Aulas práticas 1 e 2 de Inteligência Artificial@IST

Referências bibliograficas:
* Paul Graham, Ansi Common Lisp, Prentice Hall, 1996. Livro com descrição da linguagem Common-Lisp.

* António Leitão, Introdução à Linguagem Lisp. Disponível na página da cadeira. Folhas com descrição parcial da linguagem Common-Lisp.

* Hyperspec. Disponível na página da cadeira. Descrição detalhada da linguagem Common-Lisp.

Common-Lisp funciona num ciclo de read-eval-print: leitura de expressões, avaliação de expressões, escrita de resultados.

Durante a leitura de uma expressão, se o interpretador encontrar um ponto e vírgula (;), ignora o texto até ao fim da linha. Se encontrar #|, ignora o texto até |#.

Avaliação de expressões é feita de acordo com um conjunto de regras, que se apresentam a seguir de forma simplificada:

```
;- Números
;Há uma enorme variedade de números em Common-lisp. Avaliam-se para
;eles próprios.
> 1
1

> 12345425425369906840
12345425425369906840

> 3.14159
3.14159

;- Caracteres
;Escrevem-se com o prefixo "#\" e avaliam-se para eles próprios.
> #\z
#\z

> #\space
#\space

;- Strings
;Escrevem-se entre aspas (") e avaliam-se para elas próprias.
> "teste"
"teste"

;- Símbolos (diferente do SCHEME)
;São estruturas com os campos: nome (uma string); valor;
;função; lista de propriedades; e package. Componentes acessíveis
;através das funções symbol-name, symbol-value, symbol-function,
;symbol-plist e symbol-package.
;O resultado de avaliar um símbolo é o symbol-value do símbolo.

;constantes t e nil: símbolos pré-definidos cujo valor é o próprio
;símbolo.

> t
T
```

```
;(muitas implementações capitalizam os nomes dos símbolos)

> nil
NIL
;nil representa valor lógico falso. Tudo o que não for nil é verdade.
;t é normalmente usado para representar o valor lógico verdade.
;nil também representa a lista vazia.

;Símbolos que pertencem à package das keywords: escrevem-se :<nome>
;e tem como valor o próprio símbolo.
> :arg
:ARG

;- Combinações
;Escrevem-se entre parenteses em notação prefixa.
;Primeiro elemento da combinação deve ter associado a sí uma função.
;Restantes argumentos das combinações são avaliados recursivamente.
;Funções numéricas
> (+ 1 2.4 6.2)
9.6

> (- 5 (+ 1 1))
3

> (* 5 4 3 2 1)
120

> (/ 5 4)
5/4

> (/ 5 4 3)
5/12

> (1+ 4)
5

> (1- 3)
2

;Predicados numéricos
> (> 5 4 3)
T

> (> 5 4 6)
NIL

> (<= 3 2)
NIL

> (= 1 2) ;igualdade entre números
NIL

> (= 1 1 1)
T

> (zerop 0) ;reconhecedor do 0
T

> (integerp 3)
T

;Igualdade de ponteiros
> (eq  :ola :ola)
```

```
T

> (eq t t)
T

> (eq 1234567890 1234567890)
???
; Números grandes são representados por objectos estruturados.
; Se for o caso de 1234567890 ser um número grande, quando este número
; aparece pela primeira vez é criado um objecto estruturado. Quando
; aparece pela segunda vez, é criado um outro objecto estruturado para
; representar o mesmo número. Mas os objectos estruturados são diferentes.


;Predicados de ponteiros/números
;eql compara ponteiros e números, se os objectos forem números. O problema
; anterior deixa de se colocar.
> (eql 1 1)
T

> (eql 1234567890 1234567890)
T

;Funções sobre pares
;cons: constroi um par;
;consp: verifica se um objecto é um par;
;car: retorna o primeiro elemento do par;
;cdr: retorna o segundo elemento do par;
;caar, cadr, etc: composição de car com car, car com cdr, etc.
> (cons 1 2)
(1 . 2)

> (car (cons  1 2))
1

> (cdr (cons  1 2))
2

> (caar (cons (cons 1 2) (cons 3 4)))
1

;Funções sobre listas
;null: reconhecedor da lista vazia nil ou ();
;make-list: construtor de lista;
;list: construtor de lista;
;cons: construtor de lista;
;first, rest, second, ..., nineth: selectores de listas;
;equal: predicado que verifica se dois objectos lisp têm a mesma estrutura.
