Inteligência Artificial

INTRODUÇÃO

O QUE É A INTELIGÊNCIA ARTIFICIAL (IA) ?

- Muitas definições de Inteligência
- Aspectos a considerar:
  - Capacidade de resolver problemas
  - Capacidade de usar o conhecimento – raciocínio
  - Capacidade de Aprender
  - Etc.
- Medida de inteligência: QI (testes de inteligência)…
- Jogar xadrez denota inteligência?
  - Em 1980 aparentemente a resposta era consensual…
  - Hoje há máquinas que derrotam o campeão do Mundo!

DEFINIÇÃO DE TRABALHO

- Não há uma definição globalmente aceite.
- Elaine Rich: "Estudo de como fazer os computadores realizarem tarefas em que de momento as pessoas são melhores."

- A certa altura as pessoas eram melhores que as máquinas a fazer operações aritméticas!

NASCIMENTO DA IA

- A designação "Inteligência Artificial" foi inventada em 1956
  - Conferência em "Darthmouth College", EUA.
  - Por:
    - John McCarthy, … LISP (1959)
    - Marvin Minsky, … Percetrão (et. Seymour Papert) (1969)
    - Allen Newell, … Logic Theorist (1956)
    - Arthur Samuel, … Teoria de Jogos, Damas (1963)
    - Herbert Simon … Racionalidade Limitada (Premio Nobel)

LISP

Introdução

PARADIGMA FUNCIONAL

- Funções são "entidades de primeira classe" i.e. são tratadas como valores e podem ser argumentos de outra função, i.e. definir funções de "ordem superior" ou podem ser o resultado da invocação de uma função.
  - Isto é raro em linguagens imperativas ou orientadas para objectos, e na maioria dos casos não são suportadas funções de ordem superior.
  - Podem ser usadas em qualquer operação e podem ser criadas em run-time.
- Características da programação funcional pura:
  - Funções têm um parâmetro e um resultado.
    - Não há efeitos laterais; implica transparência referencial.
      - Imutabilidade dos dados: não se modificam estruturas de dados; se necessário criam-se novas estruturas que são cópias modificadas das anteriores.
    - Não há atribuição de valores a variáveis
    - Não há anomalias sintáticas ou semânticas
  - Principal estrutura de controlo: recursividade
    - tirando partido da possibilidade de se definir funções de ordem superior
- A programação funcional decorre do cálculo lambda.
- Mais informação em: https://thesocietea.org/2016/12/core-functional-programming-concepts/

CÁLCULO LAMBDA

- Sistema formal baseado em lógica matemática usado para exprimir computação com base na abstração de função e sua aplicação usando ligações de variáveis e substituições.

- λ-calculus
  - Representa um modelo de computação universal (*Turing-complete*)
- $\lambda x.x^2+1$ é uma abstração lambda para a função: $f(x)=x^2+1$
- Aplicação: f(3)=10 … $(\lambda x.x^2+1)\ 3 \rightarrow 10$
- No cálculo lambda as funções podem ser usadas como valores de input de outras funções. Exemplo:

  $((\lambda x.x^2+1)\ ((\lambda x.x-4)\ 7)) \rightarrow 10$
- As funções lambda podem ser etiquetadas (com um nome):

  f := $(\lambda x.x^2+1)$ g:= $(\lambda x.x-4)$

  E agora pode invocar-se (f (g 7)) ➔ 10

Alonzo Church

CONCEITOS BÁSICOS

- É conveniente rever os seguintes conceitos:
  - Compilação vs. Interpretação
    - Características
    - Vantagens e desvantagens
  - Gestão de memória
    - Heap vs. Stack
    - Caracteristicas
    - Vantagens e desvantagens
    - Métodos
  - Tipos de Dados
    - Semântica
    - Tipos Estáticos vs. Dinâmicos

COMPILAÇÃO VS. INTERPRETAÇÃO

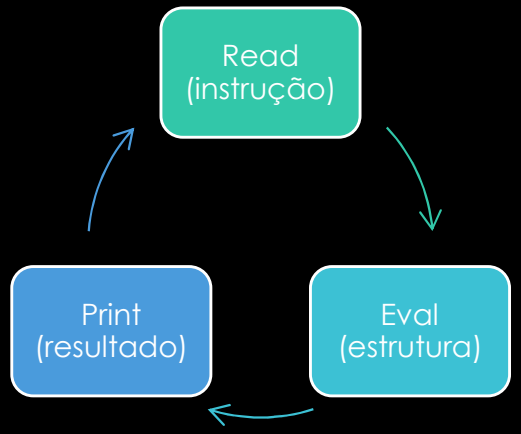
- **Compilação**: conversão para linguagem máquina antes da execução
  - Vantagens:
    - Velocidade, mas dependente da plataforma
    - Permite a deteção de erros na compilação,
    - Debug lento: Teste de novas versões obriga a recompilar o programa
- **Interpretação**: REPL
  - Vantagens:
    - Independência da plataforma
    - Reflexão (capacidade para o programa se alterar a si próprio: Assembler, Lisp, etc.)
      - Útil para testes: criação e instanciação de entidades durante a execução.
      - Essencial em metaprogramação: tratar os programas como dados.
        - https://en.wikipedia.org/wiki/Metaprogramming
    - Tipos de dados dinâmicos
      - Late binding / Duck Typing: tipo definido apenas em tempo de execução (não de compilação)
      - https://en.wikipedia.org/wiki/Type_system#Dynamic_type-checking_and_runtime_type_information
    - Tamanho de programa menor
- Para além disto:
  - As tecnologias de compilação just-in-time permitem que o gap entre linguagens compiladas e interpretadas esteja a estreitar.
  - As linguagens "bytecode" permitem compilar para uma máquina virtual que depois corre o programa de forma eficiente em diferentes plataformas, com vantagens de ambos os paradigmas.

CARACTERISTICAS DE UMA LINGUAGEM INTERPRETADA REFLEXIVA

- Interpretada: baseada num REPL
  - Compilada por partes
  - Permite testes incrementais rápidos, tirando partido do interpretador.

Ciclo Read-Eval-Print

Read-Eval-Print Loop(REPL)



- Reflexiva: trata dados como se fossem programas
  - Acesso direto ao avaliador
  - Permite construir código em tempo de execução e portanto criar estruturas de dados que são avaliadas.

GESTÃO DE MEMÓRIA: STACK VS. HEAP

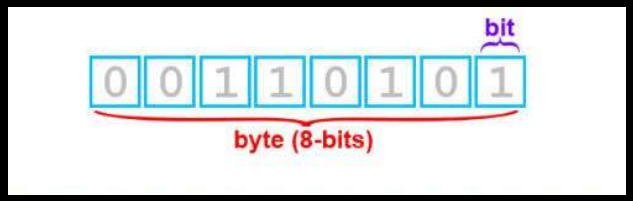
- **Stack**:
  - Segue uma disciplina LIFO, criada em RAM para cada thread, onde são guardadas as instâncias (frames) de invocação de função incluindo o código e variáveis.
    - Serve para variáveis locais, de tamanho fixo e vida limitada.
    - Rápido, pode usar a cache do CPU, mas limitado (sujeito a overflow)
      - *Possível problema com recursividade*
- **Heap**:
  - Também existe em RAM, mas permite alocação dinâmica de memória, sem intervenção do CPU.
    - Serve para variáveis globais, de tamanho variável e vida indefinida.
    - Lento, mas a limitação de memória é a memória total da máquina
    - Interage-se com o heap através de pointers
    - Gestão pode ser baseada em garbage collection (como em Java ou LISP) ou manual (como em C ou C++).
      - *Caso seja manual pode originar memory leaks.*

TIPOS DE DADOS

- O sistema de tipos de dados de uma linguagem define a semântica e as regras de comportamento do programa.

inteiro: 49 ?

carater: 1 ?



44	,	76	L	108	l
45	-	77	M	109	m
46	.	78	N	110	n
47	/	79	O	111	o
48	0	80	P	112	p
49	1	81	Q	113	q
50	2	82	R	114	r
51	3	83	S	115	s
52	4	84	T	116	t

- Vantagens:
  - Abstração e Modularidade (interoperabilidade)
  - Documentação (clarifica a intenção do programador)
  - Permite maximizar, na compilação: otimização e segurança.
- Verificação de dados:
  - Estática (no texto do código fonte)
    - Pode detetar erros durante a compilação
  - Dinâmica (durante a execução do programa)
    - Pode detetar erros durante a execução (e.g. CommonLisp)

VARIÁVEIS

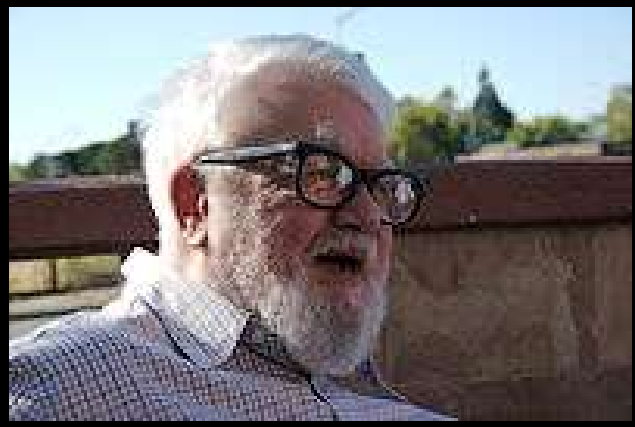
- Características principais de uma variável:
  - Tipo
  - Tempo de vida de uma variável (lifetime)
    - Desde a criação até ao desaparecimento: vide análise de stack vs. heap.
  - Âmbito de uma variável (scope)
    - Léxico – definido pela análise do texto do código fonte: uma variável pode ser referenciada em blocos internos àquele em que foi definida.
      - Este conceito está associado ao de ***closure***
    - Dinâmico – definido pela forma de encadeamento da invocação de funções ao longo da execução do programa.

