (char? #\5) devolve #t
                                ; (char? 5) devolve #f
(char->integer  c)              ; devolve o código do caracter c
                                ; (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->integer #\space) devolve 32
(char=? c1 c2)                  ; devolve #t se os caracteres c1 e c2
                                ; forem iguais
                                ; (char=? #\a #\A) devolve #f
                                ; char
(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 para além de =? têm  ; (char>? #\a #\A) devolve #t
ainda as variantes >, <, >= 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 for letra, devolve c
```

```
(char-down-case c)                ; se c for letra, devolve c minúscula,
                                  ; se não for letra, 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)              ; se c for space ou newline
                                  ;
(string ch1 ...)                  ; 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 a cad
(make-string num c)               ; devolve uma cadeia 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
                                  ; com 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)     ; (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 um símbolo
                                  ; (symbol->string 'hello) devolve "hello"
(string->symbol cadeia)           ; constrói um símbolo a partir de uma cadeia
                                  ; (string->symbol "abc") devolve abc
(string->list cadeia)             ; constrói uma lista a partir de uma cadeia
                                  ; (string->list "abc") devolve (#\a #\b #\c)
(string->number cadeia)           ; constrói um número a partir de uma cadeia
                                  ; (string->number "abc") devolve #f
                                  ; (string->number "12345") devolve 12345
(string=? string1 string2 ...)    ; (string=? "abc" "abc") devolve #t
                                  ; (string=? "abc" "ABC") devolve #f
(string-ci=? string1 string2 ...) ; (string-ci? "abc" "ABC") devolve #t
                                  ;
string e string-ci para além de =? ; (string>? "abc" "ABC") devolve #t
têm as variantes >, <, >= e <=    ; (string-ci>? "abc" "ABC") devolve #f
                                  ;
(string-fill! cad c)              ; a cadeia cad já existente, é preenchida
                                  ; com o caracter c
(string-set! cad ind c)           ; o caracter da cadeia cad com o indice ind
                                  ; é substituído pelo caracter c
```

Para processamento de Vectores

```
(vector obj1 obj2 ...)     ; constrói um vector com obj1, obj2, ...
                           ; (vector 'a 5 3 'c '(a b)) devolve #(a 5 3 c (a b))
(vector? obj)              ; devolve #t, se obj é vector
                           ; (define v1 (vector 'a 6 'abc 90)) devolve v1
                           ; (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 2 6 a 7)) devolve #(1 2 6 a 7)
(vector->list vec)        ; constrói uma lista a partir do vector vec
                          ; (vector->list (vector 'abc 3 4)) devolve (abc 3 4)
(vector-length vec)       ; devolve comprimento do vector vec
                          ; (define v1 (vector 'a 6 'abc 90)) devolve v1
                          ; (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)) devolve v1
                          ; (vector-ref v1 1) devolve 6
                          ; (vector-ref v1 2) devolve abc
(vector-copy vec)         ; devolve uma cópia do vector vec
                          ; (define v1 (vector 'a 2 't)) devolve v1
                          ; (define v2 (vector-copy v1)) devolve v2
                          ; v2 devolve #('a 2 't)
(vector-fill! vec elem)   ; o vector vec já existente, é preenchido com elem
                          ; (define v-3-elementos (vector 1 2 3))
                          ; (vector-fill! vec-3-elementos "abc")
                          ; modifica vec-3-elementos para #("abc" "abc" "abc")
(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)) devolve v1
                          ;                        v1 devolve #(0 2 4 6 8)
                          ; (vector-set! v1 2 5) devolve valor não definido
                          ;                      v1 devolve #(0 2 5 6 8)
```

Para processamento de Ficheiros

```
(load filename)               ; carrega ficheiro designado por filename
                              ; e calcula as expressões nele contidas
(open-output-file nome-fich)  ; devolve uma porta de saída que fica associada
                              ; ao ficheiro de saída nome-fich
                              ; (define port-s (open-output-file nome-ficheiro)
                              ; todas as chamadas de newline e display que
                              ; refiram port-s vão para o ficheiro associado
                              ; (display "isto vai para o ficheiro" porta-s)
                              ;
(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 port-e (open-input-file nome-ficheiro)
                              ; todas as chamadas de read que refiram port-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-s) fecha o ficheiro
                              ; de saída associado a porta-s
```

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 (var arg1 ...)    ; equivalente a:
   exp1  exp2 ...)        ;   (define var (lambda (arg1 ...) exp1 exp2 ...))
                          ; (define (frac x)
                          ;    (- x (floor x))) devolve valor não especificado
                          ; (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 e devolve valor da última
  exp1 exp2 ...)            ;
                            ; (let ((a 2) (b 3))          ; a toma valor 2 e b 3
                            ;  (let ((a 4) (c (- a b)))    ; c toma valor 2 - 3
                            ;                               ; e a toma valor 4
                            ;    (* c a)))                  ; (* -1 4)
                            ;                               ; devolve –4
(letrec ((var1 init-exp1)   ; define as variáveis var com o valor das
         (var2 init-exp2)   ; expressões init-exp. Estas definições podem ser
        ...          )      ; procedimentos recursivos ou mutuamente recursivos.
  exp1 exp2 ...)            ; 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))))
                            ;
(runtime)                   ; devolve inteiro com a indicação do número de
                            ; mili-segundos decorridos desde o início da sessão
```