UVV
Centro Universitário Vila Velha

Linguagens de Programação II

Programação Funcional usando LISP

Professor: Cristiano Biancardi

Sumário

1 - Elementos da Linguagem

Em linguagens Funcionais toda a programação é realizada por meio da aplicação de funções a argumentos. As variáveis e as operações de atribuição usadas pelas linguagens imperativas não são necessárias. Os laços de repetição são substituídos por chamadas a funções recursivas. Baseiam-se fortemente nos conceitos das funções matemáticas.

Qualquer linguagem de programação lida com duas espécies de objetos: dados e procedimentos. Os dados são os objectos que pretendemos manipular. Os procedimentos são descrições das regras para manipular esses dados.

Se considerarmos a linguagem da matemática, podemos identificar os números como dados e as operações algébricas como procedimentos e podemos combinar os números entre si usando aquelas operações. Por exemplo, 2 × 2 é uma combinação, tal como 2 × 2 × 2 e 2 × 2 × 2 × 2. No entanto, a menos que pretendamos ficar eternamente a resolver problemas de aritmética elementar, somos obrigados a considerar operações mais elaboradas que representam padrões de cálculos. Neste último exemplo, é evidente que o padrão que está a emergir é o da operação de potenciação, i.e, multiplicação sucessiva, tendo esta operação sido definida na matemática há já muito tempo.

Tal como a linguagem da matemática, uma linguagem de programação deve possuir dados e procedimentos primitivos, deve ser capaz de combinar quer os dados quer os procedimentos para produzir dados e procedimentos mais complexos e deve ser capaz de abstrair padrões de cálculo de modo a permitir tratá-los como operações simples, definindo novas operações que representem esses padrões de cálculo.

1.1 - Elementos primitivos

Elementos primitivos são as entidades mais simples com que a linguagem lida. Um número, por exemplo, é um dado primitivo.

Como dissemos anteriormente, o Lisp executa um ciclo *read-eval-print*. Isto implica que tudo o que escrevemos no Lisp tem de ser avaliado, i.e., tem de ter um valor, valor esse que o Lisp escreve no ecran.

Assim, se dermos um número ao avaliador, ele devolve-nos o valor desse número. Quanto vale um número? O melhor que podemos dizer é que ele vale aquilo que vale. Por exemplo, o número 1 vale 1.

```
> 1
1
> 12345
12345
```

```
> 4.5
4.5
```

Como se vê no exemplo, em Lisp, os números podem ser inteiros ou reais.

Exercício 2

Descubra qual é o maior real que o seu Lisp aceita. Consegue fazer o mesmo para os inteiros?

1.2 - Combinações

Combinações são entidades complexas feitas a partir de entidades mais simples. Por exemplo, na matemáticas números podem ser combinados usando operações como a soma ou o produto. Como exemplo de combinações matemáticas, temos 1 + 2 e 1 + 2 × 3. A soma e o produto de números são exemplos de operações extremamente elementares consideradas procedimentos primitivos.

Em Lisp, criam-se combinação escrevendo uma sequência de expressões entre um par de parênteses. Uma expressão é um elemento primitivo ou uma outra combinação. A expressão `(+ 1 2)` é uma combinação dos elementos primitivos `1` e `2` através do procedimento `+`. Já no caso `(+ 1 (* 2 3))` a combinação é ocorre entre `1` e `(* 2 3)`, sendo esta última expressão uma outra combinação.

Não é difícil de ver que as únicas combinações com utilidade são aquelas em que as expressões correspondem a operadores e operandos. Por convenção, o Lisp considera que o primeiro elemento de uma combinação é um operador e os restantes são os operandos.

A notação que o Lisp utiliza para construir expressões (operador primeiro e operandos a seguir) é designada por notação prefixa. Esta notação costuma causar alguma perplexidade a quem inicia o estudo da linguagem, que espera uma notação mais próxima da que aprendeu em aritmética e que é usada habitualmente nas outras linguagens de programação. Nestas, a expressão `(+ 1 (* 2 3))` é usualmente escrita na forma `1 + 2 * 3` (designada notação infixa) que, normalmente, é mais simples de ler por um ser humano. No entanto, a notação prefixa usada pelo Lisp tem largas vantagens sobre a notação infixa:

- É muito fácil usar operadores que têm um número variável de argumentos, como por exemplo:
- > (+ 1 2 3)
- 6
- > (+ 1 2 3 4 5 6 7 8 9 10)

- 55

  Numa linguagem do tipo Pascal apenas existem operadores unários ou binários, e é necessário explicitar os operador binários entre cada dois operandos: 1+2+3 ou 1+2+3+4+5+6+7+8+9+10. Um operador deste tipo diz-se infixo (*in* significa entre). Se se pretender um operador ternário (ou outro) já não se consegue escrever do mesmo modo, sendo necessário implementá-lo como uma função.

- Não existe precedência entre os operadores. Em Pascal, a expressão 1+2*3 tem de ser calculada como se se tivesse escrito 1+(2*3), e não (1+2)*3, i.e., foi criada uma precedência que elimina as ambiguidades. Essa precedência pode ser alterada através do emprego de parênteses. Em Lisp, as expressões seriam necessariamente distintas, (+ 1 (* 2 3)) ou (* (+ 1 2) 3), não podendo haver qualquer ambiguidade.
- Os operadores que nós definimos usam-se exatamente da mesma maneira que os operadores da linguagem: operador primeiro e operandos a seguir. Em Pascal, os operadores são infixos, i.e., estão entre os operandos, e as funções e procedimentos por nós definidos são prefixos, i.e., primeiro a função ou procedimento e depois os operandos. Isto impede a extensão da linguagem de uma forma coerente.

  Para exemplificar este último aspecto, consideremos a operação de exponenciação em Pascal. Para ser coerente com o resto da linguagem, deveria existir um operador, por exemplo ** que permitisse escrever 3**4 para indicar a quarta potência de 3. Como esse operador não existe na linguagem (standard), somos obrigados a criar uma função que o substitua mas, neste caso, a sintaxe muda radicalmente. Esta função, como se usa de forma prefixa não se pode tornar coerente com as restantes operações do Pascal, como a soma e a multiplicação. Em Lisp, pelo contrário, tanto podemos escrever (* 3 3 3 3) como definir uma função que permita escrever (** 3 4).

A desvantagem da notação prefixa está na escrita de combinações complexas. A expressão Pascal 1+2*3-4/5*6, equivale à expressão Lisp (- (+ 1 (* 2 3)) (* (/ 4 5) 6)) que embora seja mais simples de ler para uma máquina, pode ser mais difícil de ler para um ser humano devido à acumulação de parênteses. No entanto, esta expressão pode ser escrita de forma mais clara usando indentação.

A regra para indentação de combinações Lisp é extremamente simples. Numa linha coloca-se o operador e o primeiro operando. Os restantes operandos vêm alinhados por debaixo do primeiro.

```
(- (+ 1
      (* 2 3))
   (* (/ 4 5)
      6))
```

Quando a regra de indentação não é suficiente, usam-se pequenas variações, como seja colocar o operador numa linha e os operandos por debaixo, por exemplo:

```
(umaoperacaocomumnomemuitogrande
  1 2 3 4)
```

A indentação é fundamental em Lisp pois é muito fácil escrever código complexo. A grande maioria dos editores preparados para Lisp (Emacs, por exemplo) formatam automaticamente os programas à medida que os escrevemos e mostram o emparelhamento de parênteses. Desta forma, após algum tempo de prática, torna-se muito fácil escrever e ler os programas, por mais complexa que possa parecer a sua estrutura.

Exercício 3

Converta as seguintes expressões da notação infixa da aritmética para a notação prefixa do Lisp:
1 + 2 - 3
1 - 2 * 3
1 * 2 - 3
1 * 2 * 3
(1 - 2) * 3
(1 - 2) + 3
1 - (2 + 3)
2 * 2 + 3 * 3 * 3

Exercício 4

Converta as seguintes expressões da notação prefixa do Lisp para a notação infixa da aritmética:
(* (/ 1 2) 3)
(* 1 (- 2 3))
(/ (+ 1 2) 3)
(/ (/ 1 2) 3)
(/ 1 (/ 2 3))
(- (- 1 2) 3)
(- 1 2 3)

1.3 - Avaliação de Combinações

Como já vimos, o Lisp considera que o primeiro elemento de uma combinação é um operador e os restantes são os operandos.

O avaliador determina o valor de uma combinação como o resultado de aplicar o procedimento especificado pelo operador ao valor dos operandos. O valor de cada operando é designado de argumento do procedimento. Assim, o valor da combinação (+ 1 (* 2 3)) é o resultado de somar o valor de 1 com o valor de (* 2 3).

Como já se viu, 1 vale 1 e (* 2 3) é uma combinação cujo valor é o resultado de multiplicar o valor de 2 pelo valor de 3, o que dá 6, e que somado a 1 dá 7.

```
> (+ 1 2)
3
> (+ 1 (* 2 3))
7
```

Exercício 5

Calcule o valor das seguintes expressões Lisp:
(* (/ 1 2) 3)
(* 1 (- 2 3))
(/ (+ 1 2) 3)
(/ (/ 1 2) 3)
(/ 1 (/ 2 3))
(- (- 1 2) 3)
(- 1 2 3)
(- 1)

1.4 - Definição de Funções

Tal como em matemática, pode-se definir numa linguagem de programação a operação de elevar um número ao quadrado. Assim, se pretendermos determinar o produto de um número por ele próprio, escrevemos a combinação (* x x) sendo x o número, i.e.

```
> (* 5 5)
25
> (* 6 6)
36
```

Os números 5 e 6 e a operação * são elementos primitivos. As expressões (* 5 5) e (* 6 6) são combinações. À combinação genérica (* x x) queremos associar um nome, por exemplo, quadrado. Isso equivale a acrescentar uma nova função à linguagem.

A título de exemplo, vamos definir a função quadrado:

```
> (defun quadrado (x)
   (* x x))
quadrado
```

Para se definirem novas funções em Lisp, é necessário criar uma combinação de quatro elementos. O primeiro elemento desta combinação é a palavra defun, que informa o avaliador que estamos a definir uma função. O nome defun é uma abreviatura de ``define function''. O segundo elemento é o nome da função que queremos definir, o terceiro elemento é uma combinação com os parâmetros da

função e o quarto elemento é a combinação que determina o valor da função para aqueles parâmetros.

Quando se dá uma expressão desta forma ao avaliador, ele acrescenta a função ao conjunto de funções da linguagem, associando-a ao nome que lhe demos e devolve como valor da definição o nome da função definida.

A definição da função quadrado diz que para se determinar o quadrado de um número x, devemos multiplicar esse número por ele próprio (* x x). Esta definição associa a palavra quadrado a um procedimento. Este procedimento possui parâmetros e um corpo de expressões. De forma genérica temos:

(defun *nome* (*parâmetro-1 ...parâmetro-n*)
  corpo)

Os parâmetros de um procedimento são designados parâmetros formais e são os nomes usados no corpo de expressões para nos referirmos aos argumentos correspondentes. Quando escrevemos no avaliador de Lisp (quadrado 5), 5 é o argumento. Durante o cálculo da função este argumento está associado ao parâmetro formal x. Os argumentos de uma função são também designados parâmetros atuais.

```
> (quadrado 5)
25
> (quadrado 6)
36
```

Note-se que a regra de avaliação de combinações é também válida para as funções por nós definidas. Assim, a avaliação da expressão (quadrado (+ 1 2)) passa pela avaliação do operando (+ 1 2). Este operando tem como valor 3, valor esse que é usado pela função no lugar do parâmetro x. O corpo da função é então avaliado, i.e., o valor final será o da combinação (* 3 3).

O seguinte exemplo mostra um caso um pouco mais complexo. Nele estão apresentadas as etapas de avaliação dos operandos e de avaliação do corpo da função.

```
(quadrado (quadrado (+ 1 2)))
(quadrado (quadrado 3))
(quadrado (* 3 3))
(quadrado 9)
(* 9 9)
81
```

A definição de funções permite-nos associar um procedimento a um nome. Isto implica que o Lisp tem de possuir uma memória onde possa guardar o procedimento e a sua associação ao nome dado. Esta memória do Lisp designa-se ambiente.

Note-se que este ambiente apenas existe enquanto estamos a trabalhar com a linguagem. Quando terminamos, perde-se todo o ambiente. Isto implica que, se não queremos perder o trabalho que estivemos a escrever, devemos escrever as funções num ficheiro e ir passando-as para o Lisp. A grande maioria das implementações do Lisp permite fazer isto de forma automática.

1.5 – Funções básicas

(+) (Soma)

(+ 3.1 2.7) → 5.800000000000001
(+ 4 7 9) → 20
(+ 9/2 4) → 17/2

Note que 9/2 e 17/2 são números racionais que podem ser manipulados da mesma forma de inteiros e floats, e que a função soma não se restringe a apenas dois parâmetros.

(-) (Subtração)

(- 3.1 2.7) → 0.3999999999999999
(- 1 3 5 7) → -14
(- -4) → 4

(*) (Multiplicação)

(* 3.1 7.0) → 21.7
(* 3.1 7.1 10.0) → 220.09999999999997
(* 4 3) → 12

(/) (Divisão)

(/ 21.7 7.0) → 3.1
(/ 9 2) → 9/2
(/ 30 2 3) → 5
(/ 2) → 1/2 - *inverso do número*

Note que se houver mais de dois argumentos na divisão, o primeiro deles será dividido sucessivamente pelos demais.

sqrt (Raiz quadrada)

(sqrt 4) → 2.0
(sqrt 4.0) → 2.0
(sqrt –4) → #C(0,0 2.0)

Se o argumento for um número negativo, a função não retornará um erro mas sim um número complexo na forma #C(3 2) que é equivalente a 3 + 2i.

expt (Exponencial)

(expt 2 10) → 1024
(expt 2 10.0) → 1024.0
(expt 2/3 3) → 8/27
(expt –4 ½) → #C(1.22514845490862E-16 2.0)

Se algum dos argumentos for do tipo float, o resultado também será convertido para float, podendo ocorrer aproximações do número. No caso de 0.0, o resultado é aproximado para #C(1.22514845490862E-16 2.0).

log (Logaritmo)

(log 1) → 0.0
(log 10) → 2.302585092994046
(log 10 2) → 3.3219280948873626

abs (Valor Absoluto)

(abs 8) → 8
(abs –8) → 8
(abs –8.9) → 8.9

truncate (Trunca)

(truncate 13.5) → 13 0.5
(truncate 0.7) → 0 0.7
(truncate –1.7) → -1 -0.7

Truncar é retornar o valor inteiro de um número. Note que a função retorna dois valores: o primeiro é o valor truncado, a parte inteira, e a segunda é a parte decimal do número, “o resto”.

round (Arredonda)

(round 3.1) → 3 0.1
(round 3.7) → 4 -0.3 (round 3.5) → 4 -0.5

Esta função arredonda um número ao inteiro mais próximo. Novamente, há o retorno de dois valores: o número inteiro arredondado e a parte decimal restante do arredondamento, que pode ser positiva ou negativa.

rem (Resto)

(rem 9 3) → 0
(rem 3 9) → 3
(rem –17 4) → -1

Esta função retorna o resto inteiro da divisão.

float (Float)

(float 4) → 4.0
(float 135) → 135.0

Converte um dado do tipo inteiro em ponto flutuante (float).

Atribuição

O comando (*setf variável valor*) faz a atribuição de valores a variáveis.
(setf x 4) → 4 *;a variável x recebe o valor 4*
(setf zero 0 one 1 two 2 three 3 four 4) → 4 *; como foram feitas atribuições múltiplas, o*
; interpretador retorna apenas o último valor.