CARACTERÍSTICAS PRINCIPAIS DO LISP

- Linguagem
  - Funcional
  - Interpretada
  - Reflexiva
- Tipos de dados dinâmicos (late binding)
- Âmbito das variáveis definido de forma léxica
- Estruturas de dados dinâmicas pervasivas, tirando partido extensivo do *heap* mas em que os *pointers* são tratados de forma implícita (transparente) através do tipo de dados: "Lista".

GÉNESE

- LISP = LISt Processing 1958
- Inteligência Artificial



(John McCarthy)

Timeline of Lisp dialects (edit)

	1955	1960	1965	1970	1975	1980	1985	1990	1995	2000	2005	2010	2015
Lisp 1.5	Lisp 1.5												
Maclisp			Maclisp										
Interlisp				Interlisp									
ZetaLisp					Lisp Machine Lisp								
Scheme					Scheme								
NIL					NIL								
Common Lisp						Common Lisp							
T							T						
Emacs Lisp							Emacs Lisp						
AutoLISP							AutoLISP						
ISLISP							ISLISP						
EuLisp							EuLisp						
Racket									Racket				
Arc										Arc			
Clojure											Clojure		
LFE											LFE		
Hy												Hy	

Dialectos principais:

Stanford LISP, MacLisp, InterLisp, Franz Lisp, Zeta Lisp, Common Lisp (1984), **ANSI Common Lisp** (1999), …

Ambientes de desenvolvimento: Lispworks, ...

LISP: UMA LINGUAGEM FUNCIONAL

- O LISP tem como elemento central **a Função**.
  - Um programa em LISP é uma função: devolve um valor em vez de produzir efeitos laterais, tais como alteração de objetos ou atribuição de valores a variáveis.
- Estilo de Programação: "bottom-up" e "top-down"
  - Dividir para conquistar: "top-down"
    - Vantagem: reduz a complexidade
  - Aumentar o vocabulário da linguagem: "bottom-up"
    - Vantagens: torna os programas mais fáceis de ler, promove a reutilização de código e permite descobrir padrões úteis para o domínio de aplicação.
- A linguagem e o programa evoluem em conjunto.
- Importante criar abstrações ajustadas ao domínio de aplicação.

LISP "PURO"

- O LISP "puro" é um núcleo central da linguagem baseado exclusivamente no cálculo lambda, a que mais tarde os vários dialectos adicionaram muitas outras funcionalidades acessórias, algumas das quais desvirtuantes do paradigma funcional.
- Características principais:
  - Variáveis podem mudar de tipo durante a execução
  - Ausência de operação de atribuição
  - Ausência de sequenciação
  - Ausência de estruturas de controlo iterativas (ciclos)
  - Ausência do conceito de "pointer" para manipulação de estruturas de dados dinâmicas
  - Ausência da necessidade de gerir a memória "heap".
  - Interpretado (compilado por partes)
  - Programas e dados têm a mesma estrutura
  - Acesso direto ao avaliador o que significa que é possível fazer programas que fazem programas

ASPETOS BÁSICOS DO AVALIADOR: PROGRAMAS VS. DADOS

- Invocação de funções
  - Usa-se listas com a seguinte sintaxe:
    notação prefixa e entre parêntesis
    (+ 2 3) => 5
- Plica
  - serve para evitar avaliar a expressão à direita, tratando-a como dados e não como programa.
    - ‘a => a
    - ‘(a b) => (a b)
- Non-case-sensitive
  - Pode escrever-se os nomes das funções e variáveis sem diferenciar maiúsculas de minúsculas.

TIPOS DE DADOS ELEMENTARES

- Átomos
  - Simbolos
  - Números
    - Inteiros
      - Fixnums
      - Bignums
    - Reais
    - Etc.
  - Booleanos
  - Caracteres
  - Strings
  - Outros
- Listas
  - Estruturas de dados dinâmicas, usadas para dados e programas

LITERAIS DE: NÚMEROS, BOOLEANOS, CARATERES E STRINGS

Exemplos de literais

Inteiros: -1 1 2
Fixnum: [most-negative-fixnum, most-positive-fixnum]
Bignum: 64646765756756756752343

Reais: 1.2 5.3 4.1e7

Booleanos: T NIL

Caracteres: #\a #\b

Strings: “sou uma string”

SÍMBOLOS

- Permitem representar variáveis e funções
- Estrutura de um símbolo (slots)
    - Nome, função, valor, lista de propriedades, pacote

factorial
Name: "factorial"
Function: (lambda(...)…
Value: nil
PList: nil
Package: default

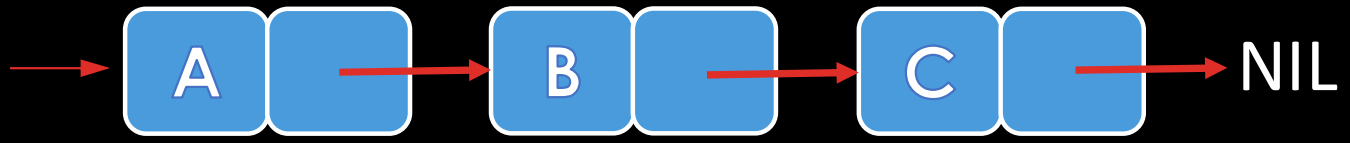
- A tabela de símbolos

Factorial	**A**	**B**	**Car**	**Cdr**	**…**

LISTAS

- Listas: Estruturas de dados dinâmicas
  - Baseada em células *cons*
    - https://en.wikipedia.org/wiki/Cons

    - Exercicio1 : representar graficamente a lista (a b c)



    - Exercicio 2: representar graficamente a lista (a (b c) d)

TIPO ABSTRATO LISTA

- Construtor
  - cons ex.: (cons ‘a ‘(b)) => (a b)
- Selectores:
  - car ex.: (car ‘(a b)) => a
  - cdr ex.: (cdr ‘(a b)) => (b)
- Exercicio: representar as estruturas de dados dos parâmetros e resultados
- Há seletores complexos que se aplicam a listas. Exemplo:
  - (cadr ‘(1 b c d)) ➔ b
  - (cddr ‘(1 b c d)) ➔ (c d)
  - (caddr ‘(1 b c d)) ➔ c
  - (cdddr ‘(1 b c d)) ➔ (d)

MACROS

- Formas especiais do LISP que definem extensões da estrutura sintática da linguagem.
- Ao contrário de funções, não implicam invocação e utilização do *stack*. São meras substituições léxicas.
- Exemplo:
  - (first '(a b c d)) ⇔ (car '(a b c d)) ➔ a
  - (rest '(a b c d)) ⇔ (cdr '(a b c d)) ➔ (b c d)
  - (second '(a b c d)) ⇔ (cadr '(a b c d)) ➔ b
  - (third '(a b c d)) ⇔ (caddr '(a b c d)) ➔ c
  - …

NOTAÇÃO BNF

BNF = Backus-Naur Form

- Simbolos terminais
  - Pertencem à linguagem que está a ser descrita e permitem construir expressões.
- Simbolos não terminais
  - Pertencem à meta-linguagem e são instanciados quando se constrói uma expressão.
- Regras de Produção
  - Definem a estrutura dos símbolos não terminais em termos de outros símbolos não terminais e de símbolos terminais.
- Simbolos da notação BNF

|	"alternativa"	a | A
<>	"delimitadores de símbolos não terminais"	
::=	"é descrito por:"	<Var>::= a | A
*	"zero ou mais ocorrências"	{prmt}*
+	"uma ou mais ocorrências"	{prmt}+

EXEMPLO: DEFINIÇÃO SINTÁTICA DO TIPO **LISTA** USANDO BNF

<Lista> ::= ({<expressão simbólica>}*) | nil

< expressão simbólica> ::= <Lista> | <Átomo>

<Átomo> ::= <número> |
<simbolo> |
<booleano> |
<carater> |
<string> |
<outro>

TIPO ABSTRATO DE DADOS

- Modelo matemático
- Tipo de dados ≠ de Estrutura de dados

- Especificação de um tipo de dados através de
  - Identificação dos seus possíveis valores e da
  - Identificação e definição das **operações** que podem ser executadas sobre esses dados.

- Um tipo abstrato de dados é independente da implementação.

TIPO ABSTRATO LISTA: ALGUMAS FUNÇÕES DE MANIPULAÇÃO

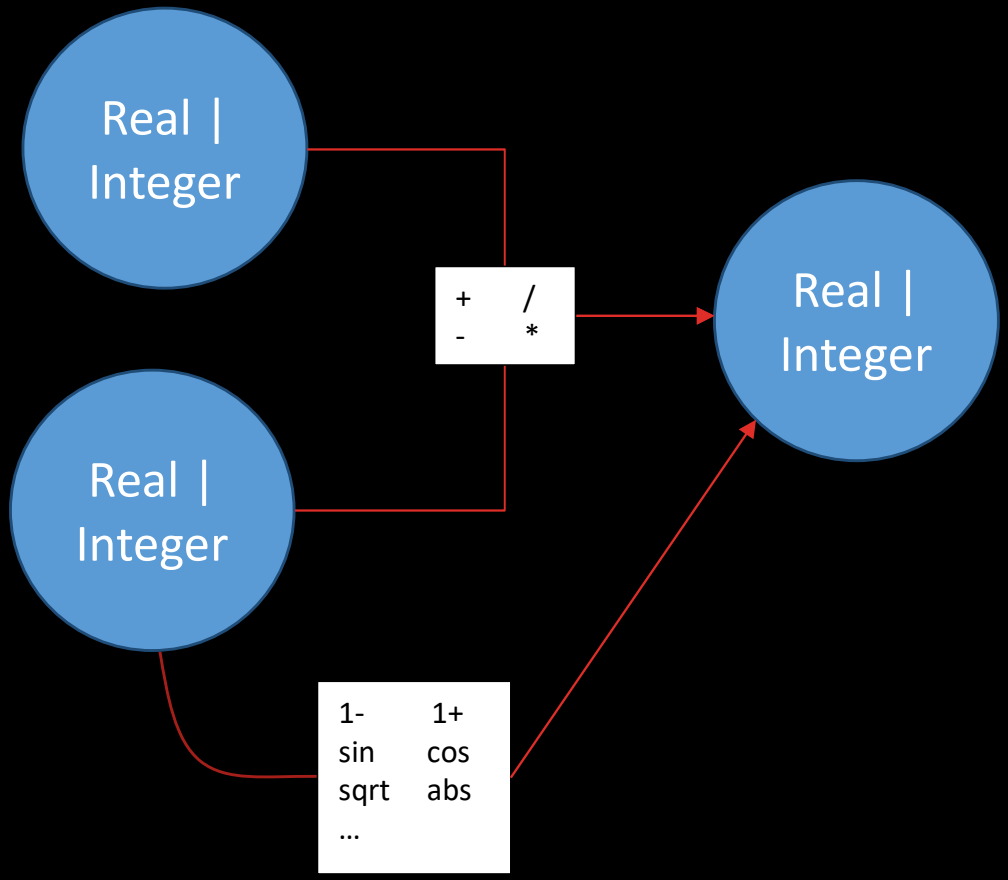
list	ex.: (list ‘a ‘b)	=> (a b)
append	ex.: (append ‘(a b) ‘(c d))	=> (a b c d)
length	ex.: (length ‘(a b))	=> 2
nth	ex.: (nth 1 ‘(a b c d))	=> b
reverse	ex.: (reverse ‘(a b c d))	=> (d c b a)
concatenate	ex.: (concatenate ‘list ‘(a) ‘(b c))	=> (a b c)