;nth, length, append: funções de alto nível;
> ()
NIL

> (null ())
T

> (list 1 2 3 4 5)
(1 2 3 4 5)

> (cons 1 ())
(1)

> (make-list 3 :initial-element 4)
(4 4 4)
```

```
> (cons () ())
(NIL)

> (cons 1 (list 2 3))
(1 2 3)

> (first (list 1 2 3))
1

> (rest (list 1 2 3))
(2 3)

> (eq (list 1) (list 1))
???

> (equal (list 1) (list 1))
T

> (nth 2 (list 0 1 2 3 4))
2

> (length (list 0 1 2 3 4))
5

> (append (list 1 2) (list 3 4))
(1 2 3 4)


; Definição de variáveis
; defparameter: define variável global e atribui-lhe um valor. As variáveis
; globais devem ser tão pouco utilizadas quanto possível e os nomes costumam-s
e
; por entre ``*'', para serem facilmente distinguidas.

> (defparameter *GLOBAL1* 4)
*GLOBAL1*

; Pode-se usar alternativamente o defvar mas é preciso cuidado adicional neste
; caso uma vez que este comando é ignorado se a variável já tiver sido
; definida anteriormente.

> (defvar *global* 4)
*GLOBAL*

> (defvar *global* 5)
*GLOBAL*

> *GLOBAL*
4

; Sugere-se a utilização do defvar sem o valor associado ou a utilização do
; defparameter.

> (defvar *global*)
*GLOBAL*


;atribuição: atribui um valor a um lugar. O lugar pode ser uma variável,
; o car de um par, etc.
; (setf lugar1 valor1
;       lugar2 valor2
;       ...)
```

```
; retorna o último valor a ter sido atribuído.
> (setf par (cons 1 2)
        (car par) 3)
3

> par
(3 . 2)

;Formas especiais:
;and: Avalia da argumentos da esquerda para a direita. Se todos os
;argumentos forem diferentes de NIL retorna o último argumento
;avaliado. Se algum dos argumentos for NIL, interrompe a avaliação e
;retorna NIL.
> (and (< 2 1) (null ()))
NIL

> (and (> 3 2 1) 3 (null ()))
T

;or: Avalia da argumentos da esquerda para a direita. Se todos os
;argumentos forem NIL retorna NIL. Se algum dos argumentos for
;diferente de NIL, interrompe a avaliação e retorna este argumento.
> (or (> 2 1) (null ()))
T

> (or (< 2 1) 3)
3


> (or (< 2 1) NIL)
NIL

; not: nega o argumento. Tudo o que não for  NIL é verdade.
; NOTA: o not não é uma forma especial.
> (not nil)
T

> (not t)
NIL

> (not 1)
NIL

;quote: não avalia o argumento.
> (quote (+ 2 3))
(+ 2 3)

> '(+ 2 3) ;transforma-se em (quote (+ 2 3)), que é avaliado para
(+ 2 3)

> ''(1 2 3)
'(1 2 3)

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

> '((1 2) . (3 4))
???

;defun: associa uma lambda à função associada a um símbolo da lista
;global de símbolos.
```

```
> (defun soma (a b) (+ a b))
SOMA

; Recursão: identificar casos triviais e casos não triviais (com if)
;Ex: potencia
> (defun potencia (base exp)
    (if (= exp 0)
        1
      (* base (potencia base (1- exp)))))
POTENCIA

; Rastreio de funções
> (trace potencia)
(POTENCIA)

>(potencia 2 3)
 0: (POTENCIA 2 3)
   1: (POTENCIA 2 2)
     2: (POTENCIA 2 1)
       3: (POTENCIA 2 0)
       3: returned 1
     2: returned 2
   1: returned 4
 0: returned 8
8

> (untrace potencia)

;Ex: Uma lista é representada como uma sequência de pares. Definir uma função
;que retorna o último par da sequência de pares que constitui uma lista (com c
ond).
> (defun ultimo (l)
    "Retorna o último par de uma lista não vazia"
    (cond ((null (rest l)) l)
          (t (ultimo (rest l)))))
ULTIMO

;Ex: construir uma nova lista que corresponde a juntar duas listas
> (defun junta (l1 l2)
    "Junta duas listas"
    (if (null l1)
        l2
      (cons (first l1)
            (junta (rest l1) l2))))
JUNTA

; Ex: função que inverte uma lista usando processo recursivo
> (defun inverte (l1)
    "Inverte uma lista"
    (if (null l1)
        ()
      (junta (inverte (rest l1))
             (list (first l1)))))
INVERTE

> (defun sublistas (l)
    "Retorna a lista de sublistas da lista recebida como argumento"
    (if (null l)
        ()
      (cons l (sublistas (rest l)))))
SUBLISTAS
```

```
; As estruturas das sublistas são partilhadas. O que acontece
; se mudarmos o elemento da última sublista?