1.6 – Testando números

A linguagem Lisp possui algumas funções de testes que são chamadas predicados. Esses predicados são funções que retornam apenas um dos dois valores: Verdadeiro (t) ou Falso (nil).

Os predicados listados a seguir podem ser usados com números e símbolos que possuam um valor numérico.

Vamos inicializar quatro variáveis com quatro valores distintos para que os exemplos a seguir sejam executados corretamente.

(setf zero 0 one 1 two 2 three 3 four 4) → 4

(>) (Ordem Descendente)

(> 100 10 one 0.1) → T
(> 10 100 1 0.1) → NIL

Retorna verdadeiro se os elementos estão dispostos em ordem descendente.

(<) (Ordem Ascendente)

(< 1 3 5 2) → NIL
(< 1 two 4) → T

Retorna verdadeiro se os elementos estão dispostos em ordem ascendente.

(=) (Igual)

(= (/ 2.0 3.0) (/ 4.0 6.0)) → T
(= (3.141592653 (/22 7)) → NIL

max (Máximo)

(max 1 -2 3 4 5 6 7) → 7
(max (* 3 7) (* 2 3)) → 21

Retorna o argumento de maior valor.

min (Mínimo)

(min 1 –2 3 4 5 6 7) → -2
(min (expt 2 10) (expt 10 2)) → 100

Retorna o argumento de menor valor.

evenp (Par)

```
( evenp 17 )          →    NIL
( evenp ( * 3 2 ) )   →    T
```

Testa se o argumento é par.

oddp (Ímpar)

```
( oddp 17 )    →    T
( oddp two )   →    NIL
```

Testa se o argumento é ímpar.

minusp (Negativo)

```
( minusp 17 )     →    NIL
( minusp –13 )    →    T
```

Verifica se o argumento é um número negativo.

zerop (Zero)

```
( zerop ( * 2 0 ) )   →    T
( zerop ( + 2 0 ) )   →    NIL
( zerop zero )        →    T
```

Testa se o argumento é igual a zero. *zerop* é mais eficiente que seu equivalente (= number 0).

1.7 – Avaliando funções

Em Lisp, + é uma função, assim uma expressão como (+ 2 3) é uma chamada de função envolvendo dois argumentos.

Quando Lisp avalia uma chamada de função, ele o faz em duas etapas:

- primeiramente os argumentos são avaliados, da esquerda para a direita. Neste contexto, cada argumento se "auto–avalia". Logo, 2 avalia para 2 e 3 valia para 3, e, assim, os valores dos argumentos são 2 e 3, respectivamente;
- os valores dos argumentos são passados a função designada pelo operador. Neste caso, é a função +, que retorna 5.

Se alguns dos argumentos são chamadas de funções, suas avaliações seguem as mesmas regras que as enunciadas anteriormente. Assim, quando (/ (- 7 1) (- 4 2)) é avaliada, observa-se as seguintes etapas:

- Lisp avalia (- 7 1), onde 7 avalia para 7 e 1 avalia para 1. Estes valores são passados para a função -, que retorna 6;
- Lisp valia (- 4 2), onde 4 avalia para 4 e 2 avalia para 2. Estes valores são passados para a função -, que retorna 2;
- Os valores 6 e 2 são enviados para a função /, que retorna 3.

Note que todos os operadores em Common Lisp são de fato funções, exceto para alguns casos (por exemplo, o quote comentado mais adiante). As chamadas de funções são sempre avaliadas da mesma maneira. Os argumentos são avaliados da esquerda para a direita, e seus valores passados para a função, que retorna o valor da expressão como um todo. Isto é conhecido em Common Lisp como a ***evaluation rule*** (regra de avaliação).

1.8 - Símbolos

A definição de funções em Lisp passa pela utilização dos seus nomes. Nomes para as funções e nomes para os parâmetros das funções.

Uma vez que o Lisp, antes de avaliar qualquer expressão, tem de a ler e converter internamente para um objeto, os nomes que criamos têm de ter um objeto associado. Esse objeto é designado por símbolo. Um símbolo é, pois, a representação interna de um nome.

Em Lisp, quase não existem limitações para a escrita de nomes. Um nome como quadrado é tão bom como + ou como 1+2*3 pois o que separa um nome dos outros elementos de uma combinação são apenas parênteses e espaços em branco. Por exemplo, é perfeitamente correto definir e usar a seguinte função:

```
> (defun x+y*z (x y z)
    (+ (* y z) x))
x+y*z
> (x+y*z 1 2 3)
7
```

Lisp atribui um significado muito especial aos símbolos. Reparemos que, quando definimos uma função, os parâmetros formais da função são símbolos. O nome da função também é um símbolo. Quando escrevemos uma combinação, o avaliador de Lisp usa a definição de função que foi associada ao símbolo que constitui o primeiro elemento da combinação. Quando, no corpo de uma função, usamos um símbolo para nos referimos a um parâmetro formal, esse símbolo tem como valor o respectivo parâmetro atual que lhe foi associado naquela combinação. Por esta descrição se vê que os símbolos constituem um dos elementos fundamentais da linguagem Lisp.

Exercício 6

- Crie uma função que some três números.
- Crie uma função que calcule o delta de uma equação do segundo grau. Delta = $b^2 – 4*a*c$

Resposta

1.9 – Inserindo comentários

Para facilitar a compreensão de um código faz-se uso de comentários. Em Lisp, os comentários devem ser precedidos de ponto-e-vírgula " ; ", caracter que indica que a respectiva linha de código não deve ser avaliada pelo interpretador.

```
( defun novogosto ( nome )
        ; adiciona nome no início da lista gosta
        ( setf gosta ( cons nome gosta ) )
        ; deleta nome da lista detesta
        ( setf detesta ( remove nome detesta) ) )
```

2 – Expressões Condicionais

Existem muitas operações cujo resultado depende da realização de um determinado teste. Por exemplo, a função matemática abs --que calcula o valor absoluto de um número-- equivale ao próprio número, se este é positivo, ou equivale ao seu simétrico se for negativo. Estas expressões, cujo valor depende de um ou mais testes a realizar previamente, permitindo escolher vias diferentes para a obtenção do resultado, são designadas expressões condicionais.

No caso mais simples de uma expressão condicional, existem apenas duas alternativas a seguir. Isto implica que o teste que é necessário realizar para determinar a via de cálculo a seguir deve produzir um de dois valores, em que cada valor designa uma das vias.

Seguindo o mesmo raciocínio, uma expressão condicional com mais de duas alternativas deverá implicar um teste com igual número de possíveis resultados. Uma expressão da forma ``caso o valor do teste seja 1, 2 ou 3, o valor da expressão é 10, 20 ou 30, respectivamente" é um exemplo desta categoria de expressões que, embora seja considerada pouco intuitiva, existe nalgumas linguagens (Basic, por exemplo). Contudo, estas expressões podem ser facilmente reduzidas à primeira. Basta decompor a expressão condicional múltipla numa composição de expressões condicionais simples, em que o teste original é também decomposto numa série de testes simples. Assim, poderíamos transformar o exemplo anterior em: ``se o valor do teste é 1, o resultado é 10, caso contrário, se o valor é 2, o resultado é 20, caso contrário é 30".

2.1 - Predicados

Desta forma, reduzimos todos os testes a expressões cujo valor pode ser apenas um de dois, e a expressão condicional assume a forma de ``se ...então ...caso contrário ...". Nesta situação, a função usada como teste é denominada predicado e o valor do teste é interpretado como sendo verdadeiro ou falso. Nesta óptica, o fato de se considerar o valor como verdadeiro ou falso não implica que o valor seja de um tipo de dados especial, como o booleano ou lógico de algumas linguagens (como o Pascal), mas apenas que a expressão condicional considera alguns dos valores como representando o verdadeiro e os restantes como o falso.

Em Lisp, as expressões condicionais consideram como falso um único valor. Esse valor é representado por `nil`. Qualquer outro valor diferente de `nil` é considerado como verdadeiro. Assim, do ponto de vista de uma expressão condicional, qualquer número é um valor verdadeiro. Contudo, não faz muito sentido para o utilizador humano considerar um número como verdadeiro ou falso, pelo que se introduziu uma constante na linguagem para representar verdade. Essa constante representa-se por `t`.

Note-se que, se `t` é diferente de `nil`, e se `nil` é o único valor que representa a falsidade, então `t` representa verdade. Desta forma, existem muitos predicados em Lisp cujo valor é `t` quando a expressão por eles designada é considerada verdadeira.

```
> (> 4 3)
t
> (< 4 3)
nil
```

Existem muitos predicados em Lisp. Os predicados numéricos mais usados são o `zerop`, =, >, <, >=, <=. O `zerop` testa se um número é zero ou diferente de zero.

```
> (zerop 1)
nil
> (zerop 0)
t
```

O fato de `zerop` terminar com a letra ``p'' deve-se a uma convenção adotada em Common Lisp segundo a qual os predicados devem ser distinguidos das restantes funções através da concatenação da letra ``p'' (de *predicate*) ao seu nome.

Apesar da adoção dos símbolos `t` e `nil`, convém alertar que nem todos os predicados devolvem `t` ou `nil` exclusivamente. Alguns há que, quando querem indicar verdade, devolvem valores diferentes de `t` (e de `nil`, obviamente).

2.2 - Operadores Lógicos

Para se poder combinar expressões lógicas entre si existem os operadores `and`, `or` e `not`. O `and` e o `or` recebem qualquer número de argumentos. O `not` só recebe um. O valor das combinações que empregam estes operadores lógicos é determinado do seguinte modo:

- O `and` avalia os seus argumentos da esquerda para a direita até que um deles seja falso, devolvendo este valor. Se nenhum for falso o `and` devolve o valor do último argumento.
- O `or` avalia os seus argumentos da esquerda para a direita até que um deles seja verdade, devolvendo este valor. Se nenhum for verdade o `or` devolve o valor do último argumento.
- O `not` avalia para verdade se o seu argumento for falso e para falso em caso contrário.

Note-se que embora o significado de falso seja claro pois corresponde necessariamente ao valor `nil`, o significado de verdade já não é tão claro pois, desde que seja diferente de `nil`, é considerada verdade.

Exercício 7

Qual o valor das seguintes expressões?

```
(and (or (> 2 3) (not (= 2 3))) (< 2 3))
(not (or (= 1 2) (= 2 3)))
(or (< 1 2) (= 1 2) (> 1 2))
(and 1 2 3)
(or 1 2 3)
(and nil 2 3)
(or nil nil 3)
```

2.3 - Seleção simples

O `if` é a expressão condicional mais simples do Lisp. O `if` determina a via a seguir em função do valor de uma expressão lógica. A sintaxe do `if` é:

```
(if condição
  consequente
  alternativa)
```

Para o `if`, a condição é o primeiro argumento, o consequente no caso de a condição ser verdade é o segundo argumento e a alternativa no caso de ser falso é o terceiro argumento.

```
> (if (> 4 3)
      5
      6)
5
```

Uma expressão `if` é avaliada determinando o valor da condição. Se ela for verdade, é avaliado o consequente. Se ela for falsa é avaliada a alternativa.

Exercício 8

Defina uma função `soma-grandes` que recebe três números como argumento e determina a soma dos dois maiores.

Exercício 9

Escreva uma função que calcule o factorial de um número.

A função `fact` é um exemplo de uma função recursiva, i.e., que se refere a ela própria.

2.4 - Seleção Múltipla

Além do `if`, existe outra expressão condicional em Lisp. O `cond` é uma versão mais potente que o `if`. É uma espécie de switch-case do C. A sua sintaxe é:

```
(cond (condição-1 expressão-1)
```

```
      (condição-2 expressão-2)
                  ⋮
      (condição-n expressão-n))
```

Designa-se o par `(condição-i expressão-i)` por cláusula. O `cond` testa cada uma das condições em sequência, e quando uma delas avalia para verdade, é devolvido o valor da expressão correspondente, terminando a avaliação. Um exemplo será:

```
> (cond ((> 4 3) 5)
        (t 6))
5
```

O `cond` permite uma análise de casos mais simples do que o `if`.

```
(defun teste (x y z w)
   (cond ((> x y) z)
         ((< (+ x w) (* y z)) x)
         ((= w z) (+ x y))
         (t 777)))
```

A função equivalente usando `if` seria mais complicada.

```
(defun teste (x y z w)
    (if (> x y)
        z
        (if (< (+ x w) (* y z))
            x
            (if (= w z)
                (+ x y)
                777))))
```

3 - Funções

3.1 - Funções Recursivas

Uma função recursiva é uma função que se refere a si própria. A idéia consiste em utilizar a própria função que estamos a definir na sua definição.

Em todas as funções recursivas existe:

- Um passo básico (ou mais) cujo resultado é imediatamente conhecido.
- Um passo recursivo em que se tenta resolver um subproblema do problema inicial.

Se analisarmos a função fatorial, o caso básico é o teste de igualdade a zero `(zerop n)`, o resultado imediato é `1`, e o passo recursivo é `(* n (fact (- n 1)))`.

Geralmente, uma função recursiva só funciona se tiver uma expressão condicional, mas não é obrigatório que assim seja. A execução de uma função recursiva consiste em ir resolvendo subproblemas sucessivamente mais simples até se atingir o caso mais simples de todos, cujo resultado é imediato. Desta forma, o padrão mais comum para escrever uma função recursiva é:

- Começar por testar os casos mais simples.
- Fazer chamadas (ou chamada) recursivas com subproblemas cada vez mais próximos dos casos mais simples.

Dado este padrão, os erros mais comuns associados às funções recursivas são, naturalmente:

- Não detectar os casos simples
- A recursão não diminuir a complexidade do problema.

No caso de erro em função recursiva, o mais usual é a recursão nunca parar. O número de chamadas recursivas cresce indefinidamente até esgotar a memória (stack), e o programa gera um erro. Em certas linguagens (Scheme) e implementações do Common Lisp, isto não é assim, e pode nunca ser gerado um erro. A recursão infinita é o equivalente das funções recursivas aos ciclos infinitos dos métodos iterativos do tipo while-do e repeat-until.

Repare-se que uma função recursiva que funciona perfeitamente para os casos para que foi prevista pode estar completamente errada para outros casos. A função fact é um exemplo. Quando o argumento é negativo, o problema torna-se cada vez mais complexo, cada vez mais longe do caso simples. `(fact -1) => (fact -2) => (fact -3) => ...`

Exercício 10

Considere uma versão extremamente primitiva da linguagem Lisp, em que as únicas funções numéricas existentes são `zerop` e duas funções que incrementam e decrementam o seu argumento em uma unidade, respectivamente, `1+` e `1-`. Isto implica que as operações `>`, `<`, `=` e similares não podem ser utilizadas. Nesta linguagem, que passaremos a designar por *nano*Lisp, abreviadamente nanoLisp , defina a função `menor`, que recebe dois número inteiros positivos e determina se o primeiro argumento é numericamente inferior ao segundo.

Exercício 11

Defina a operação `igual?` que testa igualdade numérica de inteiros positivos na linguagem nanoLisp .

Exercício 12

Até ao momento, a linguagem nanoLisp apenas trabalha com números inteiros positivos. Admitindo que as operações `1+`, `1-` e `zerop` também funcionam com números negativos, defina a função `negativo` que recebe um número inteiro positivo e retorna o seu simétrico. Assim, pretendemos obter: `(negativo 3) => -3`.

Exercício 13

Agora que a linguagem nanoLisp pode também trabalhar com números inteiros negativos, defina o predicado `positivo?`, que recebe um número e indica se ele é positivo ou não.

Exercício 14

Defina o teste de igualdade de dois números na linguagem nanoLisp contemplando a possibilidade de trabalhar também com números inteiros negativos.