Etc…

- Exercicio:
  - Escrever a sintaxe da invocação da função **append** em BNF, sabendo que pode ter qualquer número de args do tipo lista.
  - identificar dez funções de manipulação de listas existentes em LISP e escrever a respectiva sintaxe em BNF

TIPOS NUMÉRICOS FUNÇÕES ARITMÉTICAS

(+
(-
(*
(/
(mod
(sqrt
(abs
(1+
(1-
(sin
(cos
(exp
(log
(random



OPERAÇÕES COM NÚMEROS

- (/ 3 5) ➔ 3/5
- (1+ (/ 3 5)) ➔ 8/5

- (/ 3.0 5) ➔ 0.6
- (div 3 5) ➔ Error: Undefined operator DIV

- (mod 5 4) ➔ 1
- (mod 5.1 4) ➔ 1.099999
- (% 4 5) ➔ Error: Undefined operator %

- (sqrt 9) ➔ 3.0

- (sqr 3) ➔ Error: Undefined operator SQR

ALGUMAS FUNÇÕES INTERESSANTES: LOG, EXP E RANDOM

(exp <n>)

potência <n> do número de Euler (e)

Exemplo: (exp 1) = 2.71828183

(log <n>)

Logaritmo neperiano (ou natural): base 2.71828183

$e^x = n$ (log 2.71828183) = ? (log 1) = ?

(log <n> <base>) *; a base é um argumento opcional.*

Logaritmo de base diferente da de Euler

Exemplo: (log 8 2) = ?

(random <*number*>)

random aceita um número n e devolve um número aleatório do mesmo tipo (inteiro ou real) entre 0 (inclusive) e n (exclusive).

NÚMEROS MUITO GRANDES

- Tipo inteiro (numérico / lista): BIGNUM

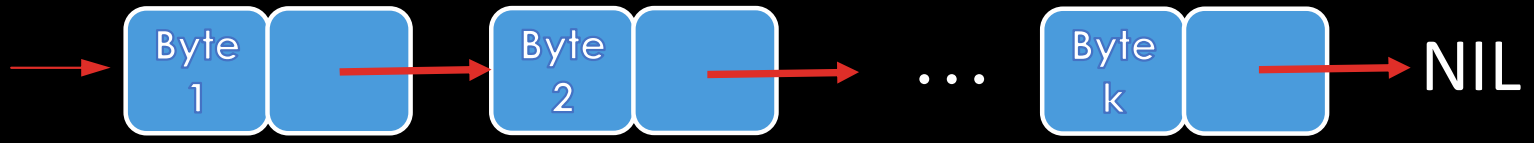
  (factorial 3) = 6

  (factorial 6) = 720

  (factorial 1000) = ??

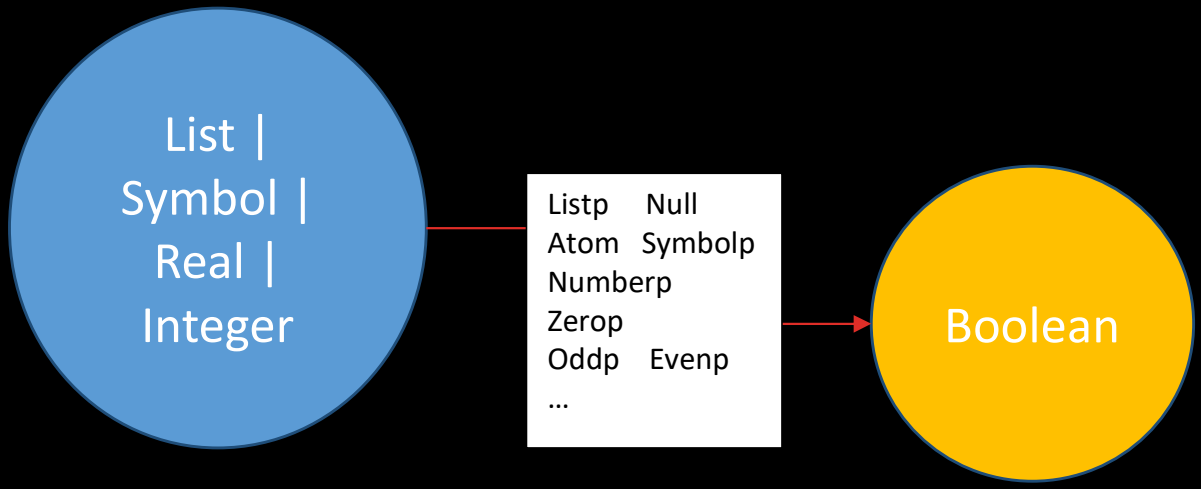
O LISP dá todos os algarismos!

Como? … usa listas:



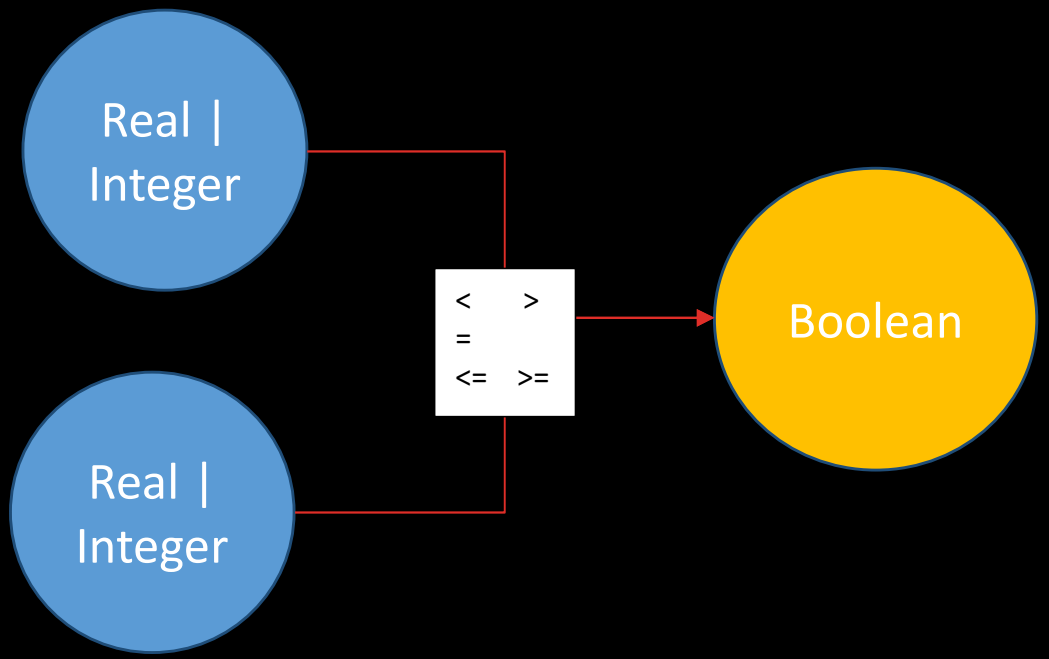
TIPO BOOLEAN / PREDICADOS

(null
(atom
(listp
(symbolp
(numberp
(zerop
(oddp
(evenp
(floatp
(integerp
(bignump



