Ficha de Laboratório Nº 3: Definição de Funções e Estruturas de Controlo

Inteligência Artificial - Escola Superior de Tecnologia de Setúbal

Prof. Joaquim Filipe
Eng. Filipe Mariano

Objetivos da Ficha

- Praticar a sintaxe para a criação de funções em Lisp
- Estruturas de controlo
- Exercícios de funções em Lisp (sem recursividade)

1. Definição de Função

Como analisado nos laboratórios anteriores, uma função é definida da seguinte forma:

```
(defun <nome da função> (<args>)
    “<documentação>”
    <corpo da função>
)
```

Notas:

- É possível definir funções sem argumentos, escrevendo apenas `()`.
- O Lisp possibilita argumentos opcionais, múltiplos e outros tipos de argumentos que serão abordados no Laboratório 5.
- A string de documentação descreve o objetivo da função. Pode ser obtida através da função `documentation`.
- O corpo de uma função pode conter várias instruções, sendo que a última será a que é retornada como valor da função. Como o intuito é programar segundo uma perspetiva de Lisp "puro" este aspeto não será tido em conta, visto que evitaremos a sequenciação de instruções.

2. Estruturas de Controlo

As estruturas de controlo são elementos de um programa que controlam a execução de instruções. Dividem-se principalmente em duas categorias:

- **Seleção:** Uma estrutura de seleção permite condicionar a execução de uma ou mais instruções em função de determinada condição booleana `if`.
- **Repetição:** Uma estrutura de repetição permite executar um bloco de instruções enquanto uma determinada condição é verdadeira. **Ao longo do semestre iremos evitar a utilização de estruturas de repetição, utilizando apenas funções recursivas ou meta-funções para atingir os mesmos objetivos (Laboratórios 4 e 5).**

Sintaxe de estrutura de seleção em Lisp:

A estrutura de seleção utilizada no Lisp "puro" é o `cond`.

```
(cond   <cláusula 1>        ;if condição 1 then
        <cláusula 2>        ;else if condição 2 then
        …
        <cláusula n> )      ;else

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

A utilização da Macro `if` também é muito comum e pode ser utilizada através da seguinte sintaxe:

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

Exemplos de estrutura de seleção em Lisp:

Com `cond`:

```
(defun entre-intervalo-cond (x)
    (cond
        ((and (numberp x) (> x 10) (< x 20)) (format t "~d é maior que 10 e menor
que 20" x))
        (t (format t "não é um número entre 10 e 20"))
    )
)
```

Com `if`:

```
(defun entre-intervalo-if (x)
    (if (and (numberp x) (> x 10) (< x 20))
        (format t "~d é maior que 10 e menor que 20" x) ; verdadeiro
        (format t "~d não é um número entre 10 e 20" x) ; falso
    )
)
```

O `numberp` é um predicado pré-definido no Lisp, que permite testar se nos encontramos perante um número ou não. A função `format`, utilizada no exemplo anterior, representa uma função de escrita no ecrã semelhante à de `printf` existente na linguagem C. O primeiro argumento `t` significa serve para indicar o destino de impressão, em que `t` indica que será o *standard output*. Ao colocar `nil`, o que é retornado é no formato de string.

Notas:
Quando apenas existe uma condição a testar, não existem vantagens na utilização do `cond` ao invés do `if`,

contudo quando existem diversas condições que necessitam de ser testadas, o `cond` torna o código muito mais legível.

3. Exercícios sobre Funções (não recursivas)

1. `entre-intervalo`: Altere a função definida anteriormente para passar a receber 2 argumentos, que são um número e uma lista. A lista deverá ter sempre 2 elementos, que representam o intervalo ao qual se pretende testar se o número pertence.

```
CL-USER > (entre-intervalo 5 '(0 10))
5 é maior que 0 e menor que 10
```

2. `max-3`: Sem recorrer à função `max` pré-definida no Lisp, use os operadores lógicos e condições que achar necessárias para encontrar o maior número entre 3 número passados como argumento.

