Ficha de laboratório Nº 2: Tipos de Dados, Debug, Manuais e Organização do Código

Inteligência Artificial - Escola Superior de Tecnologia de Setúbal

Prof. Joaquim Filipe
Eng. Filipe Mariano

Objetivos da ficha

* Tipos de Dados
* Funções Simples (não recursivas) recorrendo ao Editor do LispWorks
* _Debug_ no LispWorks
* Manual Técnico e de Utilizador (escrita em _Markdown_)
* Organização do Código

1. Tipos de Dados

O LispWorks é um IDE que permite a programação num dos dialetos do Lisp denominado de [_Common Lisp_](https://en.wikipedia.org/wiki/Common_Lisp).

Os nomes dos diferentes tipos são um pouco diferentes dos que habitualmente são conhecidos de outras linguagens e que podem ser observados recorrendo à função `type-of`. Para observar alguns desses tipos, experimentar as seguintes expressões no _listener_:

```lisp
(type-of 100)
(type-of 100000000000)
(type-of 24.589)
(type-of nil)
(type-of ())
(type-of 103.7)
(type-of 4e+5)
(type-of 12/5)
(type-of "lisp")
```

Dois dos tipos de dados que o Lisp possui que não são comuns de observar noutras linguagens são os **símbolos** e as **listas**.

O símbolo é um tipo de dados que pode não ser avaliado, por isso uma das formas frequentemente utilizadas é recorrendo à função `quote` ou à abreviação a essa função presente na linguagem que é a ' (pelica). A utilização da ' ou da função `quote` previne que a expressão abrangida seja avaliada, assim como todas as expressões nela incluídas.

Ao não ser avaliada é retornada a própria lista, enquanto que se não tiver a ' ou a função `quote` é retornado o resultado da avaliação da expressão.

Exemplos:
```lisp
CL-USER > (list 'car (+ 2 2) "exemplo")
(CAR 4 "exemplo")
```

Neste primeiro exemplo, o `car` está marcado para não ser avaliado (símbolo) mas a lista `(+ 2 2)` será avaliada e será tratada como uma chamada à função `+` (soma) com dois argumentos.
```lisp
CL-USER > (list '(+ 2 1) (+ 2 2 1))
((+ 2 1) 5)
```

Neste segundo exemplo, o `(+ 2 1)` não será avaliado enquanto a lista `(+ 2 2 1)` será avaliada e será tratada como uma chamada à função `+` (soma) com três argumentos.
```lisp
CL-USER > '(+ (- 1 2) 3)
(+ (- 1 2) 3)
```

Neste terceiro exemplo, como tudo está abrangido pela ' não será avaliada nem essa lista nem as

nela incluídas.
```lisp
CL-USER > ()
NIL
CL-USER > NIL
NIL
```

Neste quarto exemplo, ilustra duas formas diferentes de representar uma lista vazia, utilizando parênteses vazios ou o símbolo `NIL`. Independetemente da forma como a representamos, ao ser avaliada será listada como `NIL`.

Nota: Através dos exemplos demonstrados anteriormente é possível verificar uma das principais características do Lisp. Os programas em Lisp são expressos através de listas, permitindo a criação de programas que possibilitam a geração de código Lisp.

2. Funções

Por norma as funções, independentemente da linguagem, são criadas para realizar uma tarefa específica. Tal como observado no laboratório anterior as funções são criadas através da macro `defun`.

No laboratório anterior foi possível verificar a diferença entre a definição de uma função no _Editor_ em detrimento da definição da função no _Listener_ que ao fechar o LispWorks não poderá ser mais recuperada.

O _Listener_ é prático para digitar expressões curtas de Lisp, enquanto o _Editor_ é mais eficiente para escrever código Lisp extenso e guardá-lo em ficheiros.

Exemplo de utilização do Editor

1. Descarregar e abrir o ficheiro disponível no Moodle, chamado laboratorio2.lisp e salve este ficheiro no seu computador.
2. Abrir o ficheiro no IDE LispWorks.
3. De seguida, selecione no menu Buffer > Compile ou clicar com o rato no ícone da figura 1 existente na barra de ferramentas da janela do editor. Irá compilar qualquer função definida no editor (O comando **Compile Buffer** compila as funções definidas em todo o ficheiro, enquanto o comando **Compile** compila as últimas alterações do ficheiro).