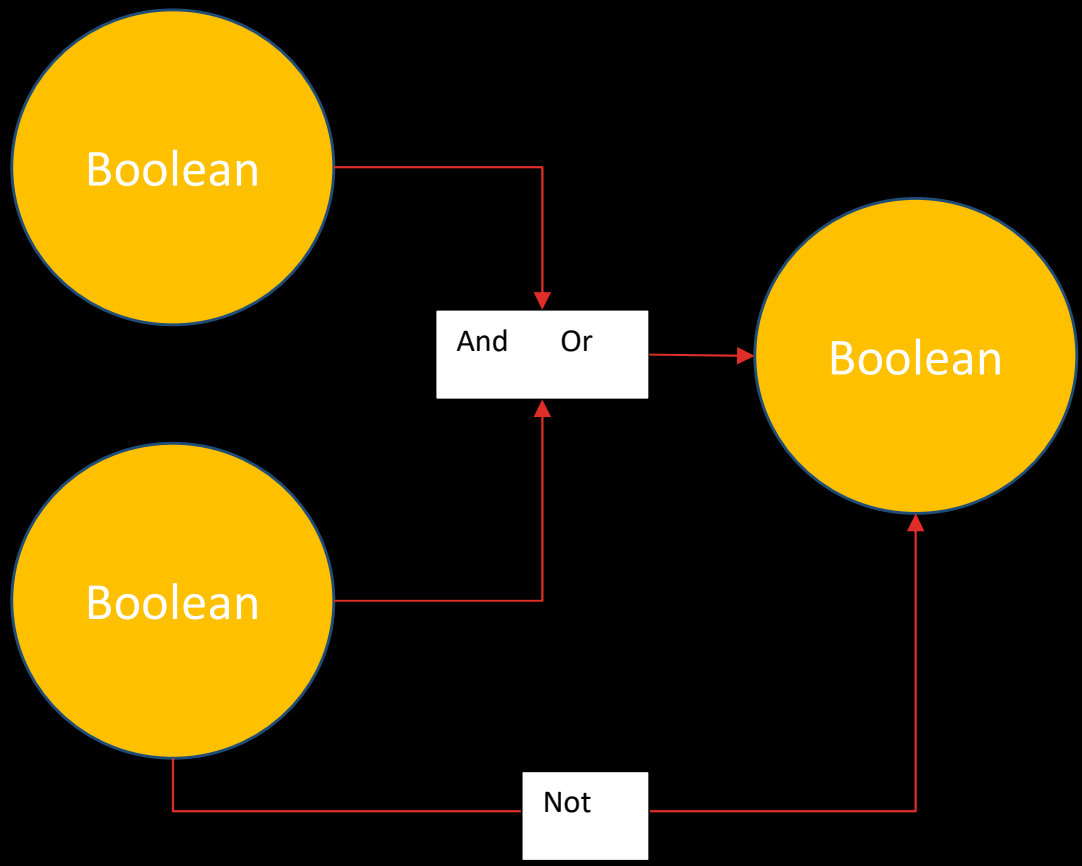
OPERADORES RELACIONAIS

(>
(<
(=
(>=
(<=



TIPO BOOLEAN OPERADORES LÓGICOS

- and
- or
- not



DETALHES DOS OPERADORES BOOLEANOS DE IGUALDADE

A diferença entre estas funções está na forma de representação/tipo de dados; ref vs. valor.

(equal
(eq
(eql
(=

(equal '(a b) '(a b)) ➔ T

(eq '(a b) '(a b)) ➔ NIL
(eql '(a b) '(a b)) ➔ NIL

(= 987654321 987654321) ➔ T
(eql 987654321 987654321) ➔ T
(equal 987654321 987654321) ➔ T

(eq 987654321 987654321) ➔ NIL

(eq #\a #\a) ➔ T
(eql #\a #\a) ➔ T
(equal #\a #\a) ➔ T

```
> (bignump 987654321)
T
```

DEFINIÇÃO DE FUNÇÕES E ESTRUTURAS DE CONTROLO

DEFINIÇÃO DE FUNÇÕES

```
(defun <nome da função> (<args>)
   "<descrição>"
   <corpo da função>)
```

```
;;; Exemplo:
(defun media (nota1 nota2 nota3)
   "faz a media aritmética de 3 notas"
    (/ (+ nota1 nota2 nota3) 3)) ; devolve um número
```

EXEMPLOS

- Função para calcular o quadrado de um número

```
(defun quadrado (numero)
   "devolve o quadrado de um número"
   (* numero numero))
```

- Função para calcular a área de um círculo

```
(defun areaCirculo (raio)
   "devolve a área de um circulo, dado o raio"
   (* 3.14159265 (quadrado raio))) ;o pi não está predefinido
```

FUNÇÕES E SÍMBOLOS DE FUNÇÕES

- Funções com nome.
  - Ex.: (defun soma3 (x) (+ x 3))
    - Símbolo "soma3" tem no slot ***function*** esta definição.

soma3
Name: "soma3"
Function: (lambda(...)
Value: nil
PList: nil
Package: default

- Funções sem nome.
  - Ex.: (lambda (x) (+ 3 x))
    - Não há nenhum símbolo para memorizar esta definição.
- O que é executável é a definição (lambda)
- Exemplo de utilização das funções lambda em LISP:
  - (**(lambda(x) (+ 3 x))** 7) => 10
  - (* 2 (**(lambda(x) (+ 3 x))** 7) => 20

DEPURAÇÃO (*DEBUG*)

(trace {<função>}*) – indica os valores dos argumentos e do resultado de cada vez que uma função é invocada.

(dribble <ficheiro>) – envia o output para o ecran e para um ficheiro, simultaneamente

(describe <função>) – dá os parâmetros e a documentação se existir

ESTRUTURAS DE CONTROLO DO LISP PURO

- Sequenciação:
  - não tem.
- Selecção:
  - Cond
- Repetição:
  - usa a recursividade

SELEÇÃO

- Sintaxe:

(cond <cláusula 1>
<cláusula 2>
…
<cláusula n>)

<cláusula> ::= (<condição> <resultado>)

COND: EXEMPLOS

```
(cond ((oddp x) 'impar)
      ((evenp x) 'par))

(cond ((oddp x) 'impar)
      (t 'par))

(cond ((= x 0) 1)
      (t (* x (fact (1- x)))))
```

MACROS

- if

  (if <condição> <se-verdade> <se-falso>)

  exemplo: (if (zerop 3) “nulo” “ok”) ➔ “ok”

- ecase

  (ecase <chave> (<item> <val>) …. (<item> <val>))

  exemplo: (ecase b (a 1) (b 2) (c 3)) ➔ 2

EXEMPLOS

- Definir a função **max** para devolver o maior de 2 números:

```
(defun max (x y)
"devolve o número maior; se os números forem iguais devolve o segundo"
   (cond ((> x y)   x)
         ( t        y)))
```

- Definir a função **max4** para devolver o maior de 4 números:

```
(defun max4 (x y w z)
   (max x (max y (max w z))))
```

RECURSIVIDADE

- Uma função recursiva tem sempre uma estrutura com 2 condições ou mais:
  - Condição de paragem
  - Condição recursiva (1 ou mais)

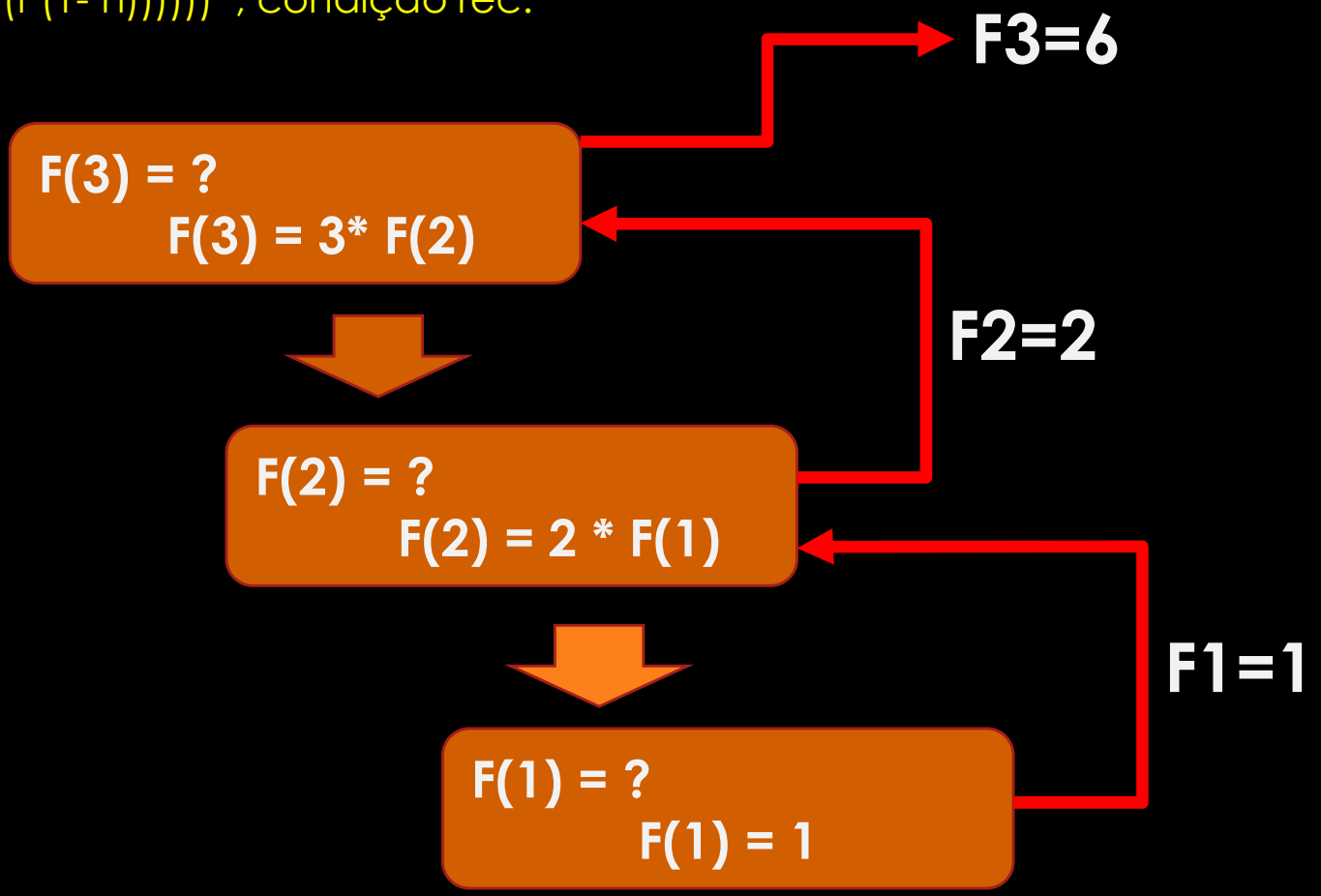
Exemplo:

```
(defun f (n)
   (cond ((<= n 1)   1)          ; condição de paragem
         (T  (* n (f (1- n)))))) ; condição recursiva
```

O que faz esta função?