Exercício 15

Defina a função `simétrico` de um número qualquer na linguagem nanoLisp .

Exercício 16

É possível definir a soma de dois números inteiros positivos em nanoLisp , i.e., apenas recorrendo às funções `1+` e `1-` que somam e subtraem uma unidade, respectivamente. Defina a operação soma.

Exercício 17

Generalize a função de soma de modo a poder receber números inteiros positivos e negativos.

Exercício 18

Do mesmo modo que a soma pode ser definida exclusivamente em termos de sucessor `1+` e predecessor `1-`, a multiplicação pode ser definida exclusivamente em termos da `soma`. Defina a função `mult` que recebe dois número e os multiplica usando a função `soma`.

3.2 – Depuração de funções

Em Lisp, é possível analisar as chamadas às funções através da forma especial `trace`. Ela recebe o nome das funções que se pretendem analisar e altera essas funções de forma a que elas escrevam no terminal as chamadas com os respectivos argumentos em cada chamada, e os valores retornados. Esta informação é extremamente útil para a depuração das funções.
Para se parar a depuração de uma função, usa-se a forma especial `untrace`, que recebe o nome da função ou funções de que se pretende tirar o trace.
Se se usar a forma especial `trace` sem argumentos ela limita-se a indicar quais as funções que estão em trace. Se se usar a forma especial `untrace` sem argumentos, são retiradas de trace todas as funções que estavam em trace.

Exercício 19
Experimentar o trace do fact.

3.3 – Funções de ordem superior

Vimos que as funções permitem-nos dar um nome a um conjunto de operações e tratá-lo como um todo. Muitas vezes, porém, há um padrão que se repete, variando apenas uma ou outra operação. Por exemplo, consideremos uma função que soma os quadrados de todos os inteiros entre *a* e *b*, $\sum_{i=a}^{b} i^2$.

```
(defun soma-quadrados (a b)
  (if (> a b)
    0
    (+ (quadrado a) (soma-quadrados (1+ a) b))))
```

```
> (soma-quadrados 1 4)
30
```

Consideremos agora uma outra função que soma as raízes quadradas de todos os inteiros entre *a* e *b*, $\sum_{i=a}^{b} \sqrt{i}$.

```
(defun soma-raizes (a b)
```

```
 (if (> a b)
   0
   (+ (sqrt a) (soma-raizes (1+ a) b))))
```

```
> (soma-raizes 1 4)
6.146264369941973
```

Em ambas as funções existe uma soma de expressões matemáticas entre dois limites, por exemplo, existe um somatório $\sum_{i=a}^{b} f(i)$. O somatório é uma abstração matemática para uma soma de números. Dentro do somatório é possível colocar qualquer operação matemática relativa ao índice do somatório. Esse índice varia desde o limite inferior até ao limite superior.
Para que se possa definir o processo do somatório na nossa linguagem de programação ela deve ser capaz de fazer abstração sobre as próprias operações a realizar, e deve poder usá-las como parâmetros do processo. O padrão a executar seria qualquer coisa do estilo:

```
(defun soma-??? (a b)
 (if (> a b)
   0
   (+ (aplica-??? a) (soma-??? (1+ a) b))))
```

O símbolo `???` representa a operação a realizar dentro do somatório, e que pretendemos transformar num parâmetro.
Em Lisp, para se aplicar uma função que é o valor de um argumento, usa-se a função `funcall`, que recebe essa função e os seus parâmetros atuais. Para se indicar que pretende-se passar a função associada a um determinado símbolo, usa-se a forma especial `function`.

```
> (funcall (function 1+) 9)
10
> (defun teste (f x y) (funcall f x y))
teste
> (teste (function +) 1 2)
3
```

Deste modo pode-se escrever a implementação do nosso somatório:

```
(defun somatorio (fun a b)
 (if (> a b)
   0
   (+ (funcall fun a) (somatorio fun (1+ a) b))))
```

Pode-se testar a função para o exemplo anterior.

```
> (somatorio (function quadrado) 1 4)
30
```

Como se vê, a função `somatorio` representa a abstração associada ao somatório matemático. Para isso, ela recebe uma função como argumento e aplica-a aos sucessivos inteiros incluídos no somatório.
As funções que recebem e manipulam outras funções são designadas funções de ordem superior.

Exercício 20
Repare-se que, tal como a função somatório, podemos escrever a abstração correspondente ao produtório (também designado piatório) $\prod_{i=a}^{b} f(i)$. Esta abstração corresponde ao produto dos valores de uma determinada expressão para todos os inteiros de um intervalo. Escreva uma função Lisp que a implemente.

Exercício 21
Escreva a função fatorial usando o produtório.

3.4 – Lambda

Se você somente deseja criar uma função temporária e não deseja perder tempo dando-lhe um nome, lambda é justamente o que você precisa. Lambda pode ser vista como uma função sem nome. A sintaxe da lambda é igual à da `defun`, mas em que se omite o nome.

```
> ((lambda (z) (+ z 3)) 2)
5
> (defun teste (fun x y) (funcall fun x y))
> (teste (function (lambda (x y) (* x y))) 2 2)
4
```

As lambdas são a essência do Lisp. A qualquer função corresponde uma lambda. Na realidade, a forma especial `defun` não faz mais do que criar uma lambda com os parâmetros e o corpo da função e associá-la ao nome da função que se está a definir. Quando a forma especial `function` recebe um nome (um símbolo) ela devolve a lambda associada a esse nome.
A designação de lambda (λ) deriva duma área da matemática que se dedica ao estudo dos conceitos de função e de aplicação de função, e que se designa por cálculo- λ. O cálculo-λ é uma ferramenta muito utilizada para estudar a semântica das linguagens de programação.

3.5 – Variáveis Locais

Imagine-se que pretendemos escrever uma função que calcula a seguinte expressão: $f(x,y) = (1 + x^2y)^2x + (1 + x^2y)y$. Em Lisp, temos a seguinte tradução:

```
(defun f (x y)
```

```
(+ (* (quadrado (+ 1 (* (quadrado x) y))) x)
   (* (+ 1 (* (quadrado x) y)) y)))
```

Como se vê, a expressão `(+ 1 (* (quadrado x) y))` aparece repetida duas vezes. Isto, além de dificultar a leitura da função torna-a menos eficiente pois aquela expressão vai ter de ser calculada duas vezes.
Quase todas as linguagens de programação fornecem os meios para se criarem variáveis locais, temporárias, para guardarem resultados parciais que vão ser utilizados em outros sítios. Em Lisp, isso pode ser obtido definindo funções intermédias:

```
(defun f (x y)
  (f* x y (+ 1 (* (quadrado x) y))))

(defun f* (x y temp)
  (+ (* (quadrado temp) x)
     (* temp y)))
```

Mas como já vimos, não há necessidade de se definir uma função `f*` no ambiente pois podemos usar as lambdas diretamente.

```
(defun f (x y)
  ((lambda (temp)
     (+ (* (quadrado temp) x)
        (* temp y)))
   (+ 1 (* (quadrado x) y))))
```

Repare-se que dentro do corpo da lambda há referências quer aos parâmetros da lambda (`temp`) quer aos parâmetros da função `f` em que a lambda está inserida.
Uma vez que não é muito conveniente separar os valores das variáveis, Lisp providencia uma forma especial designada `let` que é convertida para uma lambda. A sua sintaxe é:

```
(let ((var-1 exp-1)
      (var-2 exp-2)
          ⋮
      (var-n exp-n))
  corpo)
```

Quando se avalia um `let`, cada símbolo *`var-i`* é associado ao valor da expressão correspondente *`exp-i`* (em paralelo) e em seguida o corpo é avaliado como se as referências a *`var-i`* estivessem substituídas pelos valores correspondentes de *`exp-i`*. Esta forma especial é absolutamente equivalente a escrever:

```
((lambda (var-1 var-2 ...var-n)
   corpo)
 exp-1exp-2 ...exp-n)
```

Embora equivalentes, a utilização da forma `let` é mais fácil de ler. Este gênero de formas especiais se limita a ser uma tradução mais agradável para outras formas especiais e são designadas por açúcar sintático. O `let` é açucar sintático para uma `lambda`.

Exercício 22

Usando o `let`, reescreva a função f anterior.

Exercício 23

Qual o valor das seguintes expressões:

```
a) > (let ((x 10))
     (+ (let ((x 20))
          (+ x 5))
        (+ x 2)))
b) > (let ((x 10))
     (+ (let ((x 11) (y (+ x 4)))
          (+ y x))
        (+ x 2)))
```

3.6 – Funções Locais

Tal como se podem criar variáveis locais com a forma especial `let`, também é possível criar funções locais com a forma especial `flet`. A sua sintaxe é extremamente parecida com a do `let`, só que o valor de cada variável é a definição de uma função.

A título de exemplo, estude-se a seguinte definição:

```
(defun teste (x)
   (flet ((f-local1 (y) (+ x y))
          (f-local2 (z) (* x z))
          (f-local3 (x) (+ x 2)))
     (+ (f-local1 x) (f-local2 x) (f-local3 x))))

> (teste 2)

12

> (f-local1 2)

Error: Undefined function F-LOCAL1
```

As funções `f-local1`, `f-local2` e `f-local3` são locais à função `teste`, sendo estabelecidas a cada aplicação desta função. Tal como as variáveis do `let`, as funções locais de um `flet` não se podem referir umas às outras, pois são avaliadas em paralelo. Além disso, também não se podem referir a si própias, impedindo a criação de funções locais recursivas.

Atendendo a que a maioria das vezes as funções que definimos são recursivas, independentemente de serem locais ou não, interessa possuir um meio de o podermos fazer. A forma especial `labels` providencia esta possibilidade. A sua sintaxe é igual à do `flet`, mas a sua semântica é ligeiramente diferente. Para o `flet`, o âmbito do nome das funções definidas é apenas o corpo do `flet`. Para o `labels`, esse âmbito é extendido à própria forma especial. Isto permite que se possam definir funções locais recursivas ou mutuamente recursivas.

Observe o seguinte exemplo:

```
(defun teste (x)
  (labels ((f-local1 (y)    (if (zerop y)
                              1
                              (* y (f-local1 (- y 1))))
           )
           (f-local2 (z) (* x z))
           (f-local3 (x) (+ x 2)))
    (+ (f-local1 x) (f-local2 x) (f-local3 x))))
```

Para f-local1 é calculado o fatorial do valor associado a x.

```
> teste 2

10
```

4 – Âmbito e Duração

4.1 – Âmbito de uma referência

Designa-se por âmbito de uma referência, a zona textual em que ela pode ser corretamente referida. Assim, o âmbito de um parâmetro de uma lambda é a zona textual correspondente ao corpo da função. Isto implica que qualquer parâmetro da lambda pode ser referido dentro desse corpo, mas não fora dele.

```
> ((lambda (z) (+ z z)) 3)
6
> (+ z z)
Error: Unbound variable Z
```

Uma vez que o âmbito de um parâmetro é o corpo da lambda correspondente, é possível escrever:

```
> ((lambda (z) ((lambda (w) (+ w z)) 3) 4)
7
```

Reescrevendo o exemplo usando o `let`, temos

```
> (let ((z 4))
   (let ((w 3))
     (+ w z)))
7
```

Neste exemplo, cada `lambda` (ou cada `let`) estabelece um valor para uma variável. Quando se encontra uma referência a uma variável, o seu valor é dado pela ligação correspondente ao contexto menor. Se não existe qualquer ligação em nenhum dos contextos, a variável diz-se não ligada. A avaliação de variáveis não ligadas produz um erro.
Quando uma mesma variável aparece ligada repetidamente em contextos sucessivos, a ligação mais ``interior'' obscurece todas as ``exteriores''. Isto não quer dizer que as ligações exteriores sejam destruídas. Elas são apenas localmente substituídas durante a avaliação do corpo mais interior. Assim, temos o seguinte exemplo:

```
> (let ((x 10))
   (+ (let ((x 20))
        x)
      x))
30
```

Diz-se que uma referência é de âmbito léxico quando ela só pode ser corretamente referida dentro da região textual da expressão que a criou.
Diz-se que uma referência é de âmbito vago (ou indefinido) quando ela pode ser corretamente referida a partir de qualquer região do programa.

Exercício 24
Que tipo de âmbito possui uma variável de um `let`? Que tipo de âmbito possui o nome de uma função?

4.2 – Duração de uma referência

Designa-se por duração de uma referência o intervalo de tempo durante o qual ela pode ser corretamente referida.

Diz-se que uma referência é de duração dinâmica quando só pode ser corretamente referida no intervalo de tempo que decorre durante a avaliação da expressão que a criou.

Diz-se que uma referência é de duração vaga (ou indefinida) quando pode ser corretamente referida em qualquer instante após a avaliação da expressão que a criou.

Em Pascal, os parâmetros de uma função ou procedimento têm âmbito léxico e duração dinâmica. A ligação dos parâmetros formais aos parâmetros atuais existe apenas durante a execução da função ou procedimento.

Em Scheme ou Common Lisp, os parâmetros das lambdas têm âmbito léxico e duração vaga. Isto implica que é possível aceder a uma variável mesmo depois de a função que a criou ter terminado, desde que essa variável seja acedida dentro da região textual dessa função.

A título de exemplo, se tentarmos escrever a função que determina o máximo de uma função numérica mas de forma a que ela possa receber uma tolerância como parâmetro, podemos ser conduzidos a qualquer coisa do género:

```
(defun maximo-func (func a b tol)
  (acumulatorio (function max)
                func
                (funcall func a)
                a
                (function (lambda (x) (+ x tol)))
                b))
```

Repare-se que neste exemplo a função que estabelece o incremento refere-se à variável livre `tol`. Uma das capacidades fundamentais das lambdas é a sua referência a variáveis livres. Uma variável diz-se livre numa lambda quando não é um dos parâmetros da lambda onde é referida.

Quando se aplica uma lambda aos seus argumentos, os parâmetros tomam como valor os argumentos correspondentes, enquanto que as variáveis livres tomam como valor o valor da primeira variável igual no contexto em que a lambda é definida. É por esse motivo que quando a lambda que realiza o incremento é aplicada a um número, ela sabe qual o valor

correto de `tol`. Ele é dado pelo contexto léxico (e.x. textual) em que a lambda foi definida.

Exercício 25

Analise os seguintes casos:

```
a) ((lambda (x) (+ x y)) 10)

b) (let ((y 5)) ((lambda (x) (+ x y)) 10))
```

5 – Dados

Em todos os exemplos anteriores, são apresentadas funções essencialmente numéricas. Os números são um exemplo dos dados que os procedimentos podem usar e produzir. Neste ponto serão apresentados outros tipos de dados que se podem utilizar.

5.1 - Átomos

Os números são um dos elementos primitivos do Lisp. Os símbolos (nomes de funções e variáveis) são outro dos exemplos. Estes elementos dizem-se atômicos, pois não podem ser decompostos.

Para se testar se um elemento é atômico pode-se usar a função `atom`:

```
> (atom 1)
T
```

Exercício 26

Ao testar se o símbolo `xpto` é atômico, escreve-se a expressão `(atom xpto)` e recebe-se um erro. Explique o que se passa.