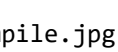
Figura 1: O ícone da ferramenta compilação.

4. Ao compilar o editor passa automaticamente para a aba de Output para mostrar o resultado da compilação. Basta clicar depois na tecla espaço para voltar à visualização.
5. Agora que compilou a função `soma-3`, pode testá-la no _listener_. Insira a expressão `(soma-3 3 2 4)` neste terminal e pressione a tecla Enter e observe o resultado.

Exercícios
1. Escreva uma função `notas-dos-alunos` que retorna 3 listas de 4 notas `(15.5 15 8.25 13)`, `(17.5 11 9 13.25)` e `(11.75 0 0 16)` obtidas por um conjunto de alunos.
```lisp
CL-USER > (notas-dos-alunos)
((15.5 15 8.25 13) (17.5 11 9 13.25) (11.75 0 0 16))
```

2. Escreva uma função `notas-do-primeiro-aluno` que retorna a lista das 4 notas `(15.5 15 8.25 13)` obtidas pelo primeiro aluno da lista.

```lisp
(notas-do-primeiro-aluno (notas-dos-alunos))
(15.5 15 8.25 13)
```

3. Escreva uma função `calcula-media-notas` que recebe uma lista de 4 valores numéricos que representam as 4 notas de um aluno e retorna a média dos valores das notas.

Nota: Como ainda não conseguem criar funções recursivas, deverão utilizar as macros `first`, `second`, `third` e `fourth` para obter os valores das notas do aluno.

```lisp
CL-USER > (calcula-media-notas '(9 10 11 10))
10
CL-USER > (calcula-media-notas (notas-do-primeiro-aluno (notas-dos-alunos)))
12.9375
```

4. Escreva uma função `maior-nota-do-aluno` que devolve a maior nota de um conjunto de quatro notas contidas numa lista recebida como argumento.

Nota: Como ainda não conseguem criar funções recursivas, deverão utilizar as macros `first`, `second`, `third` e `fourth` para obter os valores das notas do aluno.

```lisp
CL-USER > (maior-nota-do-aluno (notas-do-primeiro-aluno (notas-dos-alunos)))
15.5
CL-USER > (maior-nota-do-aluno '(11 12 13 14))
14
```

3. Debug

Quando se desenvolvem programas grandes e/ou complexos é provável que se dispenda algum tempo na correção de bugs.

Neste sentido, a ferramenta de _Debug_ permite uma análise e contextualização mais profunda aos pedaços de código que se pretende depurar. No LispWorks para utilizar esta ferramenta é necessário:

A ferramenta de _Debugging_ permite rastrear a causa de problemas no seu código-fonte.
1. Vamos ver algumas formas de utilizar o _debugger_:

- Comecemos por invocar a função `soma-3` de forma errada propositadamente, colocando no _Listener_ a seguinte chamada:

```lisp
CL-USER > (soma-3 2 3)
```

- O LispWorks reporta o erro abaixo, dado que a função foi chamada com dois argumentos enquanto precisava de três:

> Error: The call (#<Function SOMA-3 200E62FA> 1 2) does not match definition (#<Function SOMA-3 200E62FA> A B C).
  1 (continue) Return a value from the call to #<Function SOMA-3 200E62FA>.
  2 Try calling #<Function SOMA-3 200E62FA> again.
  3 Try calling another function instead of #<Function SOMA-3 200E62FA> with the same arguments.
  4 Try calling #<Function SOMA-3 200E62FA> with a new argument list.
  5 (abort) Return to level 0.
  6 Return to top loop level 0.
Type :b for backtrace or :c \<option number> to proceed.
Type :bug-form "\<subject>" for a bug report template or :? for other options.

- O _Debugger_ aparece na linha de comando do próprio _Listener_ e pode ser usado aqui. É possível obter informações sobre a forma de utilizar o _Debugger_ no _Listener_ digitando `:?` na linha de comando.

- Para chamar a ferramenta de _Debugging_ nesta fase, escolha o comando do menu **Debug \> Start GUI Debugger** ou carregue no ícone da Figura 2 que consta da barra de ferramenta do _Listener_ para iniciar a ferramenta de _Debugging_. A ferramenta de _Debugging_ aparece tal como apresentado na Figura 3.