UTILIZAÇÃO DO STACK

```
(defun f(n)
    (cond ((<= n 1) 1)            ; condição de paragem
          (t   (* n (f (1- n))))))  ; condição rec.
```



EXERCICIOS

- Defina uma função **pot** para calcular: pot(x,n)=x^n
- Defina uma função **fib** para calcular a sequência de Fibonacci:

$$F(n) = \begin{cases} 0, & \text{se } n = 0; \\ 1, & \text{se } n = 1; \\ F(n-1) + F(n-2) & \text{outros casos.} \end{cases}$$

- Defina a função **A** de Ackermann:

$$A(m,n) = \begin{cases} n+1 & \text{if } m = 0 \\ A(m-1,1) & \text{if } m > 0 \text{ and } n = 0 \\ A(m-1, A(m,n-1)) & \text{if } m > 0 \text{ and } n > 0. \end{cases}$$

USAR COM EXTREMO CUIDADO E APENAS EM CASOS EXCECIONAIS

- Sequenciação
  (progn <expr$_1$> <expr$_2$> … <expr$_n$>)

- Iteração
  (do *({<var> | (<var> [<init-form> [<step-form>]])}*)*
  (<end-test-form> <result-form>)*
  <declaration> {<tag> | <statement>}**
  <body>)

  dotimes

  dolist

EXEMPLOS

```
(do ((temp-one 1 (1+ temp-one))
     (temp-two 0 (1- temp-two)))
    ((> (- temp-one temp-two) 5) temp-one)) ➔ 4
```

Variante:

(dotimes (<counter> <limit> [<result>]) <body>)

Ex.: (dotimes (i 10) (progn (princ i) (terpri)))

(dolist (<var> <list> [<result>]) <body>)

Ex.: (dolist (x '(a b c)) (print x))

VARIÁVEIS E CONSTANTES GLOBAIS

- A usar com extremo cuidado e só em casos muito especiais.

  (defparameter <var> <valor>)

- Estas variáveis estão fora de qualquer ambiente léxico por isso são visíveis em todo o programa, correndo o risco de provocar efeitos laterais imprevisíveis.
- Geralmente usa-se a convenção de as nomear com asteriscos à esquerda e à direita.
  - Exemplo: (defparameter *pi* 3.14159265)

LIGAÇÃO DE VALORES A VARIÁVEIS EM AMBIENTES LÉXICOS

- LET – Avaliação das expressões é feita em paralelo.
  (let ({(<var> <expr>)}*)
    <corpo do let>)

- LET* - Avaliação sequencial das exprs.
  (let* ({(<var> <expr>)}*)
    <corpo do let>)

EXEMPLO LET

- Definir uma função para calcular o perímetro e a área de um círculo de raio dado pelo utilizador. Devolve uma lista com estes dois valores.

```
(defun p-a-circulo ()
    (let ((raio (read)))
      (list (* 2 3.14 raio) (* 3.14 raio raio))))
```

EXEMPLO LET*

- Definir uma função RA10 que devolve a raiz quadrada da amplitude de uma lista de números se a amplitude for maior que 10, ou zero caso contrário.
- A amplitude de valores de uma lista de números é a diferença entre máximo e mínimo: (max – min)

```
(defun RA10 (lista)
      (let*  ((maior (max lista))
              (menor (min lista))
              (amplitude (- maior menor)))
          (cond ((< amplitude 10) 0)
                (t (sqrt amplitude)))))
```

LET E LAMBDA

(let ((a_1 b_1) (a_2 b_2) ... (a_n b_n))
(corpo a_1 a_2 ... a_n))

É similar a:

((lambda (a_1 a_2 ... a_n)
(corpo a_1 a_2 ... a_n))
b_1 b_2 ... b_n)

LET* E LAMBDA

(let* ((a_1 b_1) (a_2 b_2) ... (a_n b_n))
  (corpo a_1 a_2 ... a_n))

É similar a:

((lambda (a_1)
  ((lambda (a_2)
    ...
    ((lambda (a_n)
      (corpo a_1 a_2 ... a_n))
     b_n))
   b_2))
 b_1)

LIGAÇÃO DE FUNÇÕES A VARIÁVEIS EM AMBIENTES LÉXICOS

- Enquanto as funções são definidas globalmente com defun, também é possível criar funções locais com flet e labels.

```
(flet (function-definition*)
  body-form*)

(labels (function-definition*) ;;; permite definições recursivas
  body-form*)
```

EXEMPLOS DE FLET E LABELS

```
(defun teste (x)
   (flet ( (f1 (n) (+ n n)) )
      (flet ( (f2 (n) (+ 2 (f1 n))) )   ;;; não existe flet*
         (f2 x)))
=>  TESTE
(teste 5) ➔12

(defun recursive-times (k n)
   (labels ( (temp (n)
                  (if (zerop n) 0 (+ k (temp (1- n))))) )
      (temp n)))
=>  RECURSIVE-TIMES
(recursive-times 2 3) ➔  6
```

CLOSURES

- Uma *closure* léxica é uma função que, quando invocada com argumentos executa o corpo de uma expressão lambda no ambiente léxico que foi capturado no momento da criação da *closure* aumentado com as ligações dos parâmetros da função aos respetivos argumentos.

```
(let ((e 1))
  (defun clos-1 (x) (+ x e)))

> (clos-1 3) ➔ 4
> e ➔ unbound variable
```

O que é *scope* (âmbito)?
- léxico
- dinâmico

e tempo de vida?

ALGUMAS FUNÇÕES DESTRUTIVAS USAR APENAS EM AMBIENTES LÉXICOS

setf : atribuição. Exemplo: (setf x 3)

incf : incremento destrutivo: (incf x)

```
> (let ((counter 0))
    (defun counter-next ()
      (incf counter))
    (defun counter-reset ()
      (setf counter 0)))

> (counter-next) ➔ 1
> (counter-next) ➔ 2
> (counter-next) ➔ 3
> (counter-reset) ➔ 0
> (counter-next) ➔ 1
```

OUTRO EXEMPLO

```
> (let ((password nil)
        (secret nil))
   (defun set-password (new-passwd)
    (if password
     "a password já está definida"
     (setf password new-passwd)))
   (defun change-password (old-passwd new-passwd)
    (if (eq old-passwd password)
     (setf password new-passwd)
     "Password não alterada"))
   (defun set-secret (passwd new-secret)
    (if (eq passwd password)
     (setf secret new-secret)
     "Password errada"))
   (defun get-secret (passwd)
    (if (eq passwd password)
     secret
     "Querias..."))))
```

> (get-secret 'sesame) ➔ "Querias..."
> (set-password 'valentine)
> (set-secret 'sesame 'my-secret) ➔ "Password errada"
> (set-secret 'valentine 'my-secret) ➔ MY-SECRET
> (get-secret 'fubar) ➔ "Querias..."
> (get-secret 'valentine) ➔ MY-SECRET
> (change-password 'fubar 'new-password)
➔"Password não alterada"
> (change-password 'valentine 'new-password)
➔ NEW-PASSWORD
> (get-secret 'valentine) ➔ "Querias..."
> Password ➔ Error: unbound variable
> Secret ➔ Error: unbound variable
> (setf password 'cheat)
> (get-secret 'cheat) ➔ "Querias..."

EXEMPLO 2

O exemplo anterior serve apenas para guardar 1 segredo pois de cada vez que se avalia o LET redefine-se todas as funções (*closures*).

Possível Solução:

```
> (defun make-secret-keeper ()
    (let ((password nil)
          (secret nil))
      #'(lambda (operation &rest arguments)
          (ecase operation
                  (set-password (let ((new-passwd (first arguments)))
                                  (if password "a password já está definida"
                                    (setf password new-passwd))))
                  (change-password  (let ((old-passwd (first arguments))
                                          (new-passwd (second arguments)))
                                      (if (eq old-passwd password) (setf password new-passwd)
                                        "Password não alterada")))
                  (set-secret (let  ((passwd (first arguments))
                                     (new-secret (second arguments)))
                                (if (eq passwd password) (setf secret new-secret)
                                  "Password errada")))
                  (get-secret  (let ((passwd (first arguments)))
                                 (if (eq passwd password) secret
                                   "Querias..."))))))
```

EXEMPLO 2 (CONT.)

> (defparameter secret-1 (make-secret-keeper))
> secret-1 ➔ #<LEXICAL-CLOSURE #x36AE056>
> (funcall secret-1 'set-password 'valentine) ➔ VALENTINE
> (funcall secret-1 'set-secret 'valentine 'deep-dark) ➔ DEEP-DARK

> (defparameter secret-2 (make-secret-keeper))
> (funcall secret-2 'set-password 'bloody) ➔ BLOODY
> (funcall secret-2 'set-secret 'bloody 'mysterious) ➔ MYSTERIOUS
> (funcall secret-2 'get-secret 'valentine) ➔ "Password errada"

> (funcall secret-1 'get-secret 'valentine) ➔ DEEP-DARK

EXERCICIOS

- Defina funções para
  - Rodar uma lista para a esquerda.
    - (roda-esquerda '(1 2 3 4)) => (2 3 4 1)
  - Calcular as raizes reais da equação de 2º grau.
    - (fresolvente 1 0 -1)=> (-1 1)
- Defina as seguintes funções recursivas:
  - Máximo: max(lista de números)=>valor máximo
  - Fibonacci: fib(n) => n-ésimo na sequência de Fib.
  - Binario: bin(n)=(1 0 … 0) que, dado um número inteiro positivo, constrói uma lista de zeros e uns, formando a representação binária do argumento.
  - Hexadecimal: hex(n)=(0 A 3 … F) que, dado um número inteiro positivo, constrói uma lista de símbolos entre 0 e F, formando a representação hexadecimal do argumento.

