Documentação do Código

Laboratórios de Informática I

MIEI

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 produzir 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 compreender múltiplas linhas;
- desde os caraceres `--` 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 inductivo
```

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 para uma compreensão dos programas por parte dos humanos (ao invés dos computadores) é levada ao extremo na filosofia de *Literate Programming* (`http://en.wikipedia.org/wiki/Literate_programming`) advogada por certos autores.

2 Comentários Estruturados

As linguagens de programação modernas 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

[1] Uma excepção 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).

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* que referimos na última sessão (acessível em `http://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*).

Ilustra-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. funções; tipos; construtores; etc.);
- Um comentário iniciado por "`-- ^`" diz respeito ao item imediatamente anterior (e.g. tipo envolvido numa assinatura);

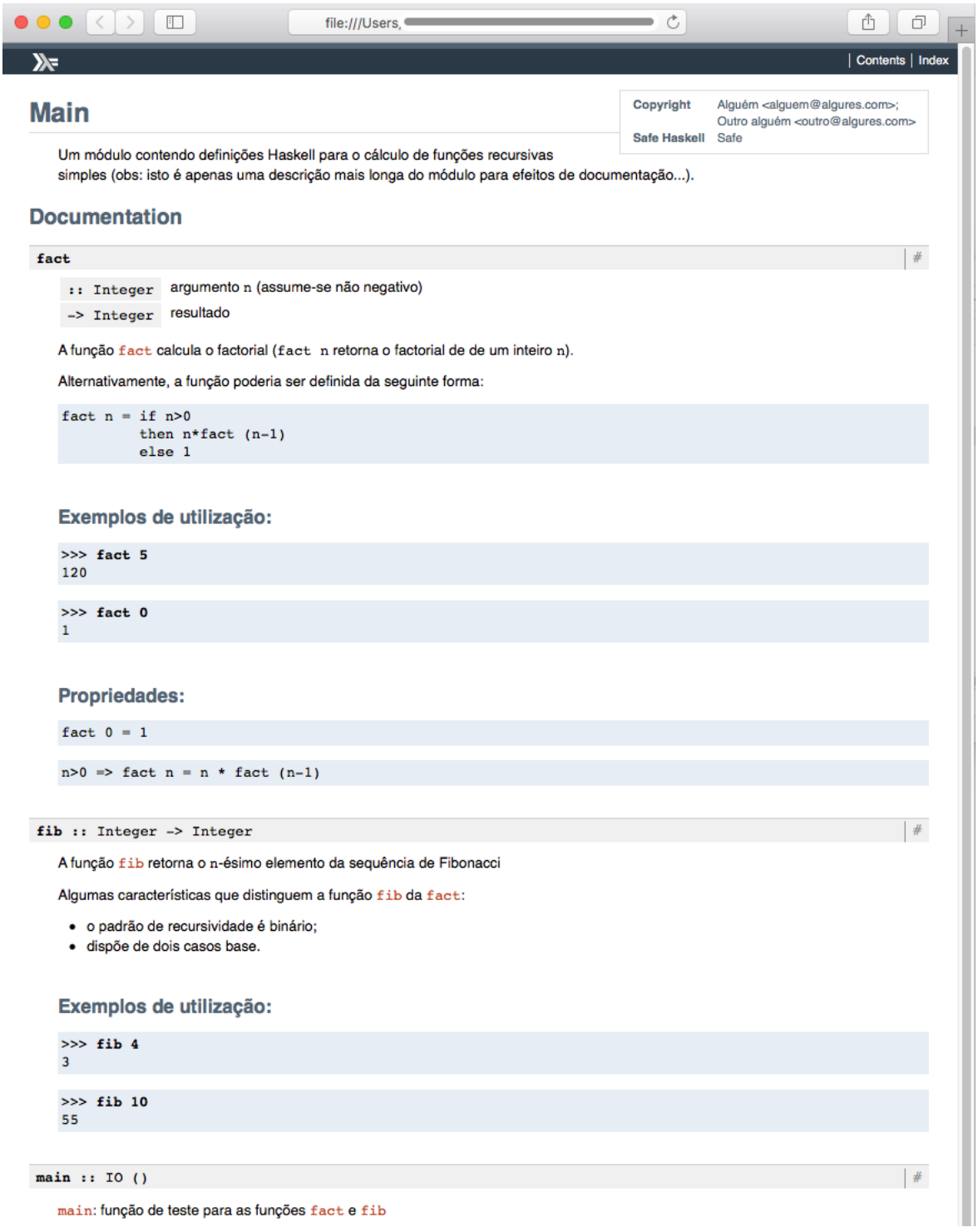

Figura 1: Exemplo de documentação gerada por `haddock`

- Nos comentários `haddock` podem-se incluir parágrafos livres que serão incluídos na documentação 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 (enumeração);
  - `= heading`, `== subheading`, `=== subsubheading`: incício de secção, sub-secção; etc.
- fragmentos de código podem ser incluídos envolvidos entre linhas iniciadas por @ (blocp), ou > (linha única);
- exemplos de utilização são apresentados em linhas prefixadas por `>>>`. O resultado esperado surge na(s) linha(s) seguinte(s).

Recomenda-se a consulta do manual do `haddock` (`http://www.haskell.org/haddock/doc/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 Main.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-)directoria `doc/html` — a documentação ficará assim acessível a partir da página `doc/html/index.html`.

3 Tarefas

Como habitualmente, não se esqueça de criar uma nova directoria de trabalho com nome `LI1`, apagando previamente alguma que possa existir com esse nome (isto se não utilizar compotador próprio, naturalmente). No final da aula deve remover a directoria de trabalho.

1. Implemente e teste (no `ghci`) algumas das funções solicitadas no grupo II do caderno de exercícios de Programação Funcional (funções recursivas).

2. Inclua, à medida vai realizando as diversas funções, comentários `haddock` para permitir a geração automática de documentação.

3. No final da aula, execute o comando `haddock` com as opções necessárias para produzir documentação em formato HTML.

A Código do programa de exemplo

```
{-|
Module      : Main
Description : Módulo Haskell contendo exemplos de funções recursivas
Copyright   : Alguém <alguem@algures.com>;
              Outro alguém <outro@algures.com>

Um módulo contendo definições Haskell para o cálculo de funções
recursivas simples (obs: isto é apenas uma descrição mais
longa do módulo para efeitos de documentação...).
-}
module Main where

import System.IO

{- | A função 'fact' calcula o factorial (@fact n@ retorna o factorial
  de 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 'n' (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)

-- | 'main': função de teste para as funções 'fact' e 'fib'
main :: IO ()
main = do hSetBuffering stdout NoBuffering   -- forçar sincronismo do "output"
          putStr "Introduza valor: "
          n_inp <- getLine                   -- ler linha do terminal
          let n = read n_inp                 -- converte String -> Int
          putStrLn (show n ++ "! = " ++ show (fact n))
          putStrLn ("fib("++show n ++ ") = " ++ show (fib n))
```