

Escola Superior de Tecnologia
Instituto Politécnico de Castelo Branco



Aulas Práticas de Fundamentos de Inteligência Artificial

Arindo Silva

Ana Paula Neves

Introdução ao Lisp

Listas, Átomos e Funções Simples

Como Começar

O Lisp é uma linguagem interpretada. Isto quer dizer que, ao contrário do C ou PASCAL, o interpretador pode processar e responder directamente a programas escritos pelo programador, sem necessidade de os compilar para código máquina.

Ao iniciar um interpretador de Lisp, este apresenta-lhe um *prompt*:

```
>
```

Caso escreva uma expressão em frente ao *prompt* seguida de *enter* o interpretador avalia a expressão e escreve o resultado, apresentando de novo o *prompt*:

```
> (+ 2 4)  
6  
>
```

Neste exemplo o interpretador leu a expressão `(+ 2 4)`, avaliou-a tendo obtido o resultado `6` que escreveu na consola. Podemos dizer que o interpretador funciona num ciclo ler-avaliar-escrever.

1. Avalie no seu interpretador as seguintes expressões:

```
(+ 1 2 3 4)
```

```
(* 3 7)
```

```
(/ (- 7 1) (- 4 2))
```

Tipos de Dados Básicos

Existem em Lisp dois tipos básicos de dados: átomos e listas. Estes dois tipos de dados são mutuamente exclusivos, exceptuando a lista vazia, que pode ser representada por `()` ou por `nil`, e que é simultaneamente uma lista e um átomo.

Os átomos são representados por sequências de caracteres. As listas são construídas recursivamente a partir de átomos, o que quer dizer que uma lista pode conter vários átomos e mesmo outras listas.

Exemplos de átomos:

```
g
123
beatriz
afs71
```

Exemplos de listas:

```
()
(g)
(g 123 beatriz)
(g (123 beatriz) ())
```

1. Quantos elementos tem a última lista nos exemplos anteriores?
2. E a lista ((um dois))?

Tanto os átomos como as listas constituem expressões válidas para o interpretador de Lisp avaliar. Nos pontos seguintes iremos descrever as regras simples que o interpretador utiliza para avaliar átomos e listas.

Avaliação de Átomos

Quando o interpretador do Lisp encontra um átomo tenta imediatamente encontrar um valor para esse átomo. A generalidade dos átomos produzirá um erro ao ser avaliada caso não lhe tenha sido previamente atribuído um valor. Podemos utilizar `setq` para atribuir um valor a um átomo:

```
> (setq ano 2002)
2002
```

O interpretador responde com o último valor avaliado, neste caso `2002`. Se pedirmos agora ao interpretador para avaliar o átomo `ano` ele responde com o valor que lhe foi anteriormente atribuído:

```
> ano
2002
```

Como pode ver os átomos podem facilmente ser encarados como variáveis, na medida em que se lhes pode atribuir um valor ou aceder a um valor já atribuído (bastando obrigar à sua avaliação). Como não existe qualquer restrição em relação aos valores que podem ser atribuídos a um átomo, podemos afirmar que em Lisp as variáveis não têm tipo nem necessitam de ser declaradas formalmente.

Números como `2`, `32.456`, `24` constituem um tipo especial de átomos tendo sido predefinido que a sua avaliação resultaria no próprio número. Além dos números existem

dois outros átomos especiais `t` e `nil` (verdadeiro e falso, respectivamente) que também são avaliados da mesma forma. O interpretador considera ainda que `nil` é idêntico à lista vazia `()`.

```
> 2
2
> 1.5
1.5
> t
T
> nil
NIL
> ()
NIL
> x
;; Error: Unbound variable X in #<function 1 #x970FC0>
```

Esta última mensagem de erro é dada pelo interpretador sempre que se tenta a avaliação de um átomo ao qual não foi ainda atribuído um valor. Isto é um erro extremamente comum e resolve-se fazendo a inicialização atempada do átomo (por exemplo usando o `setq`). Também constitui um erro tentar atribuir valores a átomos especiais: números, o `nil` e `t`.