EXERCICIOS

- Escreva funções para resolver os seguintes problemas:
- Espelho(L): Recebe uma lista, que pode ter sublistas, e inverter a ordem de todos os elementos, inclusive dentro das sublistas
- Lista-ate-n(n): Recebe um número e devolve uma lista com todos os inteiros até n ordenada por ordem crescente.
- Insere(e p L): insere um elemento e na posição p de uma lista.
- Insere-ordenado(n L-ord): insere um número numa lista que já está ordenada por ordem crescente, de forma a que o resultado continue a ser uma lista ordenada, utilizando o menor número possível de operações (procura binária).
- Alisa(L): Recebe uma lista com sublistas e devolve uma lista com os elementos pela mesma ordem mas sem sublistas.

EXERCICIOS DE CONJUNTOS

Presume-se que um conjunto é representado por uma lista de elementos sem repetição.

- Conjunto(Lista): Recebe uma lista e devolve um conjunto (elimina os repetidos)
- Intersecao(C1 C2): Recebe dois conjuntos e devolve o conjunto interseção.
- Reuniao(C1 C2): Recebe dois conjuntos e devolve o conjunto reunião.
- Diferenca(C1 C2): Recebe dois conjuntos e devolve o conjunto diferença (C1 – C2).
- Adiciona(e C1): Devolve o conjunto que resulta de adicionar e ao conjunto C1.
- Subtrai(e C1): Devolve o conjunto que resulta de retirar e ao conjunto C1.

FUNÇÕES LAMBDA, PARÂMETROS E META-FUNÇÕES

LAMBDA

- A função lambda é o elemento básico de programação em LISP.
- Decorre do cálculo lambda
- Exemplo:
  - Definição de função: (lambda (x y) (+ x y 1))
  - Invocação: ((lambda (x y) (+ x y 1)) 2 5) ➔ 8

SYNTAXE COMPLETA DOS PARÂMETROS

- (lambda ({*var*}*
  [&optional {*var* | (*var* [*initform* [*svar*]])}*]
  [&rest *var*]
  [&key {*var* | ({*var* | (*keyword var*)} [*initform* [*svar*]])}*
  [&allow-other-keys]]
  [&aux {*var* | (*var* [*initform*])}*)]
  [[{*declaration*}* | *documentation-string***]]**
  {*form*}*)

  As var e svar têm de ser símbolos

  Quando se invoca a função as keywords têm de começar pelo caracter :

  Uma initform pode ser qualquer expressão LISP.

EXEMPLOS

- &rest
  - (defun g (x &rest r) ...)
  - (g 1 2 3 4 5) ... x=1 r=(2 3 4 5)

- &optional
  - (defun f (a &optional (b 5))
    (+ a b))
  - (defun f (a &optional (b 5) &rest z)
    (cons a (cons b z)))

EXEMPLOS (CONT.)

- &key
  - (defun f (x &key y z w) ...)
  - (f 1 :w 3 :y 1) ... x=1 y=1 z=<unbound> w=3

  - (defun f (x &key y (z 0) w) ...)
  - (f 1 :w 3 :y 1) ... x=1 y=1 z=0 w=3

FUNCIONAIS (META-FUNÇÕES)

- Acesso direto ao avaliador do Lisp (mesmo que é usado no REPL)

(eval <form>)

exemplo: (defun f(n)
(eval (cons (nth n '(+ - / *)) '(2 3))))

(apply <função> <lista prmts>)

exemplo: (defun g(n)
(apply (nth n '(+ - / *)) '(2 3)))

(funcall <função> <prmts>)

exemplo: (defun h(n)
(funcall (nth n '(+ - / *)) 2 3))

FUNÇÕES DE CORRESPONDÊNCIA

mapcar *function list* &rest *more-lists*
maplist *function list* &rest *more-lists*

mapc *function list* &rest *more-lists* -- não acumulam o resultado
mapl *function list* &rest *more-lists*

mapcan *function list* &rest *more-lists* -- usam funções destrutivas
mapcon *function list* &rest *more-lists*

- Exemplos:

```
(mapcar '+ '(1 2 3) '(6 7 8))   => (7 9 11) ;; função de 2 argumentos
(mapcar #'abs '(3 -4 2 -5 -6)) => (3 4 2 5 6) ;; função de 1 argumento

(maplist #'(lambda (x) (cons 'foo x)) '(a b c d))
                              => ((foo a b c d) (foo b c d) (foo c d) (foo d))
```

EXEMPLO

- Dada uma lista de tripletes com a seguinte estrutura:

  <pessoa> ::= (<nome> <idade> <país>)

  <pessoas> ::= (<pessoa>*)

Defina uma função para listar os países das pessoas com mais de 50 anos.

```
(defun maiores50 (pessoas)
  (apply #'append ;; remove os nils de uma lista de listas
    (mapcar (lambda (pessoa)
              (cond ((> (second pessoa) 50) (cddr pessoa)))
                    (t nil) ))
            pessoas)))
```

;;; (maiores50 '((antonio 51 PT)(brad 34 UK) (charles 77 FR))) >> (PT FR)

DETALHES

- Apply #‘append …
  - Útil para remover nils de uma lista de listas.

(apply #‘append ‘((a) (b) nil (c) nil nil (d)))
>> (a b c d)

Pormenor:

((lambda (pessoa) (cond ((> (second lista) 50) (third p))) (t nil)) ‘(c 77 PT))
>> PT

((lambda (pessoa) (cond ((> (second lista) 50) (cddr p)) (t nil))) ‘(c 77 PT))
>> (PT)

EXERCICIOS

- Qual o resultado de cada uma das invocações de funções abaixo?
  - (mapcar #'1+ '(1 2 3 4 5))
  - (mapcar #'list '(1 2 3 4 5))
  - (mapcar #'list '(1 2 3 4 5) '(a b c))
- Escreva uma função para somar 2 unidades a todos os elementos de uma lista de números dada.
- Escreva uma função para somar as raízes quadradas de uma lista de números.
  - Versão recursiva
  - Versão usando funções de ordem superior
- Escrever uma função para contar o número de números ímpares numa lista.

EXERCICIOS

- Defina funções para
  - 1) multiplicar duas matrizes bidimensionais NxM e MxK
    (defun multimap (m1 m2)
  - 2) transpor uma matriz quadrada
    (defun transposta (m)

O número de linhas e colunas de cada matriz é arbitrário.

Use como representação do tipo de dados matriz uma estrutura de dados na forma de lista de listas.

```
( ( … )
  ( … )
    … )
```

FUNÇÕES DE E/S

FUNÇÕES DE E/S

- Básicas
  - Read
  - Read-line
  - Write-Line
  - Terpri
  - Format
  - …

- Estas funções têm um argumento opcional que é o <stream> ou <port>.
  - T = (*standard-io*) = ecran
  - NIL = string

LEITURA

- READ

  Argumentos:

  - (&OPTIONAL
    (STREAM *STANDARD-INPUT*)
    (EOF-ERROR-P T)
    EOF-VALUE
    RECURSIVE-P)
  - Lê o próximo valor no stream (cujo default é *standard-input*)

    - EOF-ERROR-P e EOF-VALUE especificam o que acontece se o programa tentar ler de um stream vazio:
      - Se EOF-ERROR-P tem o valor true então o Lisp gera uma excepção. Caso contrário retorna o EOF-VALUE (default nil).
    - RECURSIVE-P está reservada a funções invocadas pelo Lisp reader.

LEITURA (CONT.)

- READ-LINE

  Argumentos:

  - (&OPTIONAL

    (STREAM *STANDARD-INPUT*)

    (EOF-ERROR-P T)

    EOF-VALUE

    RECURSIVE-P)

  - Devolve a linha de texto lida do stream na forma de uma string, descartando o newline.

- Outras
  - READ-BYTE *stream &optional eof-error-p eof-value*
  - READ-CHAR *&optional stream eof-error-p eof-value recursive-p*

ESCRITA

- WRITE-LINE

  Argumentos: (STRING &OPTIONAL
  (STREAM *STANDARD-OUTPUT*)
  &KEY (START 0) END)

  Escreve uma String para o Stream dado, seguido de um newline.

- TERPRI

  Argumentos: (&OPTIONAL
  (STREAM *STANDARD-OUTPUT*))

  Escreve um newline no stream indicado.

ESCRITA

- PRINC

  Argument0s:

  (OBJECT &OPTIONAL (OUT-STREAM *STANDARD-OUTPUT*))

  Escreve uma representação esteticamente agradável do objeto no stream indicado, mas não necessariamente READable.

  Exemplo: (princ "abc") >> abc

- PRIN1

  Arguments:

  (OBJECT &OPTIONAL (OUT-STREAM *STANDARD-OUTPUT*))

  Escreve uma representação do objeto no stream indicado, que é de forma geral READable.

  Exemplo: (princ "abc") >> "abc"

ESCRITA

- FORMAT

  Argumentos:

  (STREAM CONTROL-STRING &REST FORMAT-ARGUMENTS)

  - Se STREAM= T, o resultado é escrito no *standard-output*,
  - Se é NIL, o resultado é devolvido como uma string.
  - Caso contrário, STREAM geralmente corresponde a um ficheiro.

  - A CONTROL-STRING contém a string a escrever, geralmente com diretivas embutidas, iniciadas com o carater "~".

FORMAT

DIRETIVAS:

~A or ~nA	Escreve um argumento como PRINC
~S or ~nS	Escreve um argumento como PRIN1
~D or ~nD	Escreve um argumento como um inteiro
~%	TERPRI
~&	FRESH-LINE

n é a largura do campo em que o valor é escrito

Exemplos:

- Números reais:
  - (format t "~$" pi)
    - 3.14
  - (format t "~5$" pi)
    - 3.14159

Números inteiros:
(format t "~d" 1000000)
1000000
(format t "~:d" 1000000)
1,000,000
(format t "~@d" 1000000)
+1000000

FORMAT (CONT.)

