

Laboratórios de Informática I

2024/2025

Licenciatura em Engenharia Informática

Ficha 1

Ambiente de Desenvolvimento de Programas em Haskell

1 Sistema de Ficheiros

A informação armazenada nos computadores encontra-se normalmente organizada sobre a forma de **ficheiros** e **diretorias**. Os primeiros contêm os dados propriamente ditos, enquanto os segundos contêm ficheiros ou outras diretorias, o que permite a organização dos ficheiros numa hierarquia em forma de árvore. É comum a “árvore de diretorias” conter uma porção já pre-instalada no sistema (e.g. `/usr/bin`) que contém dados e programas necessários para o funcionamento do sistema. Outras partes dessa árvore são criadas pelos utilizadores para organizar os seus dados/programas – normalmente a partir de uma diretoria específica que é atribuída pelo sistema para cada utilizador: a **home** (a sua casa).

Relativamente aos *ficheiros*, podemos distinguir diferentes tipos:

Ficheiros de texto: cujo conteúdo é inteligível pelos humanos;

Aplicações: ficheiros que podem ser “executados” pelo utilizador. Dentro destes, alguns são comandos disponibilizados pelo próprio *sistema operativo*¹ (i.e. conjunto de programas responsável por gerir os recursos oferecidos pela máquina, como disco rígido, memória, *interface* de rede, etc.); enquanto outros serão aplicações *instaladas* pelo utilizador (e.g. Microsoft Office, Firefox, ...) ou até programas desenvolvidos de raiz.

Dados de aplicações: informação armazenada num formato específico para ser interpretado por programas específicos.

Extensões aos nomes dos ficheiros são uma forma habitual de distinguir entre os diferentes tipos. Por exemplo, a extensão `.txt` é utilizada para sinalizar um ficheiro de texto; `.doc` e `.docx` para os ficheiros com dados do popular processador de texto *Word*; `.hs` para ficheiros com código *Haskell*.

1.1 Comandos UNIX para gestão de ficheiros

A linha de comandos (CLI) é uma forma de interação com o sistema operativo, permitindo, por exemplo, navegar no sistema de ficheiros, ou compilar e executar programas.

¹Nesta UC vamos utilizar um sistema operativo da família UNIX (e.g. LINUX ou MAC OS)

O sistema operativo disponibiliza comandos que permitem manipular ficheiros. O formato geral dos comandos UNIX é:

`<cmd> -<opts> <args>`

onde

- `<cmd>` é o nome do comando (tipicamente uma abreviatura),
- `<opts>` são opções (normalmente letras), e
- `<args>` os argumentos requeridos pelo comando.

Por exemplo, “`ls -l dir`” permite listar² o conteúdo da diretoria `dir` no formato longo, formato esse determinado pela opção `-l`.

Exemplos de comandos UNIX úteis para manipular ficheiros:

`pwd`: mostra o caminho para a diretoria actual

`cd <dir>`: altera a diretoria actual para `<dir>`

`ls <dir>`: lista o conteúdo da diretoria `<dir>` (ou diretoria corrente, se se omitir `<dir>`)

`mkdir <dir>`: cria diretoria com nome `<dir>`

`rmdir <dir>`: remove diretoria `<dir>`

`rm <fich>`: apaga (remove) ficheiro `<fich>`

`cat <fich>`: visualiza conteúdo de `<fich>`

`more <fich>`: visualiza conteúdo de `<fich>` de forma paginada

Cada um destes comandos admite uma variedade de opções, que pode ser consultada da página respectiva do manual. O comando `man` permite visualizar a página do manual de qualquer comando do sistema (e.g. `man ls` visualiza a página do comando `ls`, mostrando em particular a opção `-l` referida acima).

Poderá encontrar informação detalhada nos seguintes tutoriais:

- <https://ubuntu.com/tutorials/command-line-for-beginners#1-overview>
- https://w3.cs.jmu.edu/spragunr/CS139_S16/activities/unix_tutorial/index.html: um tutorial simples sobre sistema UNIX.

2 Editores de Texto

Para criar e editar ficheiros de texto deve usar-se uma aplicação desenvolvida especificamente para esse fim designada por **editor de texto**. Este tipo de ferramenta auxilia no desenvolvimento de software pois utiliza esquemas de cores para salientar algumas partes do código, tornando-o mais fácil de ler. Em alguns casos, poderá também dar sugestões de como escrever ou melhorar código.