Avaliação de Listas

O interpretador considera que qualquer lista é constituída pelo nome de uma função seguido pelos parâmetros para essa função. Por exemplo `(+ 3 4 5 6)` corresponde à chamada da função `+` com os argumentos 3, 4, 5 e 6. Se pedirmos ao interpretador para avaliar a expressão obtemos

```
> (+ 3 4 5 6)
18
```

O interpretador avaliou primeiro cada um dos argumentos por ordem. Como se tratam de números são avaliados para o seu próprio valor. Os parâmetros avaliados são passados então à função `+` que calcula o valor resultante. Os parâmetros podem não ser números, mas outros átomos ou mesmo listas. Nesses casos as regras de avaliação vão sendo aplicadas até todos os parâmetros estarem avaliados (o que pode implicar avaliar outras listas) após o que os resultados das avaliações são passados à função.

Caso tente usar um nome de função desconhecido do interpretador mais uma vez ele irá dar uma mensagem de erro:

```
(qualquer-coisa 2 3)
;; Error: Call to undefined function QUALQUER-COISA in #<function 0 #xF39428>
```

Este é também um dos erros comuns em Lisp, devendo haver um cuidado especial para não utilizar, como primeiro elemento de uma lista que irá ser avaliada, um símbolo que não é nome de função.

```
> (setq x 2)
2
> (setq y 4)
4
```

1. Após as instruções executadas acima qual o resultado da avaliação das seguintes expressões:

```
( / 8 2 )  
( * x 4 )  
( - x y )  
( + x y z )  
( r x y )  
( - ( * x 2 ) 3 )  
( / ( + y 4 ) ( + x 2 ) )
```

Algumas Funções Simples

Como qualquer linguagem o Lisp possui já um conjunto de funções implementadas, chamadas funções primitivas, incluindo as habituais funções matemáticas: +, -, *, /, exp, expt, log, sqrt, sin, cos, tan, etc.

Mais importantes para o Lisp são as funções que permitem a construção de listas e o acesso aos seus elementos.

Funções de construção

As principais funções para construção de listas em Lisp são o `cons`, o `list` e o `append`.

A função `cons` permite adicionar um novo elemento à cabeça de uma lista. Recebe dois argumentos: o elemento a adicionar e uma lista onde o colocar. Devolve uma nova lista, com o novo elemento à cabeça.

```
>(cons 1 nil)  
(1)
```

Neste exemplo o `cons` recebe como argumentos o número 1 e a lista vazia. O resultado é uma nova lista com apenas um elemento.

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

Este segundo exemplo é mais interessante. Neste caso o primeiro argumento continua a ser um número a adicionar à lista. Mas o segundo argumento é uma chamada a outra função `cons`. O interpretador avalia esta segunda chamada e o resultado é uma lista com um único elemento (o número 2). Esta é passada como segundo argumento ao primeiro `cons` que devolve então a lista (1 2).

1. Avalie as seguintes expressões:

a) `(cons 3 (cons nil nil))`

b) `(cons nil (cons 2 (cons nil nil)))`

2. Escreva utilizando `cons` uma expressão que devolva `(1 2 3)`.

3. Qual o resultado da expressão `(cons a (b c))`?

4. Confirme no interpretador se a sua resposta é correcta.

Ao tentar avaliar a expressão do exercício anterior (3) no interpretador deverá obter um erro:

```
;; Error: Unbound variable A in #<function 0 #xF3E1B4>
```

Isto acontece porque o interpretador tenta avaliar o símbolo `a` para obter o primeiro argumento para o `cons`. Como `a` não tem valor atribuído surge a mensagem de erro.

1. Qual o resultado da expressão `(cons 1 (b 2))`?

2. Confirme no interpretador se a sua resposta é correcta.

Neste exercício o resultado seria um novo erro:

```
;; Error: Call to undefined function B in #<function 0 #xF38530>
```

Desta vez o interpretador consegue encontrar um valor para o primeiro argumento (já que este é um número) mas ao chegar ao segundo (que é uma lista) avalia-o como uma chamada à função `b` com o argumento `2`. Como não existe uma função `b` a avaliação falha, surgindo a mensagem de erro correspondente.

