Laboratórios de Informática I
2024/2025

Licenciatura em Engenharia Informática

Ficha 3
Documentação Estruturada de Programas

No contexto das linguagens de programação, os *comentários* são uma construção da linguagem que permite aos programadores incluírem texto livre no meio do programa. Naturalmente que esse texto não influirá no comportamento do programa (i.e. vai acabar por ser ignorado pelo compilador ou interpretador), mas pode ser muito útil ao facilitar a compreensão do programa por parte de quem lê o código fonte.

Nesta semana iremos ver como tirar partido dos comentários em *Haskell* e, em particular, ilustrar como estes podem ser explorados para gerar documentação para o código produzido, de forma automática.

1 Comentários em *Haskell*

Em *Haskell* estão previstos dois tipos de comentário:

- bloco de texto envolvido entre as marcas `"{-"`e `"-}"`. Os blocos de comentários podem abranger múltiplas linhas;
- desde os caracteres `"--"`até ao final de linha.

Apresenta-se em seguida um fragmento de código *Haskell* onde se utilizam ambos os tipos de comentário.

```
{- "fact n" calcula o factorial de um número inteiro "n".
   obs: assume-se que o argumento "n" é não negativo.
-}
fact :: Integer -> Integer
fact 0 = 1               -- caso de base da recursividade
fact n = n * fact (n-1)  -- caso indutivo
```

Pela sua natureza, o texto inserido nos comentários é completamente livre. Existem no entanto um conjunto de boas práticas que é conveniente seguir para retirar o máximo proveito dessa construção nos programas desenvolvidos:

- um comentário não deve "repetir" por palavras o que o programa faz — quem for ler o programa deverá conhecer a linguagem, pelo que pode retirar essa informação facilmente

do próprio programa[1]. Pode no entanto explicitar a *intenção* do fragmento de código respectivo;

- certas ocasiões podem justificar que se usem os comentários para explicitar ou explicar as estratégias/algoritmos utilizados na resolução de um problema;
- durante a fase de desenvolvimento, é habitual utilizar os comentários para inserir:
  - descrição sobre funcionalidades que ainda terão de ser implementadas;
  - marcas como `FIXME` ou `TODO` que assinalam pontos que devem merecer ainda atenção por parte do(s) programador(es);
  - para excluir do programa código que se destine a teste/*debug*.

Refira-se por último que a função dos comentários como facilitadores da compreensão dos programas por parte dos humanos (ao invés dos computadores) é levada ao extremo na filosofia de *Literate Programming* (c.f. `https://en.wikipedia.org/wiki/Literate_programming`, `https://wiki.haskell.org/Literate_programming`) advogada por certos autores.

2 Comentários Estruturados

As linguagens de programação usam ainda os comentários como um mecanismo para adicionar informação ao código desenvolvido, por forma a permitir a extração automática de documentação. Nesse caso, o texto inserido em determinados comentários inclui uma estrutura própria que possibilita que ferramentas específicas processem esses comentários e extraiam de lá a informação necessária para a produção da documentação relevante.

A grande vantagem de se utilizarem sistemas de documentação integrados no próprio código é que assim se garante:

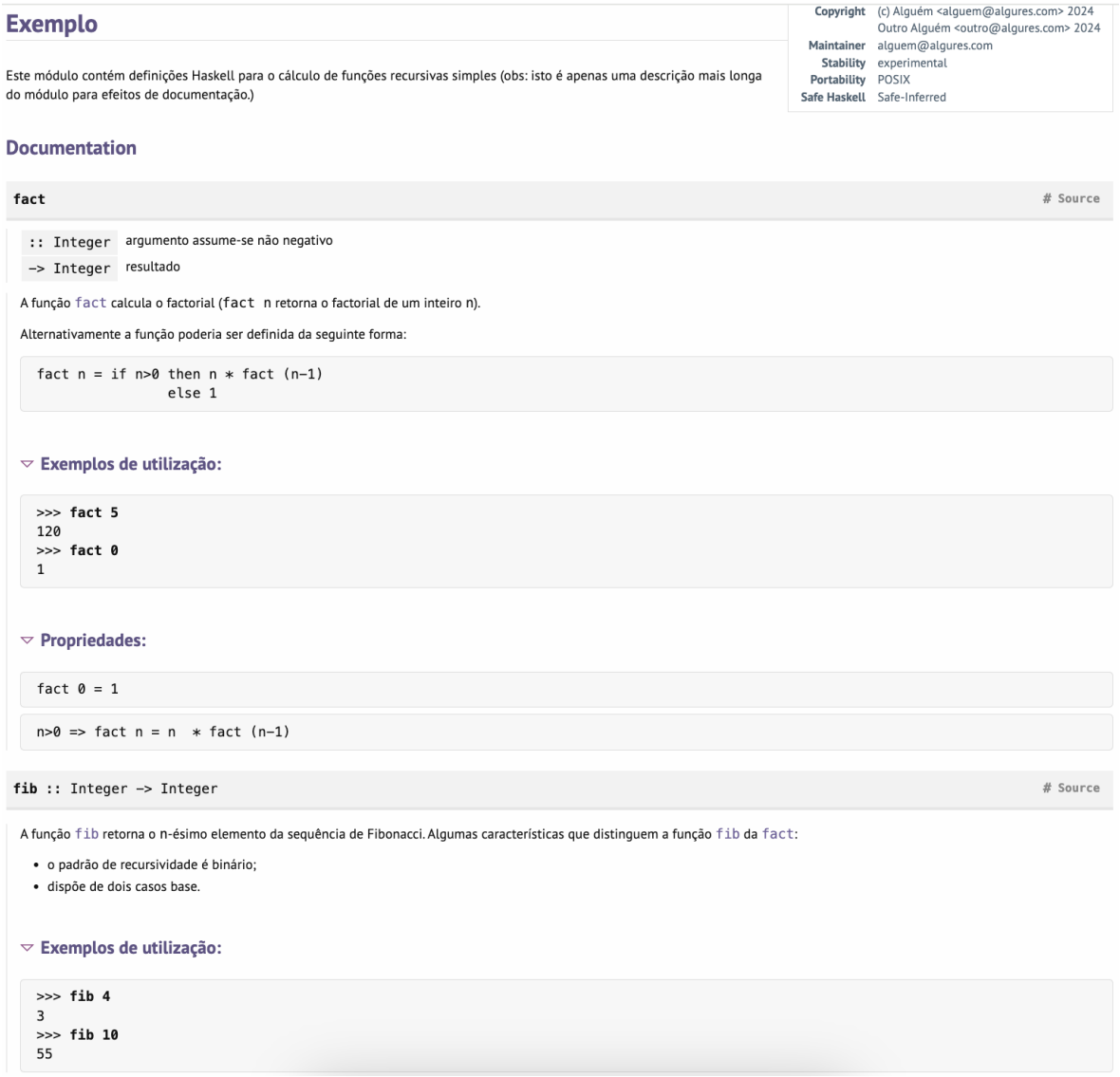
- que a documentação se encontra sempre actualizada em relação à versão do código considerada;
- que o sistema de documentação possa aceder a informação contida no próprio código, evitando assim que tenha de ser o programador a transferir essa informação para a documentação (e.g. aceder aos tipos dos argumentos e resultados das funções).

Um exemplo deste tipo de sistemas de documentação é a própria documentação das bibliotecas do *Haskell* (acessível em `https://www.haskell.org/ghc/docs/latest/html/libraries/index.html`).

2.1 Haddock

O `haddock` é uma ferramenta de extração de documentação para programas *Haskell*. Destina-se a processar programas *Haskell* com anotações apropriadas inseridas nos comentários, e produz como resultado páginas de documentação em diferentes formatos (e.g. HTML, para se visualizar em *browsers web*).

[1] Uma exceção a este princípio pode justificar-se quando o programa se destina a ser lido por quem não tiver um conhecimento aprofundado da linguagem (e.g. fins pedagógicos).



Exemplo

Este módulo contém definições Haskell para o cálculo de funções recursivas simples (obs: isto é apenas uma descrição mais longa do módulo para efeitos de documentação.)

Copyright	(c) Alguém <alguem@algures.com> 2024 Outro Alguém <outro@algures.com> 2024
Maintainer	alguem@algures.com
Stability	experimental
Portability	POSIX
Safe Haskell	Safe-Inferred

Documentation

fact # Source

`:: Integer` argumento assume-se não negativo
`-> Integer` resultado

A função `fact` calcula o factorial (`fact n` retorna o factorial de um inteiro n).