Nesta UC recomenda-se a utilização de um dos seguintes editores adequados para programação:

²`ls` é a abreviatura de *list*

- Visual Studio Code: <https://code.visualstudio.com> (recomendado)
- Sublime Text: <https://www.sublimetext.com>
- Vim: <https://www.vim.org>
- Emacs: <https://www.gnu.org/software/emacs>

3 Ambiente de Desenvolvimento

Ao longo do semestre iremos realizar um projecto de programação fazendo uso da linguagem *Haskell*. Vamos por isso necessitar de um conjunto de ferramentas de suporte para desenvolver programas nessa linguagem. Todas as ferramentas requeridas são disponibilizadas por um único pacote de instalação *GHcup* (<https://www.haskell.org/ghcup/>) para LINUX e MAC OS – aí encontrará nomeadamente:

- **ghc** (Glasgow Haskell Compiler) – um compilador para a linguagem *Haskell* (i.e. um programa que, dado um ficheiro com o código do programa *Haskell*, gera o ficheiro executável respectivo);
- **ghci** – o interpretador da linguagem *Haskell*;
- **cabal** – ferramenta para compilar e distribuir tanto bibliotecas como programas em *Haskell* (incluindo bibliotecas externas);
- ...

Ao longo das próximas semanas iremos aprender a usar estes programas (entre outros, a serem apresentados mais tarde).

3.1 Tarefa: Um primeiro programa

1. Verifique se o seu computador dispõe do **ghc** instalado (e.g. execute num terminal o comando `ghc --help`). Em caso negativo, instale a versão apropriada para o seu sistema.
2. Crie uma diretoria com o nome **LI1**. Crie dentro dessa diretoria uma subdiretoria **Aula1**.
3. Com auxílio de um dos editores de texto sugeridos anteriormente, crie na diretoria **Aula1** o ficheiro **HelloW.hs** com o seguinte conteúdo:

```
module Main where
main = do putStrLn "Hello World!"
```

4. Compile o programa. Para o efeito, execute o seguinte comando no terminal:

```
ghc HelloW.hs
```

5. Identifique, no código do programa, a posição do erro sinalizada pelo compilador. Corrija-o seguindo a sugestão apresentada na mensagem de erro.
6. Verifique a existência de um novo ficheiro com o nome **HelloW** na diretoria. Execute-o escrevendo `./HelloW`

4 Desenvolvimento de Programas em *Haskell*

Na secção anterior tivemos oportunidade de executar um programa exemplo desenvolvido na linguagem *Haskell*. Para o efeito utilizamos o compilador `ghc`, que produziu um ficheiro executável a partir do código fonte escrito em *Haskell* (no caso, o ficheiro `HelloW.hs` criado com o auxílio de um editor de texto).

Recorde-se que o executável produzido pelo compilador avalia uma função particular do programa fornecido — a função `main` do módulo `Main`. Quer isto dizer que outras funções incluídas no programa só serão executadas se forem elas próprias requeridas na avaliação da função `main` (e.g. se forem “invocadas” na função `main`). Este comportamento não é o mais apropriado numa fase de desenvolvimento, quando ainda só se implementou parte da funcionalidade e interessa testar cada função separadamente.

Iremos de seguida tomar contacto com uma outra ferramenta de desenvolvimento de programas *Haskell* mais adequada para a fase de desenvolvimento dos programas — o *interpretador*.

4.1 O Interpretador `ghci`

Ao contrário do compilador, o interpretador não produz qualquer ficheiro executável a partir de um programa *Haskell*. Em vez disso, disponibiliza ao programador um ambiente onde pode avaliar qualquer expressão *Haskell* à sua escolha.

O interpretador é invocado pelo comando `ghci`. Uma vez invocado, surge o *prompt* `ghci>`, sinalizando que o interpretador aguarda um comando do utilizador. Neste ponto pode-se avaliar uma qualquer expressão *Haskell* (e.g. `3+2*2`), ou um comando específico do interpretador (e.g. `:load Fich.hs`, que carrega o ficheiro `Fich.hs`, tornando disponíveis as várias funções aí definidas).