Para que o Lisp possa considerar um símbolo por si só, e.x., sem o considerar uma variável, é preciso usar a forma especial `quote`, que devolve o seu argumento sem o avaliar.

```
> (quote xpto)
XPTO
> (atom (quote xpto))
T
```

Existem várias funções para se testar a igualdade de elementos primitivos. Como já se viu, a igualdade de números é dada pela função `=`. Esta função compara números de todos os tipos.

```
> (= 1 1)
t
> (= 1 1.0)
t
```

Em Lisp, existe unicidade de símbolos, e.x., dados dois símbolos com o mesmo nome, eles representam necessariamente o mesmo objeto, ou seja, o mesmo espaço da memória do computador. Isto permite que a comparação entre dois símbolos possa ser feita testando se eles representam o mesmo espaço, e.x., se apontam para a mesma zona da memória. A função `eq` realiza essa operação.

```
> (eq (quote a) (quote a))
```

```
t

> (eq (quote a) (quote b))
nil
```

Para se testar símbolos e números do mesmo tipo existe uma outra função designada `eql`.

```
> (eql (quote a) (quote a))
t
> (eql 11111111111111111111111111111
       11111111111111111111111111111)
t
> (eql 1 1.0)
nil
```

5.2 – Combinação de dados

Para se combinar dados, é preciso que a linguagem possua uma ``cola" que permita agrupar esses dados. Em Lisp, essa ``cola" é implementada pela função `cons`.

A função `cons` cria um novo objeto que consiste na aglomeração de dois outros objetos, argumentos do `cons`. O `cons` é para o Lisp o mesmo que as tabelas (*arrays*) e estruturas (*records*, *structs*) são para as outras linguagens como Pascal ou C.

```
> (cons 1 2)
(1 . 2)
> (cons (cons 1 2) 3)
((1 . 2) . 3)
```

Dada uma combinação de objetos (um ``cons") podemos obter o primeiro elemento da combinação usando a função `car` e o segundo usando a função `cdr`.

```
> (car (cons 1 2))
1
> (cdr (cons 1 2))
2
```

Note-se que aplicar o `car` ou o `cdr` a um `cons` não destrói esse `cons`. O `cons` de dois objetos é designado um par com ponto (*dotted pair*). Como é natural, um `cons` não é um objeto atômico:

```
> (atom (cons 1000 2000))
nil
```

Note-se que para combinar dois objetos quaisquer é necessário arranjar espaço na memória para indicar qual o primeiro objeto e qual o segundo. É a função `cons` que arranja esse espaço. De cada vez que ela é chamada, mesmo que seja para juntar os

mesmos objetos, ela arranja um novo espaço de memória. Isto implica que a função `eq` é sempre falsa para o `cons`.

```
> (eq (cons 1 2) (cons 1 2))
nil
```

Já a função equal teste se dois objetos (com relação ao seu conteúdo) são iguais.

```
> (equal (cons 1 2) (cons 1 2))
t

> (equal (cons (cons 1 2) (cons 2 3))
         (cons (cons 1 2) (cons 2 3)))
t
```

5.3 – Abstração de dados

A abstração de dados é uma forma de aumentar a modularidade. Se decidirmos implementar números racionais, teremos de pensar em combinar dois números--o numerador e o denominador, e de os tratar como um todo. Se não fosse possível considerar aquela combinação de números como uma abstração (um racional), toda a sua utilização seria extremamente difícil. Por exemplo, para se somar dois números racionais, seria necessário usar uma operação para o cálculo do numerador, e outra operação para o cálculo do denominador, em vez de se pensar numa operação genérica, `soma-racional`, que receberia dois argumentos-- dois racionais--e calcularia um terceiro número--um racional.

Para nos abstrairmos da complexidade de um número racional, devemos definir funções que os manipulam internamente. Podemos começar por definir uma função que constrói um número racional a partir do numerador e do denominador.

```
(defun racional (numerador denominador)
   (cons numerador denominador))
```

Para sabermos qual é o numerador ou o denominador de um dado número racional podemos definir:

```
(defun numerador (racional)
  (car racional))



(defun denominador (racional)
  (cdr racional))
```

Assim, já já podemos escrever a função que calcula a soma de dois racionais, usando a fórmula $\frac{n_1}{d_1} + \frac{n_2}{d_2} = \frac{n_1 d_2 + n_2 d_1}{d_1 d_2}$.

```
(defun +racional (r1 r2)
  (racional (+ (* (numerador r1) (denominador r2))
               (* (numerador r2) (denominador r1)))
            (* (denominador r1) (denominador r2))))
```

Para simplificar a escrita de racionais, podemos definir uma função que escreve um racional de acordo com uma convenção qualquer.

```
(defun escreve-racional (racional)
  (format t "~a/~a" (numerador racional)
                    (denominador racional)))
```

Agora, já podemos calcular a seguinte expressão:

```
> (escreve-racional (+racional (racional 1 2)
                               (racional 1 3)))
5/6
```

Obs: Veja capítulo 6: Impressão

Exercício 27

Defina as restantes funções do tipo abstrato de informação `racional`: `-racional`, `*racional`, e `/racional`.

Como se vê, tratamos um número racional como um só objeto, e separamos a parte do programa que usa os racionais da parte que os implementa como pares de inteiros. Esta técnica designa-se por abstração de dados.

A abstração é a melhor maneira de lidar com a complexidade. A abstração de dados permite-nos isolar a utilização dos dados do modo como eles estão implementados, através da utilização de barreiras de abstração. Essas barreiras consistem em limitar a utilização dos dados a um pequeno conjunto de funções (`racional`, `numerador` e `denominador`) que escondem a maneira como eles estão implementados. Ao utilizador de um dado tipo de dados, apenas se diz quais as funções que ele pode usar para os manipular, e não qual o funcionamento das funções que implementam aquele tipo de dados.

Seguindo esta metodologia, se precisarmos testar a igualdade de racionais, devemos escrever uma função que o faça usando apenas as funções de manipulação de racionais, i.e., `racional`, `numerador` e `denominador`:

```
(defun =racional (r1 r2)
  (and (= (numerador r1) (numerador r2))
       (= (denominador r1) (denominador r2))))
```

Exercício 28

A função que compara dois racionais não funciona corretamente para todos os casos. Assim,

```
> (=racional (racional 4 6) (racional 4 6))
t
> (=racional (racional 4 6) (racional 2 3))
nil
```

Qual é o problema? Como é que se pode resolver?

Exercício 29

Escreva uma função que calcule o maior divisor comum entre dois números. Para isso, use o algoritmo de Euclides que diz que se *r* é o resto da divisão de *a* por *b*, então o maior divisor comum entre *a* e *b* é também o maior divisor comum entre *b* e *r*: mdc(a,b)=mdc(b,r). Como é natural, quando o resto é zero, o maior divisor comum é o próprio *b*.

Exercício 30

Empregue o método de Euclides para reescrever a função `racional` de modo a só construir números na forma reduzida.

5.4 - Tipos Abstratos de Informação

A teoria dos tipos abstratos de informação diz que o conceito fundamental para a abstração de dados é a definição de uma interface entre a implementação dos dados e a sua utilização. Essa interface é constituída por funções que se podem classificar em categorias: construtores, seletores, reconhecedores e testes. Estas funções são definidas em termos dos objetos mais primitivos que implementam o tipo de dados que se quer definir.

Os construtores são as funções que criam um objeto composto a partir dos seus elementos mais simples. Por exemplo, a função `racional` é um construtor para o tipo racional.

Os seletores são as funções que recebem um objeto composto e devolvem as suas partes. As funções `numerador` e `denominador` são exemplos de seletores.

Os reconhecedores são as funções que reconhecem certos objetos especiais do tipo de dados que se está a definir. A função `zerop` é um reconhecedor para o tipo número do Lisp.

Finalmente, os testes são funções que comparam objetos do tipo que se está a definir. A função `=racional` é um exemplo de uma função desta categoria. Como se pode verificar pela função `=racional`, por vezes, os testes são implementados usando os próprios seletores.

Para que abstração de dados seja corretamente realizada, é fundamental definir o conjunto de construtores, seletores, reconhecedores e testes. Todos os programas que pretenderem utilizar aquele tipo de dados são obrigados a usar apenas aquelas funções. Isso permite que se possa alterar a implementação do tipo de dados sem afetar os programas que o utilizam.

A implementação de estruturas de dados complexas só é corretamente realizada quando se segue esta metodologia com extremo rigor.

Exercício 31

Defina o teste `>racional`.

Quando um tipo abstrato de informação tem de interagir com um utilizador, quer para lhe pedir uma descrição de um elemento do tipo, quer para lhe apresentar uma descrição de um elemento do tipo, usa os denominados transformadores de entrada/saída. A função `escreve-racional` é um exemplo de um transformador de saída para o tipo racional. Ela limita-se a a apresentar uma representação compreensível de um número racional. O transformador de entrada realiza a operação inversa, i.e., constrói um elemento do tipo abstrato a partir de uma representação fornecida.

6 – Entrada e saída

6.1 – Leitura de dados a partir do teclado

Esta função não recebe nenhum argumento. O próprio comando não avalia sua entrada, apenas recebe os dados que o usuário digitar no teclado (sempre seguidos de <enter>). Geralmente vem combinado com outras funções.

Definindo uma função retorna a diferença de dois valores lidos

```
(defun diferenca ()
  (setf x (read))
  (setf y (read))
  (print (- x y)))
```

Avaliação da função diferença

```
>(diferença)
3
4
-1
-1
```

Definindo uma função retorna a média de quadro valores lidos a partir do teclado

```
(defun mediaquatro ()          ;a função não recebe nenhum argumento.
  (/ (+ (read)                 ;soma quatro valores e divide-os por 4.
        (read)
        (read)
        (read))   4))
```

Avaliação da função mediaquatro

```
>(mediaquatro)
2               ;Digite quatro números,
4               ;após cada número tecle Enter.
6
8
5
```

Para chamar esta função, basta digitar *(mediaquatro)* seguido por seus parâmetros (recebidos pelos comandos *read* dentro do corpo da função).

Os quatro valores passados para a função são somados e divididos por 4, resultando em 5.

6.1 – Impressão de dados

Algumas funções podem provocar uma sáida. A mais simples é print, que imprime o seu argumento e então o retorna.

```
> (print 3)
3
3
```

O primeiro 3 acima foi impresso, o segundo retornado.

Se você deseja um output mais complexo, você necessita utilizar format:

```
>(format t "An atom: ~S~%and a list: ~S~%and an integer:~D~%"
          nil (list 5) 6)
An atom: NIL
and a list: (5)
and an integer: 6
```

O primeiro argumento a format é ou t, ou NIL ou um arquivo.

- T especifica que a saída deve ser dirigida para o terminal,
- NIL especifica que não deve ser impresso nada, mas que format deve retornar um string com o conteúdo ao invés,
- uma referência a um arquivo especifica o arquivo para onde a saída vai ser redirigida.

O segundo argumento é um template de formatação, o qual é um string contendo opcionalmente diretivas de formatação, de forma similar à Linguagem "C": `"An atom: ~S~%and a list: ~S~%and an integer:~D~%"`

Todos os argumentos restantes devem ser referenciados a partir do string de formatação.

- As diretivas de formatação do string serão repostas por LISP por caracteres apropriados com base nos valores dos outros parâmetros a que eles se referem e então imprimir o string resultante.
- Format sempre retorna NIL, a não ser que seu primeiro argumento seja NIL, caso em que não imprime nada e retorna o string resultante.

No exemplo acima, há três diretivas de formatação: `~S, ~D` e `~%.`:

- A primeira,`~S`, aceita qualquer objeto LISP e é substituída por uma representação passível de ser impressa deste objeto (a mesma produzida por print).
- A segunda, `~D`, só aceita inteiros.
- A terceira, `~%`, não aceita nada. Sempre é reposta por uma quebra de linha.
- Outra diretiva útil é ~~, que é substituída por um simples ~.

Algumas Derivações de Print

A função *print* possui algumas funções derivadas que também realizam a impressão dos dados, mas de uma forma diferente. Entre elas, podemos citar:

- **terpri –** imprime uma linha em branco (não recebe nenhum argumento).
- **prin1 –** imprime o dado na linha corrente e inicia uma nova linha.
- **princ –** imprime o dado na linha corrente e não inicia nova linha.
- **print –** inicia nova linha e imprime o dado.

Exemplo:

```
(defun mediaquatro (primeiro segundo terceiro quarto)
  (princ "A média de")
  (terpri)  ; salto de linha
  (princ primeiro)
  (princ " ")
  (princ segundo)
  (princ " ")
  (princ terceiro)
  (princ " ")
  (princ quarto)
  (terpri)
  (princ "é")
  (/ (+ primeiro
        segundo
        terceiro
        quarto)
     4))
```

MEDIAQUATRO

```
> (mediaquatro 2 4 6 8)
A média de
2 4 6 8
é
5
```

7 – Listas

As listas são um dos componentes fundamentais da linguagem Lisp. O nome da linguagem é, aliás, uma abreviação de ``list processing''. Como iremos ver, as listas constituem uma estrutura de dados extremamente flexível.

7.1 Operações sobre listas

Em Lisp, quando o segundo elemento de um `cons` é outro `cons`, o Lisp escreve o resultado sob a forma de uma lista:

```
> (cons 1 (cons 2 (cons 3 (cons 4 5))))
(1 2 3 4 . 5)
```

Se o último elemento é a constante `nil`, o Lisp considera que ela representa a lista vazia, pelo que escreve:

```
> (cons 1 (cons 2 (cons 3 (cons 4 nil))))
(1 2 3 4)
```

Esta notação designa-se de lista e é esta que o Lisp usa para simplificar a leitura e a escrita. Uma lista é então uma sequência de elementos. Nesta óptica, a função `car` devolve o primeiro elemento de uma lista, enquanto a função `cdr` devolve o resto da lista. A função `cons` pode ser vista como recebendo um elemento e uma lista e devolve como resultado uma nova lista correspondente à junção daquele elemento no princípio daquela lista. Segundo esta abordagem, a função `cons` é um construtor do tipo abstrato de informação lista, enquanto as funções `car` e `cdr` são seletores.

Uma lista vazia é uma sequência sem qualquer elemento e pode ser escrita como `nil` ou ainda mais simplesmente `()`. A lista vazia é o elemento mais primitivo do tipo lista. `nil` é o construtor do elemento primitivo. Pode-se testar se uma lista é vazia com a função `null`. A função `null` é, portanto, um reconhecedor do tipo lista.

```
> (null nil)
t
> (null (cons 1 (cons 2 nil)))
nil
```

Exercício 32

Escreva uma função que calcula uma lista de todos os números desde *a* até *b*.

```
> (enumera 1 10)
(1 2 3 4 5 6 7 8 9 10)
```

Embora as listas não sejam mais do que uma estruturação particular de células `cons`, podendo por isso ser acedidas com as funções `car` e `cdr`, é considerado melhor estilo de

programação usar as funções equivalentes `first` e `rest`. `first` devolve o primeiro elemento da lista enquanto `rest` devolve o resto da lista, i.e., sem o primeiro elemento. Do mesmo modo, o predicado `null` deve ser substituído pelo seu equivalente `endp`.

```
> (first (enumera 1 10))
1
> (rest (enumera 1 10))
(2 3 4 5 6 7 8 9 10)
```

Exercício 33

Escreva uma função que filtra uma lista, devolvendo uma lista com os elementos que verificam um determinado critério. Utilize-a para encontrar os números pares entre 1 e 20.

```
> (filtra (function par?) (enumera 1 20))
(2 4 6 8 10 12 14 16 18 20)
```

Obs.: lembre-se de definir a função par?.

Esta função já existe em Lisp e denomina-se `remove-if-not`.

