```
;;;;============================================================================;;;;
;;;; Common Lisp: A Gentle Introduction to Symbolic Computation                 ;;;;
;;;; http://www.paulgraham.com/acl.html                                         ;;;;
;;;;----------------------------------------------------------------------------;;;;
;;;; Capítulo: 2                                                               ;;;;
;;;;----------------------------------------------------------------------------;;;;
;;;; Por: Abrantes Araújo Silva Filho                                          ;;;;
;;;;      abrantesasf@pm.me                                                     ;;;;
;;;;============================================================================;;;;


;;; 2.1: Lists are the most versatile data structure:
;;; -------------------------------------------------

;; Lisp = LISt Processing

;; Listas são versáteis pois servem para representar praticamente tudo:
;;    - conjuntos
;;    - tabelas
;;    - grafos
;;    - frases
;;    - funções


;;; 2.2: Uma lista se parece com o quê?
;;; -----------------------------------

;; Listas têm 2 formas de representação:
;;   - impressa (tela/papel)
;;   - interna  (na memória do computador)

;; Representação IMPRESSA:
;;   (1 2 3)
;;   (casa carro 3 nil)
;;   ((sapo perereca) (vaca boi))

;; Representação INTERNA:
;;   - Cadeia de CONS (cons cells)
;;   - Cada CONS consiste de 2 células de memória "adjacentes", e cada
;;     célula desse par (CONS) contém um PONTEIRO (assim, cada CONS contém
;;     dois ponteiros, um em cada célula):
;;        - PRIMEIRO PONTEIRO: sempre aponta para um elemento da lista;
;;        - SEGUNDO PONTEIRO:  sempre aponta para o próximo CONS na cadeira
;;          (ou para NIL se não há mais CONS na cadeia).


;;; 2.3: Lists of one element:
;;; --------------------------

;; Um símbolo e uma lista de um elemento não são a mesma coisa! A lista
;; (CASA) é representada por um CONS onde o ponteiro da primeira célula do CONS
;; aponta para o símbolo CASA:

;         |---|---|
;         |   |   ---> NIL
;         |---|---|
;           |
;           V
;         CASA
```

```
;;; 2.4: Nested Lists:
;;; ------------------

;; Listas podem ser ANINHADAS indefinidamente. Em uma lista aninhada o ponteiro
;; da primeira célula do CONS aponta para OUTRO cons que, este sim, aponta para
;; o elemento (símbolo) armazenado.

;; Listas que não são aninhadas são ditas PLANAS.


;;; 2.5: Length of lists:
;;; ---------------------

;; O tamanho de uma lista é o número de elementos que ela tem, no mesmo nível
;; (sem contar os elementos das listas aninhadas).


(length '(a (b c) d))
(length '(red green blue))
(length '(foo 937 gleep glorp))
(length '(roy (two white ducks) ((melted) (butter))))
(length '(casa))
(length 'casa)


;;; 2.6: NIL, the empty list
;;; ------------------------

;; NIL também corresponde a uma LISTA VAZIA, com 0 elementos.
;; Uma lista vazia (NIL), atenção!, NÃO TEM NENHUM CONS. Ela é representada
;; internamente apenas pelo símbolo NIL e, na forma impressa, por um par
;; de parênteses sem nenhum elemento: ()

;; NIL é a únia coisa que é um SÍMBOLO e também uma LISTA:
;;    nil = NIL = () = 'nil = 'NIL = '()

(length nil)
(length '())
(length '(a nil b))
(length '(nil nil nil nil nil))
(length '((nil nil nil nil)))


;;; 2.7: Equality of lists
;;; ----------------------

;; Duas listas são iguais (EQUAL) se seus elementos também forem iguais
;; (se as cadeias de CONS também forem iguais).

(equal '(a (b c) d) '(a b (c d)))


;;; 2.8: First, Second, Third, ..., tenth, and rest:
;;; ------------------------------------------------

;; Funções primitivas para extrair elementos de uma lista:
;;    - first          - second
;;    - third          - fourth
;;    - fifth          - sixth
;;    - seventh        - eighth
```

```
;;    - ninth          - tenth
;;    - rest

(first '(a b c d))
(second '(a b c d))
(third '(a b c d))
(rest '(a b c d))
(rest (rest '(a b c d)))

(defun my-second (lst)
  (first (rest lst)))

(my-second '(a b c d))


;;; 2.9: Functions operates on pointers:
;;; ------------------------------------

;; O input de uma função não é o objeto propriamente dito, uma lista ou um
;; símbolo, mas, sim, um PONTEIRO até o objeto.