Mais exemplos:

- (format nil "The value is: ~a" 10)
  - "The value is: 10"
- (format nil "The value is: ~a" "foo")
  - "The value is: foo"
- (format nil "The value is: ~a" (list 1 2 3))
  - "The value is: (1 2 3)“

Format Condicional:

- (format nil "~[zero~;um~;dois~]" 0) ==> "zero"
- (format nil "~[zero~;um~;dois~]" 1) ==> "um"
- (format nil "~[zero~;um~;dois~]" 2) ==> "dois"

MAIS DIRECTIVAS

- ~% new line
- ~& fresh line
- ~| page break
- ~T tab stop
- ~< justification
- ~> terminate ~<
- ~C character
- ~(case conversion
- ~) terminate ~(
- ~D decimal integer
- ~B binary integer

- ~O octal integer
- ~X hexadecimal integer
- ~bR base-b integer
- ~R spell an integer
- ~P plural
- ~F floating point
- ~E scientific notation
- ~G ~F or ~E, depending upon magnitude
- ~$ monetary
- ~A legibly, without escapes
- ~S Readably, with escapes
- ~~ ~

EXERCICIOS

- Escreva uma função denominada **escreve_numero** que receba um número e escreva no écran "O numero é ...". Use a função format.
- Escreva uma função denominada **sequencia_numeros** que leia uma sequência de números positivos via teclado, por qualquer ordem, terminada por zero, e escreva no écran a sequência de números lida ordenada por ordem crescente, um número em cada linha.
- Escreva uma função chamada **escreve_lista** que escreva o conteúdo duma lista no écran.
  - (escreve_lista '(1 2 3))

    (1 2 3)
- Redefina a pergunta anterior em **escreve_lista1** de forma a poder escrever um elemento da lista por linha. Recomenda-se o uso da função mapc.
  - (escreve_lista1 '(1 2 3))

    1
    2
    3

FICHEIROS

- open *filename*
  &key
  :direction
  :element-type
  :if-exists
  :if-does-not-exist
  :external-format

- (with-open-file (<port> <path> {keys}*)
  <corpo>)

KEYWORDS DE OPEN

- Keyword Value Stream Direction ---------- :DIRECTION
  - :INPUT input (default)
  - :OUTPUT output
  - :IO input & output
  - :PROBE none
- Keyword Value Action if File Exists ---------- :IF-EXISTS
  - :ERROR signal an error
  - :NEW-VERSION next version (or error)
  - :RENAME rename existing, create new
  - :SUPERSEDE replace file upon CLOSE
  - :RENAME-AND-DELETE rename and delete existing, create new
  - :OVERWRITE reuse existing file (position at start)
  - :APPEND reuse existing file (position at end)
- Keyword Value Action if File Does Not Exist ---------- :IF-DOES-NOT-EXIST
  - :ERROR signal an error
  - :CREATE create the file

KEYWORDS DE OPEN (CONT.)

- Keyword Value Element Type ---------- :ELEMENT-TYPE
  - :DEFAULT character (default)
  - 'CHARACTER character
  - 'SIGNED-BYTE signed byte
  - 'UNSIGNED-BYTE unsigned byte
  - *other* implementation-dependent
- Keyword Value File Format ---------- :EXTERNAL-FORMAT
  - :DEFAULT default (default)
  - *other* implementation-dependent

PATHNAME

Em diferentes sistemas operativos as funções de acesso ao sistema de ficheiros podem diferir, sendo conveniente um nível intermédio de abstração, mediante o tipo de dados "pathname":

Construtor:

make-pathname &key :host :device :directory :name :type :version :defaults :case

Seletores:

pathname-host pathname
pathname-device pathname
pathname-directory pathname
pathname-name pathname
pathname-type pathname
pathname-version pathname

Acesso especial:

user-homedir-pathname &optional *host*

Teste

probe-file namestring

Conversão para string:

namestring *pathname*

EXEMPLO

```
(make-pathname
        :host "technodrome"
        :directory '(:absolute "usr" "krang")
        :name "shredder")
  => #P"technodrome:/usr/krang/shredder“
```

Nota: Em Windows, o acesso a “c:\...” é restrito. Preferível colocar os ficheiros de trabalho num diretório abaixo da raiz ou então noutro disco.

EXEMPLO

Escreva uma função que copia as linhas impares de um ficheiro ".txt" para um novo ficheiro com o mesmo nome e extensão ".odd".

```
(defun copia-impares (file1)
       (let ((path1 (make-pathname :host "e" :directory '(:absolute "") :name file1 :type "txt"))
             (path2 (make-pathname :host "e" :directory '(:absolute "") :name file1 :type "odd")))
          (with-open-file (f1 path1 :direction :input)
             (with-open-file (f2 path2 :direction :output :if-exists :supersede)
                (copia-linha f1 f2)))))


(defun copia-linha (f1 f2)
     (let ((linha (read-line f1 nil :fim)))
        (cond ((not (eq linha :fim))(write-line linha f2) (salta-linha f1 f2))
              (t (close f2)))))


(defun salta-linha (f1 f2)
     (let ((linha (read-line f1 nil :fim)))
        (cond ((not (eq linha :fim))(copia-linha f1 f2))
              (t (close f2)))))
```

EXERCICIOS

- Usando a função with-open-file defina uma função **escreve_lista_ficheiro** que receba uma lista e um nome completo (caminho + nome) de um ficheiro e que escreva o conteúdo dessa lista – com um elemento por linha - nesse ficheiro.
- Usando a função with-open-file, e a função read defina uma função **le_elementos_ficheiro** que receba um nome completo de um ficheiro e que leia todos os elementos existentes nesse ficheiro e os escreva no écran elemento a elemento, um em cada linha.
- Escreva um função chamada **conta_se** que conte todos os carateres de um ficheiro cujo código ASCII esteja entre um dado limite inferior e um dado limite superior e retorne uma lista de pares em que cada par é constituído pelo código ASCII do carater relevante e o respetivo número de ocorrências no ficheiro.
    - Pode usar as funções: (code-char <num>) e (char-code <char>)
- Escreva uma função denominada **transforma_elementos** para ler os elementos numéricos de um dado ficheiro de entrada, e aplicar uma função **transforma** a cada elemento, a qual converte o elemento em hexadecimal, e escreve o resultado da aplicação dessa função num dado ficheiro de saída.

PACKAGES

O PROBLEMA

- Objetivo: evitar confundir símbolos que têm o mesmo nome mas existem em módulos de software diferentes.
  - Namespace problem
- O problema resolve-se ao nível do Reader no REPL (read-eval-print loop).
- Duas possibilidades principais:
  - Ocorrências de símbolos com o mesmo nome em diferentes módulos devem ser o mesmo.
  - Ocorrências de símbolos com o mesmo nome em diferentes módulos devem ser diferentes.

POSSÍVEIS SOLUÇÕES

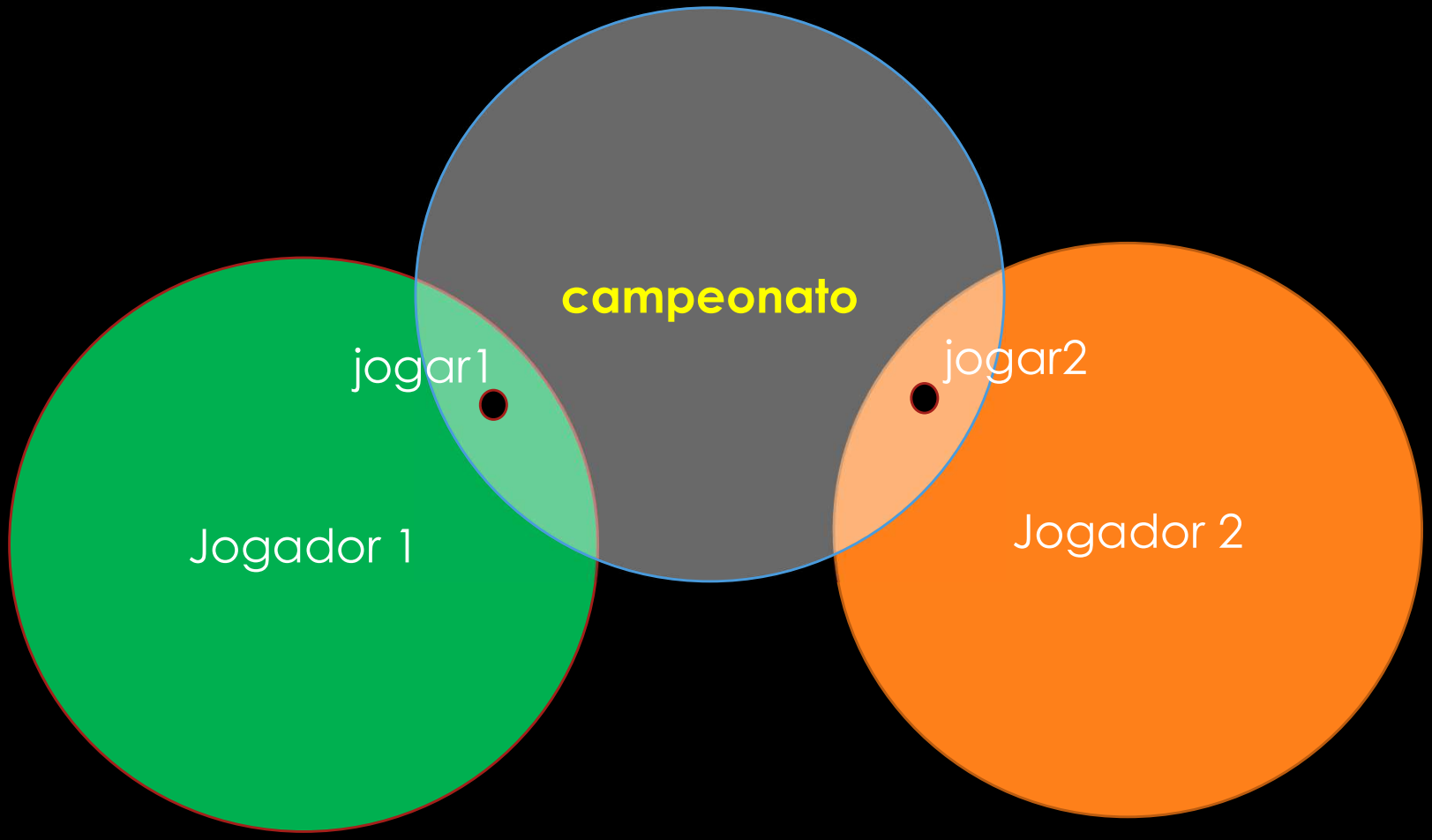
- Em módulos diferentes usar prefixos diferentes para os símbolos.
- Diferentes programadores podem construir diferentes módulos sem interferir um com o outro.