```
CL-USER > (max-3 7 3 6)
7
```

3. `restop`: Recebe três argumentos, nomeadamente o dividendo, divisor e o resto. Verifica se o resto da divisão entre os primeiros 2 números é igual ao valor passado como argumento, retornando `T` (verdadeiro) ou `NIL` (falso). Caso o divisor seja 0 deve-se retornar `NIL`.

```
CL-USER > (restop 10 5 0)
T
```

4. `aprovadop`: De uma lista de 4 notas passada como argumento, em que a primeira nota é de Matemática, a segunda de História, a terceira de Ciências e a última de Português, pretende-se saber se o aluno está aprovado. Um aluno aprova, se as notas de Matemática e Português forem superiores ou iguais a 9.5 ou se a média de todas as notas for superior ou igual a 9.5. A função representa um predicado, por isso deverá retornar `T` (verdadeiro) ou `NIL` (falso).

```
CL-USER > (aprovadop '(13 15.6 5.5 7))
T
```

5. `nota-final`: Recebe duas listas como argumento, em que a primeira contém as notas (lista com 3 elementos) e a segunda as respetivas ponderações. Deve-se calcular as notas mediante as ponderações, de modo a produzir a nota final. A lista com as ponderações somadas deve dar 100 e não devem haver notas inferiores a 0 ou superiores a 20. Exemplo: `(10 12 15) (25 25 50)` significa que as notas foram 10, 12 e 15, e que a primeira e a segunda correspondem a 25% da nota final e a terceira a 50% da nota final.

```

```

```
CL-USER > (nota-final '(10 12 15) '(25 25 50))
13
```

6. `produto-somas`: Recebe duas listas de números com 3 elementos cada. Esta função adiciona os seus membros e devolve o produto dessas adições.

```
CL-USER > (produto-somas '(1 2 3) '(2 2 2))
60
```

7. `junta-listas-tamanho-igual`: Recebe duas listas como argumento e caso tenham o mesmo tamanho junta-as numa só lista (recorrendo à função `append`). Se as listas não tiverem o mesmo tamanho, a lista a ser retornada deverá ser a de maior tamanho.

```
CL-USER > (junta-listas-tamanho-igual '(1 3 4) '(5 3 2))
(1 3 4 5 3 2)

CL-USER > (junta-listas-tamanho-igual '(1 3 4) '(5 3 2 1))
(5 3 2 1)
```

8. `dois-ultimos-elementos`: Recebe uma lista e devolve uma lista com os dois últimos elementos presentes nela. Utilize a função `reverse` ou `nth` para o auxiliar a ir buscar os dois últimos elementos. Deve testar se a lista está vazia (recorrendo à função `null`) e se o tamanho da lista é superior a 2.

```
CL-USER > (dois-ultimos-elementos '(1 2 3 4 5 6 7))
(6 7)
```

9. `palindromop`: Verificar se o inverso de uma lista é igual à própria lista passada como argumento.

```
CL-USER > (palindromop '(1 2 3 2 1))
T
```

10. `criar-pares`: Crie uma função que recebe duas listas de 3 elementos e devolve uma lista com o conjunto de pares criado a partir das duas listas. Se as listas não forem do mesmo tamanho ou estiverem vazias, devolva `NIL`.

```
CL-USER > (criar-pares '(1 2 3) '(4 5 6))
((1 4) (2 5) (3 6))
```

11. `verifica-pares`: Recebe uma lista de 4 elementos, e devolve uma lista de igual tamanho com `T` caso o elemento seja par e `NIL` caso não seja.

```
CL-USER > (verifica-pares '(1 2 3 4))
(NIL T NIL T)
```

12. `rodar`: Permite fazer a rotação à esquerda ou à direita de uma lista de 4 elementos.

```
CL-USER > (rodar '(1 2 3 4) 'esq)
(4 1 2 3)

CL-USER > (rodar '(1 2 3 4) 'dir)
(2 3 4 1)
```

13. `rodar-listas`: Implemente uma variante da função anterior que permita fazer as rotações para listas de tamanho arbitrário.