;; Quando enviamos uma lista para uma função, não enviamos a lista em si:
;; estamos enviando um PONTEIRO para o primeiro CONS da lista.

;; Quando a função retorna uma lista, não está retornando a lista em si:
;; está retornando um PONTEIRO para o primeiro CONS desse lista resultado.


;;; 2.10: Car and Cdr:
;;; ------------------

;; Um CONS é um par de células, cada uma contendo um ponteiro: a primeira célula
;; tem um ponteiro que aponta para o conteúdo, e a segunda célula tem um
;; ponteiro que aponta para NIL ou para outro CONS:

[*|*]--->NIL
 |
 v
CONTEUDO

[*|*]----->[*|*]--->NIL
 |          |
 v          v
CONTEUDO   CONTEUDO

;; A primeira célula em um CONS é chamada de:
;;    CAR: Contens of Address portion of Register

;; A segunda célula em um CONS é chamada de:
;;   CDR: Contents of Decrement portion of Regiter

;; Além disso, CAR e CDR também são funções primitivas do Lisp:
;;   - car: retorna o ponteiro da célula CAR, o ELEMENTO ou uma LISTA
;;   - cdr: retorna o ponteiro da célula CDR, sempre no formato de uma LISTA

(car '(a b c d e))
(cdr '(a b c d e))

;; Note que a função FIRST retorna o CAR de uma lista, e
;; a função REST retorna o CDR de uma lista.
```

```
;; Se a lista tem um único elemento, CDR retorna NIL:

[*|*]--->NIL
 |
 v
 A

(cdr '(a))

[*|*]--->NIL
 |
 v
[*|*]--->[*|*]--->NIL
 |        |
 v        v
BOI      VACA

(car '((boi vaca)))
(cdr '((boi vaca)))

;; As funções CAR e CDR podem ser combinadas de diversas formas, e Lisp
;; tem várias funções built-in com essas combinações (30 combinações!),
;; da forma: C<a|d>[a|d][a|d][a|d]R
;; Lembre-se: as combinações são "aplicadas" da "direita para esquerda",
;; ou seja, os "A" e os "D" são aplicados da direita para a esquerda.

Isso...         É equivalente a isso...
(car x)         (car x)
(cdr x)         (cdr x)
(caar x)        (car (car x))
(cadr x)        (car (cdr x))
(cdar x)        (cdr (car x))
(cddr x)        (cdr (cdr x))
(caaar x)       (car (car (car x)))
(caadr x)       (car (car (cdr x)))
(cadar x)       (car (cdr (car x)))
(caddr x)       (car (cdr (cdr x)))
(cdaar x)       (cdr (car (car x)))
(cdadr x)       (cdr (car (cdr x)))
(cddar x)       (cdr (cdr (car x)))
(cdddr x)       (cdr (cdr (cdr x)))
(caaaar x)      (car (car (car (car x))))
(caaadr x)      (car (car (car (cdr x))))
(caadar x)      (car (car (cdr (car x))))
(caaddr x)      (car (car (cdr (cdr x))))
(cadaar x)      (car (cdr (car (car x))))
(cadadr x)      (car (cdr (car (cdr x))))
(caddar x)      (car (cdr (cdr (car x))))
(cadddr x)      (car (cdr (cdr (cdr x))))
(cdaaar x)      (cdr (car (car (car x))))
(cdaadr x)      (cdr (car (car (cdr x))))
(cdadar x)      (cdr (car (cdr (car x))))
(cdaddr x)      (cdr (car (cdr (cdr x))))
(cddaar x)      (cdr (cdr (car (car x))))
(cddadr x)      (cdr (cdr (car (cdr x))))
(cdddar x)      (cdr (cdr (cdr (car x))))
(cddddr x)      (cdr (cdr (cdr (cdr x))))

(cdar '((fee fie) (foe fum)))
```

```
;; Note que as funções FIRST...TENTH são definidas em termos das combinações
;; de CAR e CDR:

(first list)    ==   (car list)                          == (car list)
(second list)   ==   (car (cdr list))                    == (cadr list)
(third list)    ==   (car (cddr list))                   == (caddr list)
(fourth list)   ==   (car (cdddr list))                  == (cadddr list)
(fifth list)    ==   (car (cddddr list))
(sixth list)    ==   (car (cdr (cddddr list)))
(seventh list)  ==   (car (cddr (cddddr list)))
(eighth list)   ==   (car (cdddr (cddddr list)))
(ninth list)    ==   (car (cddddr (cddddr list)))
(tenth list)    ==   (car (cdr (cddddr (cddddr list))))
(rest list)     ==   (cdr list)                          == (cdr list)

;; Lembre: CDR sempre, SEMPRE, retorna uma LISTA (nil também é lista!).