Alternativamente a função poderia ser definida da seguinte forma:

```
fact n = if n>0 then n * fact (n-1)
                else 1
```

Exemplos de utilização:

```
>>> fact 5
120
>>> fact 0
1
```

Propriedades:

```
fact 0 = 1
```

```
n>0 => fact n = n  * fact (n-1)
```

fib `:: Integer -> Integer` # Source

A função `fib` retorna o n-ésimo elemento da sequência de Fibonacci. Algumas características que distinguem a função `fib` da `fact`:

- o padrão de recursividade é binário;
- dispõe de dois casos base.

Exemplos de utilização:

```
>>> fib 4
3
>>> fib 10
55
```

Figura 1: Exemplo de documentação gerada por `haddock`

Ilustram-se diferentes construções sintácticas do `haddock` por intermédio de um pequeno exemplo apresentado na Figura 1. O código que deu origem a este exemplo é apresentado no Anexo A.

Uma apresentação sumária de algumas facilidades do `haddock` utilizadas neste exemplo:

- Um comentário iniciado por "{- |" (para blocos), ou "- |" (para linhas) será interpretado pelo `haddock` como um comentário referente à declaração que se segue (e.g. declaração de funções, de tipos, de construtores, etc.);
- Um comentário iniciado por "- ^" diz respeito ao item imediatamente anterior (e.g. tipo envolvido numa assinatura);
- Nos comentários `haddock` podem-se incluir parágrafos livres que serão incluídos na do-

cumentação gerada. Podem-se ainda utilizar marcas para formatação do texto, como:

  - Parágrafos são separados por linhas em branco;
  - `/text/`: imprime `text` em *itálico*;
  - `__text__`: imprime `text` em **bold**;
  - `@text@`: imprime `text` em `mono-espaçamento`;
  - `'name'`: insere hiper-ligação para entidade `name`;
  - `* item` (no inicio de um parágrafo): item de uma lista (não enumerada);
  - `1. item` (no inicio de um parágrafo): item de uma lista (enumerada);
  - `= heading`, `== subheading`, `=== subsubheading`: incício de secção, sub-secção; etc.

- fragmentos de código podem ser incluídos envolvidos entre linhas iniciadas por @ (bloco), ou > (linha única);

```
@
fact n = if n>0
         then n*fact (n-1)
         else 1
@
```

- exemplos de teste de funções são apresentados em linhas prefixadas por `>>>`. O resultado esperado surge na(s) linha(s) seguinte(s).

```
>>> fact 5
120
>>> fact 0
1
```

Recomenda-se a consulta do manual do `haddock` (`https://www.haskell.org/haddock/doc/html/index.html`) para uma explicação mais detalhada de cada uma destas construções, assim como de outras construções não referidas neste resumo.

Invocação do `haddock`

Dispondo de um programa *Haskell* com as anotações apropriadas, torna-se necessário invocar o comando `haddock` para se produzir efectivamente a documentação pretendida. Uma linha de comando apropriada para se produzir a documentação a partir do programa de exemplo apresentado no Anexo A é:

```
haddock -h -o doc/html Exemplo.hs
```

A opção `-h` indica que se pretende que a documentação seja produzida no formato HTML; a opção `-o doc/html` indica que os ficheiros resultantes devem ser gravados na (sub-)diretoria `doc/html` — a documentação ficará assim acessível a partir da página `doc/html/index.html`. Para incluir na documentação apontadores para o código *Haskell* poderá executar:

```
haddock -h -o doc/html --hyperlinked-source Exemplo.hs
```

2.1.1 Tarefas

1. Copie o código *Haskell* apresentado no anexo A para um ficheiro chamado `Exemplo.hs`. Teste no interpretador as funções `fact` e `fib`.

2. Execute no terminal o comando `haddock` com as opções necessárias para produzir a documentação em formato HTML do módulo `Exemplo.hs`. A documentação deverá ser criada na sub-diretoria `doc/html/`

3. Compare com o resultado apresentado na Figura 1.

3 Funções recursivas sobre listas ou sobre inteiros, em Haskell

Pretende-se de seguida continuar o estudo de funções recursivas em *Haskell* e utlizar o `haddock` para documentar o código criado. Para o efeito, crie o ficheiro `Aula3.hs` na diretoria `Aula3`. Escreva nesse ficheiro as diversas funções a definir. Inclua, à medida que vai resolvendo as funções, comentários `haddock` para permitir a geração automática de documentação. No final da aula, execute o comando `haddock` com as opções necessárias para produzir a documentação em formato HTML do módulo `Aula3.hs` e gravar o resultado em `doc/html`.