> (setf sublistas (sublistas '(1 2 3)))
((1 2 3) (2 3) (3))

> (ultimo sublistas)
((3))

> (car (ultimo sublistas))
(3)

> (setf (car (car (ultimo sublistas))) 1)
1

> (car (ultimo sublistas))
(1)

> sublistas
((1 2 1) (2 1) (1))
; Porquê? Percebe-se facilmente a razão se os cons forem representados grafica
mente.



; Não se deve usar defun dentro de defun

> (defun iterativo (n l)
    (mapcar #'(lambda (el) (+ n el)) l))
ITERATIVO

> (iterativo 2 '(4 5 6))
(6 7 8)

;Ex: soma n args
> (defun soma (&rest l)
    (defun iterativo (res l)
      (if (null l)
          res
        (iterativo (+ (first l) res) (rest l))))
    (iterativo 0 l))
SOMA

> (soma 1 2 3)
6

> (iterativo 2 '(4 5 6))
17

;função iterativo alterada relativamente a definição anterior.
;O defun associa funções ao slot symbol-function dos símbolos
;de uma lista global. Quando se faz o defun da função iterativo
;dentro da função soma, NÃO estamos a alterar a função local,
;como acontecia em Scheme, mas sim a função global. Alternativamente,
> (defun soma (&rest l)
    (labels ((iterativo (res l)
               (if (null l)
                   res
                  (iterativo (+ (first l) res) (rest l)))))
       (iterativo 0 l))
SOMA
```

```
;Outros exemplos
> (defun meu-equal (obj1 obj2)
    "Compara estruturas de cons de ponteiros/numeros"
    (cond ((atom obj1) (eq obj1 obj2))
          ((consp obj1)
           (and (consp obj2)
                (or (eql (car obj1) (car obj2))
                    (meu-equal (car obj1) (car obj2)))
                (or (eql (cdr obj1) (cdr obj2))
                    (meu-equal (cdr obj1) (cdr obj2)))))))
MEU-EQUAL

> (defun posicao (obj lista)
    "Retorna a posição de um objecto numa lista (-1 se não encontrar)"
    (posicao-aux obj 0 lista))
POSICAO

> (defun posicao-aux (obj pos l)
    (if (null l)
        -1
      (if (equal obj (first l))
          pos
        (posicao-aux obj (1+ pos) (rest l)))))
POSICAO-AUX

> (defun retira (obj l)
    "Retira todas as ocorrências de um objecto de uma lista"
    (cond ((null l) ())
          ((equal obj (first l)) (retira obj (rest l)))
          (t (cons (first l) (retira obj (rest l))))))
RETIRA

> (defun substitui (velho novo l)
    "Substitui todas as ocorrências de velho por novo na lista"
    (if (null l)
        ()
      (if (equal velho (first l))
          (cons novo (substitui velho novo (rest l)))
        (cons  (first l) (substitui velho novo (rest l)))
SUBSTITUI




;Tipos de parâmetros formais (versão simplificada)
; (var*
   [&optional {var | (var [init-form [supplied-p-parameter]])}*]
   [&rest var]
   [&key {var | (var init-form)}]
   [&aux {var | (var [init-form])}*])

> (defun inverte (l &optional res)
    (if (null l)
        res
      (inverte (rest l) (cons (first l) res))))
INVERTE

> (defun meu-list (&rest args)
    args)
MEU-LIST
```

```
> (defun f (a &key (b 2) c d)
    (list a b c d))
F

> (f 1 :b 5)
(1 5 NIL NIL)

> (f 1 :d 4)
(1 2 NIL 4)


;;Tabelas
;
;
;Construtor
;(make-array <dimensões> [...])