;;         CAR pode retornar um ELEMENTO ou uma LISTA.

;; CAR e CDR de nil, sempre retornam nil.
(car nil)
(cdr nil)


;;; 2.11: CONS:
;;; -----------

;; A função CONS (de CONStruct) cria novas células CONS, a partir de 2 inputs.
;; A célula CAR do novo cons tem um ponteiro que aponta para o primeiro input,
;; e a célular CDR do novo cons tem um ponteiro que aponta para o segundo input.

;; De forma "genérica": CONS acrescenta um elemento na frente de uma lista.

(cons 'a nil)
(cons 'a '(b c d))

(defun greet (x)
  (if (listp x)
      (cons 'hello x)
      (cons 'hello (list x))))

(greet 'world)
(greet '(there miss doolittle))

(cons nil nil)
(cons '(phone home) nil)

;; Se o primeiro input da função CONS for uma lista, o resultado será uma
;; lista aninhada.

(cons '(fred) '(and ginger))
(cons '(fred) '((and ginger)))

;; Podemos usar CONS para construir listas a partir do zero:

(cons '1 (cons '2 (cons '3 (cons '4 ()))))


;;; 2.12: Symmetry of CONS e CAR/CDR:
;;; ---------------------------------
```

```
;; Para listas não vazias, x, temos a seguinte relação:
;;    x = (cons (car x) (cdr x))
;; Ou seja: a lista x é o CONS do CAR de x, com o CDR de x.

;; Essa relação NÃO VALE para listas vazias, NIL, pois se
;; fizermos "(cons (car nil) (cdr nil))" NÃO OBTEREMOS a lista
;; vazia NIL novamente, obteremos a lista (NIL) que é uma lista
;; com 1 elemento: outra lista vazia.

;; NIL é diferente de (NIL)!
;; NIL não é uma célula cons! Nil é nil!

(sdraw nil)

NIL

(sdraw '(nil))

[*|*]--->NIL
 |
 v
NIL


;;; 2.13: LIST:
;;; -----------

;; Criar listas usando CONS é trabalhaso pois o comando CONS só cria uma célula
;; cons de cada vez (e assim temos que usar vários CONS aninhados). Para
;; resolver esse problema temos o comando LIST:

(list 'a 'b 'c)

;; LIST cria células cons para cada argumento e os ponteiros necessários,
;; retornando assim a lista final pronta.

(list '(foo))
(list 'sun nil)
(list 'a '(b c) 'd)
(list nil)

(defun blurt (x y)
  (list x 'é 'um y))

(blurt 'abrantes 'estudioso)


;;; 2.14: Replacing the first element of a list:
;;; --------------------------------------------

(defun say-what (lst)
  (cons 'what (rest lst)))

(say-what '(take a nap))

(defun say-what2 (lst)
  (format t "~A" (cons 'what (rest lst)))
  lst)
```

```
(say-what2 '(take a nap))


;;; 2.15: LISTP, CONSP, ATOM, NULL, predicates:
;;; -------------------------------------------

;; LISTP é um predicado que retorna T se o input é uma lista, ou NIL caso não.

(listp 'stitch)
(listp '(a b c))

;; CONSTP é um predicado que retorna T se o input é uma cons cell, ou NIL.

(consp 'a)
(consp '(a))

;; Atenção: NIL é uma lista, mas NÃO É uma cons cell!

(listp nil)  =>  T
(consp nil)  =>  NIL

;; ATOM é um predicado que retorna T se o input NÃO É uma cons cell, ou NIL.

(atom 'a)    => T   (não é uma cons cell)
(atom '(a))  => nil (é uma cons cell)
(atom nil)   => T   (não é uma cons cell)

;; Note que ATOM e CONSP são opostos: quando um retorna T o outro retorna NIL:
;;   CONSP: retorna T se o input   É   uma cons cell
;;   ATOM:  retorna T se o input NÃO É uma cons cell

;; Lembre-se: NIL é uma lista;
;;            NIL não é uma cons cell
;; Por isso que:

(listp nil)  => T   (nil é uma lista)
(consp nil)  => NIL (nil não é uma cons cell)
(atom  nil)  => T   (nil não é uma cons cell)

;; NULL é um predicado que retorna T se o input é NIL

(null nil)     => T   (o input é NIL)
(null ())      => T   (o input é NIL)
(null '(nil))  => nil (é uma lista com 1 elemento, o NIL)

;; Lembre-se: NIL = 'NIL = () = '()
```