4.1.1 Alguns comandos do `ghci`

- `:?` ou `:help` — mostra informação sobre comandos do `ghci`;
- `:quit` — sai do interpretador;
- `:cd <dir>` — altera diretoria corrente para `<dir>`;
- `:load <mod>` — carrega módulo `<mod>` (ficheiro);
- `:reload` — recarrega último módulo;
- `:type <expr>` — imprime tipo da expressão `<expr>`
- `:info <symb>` — imprime informação sobre símbolo `<symb>`
- `!:<cmd>` — invoca o comando UNIX `<cmd>`

Quando não existir ambiguidade, o `ghci` aceita também abreviaturas dos comandos – por exemplo, o comando `:load HelloW` pode simplesmente ser escrito `:l HelloW`.

4.2 Utilização de Bibliotecas

A linguagem *Haskell*, tal como a generalidade das linguagens de programação, disponibiliza um conjunto de *bibliotecas* que oferecem ao programador um vasto leque de funcionalidades. Como regra, para utilizar uma biblioteca é necessário *importar* o respectivo módulo. A título de exemplo, no módulo `Data.Char` encontramos funções para manipular valores do tipo `Char` (a representação dos *caracteres*) — para ter acesso a essa funcionalidade é então necessário incluir a declaração `import Data.Char` no início do programa.

4.3 Documentação

Um recurso particularmente útil quando recorremos às bibliotecas oferecidas pela linguagem é a sua documentação — é aí que encontramos qual a funcionalidade oferecida (quais os módulos; tipos e funções disponibilizados), assim como uma descrição sumária de cada função (incluindo o seu tipo).

Na página da documentação do *Haskell*, em *Library Documentation*, encontra informação sobre as bibliotecas. No item *Haddocks for Libraries included with GHC* encontrará as bibliotecas instaladas pelo GHC. Exemplos de bibliotecas que teremos oportunidade de explorar:

- `Prelude` — conjunto de tipos e funções pré-carregados (i.e. não é necessário importar explicitamente qualquer módulo);
- `Data.Char` — funções de manipulação de caracteres;
- `Data.String` — funções de manipulação de *strings* (i.e. sequências de caracteres);
- `Data.List` — funções de manipulação de listas;
- `Data.Maybe` — funções para manipulação do tipo `Maybe`.

Outros apontadores *web* úteis são:

- <https://www.haskell.org>: página oficial da linguagem, que inclui apontadores para todo o tipo de documentação sobre a linguagem (em particular, a própria especificação da linguagem);
- <https://www.haskell.org/hoogle/>: disponibiliza um mecanismo de busca sobre a documentação das bibliotecas;
- <https://hackage.haskell.org>: sítio que agrega contribuições (*packages*) desenvolvidas em *Haskell*.
- <https://www.fpcomplete.com/haskell/learn/>: tutoriais *online* da linguagem;
- https://www.tutorialspoint.com/compile_haskell_online.php: ambiente *online* para compilar/executar programas *Haskell*;

4.4 Tarefas

Vamos de seguida programar em *Haskell* e testar (no **ghci**) algumas funções simples. Para o efeito crie o ficheiro **Ficha1.hs** usando um editor de texto, e grave-o na directoria **LI1/Aula1/** criada anteriormente. Escreva nesse ficheiro as diversas funções. Adicione a cada uma das funções implementadas a respectiva anotação de tipo. Para testar, escreva no interpretador **:load Ficha1.hs** (ou de forma abreviada **:l Ficha1**) para carregar o ficheiro **Ficha1.hs**, e teste cada função escrevendo o nome da função seguido dos respectivos argumentos.

1. Considere as seguintes funções pré-definidas do Haskell:

- **length l**: o número de elementos de uma lista *l*
- **head l**: a cabeça da lista (não vazia) *l*
- **tail l**: a cauda da lista (não vazia) *l*
- **init l**: o segmento inicial da lista (não vazia) *l*
- **last l**: o último elemento da lista (não vazia) *l*
- **elem x l**: verifica se *x* é elemento da lista *l*
- **xs ++ ys**: junta duas listas (a lista *ys* ao final da lista *xs*)
- **div x y**: o quociente da divisão inteira de *x* por *y*
- **mod x y**: o resto da divisão inteira de *x* por *y*
- **fst (x,y)**: a primeira componente, *x*, do par
- **snd (x,y)**: a segunda componente, *y*, do par

Verifique qual o tipo de cada uma destas funções. Teste-as no interpretador.