- Problemas:
  - Gestão de nomes dependente da auto-disciplina do programador.
  - Dificuldade em combinar que um nome é o mesmo símbolo (com o mesmo valor e/ou mesma definição de função) em vários módulos.

EXEMPLO CONCRETO

- Campeonato de um Jogo de 2 jogadores
- Necessário:
  - Programa para gerir o jogo, invocando a função principal de cada jogador (recebe o estado e devolve a jogada).
  - Programa que determina a jogada do jogador 1.
  - Programa que determina a jogada do jogador 2.
- O program de gestão do jogo tem de usar símbolos que pertencem a cada um dos outros programas (a função de jogar)
- Cada um dos programas de cada jogador tem de encapsular todos os outros símbolos para evitar conflitos.

EXEMPLO (ESQUEMÁTICO)



A MELHOR SOLUÇÃO: PACKAGES

- O LISP tem namespaces implementados na forma de packages (pacotes de símbolos)
- Existem alguns packages pre-definidos. Exemplo:
  - Common-LISP (nickname: CL)
  - Common-LISP-User (nickname: CL-User)
    - O CL-User usa o CL
- Há uma constante *package* cujo valor é o package de default (CL-User)
- A definição de packages faz-se com defpackage

EXEMPLO SIMPLES DE UTILIZAÇÃO

```
(defpackage :jogador1)
(defpackage :jogador2)

(load "w:\\j1.lisp")
(load "w:\\j2.lisp")
```

PROGRAMA J1.LISP

```
(in-package :jogador1)

(defun alfabeta (...)
        ...)
(defun sucessores (e)
        ...)
(defun jogar (x)
        ...)
```

PROGRAMA J2.LISP

```
(in-package :jogador2)

(defun alfabeta (...)
        ...)
(defun sucessores (e)
        ...)
(defun jogar (x)
        ...)
```

RECOMENDAÇÕES

- Cada ficheiro deve indicar o package em que os símbolos são definidos com in-package
- Apenas deve haver um in-package por ficheiro e deverá ser a primeira linha do ficheiro.
- Para se poder usar o in-package tem de se definir o package antes (defpackage).

DEFINIR PACKAGES

(defpackage name &rest options)

Name = keyword, correspondente ao nome do package

options::=

(:nicknames nickname*)* |
(:documentation string) |
(:use package-name*)* |
(:shadow {symbol-name}*)* |
(:shadowing-import-from package-name {symbol-name}*)* |
(:import-from package-name {symbol-name}*)* |
(:export {symbol-name}*)* |
(:intern {symbol-name}*)* |
(:size integer)

OPÇÕES PRINCIPAIS

- :use
  - conjunto de packages de que o package que está a ser definido herda. Por default, caso não seja dado nada, assume um valor dependente da implementação – habitualmente :Common-LISP.
- :import-from
  - Os símbolos definidos como argumento são importados para o package que está a ser definido
- :export
  - Os símbolos definidos como argumento são encontrados ou criados no package que está a ser definido e exportados, por forma a ser partilhados com os packages que usem o package que está a ser definido

EXPORT E USE-PACKAGE

- Mecanismo para importar todos os símbolos relevantes de um dado package:
  - Todos os packages mantêm uma lista de símbolos que é suposto serem usados por outros packages, designada por exported symbol list.
  - Para adicionar um simbolo a esta lista usa-se a função export.
  - Para remover um simbolo desta lista usa-se a função unexport.
  - Para importar todos os símbolos exportados por um package usa-se a função use-package.
  - Para desfazer a operação anterior usa-se a função unuse-package.

SHADOWING

- Shadowing symbols list
  - Lista associada a um package que contém os símbolos isentos de "erros de conflito de símbolos" detetados quando outros packages são ":used"
- :shadow
  - Conjunto de símbolos criados no package que está a ser definido e que passam a pertencer à shadowing symbols list
- :shadowing-import-from
  - Conjunto de símbolos importados do package indicado para o package que está a ser definido e que passam a pertencer à shadowing symbols list.
    - Caso exista um símbolo com o mesmo nome no package que está a ser definido esse símbolo é removido do package.

INTERN / UNINTERN

- Todos os símbolos que pertencem a um package são "interned".
- Os símbolos são interned no package corrente, que por default é o CL-User.
- As keywords são interned num package especial com o nome KEYWORD.
- Um simbolo interned num package pode ser uninterned, deixando de estar acessível. Exemplo, se o package corrente tiver o simbolo fatorial:

```
> (unintern 'fatorial)
T
```

LISTA DE SÍMBOLOS DE UM PACKAGE

Para obter todos os simbolos definidos num package:

do-symbols *(var [package [result-form]]) declaration* {tag | statement}**

Há 3 macros:

DO-SYMBOLS, DO-EXTERNAL-SYMBOLS, DO-ALL-SYMBOLS

Exemplo:

```
(do-external-symbols (s (find-package “KEYWORD"))
      (print s))
```

EXERCICIOS

- Considere um jogo de 2 jogadores em que cada jogador tenta obter um número (objetivo) com uma sequência de N lançamentos de um dado com 6 faces.
- Ganha o jogador que ficar mais próximo do número.
- Fazer um programa para o jogador 1 jogar. Use o package jogador-1.
- Fazer um programa para o jogador 2 jogar. Use o package jogador-2.
- Fazer um programa jogar num package designado por "campeonato" que jogue alternadamente as funções jogar exportadas por cada um dos packages "jogador-1" e "jogador-2".
  - As funções jogar recebem um estado (na forma de uma lista de 3 elementos: número a atingir, número corrente do próprio jogador e número corrente do adversário) e devolvem uma jogada (na forma de um boolean: t = joga; nil=pára)
  - A função jogar no campeonato lança o dado (gera um número aleatório) que adiciona ao jogador seguinte, parando o jogo quando ambos os jogadores pararem ou quando um deles ultrapassar o objetivo, caso em que perde.

PROGRAMAS COMPLETOS

TIPOS ABSTRATOS DE DADOS

- O LISP é uma linguagem extensível:
  - Permite definir uma linguagem mais abstrata, mais adaptada ao domínio de aplicação
    - Vantagens:
      - Compreensibilidade do código
      - Reutilização de código complexo
      - Facilidade de mudança da representação
  - => Abordagem Bottom-Up

EXEMPLO 1

- Tipo de dados "Turma"
  Turma é um conjunto de alunos
  Necessário definir o tipo "Aluno"

  Construtor do tipo Aluno (Aluno-novo {:<prop> <val>}*)
  <prop> é o nome de uma propriedade qualquer
  <val> o respetivo valor
  no mínimo é preciso o # de aluno.
  Seletor do tipo Aluno (Aluno-<prop> <aluno>)

  Construtor do tipo Turma (Turma-nova {<alunos>})
  Seletor do tipo Turma (Turma-alunos {:<prop> <val>}*)

  É habitual os tipos abstratos terem ainda operações de:
  comparação, predicados, leitura, escrita, para além de toda as operações especificas do tipo abstrato.

EXERCICIO 1

- Fazer um programa para gerir uma pauta de alunos de uma turma:
  - Adicionar alunos a uma turma
    - Por leitura de um ficheiro
    - Interativamente
      - Cada aluno pode ser representado por uma lista com: numero de aluno, nome, nota do projecto 1, nota do projecto 2, nota do exame e média final.
  - Modificar a nota de um aluno
  - Procurar os alunos com notas positivas
  - Procurar os alunos cujo nome começa por uma dada letra
  - Calcular a média e a moda das notas da turma
  - Calcular o histograma baseado em quartis
  - Ordenar a pauta por nota ou por nome
  - Escrever a pauta
    - Num ficheiro
    - No ecran do computador

EXEMPLO 2

- Jogo dos animais: o computador faz perguntas ao utilizador, para tentar descobrir o animal em que este pensou. Se não descobrir, aprende o animal.
- Este jogo é baseado num tipo abstrato de dados que é uma árvore de decisão binária.
  - Admita que a base de dados é constituída por uma estrutura recursiva do tipo:

    (<pergunta> <sim> <não>)

  - A estrutura de dados inicial poderia ser a seguinte:
    - ("Tem asas" ("Voa" Pardal Avestruz) ("Mamífero" Cão Rã))
  - Depois da interacção passaria a ser:
    - ("Tem asas" ("Voa" Pardal
      ("Nada" Pinguim Avestruz))
      ("Mamífero" Cão Rã))

EXERCICIO 2

- Construa um programa para:

1. Ler de um ficheiro "animais.dat" uma estrutura de dados inicial, representando a árvore binária do jogo.
2. Interagir com o utilizador da forma normal para este jogo,
3. Actualizar a estrutura de dados da forma descrita no exemplo,
4. E, no fim do jogo, escrevê-la no mesmo ficheiro.

EXERCICIO 2 (IMPLEMENTAÇÃO)

É necessário definir e implementar o tipo abstrato de dados "árvore binária" através de um tipo "nó" recursivo de 3 elementos.

Exemplo de implementação:

```
(defun escrever-arvore (no &optional (g t))
  (format g "~A" no))

(defun ler-arvore (&optional (f t))
  (read f))
```

```
(defun faz-no (pergunta sim nao)
  (list pergunta sim nao))

(defun no-pergunta (no)
  (first no))

(defun no-sim (no)
  (second no))

(defun no-nao (no)
  (third no))

(defun no-terminalp (no)
  (atom no))
```