O quote

Para evitar este problema podemos utilizar o `quote` (representado por uma plica simples). Quando colocado à frente de algo evita que o interpretador avalie esse algo. Por exemplo:

```
> `a  
A
```

```
> `(a b)  
(A B)
```

Podemos agora tentar fazer `(cons `a (b c))` e neste caso o interpretador avança para a avaliação do segundo argumento. `(b c)` é avaliado como sendo uma chamada à

função `b` mas como não existe uma função com este nome daqui resulta um novo erro. A expressão correcta seria:

```
> (cons 'a '(b c))  
(A B C)
```

Note-se que o `quote` quando colocado à frente de uma lista impede a avaliação de todos os elementos dessa lista, não sendo por isso necessário utilizar um `quote` para cada elemento.

Na realidade a utilização da `plica` é apenas uma abreviatura da utilização da função `quote` do Lisp. O interpretador expande expressões do tipo `'a` para `(quote a)`.

1. Escreva, utilizando o `cons`, três expressões diferentes que retornem `(a b c)`.
2. Indique qual seria o resultado da avaliação de cada uma das seguintes expressões

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

Outras funções que permitem criar listas são o `list` e o `append`. O `list` recebe um número variável de argumentos e devolve uma nova lista em que os elementos são o resultado da avaliação de cada argumento (lembre-se que deve utilizar o `quote` quando quer evitar a avaliação):

```
> (list 1 2 3)  
(1 2 3)  
  
> (list (+ 1 3) 4)  
(4 4)  
  
> (list 'a 'b 'c)  
(A B C)  
  
> (list 'a '(b c))  
(A (B C))
```

1. Utilizando apenas o `list` escreva, para cada uma das listas seguintes, uma expressão cuja avaliação resulte na lista em causa.

```
a) ((1 2) ((2 3)))  
b) ((nil) (b d))  
c) (a (b (c nil)))
```

O `append` recebe um número variável de argumentos as quais deverão ser listas. O resultado devolvido consiste numa lista resultante da concatenação das listas recebidas como argumentos:

```
> (append '(a b) '(c d))
(A B C D)
```

Funções de Acesso

As funções mais importantes de acesso a listas são `first` e `rest` (também conhecidas por `car` e `cdr`). Podemos utilizar `first` para obter o primeiro elemento de uma lista (a cabeça da lista):

```
> (first '(1 2 3))
1
```

```
> (first '((1 2) 3))
(1 2)
```

`rest` recebe uma lista e devolve uma nova lista com todos os elementos dessa lista excepto o primeiro:

```
> (rest '(1 2 3))
(2 3)
```

```
> (rest '((1 2) 3))
(3)
```

`first` e `rest` podem ser utilizados de forma encadeada para devolver qualquer elemento de uma lista. Por exemplo, para aceder ao segundo elemento de `(a b c)` utilizamos

```
> (first (rest '(a b c)))
B
```

1. Para cada uma das seguintes expressões indique qual seria o resultado da sua avaliação:

a) `(rest '(a))`

b) `(first '((a b) c))`

c) `(first (first '((a b))))`

d) `(rest (first (rest '(a (b c) d))))`

2. Utilizando `first` e `rest` escreva uma expressão que devolva o quarto elemento da

lista `(a b c d e)`.

Efeito não Destrutivo das Funções em Lisp

```
> (setq x '(a b c))  
(A B C)
```

Após a avaliação da expressão acima o que é que acontece a `x` se fizermos

```
> (cons 'y x)  
(Y A B C)
```

Podemos pensar que `x` conteria a lista resultante `(Y A B C)` mas isso não é verdade, de facto se tentarmos a avaliação de `x` obtemos:

```
> x  
(A B C)
```

Na realidade, as funções do Lisp raramente alteram os argumentos que recebem e isto é uma característica que devemos tentar manter nas nossas próprias funções. Se pretendemos guardar a nova lista em `x` devemos fazer:

```
> (setq x (cons 'y x))  
(Y A B C)
```

Agora sim, `x` conterà a lista modificada:

```
> x  
(Y A B C)
```

A utilização de `setq`, `cons`, `first` e `rest` é suficiente para manipular completamente as listas em Lisp. As funções `list` e `append`, embora por vezes sejam muito úteis, não são essenciais à manipulação de listas.