Quando se pretendem construir listas pode-se usar também a função `list`. Esta função recebe qualquer número de argumentos e constrói uma lista com todos eles.

```
> (list 1 2 3 4)
(1 2 3 4)
> (first (list 1 2 3 4))
1
> (rest (list 1 2 3 4))
(2 3 4)
> (list 1 2 (list 10 20) 3 4)
(1 2 (10 20) 3 4)
```

Como se vê é possível construir listas dentro de listas. Lisp permite também a construção de listas diretamente no avaliador. Idealmente, bastaria escrever (1 2 3 ...), só que isso seria avaliado segundo as regras de avaliação das combinações. O número 1 seria considerado um operador e os restantes elementos da lista os operandos. Para evitar que uma lista possa ser avaliada podemos usar a forma especial `quote`, que devolve o seu argumento sem o avaliar.

```
> (quote (1 . (2 . (3 . nil))))
(1 2 3)
> (quote (1 2 3 4 5 6 7 8 9 10))
(1 2 3 4 5 6 7 8 9 10)
> (filtro (function par?) (quote (1 2 3 4 5 6 7 8 9 10)))
(2 4 6 8 10)
```

Uma vez que as formas especiais `quote` e `function` são bastante utilizadas, Lisp fornece um meio de se simplificar a sua utilização. Se dermos ao avaliador uma expressão

precedida por uma plica (quote em Inglês), é como se tivessemos empregue a forma especial `quote`. A substituição é feita durante a leitura da expressão. Do mesmo modo, se precedermos uma função ou uma lambda por `#'` (cardinal-plica) é como se tivéssemos empregue a forma especial `function`. `'`*exp* é equivalente a `(quote` *exp*`)`, enquanto que `#'`*exp* é equivalente a `(function exp)`.

```
> '(1 2 3 4 5)
(1 2 3 4 5)
> (filtra #'par '(1 2 3 4 5 6))
(2 4 6)
```

Exercício 44

O que é que o avaliador de Lisp devolve para a seguinte expressão: `(first ''(1 2 3))`?

Como é natural, as operações `car` e `cdr` podem ser encadeadas:

```
> (car '(1 2 3))
1
> (cdr '(1 2 3))
(2 3)
> (car (cdr '(1 2 3))
2
> (car (cdr (cdr '(1 2 3))))
3
```

Dado que aquele gênero de expressões é muito utilizado em Lisp, foram compostas as várias combinações, e criaram-se funções do tipo `(caddr` *exp*`)`, que correspondem a `(car (cdr (cdr` *exp*`)))`. O nome da função indica quais as operações a realizar. Um ``a'' representa um `car` e um ``d'' representa um `cdr`.

```
> (cadr '(1 2 3))
2
> (cdddr '(1 2 3))
nil
```

7.2 – Funções úteis

Exercício 35

Escreva uma função `n-esimo` que devolva o *n*-ésimo elemento de uma lista. Note que o primeiro elemento da lista corresponde a *n* igual a zero.

Esta função já existe em Lisp e denomina-se `nth`.

Exercício 36

Escreva uma função `muda-n-esimo` que recebe um número *n*, uma lista e um elemento, e substitui o *n*-ésimo elemento da lista por aquele elemento. Note que o primeiro elemento da lista corresponde a *n* igual a zero.

Exercício 37

Escreva uma função que calcula o comprimento de uma lista, i.e., determina quantos elementos ela tem.

Esta função já existe em Lisp e denomina-se `length`.

Exercício 38

Escreva uma função que recebe um elemento e uma lista que contém esse elemento e devolve a posição desse elemento na lista.

Exercício 39

Escreva uma função que calcula o número de átomos que uma lista (possivelmente com sublistas) tem.

Exercício 40

Escreva uma função `junta` que recebe duas listas como argumento e devolve uma lista que é o resultado de as juntar uma à frente da outra.

Esta função já existe em Lisp e denomina-se `append`.

Exercício 41

Defina uma função `inverte` que recebe uma lista e devolve outra lista que possui os mesmos elementos da primeira só que por ordem inversa.

Esta função já existe em Lisp e denomina-se `reverse`.

Exercício 42

Escreva uma função designada `inverte-tudo` que recebe uma lista (possivelmente com sublistas) e devolve outra lista que possui os mesmo elementos da primeira só que por ordem inversa, e em que todas as sublistas estão também por ordem inversa, i.e.:

```
> (inverte-tudo '(1 2 (3 4 (5 6)) 7))
(7 ((6 5) 4 3) 2 1)
```

Exercício 43

Escreva uma função `mapear` que recebe uma função e uma lista como argumentos e devolve outra lista com o resultado de aplicar a função a cada um dos elementos da lista.

Esta função já existe em Lisp e denomina-se `mapcar`.

Exercício 44

Escreva uma função denominada `alisa` que recebe uma lista (possivelmente com sublistas) como argumento e devolve outra lista com todos os átomos da primeira e pela mesma ordem, i.e.

```
> (alisa '(1 2 (3 4 (5 6)) 7))
(1 2 3 4 5 6 7)
```

Exercício 45

Escreva uma função `membro?` que recebe um objecto e uma lista e verifica se aquele objecto existe na lista.

Esta função já existe em Lisp e denomina-se `member`. Quando ela encontra um elemento igual na lista devolve o resto dessa lista.

```
> (member 3 '(1 2 3 4 5 6))
(3 4 5 6)
```

Exercício 46

Escreva uma função `elimina` que recebe um elemento e uma lista como argumentos e devolve outra lista onde esse elemento não aparece.

Esta função já existe em Lisp e denomina-se `remove`.

Exercício 47

Escreva uma função `substitui` que recebe dois elementos e uma lista como argumentos e devolve outra lista com todas as ocorrências do segundo elemento substituídas pelo primeiro.

Esta função já existe em Lisp e denomina-se `subst`.

Exercício 48

Escreva uma função `remove-duplicados` que recebe uma lista como argumento e devolve outra lista com todos os elementos da primeira mas sem duplicados, i.e.:

```
> (remove-duplicados '(1 2 3 3 2 4 5 4 1))
(3 2 5 4 1)
```

Esta função não mantém a anterior ordenação da lista. Se pretendermos preservar a ordem original, temos de testar se um elemento já existe na lista que estamos a construir e não na que estamos a analisar.

```
> (remove-duplicados2 '(1 2 3 3 2 4 5 4 1))
(1 2 3 4 5)
```

Esta função já existe em Lisp e denomina-se `remove-duplicates`.

Exercício 49

Escreva as funções inversas do `cons`, `car` e `cdr`, designadas `snoc`, `rac` e `rdc`. O `snoc` recebe um elemento e uma lista e junta o elemento ao fim da lista. O `rac` devolve o último elemento da lista. O `rdc` devolve todos os elementos da lista menos o primeiro.

A função `snoc` já existe em Lisp através da combinação das funções `append` e `list`.

A função `rac` já existe em Lisp através da combinação das funções `first` e `last`.

A função `rdc` já existe em Lisp e denomina-se `butlast`.

Exercício 50

As funções `todos?`, `algum?`, `nenhum?` e `nem-todos?` são predicados que recebem uma função e uma lista e verificam, respectivamente, se a função é verdade para todos os elementos da lista, se é verdade para pelo menos um, se não é verdade para todos e se não é verdade para pelo menos um. Defina estas funções.

Esta função já existe em Lisp e denomina-se `every`.

Esta função já existe em Lisp e denomina-se `some`.

Esta função já existe em Lisp e denomina-se `notany`.

Esta função já existe em Lisp e denomina-se `notevery`.

7.4 – Usando as operações

Quando se pretendem construir listas pode-se usar também a função `list`. Esta função recebe qualquer número de argumentos e constrói uma lista com todos eles.

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

Retorna o primeiro elemento:

> (first (list 1 2 3 4))
```

```
1

Retorna o resto de uma lista: a lista restante a partir do primeiro
elemento:

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

Construindo uma lista dentro de outra lista:

> (list 1 2 (list 10 20) 3 4)
(1 2 (10 20) 3 4)
```

Como se vê é possível construir listas dentro de listas. Lisp permite também a construção de listas diretamente no avaliador. Idealmente, bastaria escrever (1 2 3 ...), só que isso seria avaliado segundo as regras de avaliação das combinações. O número 1 seria considerado um operador e os restantes elementos da lista os operandos. Para evitar que uma lista possa ser avaliada podemos usar a forma especial `quote`, que devolve o seu argumento sem o avaliar.

```
> (quote (1 2 3)
(1 2 3)

> (quote (1 2 3 4 5 6 7 8 9 10))
(1 2 3 4 5 6 7 8 9 10)
```

Uma vez que as formas especiais `quote` e `function` são bastante utilizadas, Lisp fornece um meio de se simplificar a sua utilização. Se dermos ao avaliador uma expressão precedida por uma plica (quote em Inglês), é como se tivessemos empregue a forma especial `quote`. A substituição é feita durante a leitura da expressão. Do mesmo modo, se precedermos uma função ou uma lambda por `#'` (cardinal-plica) é como se tivéssemos empregue a forma especial `function`. `'`*exp* é equivalente a `(quote` *exp*`)`, enquanto que `#'`*exp* é equivalente a `(function exp)`.

```
> '(1 2 3 4 5)
(1 2 3 4 5)

> (remove-if-not #'evenp '(1 2 3 4 5 6))
(2 4 6)
```

Devolve o *n*-ésimo elemento de uma lista. Note que o primeiro elemento da lista corresponde a *n* igual a zero.

nth 0 '(1 2 3)
1

Devolve o comprimento de uma lista, i.e., determina quantos elementos ela tem.

length '(1 2 3 4)
4

Recebe duas listas como argumento e devolve uma lista que é o resultado de as juntar uma à frente da outra.

append (list 1 2 3) (list 4 5 6)
(1 2 3 4 5 6)

`Inverte` uma lista e devolve outra lista que possui os mesmos elementos da primeira só que por ordem inversa.

```
reverse (list 1 2 3)
(3 2 1)
```

Escreva uma função designada `inverte-tudo` que recebe uma lista (possivelmente com sublistas) e devolve outra lista que possui os mesmo elementos da primeira só que por ordem inversa, e em que todas as sublistas estão também por ordem inversa, i.e.:

Recebe uma função e uma lista como argumentos e devolve outra lista com o resultado de aplicar a função a cada um dos elementos da lista.

mapcar (function sqrt) (list 1 2 3)
`(1.0 1.4142135623730951 1.7320508075688772)`

Recebe um objeto e uma lista e verifica se aquele objeto existe na lista. Esta função já existe em Lisp e denomina-se `member`. Quando ela encontra um elemento igual na lista devolve o resto dessa lista.

```
> (member 3 '(1 2 3 4 5 6))
(3 4 5 6)
```

Recebe um elemento e uma lista como argumentos e devolve outra lista onde esse elemento não aparece.

remove 3 (list 12 3 4)
(12 4)

Recebe dois elementos e uma lista como argumentos e devolve outra lista com todas as ocorrências do segundo elemento substituídas pelo primeiro.

subst 1 2 (list 2 3 4 3 2 2)
(1 3 4 3 1 1)

Recebe uma lista como argumento e devolve outra lista com todos os elementos da primeira mas sem duplicados.

remove-duplicates (list 1 1 1 2 2 2 3 3 3 4 4 4)
(1 2 3 4)

Recebe uma lista e devolve uma outra lista sem o último elemento.

butlast (list 2 3 4)
(2 3)

Recebe uma função e uma lista e verifica se se a função é verdade para todos os elementos da lista

every (function zerop) (list 0 1 2)
NIL

every (function zerop) (list 0 0 0)
T

Recebe uma função e uma lista e verifica se se a função é verdade para pelo menos um elemento da lista

some (function zerop) (list 0 1 2)
T

some (function zerop) (list 2 3 4)
NIL

7.5 – Listas de argumentos

Sendo as listas uma das estruturas básicas do Lisp, a linguagem permite aplicar funções directamente a listas de argumentos através da função `apply`. Esta é em tudo idêntica à função `funcall`, mas em vez de ter os argumentos da função a aplicar como argumentos da função funcall, tem-nos numa lista, i.e.:

```
(funcall func arg-1 arg-2 ...arg-n) <=>
(apply func (list arg-1 arg-2 ...arg-n))
```

Na linguagem Common Lisp, a função `apply` é uma fusão entre a função `funcall` e a função `apply` primitiva, pois é da forma:

```
(apply func arg-1 arg-2 ...rest-args)
```

Nesta aplicação o último argumento *`rest-args`* é uma lista com os restantes argumentos. Desta forma, pode-se escrever qualquer das seguintes equivalências:

```
(funcall func arg-1 arg-2 ...arg-n)  <=>
(apply func (list arg-1 arg-2 ...arg-n))  <=>
(apply func arg-1 (list arg-2 ...arg-n))  <=>
(apply func arg-1 arg-2 ...(list arg-n))  <=>
(apply func arg-1 arg-2 ...arg-n nil))
```

7.6 – Tipos aglomerados

Um tipo aglomerado é um tipo abstrato de informação que é composto exclusivamente pela aglomeração de outros tipos abstratos. O conjunto dos racionais é um exemplo pois, como vimos, um racional não é mais do que uma aglomeração de dois inteiros. As operações fundamentais de um tipo aglomerado são os seus construtores e seletores, embora possam existir outras. Como vimos, para um racional, as operações mais utilizadas eram o construtor racional e os seletores numerador e denominador, mas também foram definidos alguns testes e os transformadores de entrada/saída.

Os tipos aglomerados são extremamente utilizados. Por este motivo é costume as linguagens de alto nível fornecerem ferramentas próprias para os tratar. Pascal, por exemplo, permite defini-los com a forma especial record, enquanto que a linguagem C usa, para o mesmo efeito, o struct.

Para exemplificarmos a utilização de tipos aglomerados podemos considerar a definição de um automóvel. Um automóvel é caracterizado por uma marca, um modelo, um dado número de portas, uma cilindrada, uma potência, etc. Para simplificar, podemos considerar só as três primeiras. O construtor de um objeto do tipo automóvel não tem mais que agrupar as informações relativas a cada uma daquelas características. Para isso, podemos usar a função `list`. Assim, criamos o construtor do tipo da seguinte forma:

```
(defun novo-automovel (marca modelo portas)
  (list marca modelo portas))
```

Os seletores do tipo automóvel limitam-se a determinar de que é que um dado objeto daquele tipo é composto:

```
(defun automovel-marca (automovel)
  (nth 0 automovel))

(defun automovel-modelo (automovel)

  (nth 1 automovel))

(defun automovel-portas (automovel)

  (nth 2 automovel))
```

Estando na posse destas funções, podemos criar um automóvel específico, por exemplo:

```
> (novo-automovel 'honda 'civic 2)
(honda civic 2)
```

Dado aquele objeto do tipo automóvel, podemos estar interessados em alterar-lhe o número de portas, passando-as de 2 para 4, por exemplo. Contudo, para manipularmos um tipo abstrato devemos restringirmo-nos às operações desse tipo. Precisamos, portanto,

de criar novas operações que nos permitem modificar um objeto. Para o tipo automóvel poderíamos definir:

```
(defun muda-automovel-marca (automovel nova-marca)
  (muda-n-esimo 0 automovel nova-marca))

(defun muda-automovel-modelo (automovel novo-modelo)
  (muda-n-esimo 1 automovel novo-modelo))

(defun muda-automovel-portas (automovel novo-portas)
  (muda-n-esimo 2 automovel novo-portas))
```

A função `muda-n-esimo` recebia um número *n*, uma lista e um novo elemento, e substitua o *n*-ésimo elemento da lista pelo novo elemento. Esta função não alterava a lista original, produzindo uma nova lista. Desta forma, qualquer destas funções do tipo automóvel deixa o automóvel a modificar absolutamente inalterado, produzindo um novo automóvel. Por este motivo, estas operações devem ser vistas como construtores do tipo automóvel, pois elas criam um novo automóvel a partir de um outro já existente. Elas não permitem alterar um automóvel já criado.

8 – Programação imperativa

Todas as funções que apresentadas anteriormente realizam operações muito variadas e algumas são até relativamente complexas, mas nenhuma afeta os seus argumentos. Elas limitam-se a produzir novos objetos a partir de outros já existentes, sem alterar estes últimos seja de que forma for. Até as próprias variáveis que introduzimos nas funções e que se destinavam a guardar valores temporários não eram mais do que parâmetros de uma lambda, e a sua inicialização correspondia a invocar a lambda com os valores iniciais como argumentos, sendo por isso inicializadas uma única vez e nunca modificadas. Por este motivo, nem sequer foi apresentado nenhum operador de atribuição, tão característico em linguagens como C e Pascal.

Este estilo de programação, sem atribuição, sem alteração dos argumentos de funções, e em que estas se limitam a produzir novos valores, é designado programação funcional. Neste paradigma de programação, qualquer função da linguagem é considerada uma função matemática pura que, para os mesmos argumentos produz sempre os mesmos valores. Nunca nada é destruído. Uma função que junta duas listas produz uma nova lista sem alterar as listas originais. Uma função que muda o número de portas de um automóvel produz um novo automóvel.

A programação funcional tem muitas vantagens sobre outros estilos de programação, em especial no que diz respeito a produzir programas muito rapidamente e minimizando os erros. Contudo, tem também as suas limitações, e a sua incapacidade em modificar seja o que for é a maior. A partir do momento em que introduzimos a modificação de objetos, estamos a introduzir o conceito de destruição. A forma anterior do objeto que foi modificado deixou de existir, passando a existir apenas a nova forma. A modificação implica a introdução do conceito de tempo. Os objetos passam a ter uma história, e isto conduz a um novo estilo de programação.

8.1 – Atribuição

Para se alterar o valor de uma variável Lisp possui um operador de atribuição. A forma especial `setq` recebe uma variável e um valor e atribui o valor à variável.