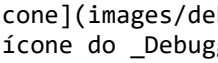
Figura 2: O ícone do _Debugger_.

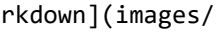

Figura 3: A janela do _Debugger_ composta por botões de controlo, condição de erro e _Backtrace_ (zona que permite analisar mais ao detalhe o estado da função e seus argumentos em que ocorreu o erro).

- Por um momento examine a listagem impressa na janela do erro de _Backtrace_.
O _Debugger_ mostra todos os registos de chamadas da função (no painel _Backtrace_), mostrando as funções que estão na pilha, na altura em que o erro ocorreu, assim como os argumentos passados para a função.

2. Outra ferramenta útil do LispWorks para _Debugging_ é o _Stepper_ que permite a colocação de _Breakpoints_ no código.

- Recorrendo novamente à função `soma-3` coloque um _Breakpoint_ na linha em que é efetuada a soma. O _Breakpoint_ pode ser colocado através do _Editor_, com o ficheiro _laboratorio2.lisp_ aberto, e clicando no seguinte ícone:

Figura 4: O ícone do _Debugger_.

Figura 5: Editor com o _Breakpoint_ colocado na linha pretendida (a vermelho).

- Ao invocar a função `soma-3` no _Listener_ da forma apresentada em baixo, a janela do _Stepper_ irá ser apresentada automaticamente quando a execução do código parar na linha marcada com o _Breakpoint_.

```lisp
CL-USER > (soma-3 1 2 3)
```


Figura 6: Janela do _Stepper_ indicando a linha em que ficou parado através do _Breakpoint_. Na aba de _Backtrace_ é possível analisar mais ao detalhes o que está a ser executado dentro da função e como estão os valores na altura em que a execução parou no _Breakpoint_. Para mais informações dos botões de controlo consultar a [documentação] (http://www.lispworks.com/documentation/lw70/IDE-W/html/ide-w-205.htm) do LispWorks.

4. Manuais

Quando se desenvolve um software é de enorme importância saber documentá-lo. Existem dois tipos de manuais que costumam ser elaborados na realização de um software, que são o Manual Técnico e o de Utilizador.

No âmbito da Unidade Curricular de Inteligência Artificial pretende-se que os alunos pratiquem a escrita de documentos recorrendo à linguagem de marcação ***Markdown***, que é amplamente utilizada para os ficheiros ***ReadMe*** e no ***GitHub***.

Como ferramenta para escrita dos documentos utilizaremos o [Visual Studio Code] (https://code.visualstudio.com/download) (VSCode), já conhecido de outras Unidades Curriculares.
No VSCode para poder escrever documentos **Markdown** é necessário efetuar os seguintes passos:
1. Abrir o VSCode
1. Pressionar CTRL + P para abrir a barra de acesso rápido
1. Escrever _ext install markdownlint_ para instalar a extensão apropriada
1. Clicar no botão de instalar e depois no de _reload_
1. **OPCIONALMENTE** poderá instalar também a extensão _Markdown PDF_ para poder fazer a exportação de _Markdown_ para PDF

Exemplo de utilização:

Figura 7: Exemplo de utilização do VSCode para escrita de documentos _Markdown_.

Para visualizar alguns dos comandos do _Markdown_ consultar o seguinte [link] (https://github.com/adam-p/markdown-here/wiki/Markdown-Cheatsheet).

4.1 Manual Técnico

O Manual Técnico representa um importante auxílio (principalmente) para os técnicos que não

participaram no desenvolvimento do projeto, proporcionando uma descrição técnica das diversas especificidades que suportaram as decisões e o desenvolvimento do software. Um dos principais objetivos deste manual é munir os técnicos que precisarão de conhecer o que foi desenvolvido, a compreender as especificidades técnicas que orientaram a implementação e as decisões tomadas, de forma a que consigam eles próprios prosseguir com o desenvolvimento ou corrigir algum aspeto que tenha ficado menos bem.