Outras Funções

Outras funções úteis que podemos utilizar com listas são:

`length` – recebe uma lista e devolve o número de elementos da mesma:

```
> (length '(a b c d))  
4
```

`nth` – recebe um inteiro e uma lista e devolve o elemento da lista que se encontre na posição representada pelo inteiro (a função é indexada a 0):

```
> (nth 2 '(a b c d))  
C
```

`listp` e `atomp` – a primeira devolve `t` quando o seu argumento é uma lista e `nil` em qualquer outro caso; a segunda devolve `t` quando o argumento é um átomo e `nil` em qualquer outro caso. Funções que devolvem sempre `t` ou `nil` são chamadas predicados.

Outras funções potencialmente úteis (ver help) são: second, third, ..., tenth, nthcar, nthcdr, last, butlast, nbutlast, reverse, ...

Atribuição com setf

Já vimos que setq é uma função útil que nos permite modificar o valor de variáveis. Com setq podemos por exemplo fazer

```
> (setq numero 25)
25
> numero
25
```

Mas o que é que acontece se tentarmos alterar, por exemplo, o terceiro elemento de uma lista?

```
> (setq lista '(a b c d))
(A B C D)
> (setq (first (rest (rest lista))) 1)
;; Error: Compilation error: SETQ malformed: (SETQ (FIRST (REST (REST LISTA))) 1)
```

O setq não avalia o primeiro argumento e por isso temos um erro. A solução é utilizar o setf em vez de setq:

```
> (setf (first (rest (rest lista))) 1)
1
> lista
(A B 1 D)
```

Note-se que ao utilizarmos o setf desta forma a lista original é alterada!

1. Em cada uma das seguintes expressões substitua as ocorrências de x e y pela função de atribuição mais apropriada: setq ou setf.

- a) (**x** a 1)
- b) (**x** b (**y** (first '(a b c))1))
- c) (**x** (first (rest (list 'a 'b 'c))) (length '(a b)))
- d) (**x** (first '(a b c)) (**y** a (length '(2 3 4))))

Exercícios

1. Avalie as seguintes expressões:
- a) (first '(((a)) (b c d e)))
 - b) (rest '((((f))))))
 - c) (first '(rest (a b c)))

- d) (first '(rest (rest (a b c))))
- e) (cons '(my life as) '(a dog))
- f) (append '(my life as) '(a dog))
- g) (list '(my life as) '(a dog))
- h) (cons (rest nil) (first nil))
- i) (abs (- (length (rest '(((a b) (c d))))) 5))
- j) (reverse (cons '(rest (reverse '(its gut na mur ta give captin)))))

2. Usando o first e o rest acesse ao átomo "jim" presente nas seguintes listas:

- a) (he is dead jim)
- b) (captain (((jim) kirk)))
- c) (((((spock) asked) jim) if) he was all right)
- d) (after (looking at the (lizard man) (((jim))) asked for warp 9))

3. Qual o valor devolvido pelo interpretador na avaliação de cada uma das seguintes expressões. Assuma que elas são avaliadas pela ordem em que são apresentadas.

- a) (setf trek '(picard riker laforge worf))
- b) (cons 'data trek)
- c) trek
- d) (length (cons 'troi trek))
- e) (setf trek (cons 'data trek))
- f) (length (cons 'troi trek))

4. Atendendo à seguinte definição:

```
(setf mylist '((bush broccoli) (nixon watergate)
              (letterman (viewer mail))
              (you are no jack kennedy)
              (and please) (scorsese (robert deniro))))
```

construa as seguintes listas usando as funções apresentadas.

- a) (no broccoli please)
- b) ((scorsese and deniro) are no robert kennedy)
- c) (watergate and no viewer)
- d) (bush nixon kennedy)
- e) ((bush broccoli) (nixon watergate) (letterman mail))