> (setf exemplo1 (make-array '(2 3)))
#2A((nil nil nil) (nil nil nil))

> (setf exemplo2 (make-array '(2 3) :initial-element :b))
#2A((:B :B :B) (:B :B :B))

> (setf exemplo3 (make-array '(4) :element-type 'bit :initial-contents '(0 1 0
 1)))
#1A((0 1 0 1))

> (setf exemplo4 #2A((1 2 3) (4 5 6)))
#2A((1 2 3) (4 5 6))

;Selectores: aref, array-dimensions, array-dimension
> (aref exemplo4 1 2)
6

> (array-dimensions exemplo4)
(2 3)

> (array-dimension exemplo4 0)
2

;Predicados: arrayp, array-in-bounds-p
> (arrayp #2A((1 2 3) (4 5 6)))
T

> (arrayp '((1 2 3) (4 5 6)))
NIL

> (array-in-bounds-p #2A((1 2 3) (4 5 6)) 1 2)
T

> (array-in-bounds-p exemplo4 1 3)
NIL

;Modificadores: setf
> (setf (aref exemplo4 1 2) 7)
7

> exemplo4
#2A((1 2 3) (4 5 7))



;Ciclos dolist e dotimes
;
```

```
;(dolist (<var> <lista> [<resultado>]) <corpo>)
;Valor de <var> varia desde o primeiro elemento da lista ao último. Para cada
;valor de var é avaliado o corpo.
;
;(dotimes (<var> <inteiro> [<resultado>]) <corpo>)
;Valor de <var> varia desde o 0 ao inteiro. Para cada
;valor de var é avaliado o corpo.
;
;<resultado> é valor a retornar no fim do ciclo (nil por omissão).
;
;(return <exp>) interrompe ciclo retornando avaliação de <exp>
;(return-from <função> <exp>) retorna da <função> o resultado de
;avaliar <exp> (útil para sair de 2 ou mais ciclos embebidos).

> (defun procura (lista obj)
    (dolist (el lista nil)
      (when (equal el obj)
        (return obj))))
PROCURA

> (defun procura-matriz (matriz obj)
    "Predicado que verifica se uma matriz tem um elemento"
    (dotimes (linha (array-dimension coluna 0) nil)
      (dotimes (coluna (array-dimension matriz 1))
        (when (equal obj (aref matriz linha coluna))
          (return-from procura-matriz obj)))))
PROCURA-MATRIZ-P

; ciclo do e do*
;
;(do ({<var> | (<var> [<valor-inicial> [<incremento>]])}*)
;    (<teste-fim-ciclo> <expressão>*)
;  <corpo>)
;
;do* igual mas as variáveis são definidas em paralelo.
;
;
> (defun inverte (lista)
    (do ((iter lista (rest iter))    ;var iter
         (res '()))                  ;var res
        ((null iter) res)            ;teste fim de ciclo
      (setf res (cons (first iter) res))))
INVERTE

;ciclo loop
;
;(loop <exp>*)
;
> (defun inverte (lista &aux res)
    (loop
     (when (null lista)
       (return res))
     (setf res (cons (first lista) res)
           lista (rest lista))))
INVERTE


; Estruturas
;
;Definição de estruturas: (defstruct <tipo> <campo>+)
;Cria automaticamente construtor (make-<tipo> [:<campo> <valor>]*)
;selectores <tipo>-<campo>, predicado <tipo>-p, copiador (copy-<tipo> <obj>)
;e modificadores (setf (<tipo>-<campo> <obj>) <valor>).
```

```
> (defstruct personagem nome outro-nome (tipo :HEROI))
PERSONAGEM

> (setf costureira (make-personagem :nome "Edna Moda" :tipo :personagem-secund
aria))
#S(PERSONAGEM :NOME "Edna Moda" :OUTRO-NOME NIL :TIPO :PERSONAGEM-SECUNDARIA)

> (setf heroi #S(PERSONAGEM :NOME "Roberto Pera" :OUTRO-NOME "Sr. Incrivel" :T
IPO :HEROI))
#S(PERSONAGEM :NOME "Roberto Pera" :OUTRO-NOME "Sr. Incrivel" :TIPO :HEROI)

> (personagem-nome heroi)
"Roberto Pera"

> (personagem-outro-nome heroi)
"Sr. Incrivel"

> (personagem-p heroi)
T

> (setf clone (copy-personagem heroi))
#S(PERSONAGEM :NOME "Roberto Pera" :OUTRO-NOME "Sr. Incrivel" :TIPO :HEROI)

> (eq heroi clone)
NIL

> (setf (personagem-tipo clone) :clone)
:CLONE

> clone
#S(PERSONAGEM :NOME "Roberto Pera" :OUTRO-NOME "Sr. Incrivel" :TIPO :CLONE)

> (personagem-outro-nome clone)
"Sr. Incrivel"

> (setf (aref (personagem-outro-nome clone) 0) #\M)
#\M

> (personagem-outro-nome clone)
"Mr. Incrivel"

> (personagem-outro-nome heroi)
???