1. Defina uma função que recebe uma lista de strings e remove todas as strings iniciadas por um dado caractere.

2. Defina uma função recursiva que recebe uma lista de pares de inteiros e adiciona um valor dado à primeira componente de cada par.

3. Defina uma função recursiva que recebe uma lista, não vazia, de pares de inteiros e calcula qual o maior valor da segunda componente.

4. Considere o seguinte tipo de dados para representar posições de uma pessoa num sistema de coordenadas cartesiano:

```
type Nome = String
type Coordenada = (Double, Double)
data Movimento= N | S | E | W deriving (Show,Eq) -- norte, sul, este, oeste
type Movimentos = [Movimento]
data PosicaoPessoa = Pos Nome Coordenada deriving (Show,Eq)
```

   (a) Dada uma lista de posições de pessoas, atualize essa lista depois de todas executarem um movimento dado:

   ```
   posicoesM:: [PosicaoPessoa] -> Movimento -> [PosicaoPessoa]
   ```

   (b) Defina uma função que calcule a posição de uma pessoa depois de executar uma sequência de movimentos:

   ```
   posicao:: PosicaoPessoa -> Movimentos -> PosicaoPessoa
   ```

   (c) Dada uma lista de posições de pessoas, atualize essa lista depois de todas as pessoas executarem uma mesma sequência de movimentos. Use as funções anteriormente definidas.

```
posicoesMs:: [PosicaoPessoa] -> Movimentos -> [PosicaoPessoa]
```

(d) Defina uma função que recebe uma lista de posições de pessoas e indica qual a posição mais a norte (em caso de igualdade, escolha uma posição).

```
pessoaNorte:: [PosicaoPessoa] -> Maybe PosicaoPessoa
```

(e) Defina uma função que calcule quais as pessoas posicionadas mais a norte:

```
pessoasNorte:: [PosicaoPessoa] -> [Nome]
```

5. Defina uma função recursiva que recebe uma lista e desloca cada elemento da lista, n posições para a direita. Os elementos do final da lista são colocados no início (e.g. [1,2,3,4,5] daria [4,5,1,2,3], para n=2).

6. Defina uma função recursiva semelhante à anterior, mas que desloca os elementos para a esquerda.

7. Defina uma função recursiva que procure a posição de um elemento numa lista (posição da primeira ocorrência). Devolve (-1) caso o elemento não ocorra na lista. O índice de posições começa em zero.

8. Defina uma função recursiva que substitua um elemento de uma posição numa lista, por outro valor dado. Por exemplo: dada a lista "abcdefg", a posição 3 e o elemento 'X', devolve "abcXefg".

9. Considere as seguintes funções pré-definidas:

```
(!!), concat, words, unwords, lines, unlines,
take, drop, splitAt, zip, unzip, replicate
```

   Investigue o seu tipo e teste-as.

10. Use as funções pré-definidas `take`, `drop` e/ou `splitAt` para reformular algumas das funções anteriores.

A Código do programa de exemplo

```
{-|
Module      : Exemplo
Description : Módulo Haskell contendo funções recursivas.
Copyright   : (c) Alguém <alguem@algures.com>, 2024
              Outro  Alguém  <outro@algures.com>, 2024
Maintainer  : alguem@algures.com
Stability   : experimental
Portability : POSIX

Este módulo contém definições Haskell para o cálculo de funções recursivas simples (obs: isto é apenas uma descriçã
-}
module Exemplo where
{-| A função 'fact' calcula o factorial (@fact n@ retorna o factorial de um inteiro @n@).

Alternativamente a função poderia ser definida da seguinte forma:

@
fact n = if n>0 then n * fact (n-1)
                else 1
@

==  __Exemplos de utilização:__
>>> fact 5
120
>>> fact 0
1

== __Propriedades:__
prop> fact 0 = 1
prop> n>0 => fact n = n  * fact (n-1)
-}
fact ::
  Integer       -- ^ argumento assume-se não negativo
  -> Integer    -- ^ resultado
fact 0 = 1
fact n = n * fact (n-1)

{-| A função 'fib' retorna o @n@-ésimo elemento da sequência de Fibonacci.
Algumas características que distinguem a função 'fib' da 'fact':

* o padrão de recursividade é binário;

* dispõe de dois casos base.

== __Exemplos de utilização:__
>>> fib 4
3
>>> fib 10
55
-}
fib :: Integer -> Integer
fib 0 = 0
fib 1 = 1
fib n = fib (n-1) + fib (n-2)
```