```
> (let ((x 2))
    (setq x (+ x 3))
    (setq x (* x x))
    (setq x (- x 5))
    x)
20
```

Cada vez que se realiza uma atribuição, perde-se o valor anterior que a variável possuía. Muito embora a forma especial `setq`, como todas as funções e forma especiais, retorne um valor, esse valor não possui qualquer interesse. O operador de atribuição serve apenas para modificar o estado de um programa, neste exemplo, alterando o valor de `x`. Por este motivo, a atribuição assemelha-se mais a um comando do que a uma função. A atribuição é uma ordem que tem de ser cumprida e de que não interessa o resultado. A ordem é a

operação fundamental da programação imperativa, em que um programa é composto por sucessivos comandos. Neste estilo de programação, os comandos funcionam por efeitos secundários. O valor de cada comando não tem interesse e muitas vezes nem sequer tem significado falar dele.

8.2 – Sequenciação

A forma especial `progn` está especialmente vocacionada para este gênero de utilização. Ela recebe um conjunto de expressões que avalia sequencialmente retornando o valor da última. Desta forma é possível incluir várias ações no consequente ou alternativa de um if, por exemplo.

```
(if (> 3 2)
  (progn
    (print 'estou-no-consequente)
    (+ 2 3))
  (progn
    (print 'estou na alternativa)
    (* 4 5)))
```

Algumas das formas especiais do Lisp incluem um `progn` implícito. Vimos atrás um exemplo de um `let` que realizava quatro operações em sequência. Isto implica que o `let` e, por arrastamento, as lambdas e tudo o que for definido com `defun`, possuem também a capacidade de realizar operações em sequência. O `cond` está também nesta categoria. A sua sintaxe é, na realidade, a seguinte:

```
(cond (condição-1 expressão-11...expressão-1l)
      (condição-2 expressão-21...expressão-2m)
                          ⋮
      (condição-n expressão-n1...expressão-nk))
```

O `cond` testa cada uma das condições em sequência, e quando uma delas avalia para verdade, são avaliadas todas as expressões da cláusula correspondente sendo devolvido o valor da última dessas expressões.

Usando o `cond`, o exemplo anterior ficará mais elegante:

```
(cond ((> 3 2)
       (print 'estou-no-consequente)
       (+ 2 3))
      (t
       (print 'estou na alternativa)
       (* 4 5)))
```

A forma especial `prog1` é semelhante ao `progn`. Ela recebe um conjunto de expressões que avalia sequencialmente retornando o valor da primeira. A expressão equivalente ao exemplo anterior mas utilizando o `prog1` será:

```
(if (> 3 2)
```

```
(prog1
  (+ 2 3)
  (print 'estou-no-consequente))
(prog1
  (* 4 5)
  (print 'estou na alternativa)))
```

Note-se que a ordem de avaliação das expressões de um `prog1` é igual à de um `progn`. Apenas o valor retornado é diferente: é o primeiro no caso do `prog1` e o último no caso do `progn`.

A sequenciação é também suportada por qualquer lambda. Em consequência, as formas especiais que implicam a criação de lambdas, como o `let` e o próprio `defun` permitem também especificar mais do que uma expressão, sendo estas avaliadas em sequência e devolvido o valor da última.

8.3 – Alteração de dados

A atribuição não está restricta a variáveis. É também possível alterar o conteúdo da maioria dos tipos de dados Lisp. Uma célula `cons`, por exemplo, pode ser alterada com as funções `rplaca` e `rplacd`, significando, respectivamente ``replace-car'' e ``replace-cdr''.

```
> (let ((x (cons 1 2)))
    (rplaca x 3)
    x)
(3 . 2)

> (let ((x (cons 1 2)))
    (rplacd x 3)
    x)
(1 . 3)
```

Note-se que estas funções são uma forma de atribuição e, como tal, destroem o conteúdo anterior da estrutura a que se aplicam. Por este motivo, este género de funções diz-se destrutivo. Na sequência lógica da convenção usual em Lisp para a definição de reconhecedores, que terminam sempre com um ponto de interrogação (ou com a letra ``p'' de predicado), deve-se colocar um ponto de exclamação no fim do nome das funções destrutivas para salientar o seu carácter imperativo.

Exercício 51

Escreva uma função `junta!` (note-se o ponto de exclamação) que recebe duas listas como argumento e devolve uma lista que é o resultado de as juntar destrutivamente uma à frente da outra, i.e., faz a cauda da primeira lista apontar para a segunda.

Esta função já existe em Lisp e denomina-se `nconc`.

Exercício 52

Analise o seguinte exemplo funcional e imperativo:

```
> (let ((x '(1 2 3)))
    (junta x x))
(1 2 3 1 2 3)

> (let ((x '(1 2 3)))

    (junta! x x))

(1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 ...
```

Exercício 53

Escreva uma função `muda-n-esimo!` (note-se o ponto de exclamação) que recebe um número *n*, uma lista e um elemento, e substitui o *n*-ésimo elemento da lista por aquele elemento. Note que o primeiro elemento da lista corresponde a *n* igual a zero.

Exercício 54

Reescreva as operações do tipo abstrato de informação automóvel que alteravam as características de um elemento do tipo de forma a torná-las destrutivas.

Quando as operações de um tipo abstrato alteram um elemento do tipo, essas operações são classificadas como modificadores do tipo abstrato. Os modificadores, como caso especial da atribuição, são muito empregues em programação imperativa. Note-se que os modificadores possuem todos os problemas da atribuição simples, nomeadamente a alteração ser destrutiva. Isto levanta problemas quando se testa igualdade em presença de modificação.

```
> (let ((x (novo-automovel 'honda 'civic 2)))
    (let ((y (muda-automovel-portas x 4)))
      (eql x y)))
NIL

> (let ((x (novo-automovel 'honda 'civic 2)))
    (let ((y (muda-automovel-portas! x 4)))
      (eql x y)))
T
```

Repare-se que no primeiro exemplo (funcional), o automóvel modificado é, logicamente, diferente do automóvel original. `x` e `y` representam automóveis diferentes. No segundo exemplo (imperativo), a modificação do automóvel que `x` representa é realizada sobre esse próprio automóvel, de modo que `x` e `y` acabam por representar um mesmo automóvel (modificado). Muito embora esta situação possa ter vantagens, ela permite também a introdução de erros muito subtis e extremamente difíceis de tirar. Para dar apenas um pequeno exemplo, repare-se que o automóvel que `x` representa passou a ter quatro portas embora uma leitura superficial do código sugira que ele foi criado com apenas duas.

8.4 – Repetição

Para além dos operadores de atribuição (`setq`, `rplaca`, `rplacd`, etc.) e de sequenciação (`progn`, `prog1`, etc.) a linguagem Common Lisp possui muitas outras formas especiais destinadas a permitir o estilo de programação imperativa. De destacar são as estruturas de controle de repetição, tais como o `loop`, o `do`, o `dotimes` e ainda outras adequadas para iterar ao longo de listas.

O `loop` é a mais genérica de todas as formas de repetição. Ela recebe um conjunto de expressões que avalia sequencialmente, repetindo essa avaliação em ciclo até que seja avaliada a forma especial `return`. Esta última recebe uma expressão opcional e termina o ciclo em que está inserida, fazendo-o devolver o valor daquela expressão.

A seguinte expressão exemplifica um ciclo que escreve todos os números desde 0 até 100, retornando o símbolo `fim` no final do ciclo.

```
(let ((n 0))
  (loop
    (print n)
    (setq n (1+ n))
    (when (> n 100)
      (return 'fim))))
```

A forma especial `do` é um pouco mais sofisticada que o `loop`. Ela permite estabelecer variáveis, inicializá-las e incrementá-las automaticamente, testar condições de paragem com indicação do valor a retornar e repetir a execução de código. Se reescrevermos o exemplo anterior usando a forma especial `do`, obtemos:

```
(do ((n 0 (1+ n)))
    ((> n 100) 'fim)
  (print n))
```

Tal como o `loop`, a forma especial `do` pode ser interrompida em qualquer altura com um `return`, retornando o valor opcional fornecido com o `return`.

Apesar do estilo mais utilizado na maioria das linguagens de programação ser o imperativo, ele é muito pouco natural em Lisp.

A falta de naturalidade resulta, por um lado, de os programas em Lisp se decomporem geralmente em pequenas funções que calculam valores, invalidando uma abordagem baseada em ciclos de alteração de variáveis, típica da programação imperativa.

Por outro lado, a grande maioria de tipos de dados existentes em Lisp são inerentemente recursivos, o que dificulta o seu tratamento segundo o estilo imperativo.

Apesar de muito pouco prático para usar em Lisp, a programação imperativa tem algumas vantagens, das quais a possibilidade de atribuição é a maior (e também a mais perigosa).

9 – Modelos de ambientes

Até agora vimos que as variáveis eram apenas designações para valores. Quando se avaliava uma expressão, as variáveis desapareciam, sendo substituídas pelos seus valores. A partir do momento em que podemos alterar o valor de uma variável, o seu comportamento torna-se menos claro.

Para se explicar correctamente este comportamento é necessário passar para um modelo de avaliação mais elaborado designado modelo de avaliação em ambientes.

Neste modelo, uma variável já não é uma designação de um valor mas sim uma designação de um objecto que contém um valor. Esse objecto pode ser visto como uma caixa onde se guardam coisas. Em cada instante, a variável designa sempre a mesma caixa, mas esta pode guardar coisas diferentes. Segundo o modelo de ambientes, o valor de uma variável é o conteúdo da caixa que ela designa. A forma especial `setq` é a operação que permite meter valores dentro da caixa.

As variáveis são guardadas em estruturas denominadas enquadramentos. Por exemplo, cada vez que usamos a forma `let` é criado um novo enquadramento para conter as variáveis estabelecidas pelo `let`. Todas as expressões pertencentes ao corpo do `let` serão avaliadas em relação a este enquadramento. Imaginemos agora a seguinte situação:

```
(let ((x 1))
  (let ((y 2)
        (z 3))
    (+ x y z)))
```

Neste exemplo, o corpo do primeiro `let` é um novo `let`. Existem portanto dois enquadramentos. Estes enquadramentos estão organizados de modo a que o corpo do segundo `let` consiga fazer referência às três variáveis `x`, `y` e `z`.

Para isso, os enquadramentos são estruturados sequencialmente, desde aquele que for textualmente mais interior até ao mais exterior. Essa sequência de enquadramentos é designada por ambiente.

Cada enquadramento é uma tabela de ligações, que associa as variáveis aos seus valores correspondentes. Uma variável nunca pode estar repetida num enquadramento, embora possa aparecer em vários enquadramentos de um ambiente. Cada enquadramento aponta para o ambiente envolvente, excepto o ambiente global, que é composto por um único enquadramento sem ambiente envolvente. É no ambiente global que estão guardadas todas as funções que usamos normalmente.

9.1 – Âmbito léxico

A regra de avaliação de variáveis em ambientes diz que o valor de uma variável em relação a um ambiente é dado pela ligação dessa variável no primeiro enquadramento em que ela surja ao longo da sequência de enquadramentos que constituem esse ambiente. Se nenhum enquadramento possui uma ligação para essa variável ela diz-se não ligada. É um erro avaliar variáveis não ligadas.

Uma vez que os enquadramentos de um ambiente estão associados lexicamente às formas que os criaram, é possível determinar o âmbito de uma variável qualquer simplesmente observando o texto do programa.

Usando o modelo de avaliação em ambientes é muito fácil perceber o comportamento da forma especial `let` (que não é mais do que uma simplificação de uma lambda) e da forma especial `setq`. Cada `let` aumenta o ambiente em que é avaliado com um novo enquadramento, estabelecendo ligações para as suas variáveis. Quando se pretende saber o valor de uma variável percorre-se o ambiente, começando pelo primeiro enquadramento até se encontrar a ligação correspondente. Se ela não aparecer, vai-se passando de enquadramento em enquadramento até se atingir o ambiente global, e se aí também não existir nenhuma ligação para aquela variável é gerado um erro de variável não ligada. O `setq` altera o valor da variável que aparece estabelecida no enquadramento mais próximo do ponto onde o `setq` é avaliado. Se se atingir o ambiente global, e se aí também não existir nenhuma ligação para aquela variável, é criada essa ligação no ambiente global.

```
> (let ((x 10))
    (+ x y))
Error: Unbound variable: Y
> (setq y 20)
20
> (let ((x 10))
    (+ x y))
30
```

Como se vê pelo exemplo. A partir do momento em que se estabeleceu uma ligação no ambiente global para a variável `y`, já é possível avaliar aquele `let` apesar de ele fazer referência a uma variável livre. No entanto, a utilização da forma especial `setq` para criar variáveis globais é considerada pouca correcta e o compilador emitirá um aviso se encontrar uma destas formas de utilização. A utilização do `setq` deve ser restricta a modificações do valor de variáveis previamente estabelecidas.

As regras de avaliação do modelo de ambientes são, em tudo, equivalentes às do modelo clássico, excepto no que diz respeito à aplicação de funções.

No modelo de ambientes todas as funções possuem um ambiente associado, que corresponde àquele que existia quando a função foi definida. Quando se aplica uma função aos seus argumentos, cria-se um novo ambiente, cujo primeiro enquadramento contém as ligações dos parâmetros formais da função aos seus argumentos e cujo

ambiente envolvente é aquele em que a função foi definida. É em relação a este novo ambiente que se avalia o corpo da função.

Note-se que a forma especial `defun` define funções (i.e., cria uma ligação entre o nome da função e a lambda correspondente ao seu corpo) sempre no ambiente global, enquanto que `setq` altera a ligação de uma variável no primeiro enquadramento do ambiente em que a forma especial é avaliada. Só se a variável não for encontrada na sequência de enquadramentos é que o `setq` cria uma no ambiente global.

Exercício 55

Interprete o comportamento da seguinte função:

```
(let ((valor 0))
  (defun incrementa ()
     (setq valor (1+ valor))
     valor))

> (incrementa)
1
> (incrementa)
2
> (incrementa)
3
```

Exercício 56

Uma excelente aplicação de funções com estado local é na criação de geradores de sequências de números, i.e., funções sem argumentos que, a cada invocação, devolvem o elemento que sucede logicamente a todos os que foram gerados anteriormente. Seguindo este paradigma, defina o gerador da sequência de Fibonacci, que é representada pela sucessão crescente 1, 2, 3, 5, 8, ..., em que cada número é a soma dos dois últimos que o precedem.