As secções base de um Manual Técnico são:

1. Capa
– identificação da UC, do projeto e do(s) aluno(s)
2. Arquitetura do sistema
– identificação dos módulos, seus objetivos individuais, suas conexões e como se relacionam
– que informação circula dentro de cada módulo e entre módulos
3. Entidades e sua implementação
– poderá recorrer-se à identificação e descrição de tipos abstratos de dados, objetos, ou usar outras formas de explicação
– convém fazer um paralelo entre as entidades do domínio de aplicação e as entidades programáticas, sem as confundir
4. Algoritmos e sua implementação
– poderá segmentar-se o sistema em módulos e explicar um de cada vez, identificando em cada caso as suas peculiariedades
– poderá haver interesse em referir métricas de desempenho
– Se possível apresentar os testes e validação
5. Descrição das opções tomadas
– descrever as opções de implementação que foram tomadas em detrimento de outras, por vezes essas opções podem não ser as mais óbvias e como tal devem ser documentadas
6. Limitações técnicas e ideias para desenvolvimento futuro
– requisitos não implementados
– refactoring que se percebe ser necessário fazer no futuro mas que não houve tempo para fazer
– melhoramentos potenciais de desempenho

Exercício
Tendo em conta as secções mencionadas escreva um documento em _Markdown_ que contemple um exemplo da estrutura do seu futuro Manual Técnico. O documento criado servirá de estrutura para o Manual Técnico dos Projetos a serem realizados.

4.2 Manual de Utilizador

O Manual de Utilizador é de mais fácil compreensão do que representa, uma vez que os encontramos em diversos objetos que se compra, seja um electrodoméstico, um carro ou um software. Este manual serve como um guião para proporcionar ao utilizador uma descrição do que fazer em diversas situações. No caso do software o mais comum é descrever todas as Funcionalidades existentes.

As secções base de um Manual de Utilizador são:

1. Capa
– identificação da UC, do projeto e do(s) aluno(s)
2. Acrónimos e Convenções usadas
– Explicação das convenções: abreviaturas comuns, definir formulário específico (por exemplo, para software, comandos digitados são fontes monoespaçadas, nomes dos botões utilizados, ...); etc.
3. Introdução
– Para quem é , para que serve? Que problemas resolve?
– Descrever de forma geral os requisitos que o programa satisfaz
4. Instalação e utilização
– Descrever como instalar e configurar o programa, para que este fique pronto a ser usado
– Descrever quais os comandos necessários para usar o programa
– Nas descrições anteriores poderá ser necessário explicar as diferenças de configuração e "arranque" em diferentes ambientes, se necessário.
5. Input/Output
- Qual a informação que o programa recebe (seja interativamente seja por ficheiros) e que informação produz (écran, ficheiros)
– Se a informação for estruturada, convém definir exatamente a sintaxe a semântica de cada tipo de entidade.
6. Exemplo de aplicação
– Será necessário descrever os comandos que o programa pode receber para funcionar de forma adequada e conduzir à resolução do problema para que foi concebido
– Deverá explicar-se a linha de sucesso, ou seja: o funcionamento típico, e algumas linhas de

insucesso, ou seja: possíveis comportamentos de exceção, erros, etc.

Exercício
Tendo em conta as secções mencionadas escreva um documento em _Markdown_ que contemple um exemplo da estrutura do seu futuro Manual de Utilizador. O documento criado servirá de estrutura para o Manual de Utilizador dos Projetos a serem realizados.

5. Organização do Código

A organização do código num programa é por vezes displicente, o que leva a uma má estruturação do programa e a uma futura manutenção do mesmo muito mais dispendiosa. No âmbito desta Unidade Curricular e no intuito de se respeitar algumas das boas normas da programação, o código deve ser estruturado de acordo com o seguinte ponto de vista:

* Dependente do Domínio de Aplicação
* Independente do Domínio de Aplicação
* Testes

Desta forma, as funções genéricas que não dependem do problema que se está a resolver, deverão ficar separadas daquelas que são criadas fruto da análise ao problema colocado e que dependem do domínio de aplicação. Os testes "programados" também deverão estar separados do código desenvolvido para o programa. **Estas informações são particularmente importantes no que diz respeito aos Projetos desta Unidade Curricular.**

Nota:
Para os laboratórios realizados ao longo do semestre, o aluno deverá criar um ficheiro (ou mais) onde guardará as funções criadas para resolver a ficha de laboratório. Como boa norma de programação e de forma a que as funções sejam devidamente documentadas, as mesmas deverão ter uma explicação `docstring` tal como demonstrado no laboratório anterior.