2. Use as funções pré-definidas do item anterior para definir as funções seguintes:

- (a) Defina uma função que recebe uma lista não vazia e devolve um par com o primeiro e último elemento da lista.
- (b) Defina uma função que recebe uma lista de strings, representando os nomes de uma pessoa, e produz um par com o primeiro e o último nome dessa pessoa.
- (c) Defina uma função que recebe uma lista de strings com os nomes de uma pessoa e calcula uma única string com o primeiro nome e com o apelido. Considere que o apelido é o último nome da lista. Se a lista de nomes recebida for vazia, devolve a string vazia.
- (d) Defina uma função que recebe uma lista de strings com os nomes de uma pessoa e calcula o tamanho do apelido dessa pessoa. Se a lista recebida for vazia, devolve o valor zero.
- (e) Defina uma função que verifique se um caractere pertence a uma string.
- (f) Defina uma função que recebe uma lista não vazia e remove o primeiro elemento da lista, se ela tiver um número par de elementos. Se a lista tiver um número ímpar de elementos, remove o último elemento.
- (g) Considere que representamos um turno como a lista dos números dos seus alunos. Pretende-se juntar os alunos de dois turnos num único turno.

- i. Defina uma função que calcule a lista dos alunos do turno resultante da junção.
 - ii. Defina uma função que calcule o número de alunos do turno resultante da junção.
 - iii. Defina uma função que calcule a diferença entre o número de alunos dos turnos. O resultado deve ser um número não negativo.
 - iv. Defina uma função que recebe um turno e o número de um aluno, e verifique se o aluno pertence a esse turno (deve devolver “Sim” caso o aluno pertença ao turno e “Nao” caso contrário).
- (h) Defina uma função que recebe duas listas de inteiros e faz a sua junção, colocando em primeiro lugar a lista com menos elementos.
- (i) Defina uma função que recebe duas listas de inteiros e faz a sua junção, colocando em primeiro lugar a lista que tem o menor elemento à cabeça.
- (j) Defina uma função que recebe um par de listas (xs, ys) e devolve um par em que a primeira componente é o elemento que está à cabeça da primeira lista xs , e a segunda componente é a lista ys inalterada.
- (k) Defina uma função que recebe uma lista de nomes de uma pessoa e produz uma string com a inicial do primeiro nome, seguida do caracter '.' seguida do último nome. Por exemplo, se a função receber a lista `["Joaquim", "Francisco", "Alves", "Martins"]`, deve devolver o valor `"J.Martins"`
- (l) Dada uma lista de pares de inteiros, defina uma função que selecione a segunda componente do par à cabeça da lista. Se a lista for vazia, devolve zero.
- (m) Dada uma lista de pares de inteiros, defina uma função que adiciona os inteiros do último par da lista. Se a lista for vazia, devolve zero.
- (n) Representa-se informação acerca do nome e idade de uma pessoa, num par do tipo `(String, Int)`. Defina uma função que recebe informação de duas pessoas e indica o nome da pessoa mais nova.
- (o) Vamos representar um ponto como um par de coordenadas inteiras num plano cartesiano. Defina uma função que recebe as coordenadas do vértice superior esquerdo de um quadrado e o comprimento do lado (os lados são paralelos aos eixos), e calcula as coordenadas do vértice inferior direito.

```
f :: (Int, Int) -> Int -> (Int, Int)
f (x,y) lado = ...
```

- (p) Vamos representar um ponto como um par de coordenadas inteiras num plano cartesiano. Vamos representar um quadrado como um par, em que a primeira componente é vértice do canto superior esquerdo, e a segunda componente é o comprimento do lado. Considere que os lados do quadrado estão paralelos aos eixos. Defina uma função que recebe dois quadrados e calcula a área do quadrado "mais acima" no plano cartesiano. Em caso de igualdade, indique a menor área.

```
g :: ((Int, Int), Int) -> ((Int, Int), Int) -> Int
g ((x1,y1), l1) ((x2,y2), l2) = ...
```

3. Defina e teste outras funções da Ficha 1 de Programação Funcional (funções não recursivas).