```
> (gera-fib)
1
> (gera-fib)
2
> (gera-fib)
3
```

Exercício 57

Infelizmente, a função `gera-fib` não sabe recomeçar. Para isso, é necessário defini-la (compilá-la) outra vez. Este processo, como é lógico, é muito pouco prático, em especial porque obriga o utilizador a ter acesso ao código do gerador. Complemente a definição da função `gera-fib` com uma outra função denominada `repoe-fib` que repõe o gerador em condições de recomeçar de novo desde o princípio.

```
> (gera-fib)
1
```

```
> (gera-fib)
2
> (repoe-fib)
0
> (gera-fib)
1
```

9.2 – Âmbito dinâmico

A criação de variáveis globais em Common Lisp deve ser feita usando as formas especiais `defconstant` para criar constantes, `defvar` para criar variáveis inicializáveis uma única vez e `defparameter` para criar variáveis inicializáveis várias vezes. Esta diferença é relevante sobretudo durante a fase de desenvolvimento e depuração de programas.

```
> (defconstant acelaracao-gravidade 9.8)
ACELARACAO-GRAVIDADE
> (defvar *y*)
*Y*
> (defparameter *z* 10)
*Z*
```

Note-se a convenção adotada para as variáveis globais de usar nomes compreendidos entre um par de asteriscos. Quando se definem constantes essa convenção não se aplica.

O fato de podermos ter variáveis globais introduz uma alteração nas regras de avaliação. Tínhamos visto que as variáveis que eram parâmetros de funções (e, como tal, as variáveis introduzidas por um `let`) tinham âmbito léxico, ou seja, apenas podiam ser referidas dentro da região textual que as introduziu. No entanto, as variáveis globais como `aceleracao-gravidade`, `*y*` ou `*z*` podem ser referidas de qualquer ponto do programa, fazendo com que o seu âmbito passe a ser vago. No entanto, apesar de ser possível referênciar a variável `*y*`, será produzido um erro quando tentarmos determinar o seu valor, uma vez que ele ainda está indefinido. Será preciso ligarmos um valor àquela variável antes de a podermos avaliar e, para isso, podemos usar um `let`.

```
> (defvar *y*)
*Y*
> (defun teste () (+ *y* *y*))
TESTE
> (teste)
Error: Unbound variable: *Y*
> (let ((*y* 10)) (teste))
20
>*y*
Error: Unbound variable: *Y*
```

Repare-se que apesar de, momentaneamente, termos atribuído um valor à variável `*y*` por intermédio de um `let`, ela perdeu esse valor assim que terminou o `let`. A duração da variável `*y*` é, assim, dinâmica. Apenas as variáveis léxicas possuem duração indefinida.

Variáveis como `*y*` e `*z*` dizem-se especiais e possuem âmbito vago e duração dinâmica. Esta combinação dá origem a um comportamento que se designa de âmbito dinâmico.

Um dos aspectos mais críticos na utilização de variáveis de âmbito dinâmico é o fato de, geralmente, não ser suficiente ler o código de um programa para perceber o que é que ele vai fazer--é também preciso executá-lo. O seguinte exemplo explica este ponto.

Imaginemos as seguintes definições:

```
> (let ((x 2))
    (defun soma-2 (y)
      (+ x y)))
SOMA-2
> (let ((x 1000))
    (soma-2 1))
3
> (defparameter *x* 1)
*X*
> (let ((*x* 2))
    (defun soma-2 (y)
      (+ *x* y)))
SOMA-2
> (let ((*x* 1000))
    (soma-2 1))
1001
```

O primeiro exemplo envolve apenas variáveis léxicas. Daí que baste observar o texto da função `soma-2` para se perceber que a variável `x` usada em `(+ x y)` toma sempre o valor `2`.

No segundo exemplo, a única diferença está no fato de a variável `*x*` ser especial. Nesta situação a função `soma-2` não usa o valor de `*x*` que existia no momento da *definição* da função, mas sim o valor de `*x*` que existe no momento da *execução* da função. Desta forma, já não é suficiente observar o texto da função `soma-2` para perceber o que ela faz. Por este motivo, o uso excessivo de variáveis dinâmicas pode tornar um programa difícil de ler e, consequentemente, difícil de desenvolver e difícil de corrigir.

10 – Parâmetros especiais

10.1 – Parâmetros opcionais

Como referimos anteriormente, a forma especial `return`, que se pode utilizar dentro de um `loop`, possui um parâmetro opcional, que é o valor a retornar do ciclo. Isto quer dizer que o argumento que deveremos passar para esse parâmetro pode ser omitido. Esta característica é também muito útil para a definição de funções, permitindo que ela possa assumir certos parâmetros por omissão.

Para definirmos funções que têm parâmetros opcionais temos de usar um qualificador especial designado `&optional` na lista de parâmetros formais. Esse qualificador indica que todos os parâmetros que se lhe seguem são opcionais e que, se os argumentos correspondentes forem omitidos, eles valem `nil`. Se pretendermos um valor diferente para um parâmetro, podemos inserir o parâmetro numa lista com o seu valor. A seguinte função mostra como se pode definir a função `incr` que incrementa o seu argumento de uma unidade, ou de uma quantidade que lhe seja fornecida.

```
(defun incr (x &optional (i 1))
  (+ x i))



> (incr 10)

11

> (incr 10 5)

15
```

Exercício 58

Defina a função `eleva`, que eleva um número a uma determinada potência. Se a potência não for indicada deverá ser considerada 2. Nota: a expressão `(expt x y)` determina a potência y de x, i.e., x^y.

Exercício 59

Reescreva a função factorial de forma a gerar um processo iterativo mas sem usar funções auxiliares.

10.2 – Parâmetros de resto

Para além do qualificador `&optional` existem ainda o `&rest` e o `&key`. O `&rest` só pode qualificar o último parâmetro de uma função, e indica que esse parâmetro vai ficar ligado a uma lista com todos os restantes argumentos. A título de exemplo, temos:

```
> ((lambda (x y &rest z) (list x y z)) 1 2 3 4 5 6)
(1 2 (3 4 5 6))
```

O qualificador `&rest` permite assim construir funções com qualquer número de argumentos.

Exercício 60

Defina a função lista que recebe qualquer número de argumentos e constrói uma lista com todos eles.

```
>(lista 1 2 3 4 5 6 7 8 9 0)
(1 2 3 4 5 6 7 8 9 0)
```

10.3 – Parâmetros de chave

O qualificador `&key` informa o avaliador que os parâmetros qualificados são ligados através de uma indicação explícita de quem chama a função. Essa indicação é feita designando o nome de cada parâmetro precedido por dois pontos e indicando qual o valor a que ele deve estar ligado. Os parâmetros que não forem ligados comportam-se como se fossem opcionais. Este gênero de parâmetros dizem-se de chave (keyword).

Desta forma, o `&key` permite trocar a ordem dos argumentos. O seguinte exemplo mostra o funcionamento do `&key`.

```
> ((lambda (x y &key z (w 4) k) (lista x y z w k))
   1 2 :k 5 :z 3)
(1 2 3 4 5)
```

A grande maioria das funções pré-definidas na linguagem para manipular listas possui parâmetros opcionais e de chave. Repare-se que as chaves são tratadas de forma especial pelo avaliador. Efetivamente, se assim não fosse, quando especificávamos os argumentos de uma função com parâmetros de chave, o avaliador iria tentar determinar o valor das chaves, gerando então um erro por estas não terem valor. Na realidade, quando um símbolo qualquer é precedido por dois pontos, esse símbolo é considerado como especial, pertencendo ao conjunto dos símbolos chaves, e avaliando para si próprio.

```
> ola
Error: Unbound variable: OLA
> 'ola
OLA
> :ola
:OLA
```

Os tipos aglomerados constituem uma das grandes utilizações dos parâmetros de chave. Nestes tipos de dados, os construtores limitam-se a realizar um agrupamento de valores para os diversos constituintes do tipo. Vimos o exemplo de um automóvel, que era constituído por uma marca, um modelo, um número de portas, etc. Embora não exista qualquer razão para que a marca de um automóvel seja mais importante que o seu número de portas, infelizmente a ordenação implícita dos argumentos das funções Lisp impõem que assim seja. Podemos resolver este problema usando parâmetros de chave, de forma a eliminar a ordenação dos argumentos e permitir ao utilizador especificá-los pela ordem que entender, por exemplo:

```
(defun novo-automovel (&key marca modelo portas)
  (list marca modelo portas))

> (novo-automovel :portas 2 :marca 'honda :modelo 'civic)
(honda civic 2)
```

Para além da possibilidade de alteração da ordem dos argumentos, o qualificador `&key` ajuda à legibilidade do programa, ao tornar explícito o papel que cada argumento tem na função.

11 – Macros

Como referimos na apresentação da linguagem Lisp, existem certas formas da linguagem que não obedecem às regras de avaliação usuais. Essas formas designam-se formas especiais e o `if` é um exemplo. Cada forma especial possui a sua própria regra de avaliação. Vimos que, por isso, era impossível definir o `if` como se fosse uma função, pois todos os operandos (o teste, o consequente e a alternativa) seriam avaliados.

Embora a linguagem Lisp possua muitas formas especiais, é possível ``criar'' outras formas especiais através da utilização de macros. Uma macro é uma forma que a linguagem expande para outra forma, superando assim as dificuldades inerentes à avaliação dos argumentos que as funções realizam. Na realidade, Lisp possui muito poucas formas especiais reais. A grande maioria das formas especiais são implementadas através de macros, usando a forma especial `defmacro` cuja sintaxe é igual à da `defun`.

11.1 – Avaliação de macros

A utilização de uma macro implica duas avaliações. Na primeira, a macro produz uma expressão Lisp a partir dos seus argumentos, que se designa a expansão da macro. Esta expressão é então avaliada uma segunda vez para produzir o valor final. A título de exemplo, se definirmos o `if` como uma macro que expande para um `cond` e avaliarmos a expressão `(if (> 3 4) (+ 1 2) (- 5 2))`, a primeira avaliação deverá produzir a expressão `(cond ((> 3 4) (+ 1 2)) (t (- 5 2)))`, que será avaliada segunda vez para determinar o seu valor.

Note-se que, neste exemplo, se a forma `cond` fosse, também ela, uma macro, o processo era aplicado recursivamente até que não surgisse mais nenhuma macro. Nessa altura, o Lisp usava a regra usual de avaliação para determinar o valor final da expressão

11.2 – Escrita de macros

A escrita de uma macro é inerentemente mais complexa que a escrita de uma função, sendo decomposta em quatro fases:

1. Decidir se a macro é realmente necessária. Esta fase é de grande importância, pois cada vez que se define uma macro está-se a aumentar a linguagem com uma nova forma especial. Quem pretende ler um programa que usa a macro é obrigado a conhecer a sua sintaxe e semântica, e se o número de macros é muito grande, pode ser difícil perceber o código.
2. Escrever a sintaxe da macro. Nesta fase pretende-se definir qual vai ser a forma de utilização da macro. A sintaxe deve ser o mais simples possível e o mais coerente possível com as restantes formas da linguagem para não complicar a sua leitura.

3. Escrever a expansão da macro. Nesta fase determina-se a expressão Lisp que a macro deve produzir quando expandida. A expansão é qualquer expressão Lisp, que pode inclusive fazer referência a outras macros.
4. Escrever a macro usando a forma especial `defmacro`. É esta a fase mais delicada do processo, em que se programa um processo de transformar a forma especial que queremos definir numa expressão que use outras formas especiais já definidas.

A título de exemplo vamos definir a forma especial `meu-if` cujo objetivo é simplificar o uso do `cond` quando só existe um teste, um consequente e uma alternativa.

A sintaxe da forma `meu-if` é:

```
(meu-if teste consequente alternativa)
```

A expansão da macro será qualquer coisa da forma:

```
(cond (teste consequente)
      (t alternativa))
```

A definição da macro é:

```
(defmacro meu-if (teste consequente alternativa)
  (list 'cond
        (list teste consequente)
        (list t alternativa)))
```

11.3 – Depuração de macros

Uma vez que a aplicação de uma macro mostra apenas o resultado final, depois de a macro ter sido expandida e a sua expansão avaliada, é necessário um meio auxiliar para visualizarmos se a expansão está a ser feita de modo correto. Para isso o Lisp fornece as funções `macroexpand-1` e `macroexpand` que realizam a expansão da macro uma única vez ou todas as vezes possíveis, respectivamente.

```
> (macroexpand-1 '(meu-if (> 3 4) (+ 2 3) (- 5 3)))
(COND ((> 3 4) (+ 2 3)) (T (- 5 3)))
```

Exercício 61

Implemente a macro `quando`, que recebe um teste e um conjunto de expressões. Esta forma especial avalia o teste e, quando este é verdade, avalia sequencialmente as expressões, devolvendo o valor da última. Se o teste é falso, a forma retorna `nil` sem avaliar mais nada.

Esta macro já existe em Lisp e denomina-se `when`.

11.4 – Caracteres de macro

Geralmente, a parte mais difícil na escrita de uma macro é a determinação das expressões Lisp que quando avaliadas produzem uma nova expressão Lisp que realiza o nosso objetivo. Para simplificar esta tarefa é usual utilizarem-se caracteres especiais que, à semelhança da plica (`quote`) e do cardinal-plica (`function`) são transformados na leitura para outras expressões. Cada um dos caracteres especiais possui uma função Lisp associada que é avaliada quando a linguagem, durante a leitura das expressões, encontra um desses caracteres. Essa função Lisp pode então realizar mais leituras e retornar o que achar mais conveniente.

Estes caracteres especiais são designados caracteres de macro pois eles são transformados (são expandidos) na leitura em outras expressões, um pouco à imagem do que acontecia com as macros. A diferenção está no instante em que a expansão ocorre. Uma macro é expandida em tempo de compilação (ou, em algumas implementações de Lisp, em tempo de execução). Um caracter de macro é expandido na leitura de expressões.

Qualquer caracter pode ser considerado especial, bastando, para isso, usar a função `set-macro-caracter` que recebe um caracter e uma função a aplicar sempre que o caracter for lido. A função a aplicar deve possuir dois parâmetros que receberão, o primeiro, o local donde o Lisp estava a ler (terminal, ficheiro, etc) para que a função possa continuar a leitura do mesmo sítio, e o segundo, o próprio caracter de macro.

Para se indicar um caracter em Lisp é necessário precedê-lo dos caracteres ``#\''. Por exemplo, o caracter `$` é indicado por ``#\$''. Como já é sabido, se não se incluíssem os caracteres ``#\'', o Lisp consideraria o objeto lido como um símbolo e não como um caracter.

Para se compreender a utilização dos caracteres de macro, podemos admitir que não existia o caracter de plica e que pretendíamos defini-lo. A expressão `'ola` representa, como sabemos, `(quote ola)`, logo a função que seria invocada quando se encontrasse a plica necessitaria de ler o objeto que estava após a plica (usando a função `read`) e construiria uma lista com o símbolo `quote` à cabeça e o objeto lido no fim, ou seja:

```
(defun plica (canal-leitura caracter)
  (list (quote quote) (read canal-leitura)))

> (set-macro-character #´ (function plica))
T
> 'ola
OLA
```

A linguagem Lisp possui, previamente definidos, vários caracteres de macro. A plica é um deles, o ponto e vírgula (que representa um comentário) é outro, etc. Alguns dos caracteres de macro são precedidos por um caracter especial, considerado o caracter de despacho, permitindo aumentar o número de possíveis caracteres de macro sem reduzir o número de caracteres normais utilizáveis. O caracter de despacho mais usado é o

cardinal, mas pode ser qualquer outro. Como exemplos destes caracteres de macro com despacho temos o cardinal-plica para indicar funções, o cardinal-barra para indicar caracteres, etc.

De todos os caracteres de macro, aqueles que são particularmente úteis para a escrita de macros são o plica para trás (`--`backquote), a vírgula (`,`--`comma`) e o vírgula-arroba (`,@`--`comma-at`).

O `backquote` indica ao avaliador que não deve avaliar uma expressão exceto quando for indicado em contrário. Quando usado isoladamente, o `backquote` funciona exactamente como o `quote`. No entanto, a sua combinação com o `comma` e o `comma-at` permite simplificar imenso a escrita de expressões complexas. O `comma` só pode ser utilizado numa expressão (uma lista, tipicamente) precedida do `backquote`, e informa o avaliador que a expressão que se segue é para avaliar e o seu resultado inserido na lista. O `comma-at` é idêntico mas o resultado da avaliação tem de ser uma lista cujos elementos são inseridos.

A título de exemplo, temos:

```
> `((+ 1 2) ,(+ 1 2) (list 1 2) ,(list 1 2) ,@(list 1 2))
((+ 1 2) 3 (LIST 1 2) (1 2) 1 2)
```

11.5 – Macros utéis

Exercício 62

Reescreva a macro `quando` usando o `backquote`, o `comma` e o `comma-at`.

Exercício 63

Escreva a macro `a-menos-que`, que recebe um teste e um conjunto de expressões. Esta forma especial avalia o teste e, quando este é falso, avalia sequencialmente as expressões, devolvendo o valor da última. Se o teste é verdade, a forma retorna `nil` sem avaliar mais nada.

Esta macro já existe em Lisp e denomina-se `unless`.

Exercício 64

Escreva uma implementação da macro `meu-cond` usando a forma especial `if`.

Exercício 65

A implementação da macro `meu-cond` anterior implica que a expansão é apenas parcial, i.e., enquanto houver cláusulas, o `meu-cond` expande para outro `meu-cond`. Implemente a macro `meu-cond` de modo a realizar a expansão de uma só vez.

Exercício 66

A macro `seja` implementa a mesma funcionalidade que a forma especial `let`. Como se sabe, o `let` não é mais do que uma macro que expande para uma `lambda`. A idéia será definir a macro `seja` exatamente da mesma forma, i.e., deverá existir uma correspondência entre a expressão:

```
(seja ((x 10) (y 20))
  (+ x y))
```

e a sua expansão:

```
((lambda (x y) (+ x y)) 10 20)
```

Defina a macro `seja` de modo a implementar essa correspondência.

```
> (seja ((x 10) (y 20)) (+ x y))

30
```

Como se disse, esta macro já existe em Lisp e designa-se `let`.

Exercício 67

Escreva a macro `enquanto`, que recebe um teste e um conjunto de expressões. A forma `enquanto` deve avaliar o teste e, caso seja verdade, avaliar sequencialmente as expressões e voltar ao princípio. Caso o teste seja falso, deve terminar com o valor `nil`.

Exercício 68

Escreva a macro `caso`, que recebe uma expressão e um conjunto de pares átomo-expressões. A forma especial `caso` avalia a primeira expressão, e compara o resultado (usando a função `eql`) com cada um dos átomos em sequência. Se um dos átomos emparelhar, são avaliadas as expressões a ele associadas e retornado o valor da última.

Um exemplo de utilização seria:

```
(defun inverso-fact (x)
  (caso x
    (1 (print 'basico) 1)
    (2 (print 'menos-basico) 2)
    (6 (print 'ainda-menos-basico) 3)))

> (inverso-fact 1)
```

```
BASICO

1
```

Esta forma especial já existe em Lisp e denomina-se `case`.

11.6 – Iteradores

Como se pode depreender dos exemplos apresentados, as macros destinam-se essencialmente à criação de açúcar sintático, i.e., de expressões que sejam mais simples de utilizar que outras já existentes. Esta característica torna as macros ferramentas extremamente úteis para criação de tipos abstratos de informação capazes de dar ao utilizador iteradores sobre os objetos desse tipo.

A título de exemplo, vamos considerar a definição de um iterador para os elementos de uma lista. Este iterador deverá ser uma forma especial que recebe um símbolo (uma variável), uma lista e um conjunto de expressões, e itera aquelas expressões com o símbolo ligado a cada elemento da lista. Apresenta-se agora um exemplo da sintaxe da forma especial:

```
(itera-lista (elem (list 1 2 3 4 5))
  (print elem))
```

A sua definição é relativamente simples:

```
(defmacro itera-lista (var-e-lista &rest exprs)
  `(let ((lista ,(cadr var-e-lista))
         (,(car var-e-lista) nil))
     (loop
       (unless lista (return nil))
       (setq ,(car var-e-lista) (car lista)
             lista (cdr lista))
       ,@exprs)))



> (itera-lista (x '(1 2 3)) (print x))
1
2
3

NIL
```

Infelizmente, nem tudo está bem. Reparemos no seguinte exemplo:

```
> (let ((lista '(1 2 3)))
    (itera-lista (x '(4 5 6))
      (print (cons x lista))))
(4 5 6)
(5 6)
```

```
(6)
NIL
```

O problema está no fato de a macro estabelecer uma variável denominada `lista`, que interfere com a variável exterior do segundo exemplo, pois tem o mesmo nome, ficando assim obscurecida. A referência a `lista` feita no corpo da macro refere-se assim à variável interna da macro, e não à que seria desejável. Nesta situação diz-se que a macro capturou variáveis.

A solução para este problema está na utilização de variáveis que não possam interferir de modo algum com outras já existentes. Um remédio possível será criar as variáveis necessárias às macros com nomes estranhos, com pouca probabilidade de serem usados pelo utilizador da macro, como por exemplo, `%%%$$$lista$$$%%%`. No entanto esta solução não é perfeita. O melhor a fazer é usar novos símbolos que não se possam confundir com os já existentes. A função `gensym` produz um símbolo novo e único de cada vez que é chamada, sendo ideal para resolver estas dificuldades.

Exercício 69

Defina a macro `itera-lista` usando a referida função `gensym` para proteger as variáveis do utilizador de uma captura indevida.

```
> (let ((lista '(1 2 3)))
    (itera-lista (x '(4 5 6))
      (print (cons x lista))))
(4 1 2 3)
(5 1 2 3)
(6 1 2 3)
NIL
```

Esta forma especial já existe em Lisp e denomina-se `dolist`.

Exercício 70

Defina de novo a forma especial `itera-lista`, mas recorrendo desta vez à abordagem da programação funcional, i.e., sem utilizar formas especiais para ciclos nem atribuição de valores a variáveis.

Exercício 71

A forma especial `caso` definida anteriormente sofria do mesmo problema do `itera-lista`, pois a variável usada para guardar o valor temporário pode obscurecer variáveis idênticas declaradas exteriormente. Redefina a macro de forma a evitar esse perigo.

11.7 – Fichas

Como vimos, uma ficha (no sentido da linguagem Pascal) não é mais do que um tipo aglomerado. Uma ficha possui um nome e é composta por um conjunto de campos, cada um com um nome distinto. Cada elemento de um dado tipo de ficha corresponde a um conjunto de valores apropriados para cada campo desse tipo. A utilização de fichas é de tal modo simples que todas as linguagens de programação possuem capacidades próprias para lidar com elas. Um dado tipo de ficha não necessita mais do que um construtor para os elementos desse tipo, um seletor e um modificador para cada um dos campos da ficha e, possivelmente, um reconhecedor de fichas de um dado tipo.

Dadas as capacidades da linguagem Lisp em aglomerar facilmente objetos, a implementação de fichas é uma tarefa de tal modo simples que é bastante fácil automatizá-la.

Vimos na descrição do tipo aglomerado `automóvel` que ele era definido por um construtor:

```
(defun novo-automovel (&key marca modelo portas) ...)
```

e pelos selectores:

```
(defun automovel-marca (automovel) ...)

(defun automovel-modelo (automovel) ...)

(defun automovel-portas (automovel) ...)
```

e ainda pelos modificadores:

```
(defun muda-automovel-marca! (automovel nova-marca) ...)

(defun muda-automovel-modelo! (automovel novo-modelo) ...)

(defun muda-automovel-portas! (automovel novo-portas) ...)
```

Vamos também incluir um reconhecedor de automóveis muito útil para distinguirmos os diversos tipos de fichas:

```
(defun automovel? (obj) ...)
```

Repare-se que o conjunto de funções que apresentamos constituem um modelo para a definição de fichas. A ficha `automóvel` possui um construtor criado através da concatenação da palavra ``novo'' ao nome da ficha. Cada seletor é dado pela concatenação do nome da ficha ao nome do campo a que ele diz respeito. Cada modificador é dado pela concatenação da palavra ``muda'' ao nome do seletor correspondente e terminado com a letra ``!''. O reconhecedor é dado pelo nome da ficha terminado com a letra ``?''. Qualquer outra ficha seria definida de forma idêntica, pelo que podemos fazer uma abstração da definição de fichas, criando uma macro que encapsule a definição de todas aquelas funções.

Um dos problemas que temos de resolver é a implementação das fichas. Vimos no exemplo do automóvel que o podíamos implementar como uma lista com os valores dos vários campos. Infelizmente, aquela implementação não nos permite distinguir o tipo de registo automóvel de outros tipos de registo, pelo que temos de modificar a implementação de modo a isso ser possível. Para minimizar as alterações, podemos considerar que cada registo é implementado por uma lista cujo primeiro elemento é o nome da ficha e cujos restantes elementos são os valores dos campos da ficha. Isto permite manter a mesma lógica de acesso e modificação dos campos desde que os índices desses campos na lista sejam incrementados de uma unidade.

Assim sendo, o construtor ficará qualquer coisa do gênero:

```
(defun novo-automovel (&key marca modelo portas)
  (list 'automovel marca modelo portas))
```

Um dado selector será :

```
(defun automovel-marca (automovel)
  (nth 1 automovel))
```

Um modificador será:

```
(defun muda-automovel-marca! (automovel nova-marca)
  (muda-n-esimo! 1 automovel nova-marca))
```

O reconhecedor de automóveis ficará:

```
(defun automovel? (obj)
  (and (listp obj) (eql (first obj) 'automovel)))
```

Podemos agora definir uma macro designada `ficha`, cuja utilização será da seguinte forma:

```
(ficha automovel
  marca modelo portas)
```

A expansão da macro deverá ser qualquer coisa da forma:

```
(progn
 (defun novo-automovel (&key marca modelo portas) ...)
 (defun automovel-marca (automovel) ...)
 (defun automovel-modelo (automovel) ...)
 (defun automovel-portas (automovel) ...)
 (defun muda-automovel-marca! (automovel nova-marca) ...)
 (defun muda-automovel-modelo! (automovel novo-modelo) ...)
 (defun muda-automovel-portas! (automovel novo-portas) ...)
 (defun automovel? (obj) ...)
 'carro)
```

A definição da macro é, assim, relativamente simples:

```
(defmacro ficha (nome &rest campos)
  `(progn
    ,(construtor-ficha nome campos)
    ,@(selectores-ficha nome campos)
    ,@(modificadores-ficha nome campos)
    ,(reconhecedor-ficha nome)
    ',nome))
```

Para definirmos as várias operações vamos necessitar de concatenar símbolos. Para isso, é necessário criar um símbolo, através da função `intern`, cujo nome seja a concatenação, através da função `concatenate`, dos nomes dos símbolos, obtidos mapeando a função `string` nesses símbolos, i.e.:

```
(defun junta-nomes (&rest nomes)
  (intern (apply #'concatenate
                 'string
                 (mapcar #'string nomes))))
```

Com a ajuda desta função, já podemos definir as restantes funções da macro.

```
(defun construtor-ficha (nome campos)
  `(defun ,(junta-nomes 'novo- nome) (&key ,@campos)
     (list ',nome ,@campos)))

(defun selectores-ficha (nome campos)
  (mapcar
    #'(lambda (campo)
        `(defun ,(junta-nomes nome '- campo) (,nome)
           (nth ,(1+ (position campo campos)) ,nome)))
    campos))

(defun modificadores-ficha (nome campos)
  (mapcar
    #'(lambda (campo)
        `(defun ,(junta-nomes 'muda- nome '- campo '!)
                (,nome novo)
           (muda-n-esimo! ,(1+ (position campo campos))
                          ,nome novo)))
    campos))

(defun reconhecedor-ficha (nome)
  `(defun ,(junta-nomes nome '?) (obj)
     (and (listp obj) (eql (first obj) ',nome))))
```

Podemos agora experimentar a macro e visualizar os resultados:

```
> (macroexpand-1 '(ficha automovel marca modelo portas)))
(PROGN
  (DEFUN NOVO-AUTOMOVEL (&KEY MARCA MODELO PORTAS)
    (LIST 'AUTOMOVEL MARCA MODELO PORTAS))
  (DEFUN AUTOMOVEL-MARCA (AUTOMOVEL)
    (NTH 1 AUTOMOVEL))
  (DEFUN AUTOMOVEL-MODELO (AUTOMOVEL)
    (NTH 2 AUTOMOVEL))
```

```
  (DEFUN AUTOMOVEL-PORTAS (AUTOMOVEL)
    (NTH 3 AUTOMOVEL))
  (DEFUN MUDA-AUTOMOVEL-MARCA! (AUTOMOVEL NOVO)
    (MUDA-N-ESIMO! 1 AUTOMOVEL NOVO))
  (DEFUN MUDA-AUTOMOVEL-MODELO! (AUTOMOVEL NOVO)
    (MUDA-N-ESIMO! 2 AUTOMOVEL NOVO))
  (DEFUN MUDA-AUTOMOVEL-PORTAS! (AUTOMOVEL NOVO)
    (MUDA-N-ESIMO! 3 AUTOMOVEL NOVO))
  (DEFUN AUTOMOVEL? (OBJ)
    (AND (LISTP OBJ) (EQL (first OBJ) 'AUTOMOVEL)))
  'AUTOMOVEL)
```

Exercício 71

Um dos melhoramentos que seria interessante incluir na definição de fichas seria a inclusão de valores por omissão para qualquer campo. Poderíamos indicar que um automóvel possui geralmente quatro portas da seguinte forma:

```
(ficha automovel
  marca
  modelo
  (portas 4))
```

Assim, se se criasse um automóvel sem especificar o número de portas, ele seria de 4. Implemente essa alteração.

```
> (macroexpand-1 '(ficha automovel marca modelo (portas 4))))
(PROGN
  (DEFUN NOVO-AUTOMOVEL (&KEY MARCA MODELO (PORTAS 4))
    (LIST 'AUTOMOVEL MARCA MODELO PORTAS))
  (DEFUN AUTOMOVEL-MARCA (AUTOMOVEL)
    (NTH 1 AUTOMOVEL))
  ...
```

Esta forma de criação de fichas já existe em Common Lisp através da macro `defstruct`.

Referências

Steelem, Guy L..**Commom Lisp the Language**. Ed. Digital Press, 1990. dispoível online: http://www.supelec.fr/docs/cltl/cltl2.html.