;Porquê?



;;Funções de ordem superior
;
; (function <obj>): função associada ao objecto
> (function cons)
#(function cons)

> #'cons ;é transformado durante a leitura para (function cons), que avalia pa
ra
#(function cons)

;
;Forma lambda: (lambda (<parametros>) <corpo>)
> (setf soma-e-dobra (function (lambda (x y) (* 2 (+ x y)))))
...

> (soma-e-dobra 3 2)
```

```
Erro
;porque o símbolo soma-e-dobra não tem função associada (tem apenas um valor q
ue
;é uma função). Para aplicar uma função a argumentos utiliza-se funcall ou app
ly:
> (funcall soma-e-dobra 3 2)
10

> (apply soma-e-dobra '(3 2))
10

;Ex: Função meu-find-if procura um elemento da lista que satisfaz um predicado
> (defun meu-find-if (pred l)
    (if (null l)
        ()
      (if (funcall pred (first l))
          T
        (meu-find-if pred (rest l)))))
MEU-FIND-IF

;Regras da avaliação das combinações são alteradas para tratar combinações:
;o primeiro elemento da combinação pode ser uma forma lambda.
> ((lambda (x y) (+ (* 2 x) y)) 5 3)
13

;Quando a forma lambda é aplicada aos parâmetros actuais, são criadas variávei
s locais
;a que são inicialmente atribuidos o valor dos parâmetros actuais corresponden
tes. Estas
;variáveis podem ser referidas apenas no corpo da forma lambda que as criou e
têm duração
;infinita depois de criadas.

;A forma let é açucar sintáctico para a utilização das lambda.
;
;((lambda (<p1>...<pn>) <corpo>) <v1>...<vn>) corresponde a
;(let ((<p1> <v1>)...(<pn> <vn>)) <corpo>)
;Exemplo anterior equivale a
> (let ((x 5)
        (y 3))
    (+ (* 2 x) y))
13

> (let ((x 1)
        (y (+ 2 x)))
    (+ x y))
???
;Porquê?

;Devia ser
> (let ((x 1))
    (let ((y (+ 2 x)))
      (+ x y)))
4

; ou, alternativamente, utilizar o let*
> (let* ((x 1)
         (y (+ 2 x)))
    (+ x y))
4


;Closures (ambientes léxicos)
```

```
;
;
;Ex: contador
> (let ((contador 0))
    (defun incrementa ()
      (incf contador)))
INCREMENTA
> (incrementa)
1
> (incrementa)
2

;Ex: contador
> (defun faz-contador ()
    (let ((contador 0))
      #'(lambda () (incf contador))))
FAZ-CONTADOR
> (setf incrementa1 (faz-contador))
> (funcall incrementa1)
1
> (funcall incrementa1)
2
> (funcall incrementa1)
3
> (setf incrementa2 (faz-contador))
> (funcall incrementa2)
1
> (funcall incrementa2)
2
> (funcall incrementa1)
4


;Ex: meu-cons
> (defun meu-cons (car cdr)
    #'(lambda (mensagem)
        (case mensagem
              (:car car)
              ((:cdr) cdr) ;podia ser :cdr em vez de (:cdr)
              (otherwise (error "Mensagem desconhecida: ~a" mensagem)))))
MEU-CONS

> (defun meu-car (meu-cons)
    (funcall meu-cons :car))
MEU-CAR

> (defun meu-cdr (meu-cons)
    (funcall meu-cons :cdr))
MEU-CDR

> (defun meu-print (meu-cons)
    (format t "(~a . ~a)" (meu-car meu-cons) (meu-cdr meu-cons)))
MEU-PRINT

> (meu-print (meu-cons 1 2))
(1 . 2)
NIL

> (defun meu-listp (obj)
    (if (atom obj)
        (null obj)
      (meu-listp (meu-cdr obj))))
MEU-LISTP
```

```
; Funciona se a lista não for circular. Sugira alterações para o caso da lista

; ser circular.

> (format t "~%") ;muda de linha. Ver descrição do Format no hyperspec.

NIL

; Para carregar um ficheiro fonte para o ambiente Lisp
; fazer (load <nome-ficheiro>). Por exemplo
> (load "projecto.lisp").
T

; Para compilar um ficheiro
> (compile-file "projecto")
#P"/Users/pedromatos/Desktop/projecto/projecto.fasl"
NIL
NIL

; Compilação não deve identificar warnings! Para carregar o último ficheiro
; bem compilado:
> (load "projecto")
T
```