Programação Lógica e Funcional

Paradigma Funcional com Haskell (Parte 4)

Introdução

Este material contém notas de aula sobre Geração de Arquivo Executável e Entrada e Saída – E/S em Haskell, seguindo o material sobre o *Paradigma Funcional*. O objetivo principal desta aula é compreender como é feita a geração de arquivos executáveis utilizando recursos de E/S em Haskell. Os objetivos específicos são:

- Compreender como se usa a ferramenta GHC (*Glasgow Haskell Compiler*).
- Conhecer as principais funções de leitura de dados
- Conhecer as principais funções de escrita de dados.

Assume-se que o leitor tenha familiaridade com o Paradigma Funcional e a linguagem Haskell, bem como tenha realizado a leitura dos materiais anteriores, citados abaixo:

- Parte 1, que aborda principalmente aspectos como conceitos do Paradigma Funcional, apresentação de Haskell, suas aplicações, tipos, ferramentas, sintaxe, operadores, funções, listas, tuplas, avaliação preguiçosa, entre outros.
- Parte 2, que aborda estruturas de decisão, recursividade, *let*, aprofundamento sobre listas, funções de alta ordem e sinônimos de tipo.
- Parte 3, que aborda mais sobre funções de alta ordem, currificação e composição de funções.

Criação de Arquivos Executáveis em Haskell

Os materiais anteriores abordaram majoritariamente a criação e o uso de funções por meio do GHCi (*Glasgow Haskell Compiler Interactive*), embora a ferramenta GHC tenha sido citada no material da Parte 1. Neste material, foi apresentada como uma das formas de uso do Haskell a criação de um arquivo de extensão `.hs`, a ser compilado com o GHC e executado, mas ainda sem detalhes sobre como utilizar o GHC.

RELEMBRANDO

O GHCi é um ambiente de execução interativo para Haskell, usado para testar funções, geralmente implementadas com pequenos trechos de código. É uma ferramenta de linha de comando (comando `ghci`) que permite escrever código Haskell e executá-lo imediatamente, sem precisar compilar um arquivo separado.

O uso do GHC se torna interessante quando deseja-se portabilidade, ou seja, criar arquivos executáveis que poderão ser utilizados em outros ambientes.

GHC

O GHC é o compilador Haskell. Assim como qualquer compilador, ele é responsável por fazer a tradução de um código-fonte Haskell para um código de máquina. Como já visto, um código-fonte Haskell deve ser escrito em um arquivo com extensão `.hs`, que significa *Haskell Source-Code*.

Este compilador aceita diversos parâmetros, os quais podem ser conhecidos a partir da leitura da seção "`OPTIONS`" do *manual da ferramenta*[1].

[1] O manual do GHC pode ser obtido digitando o comando **`man ghc`** a partir do terminal de um Sistema Operacional Unix-like.

Geração do Arquivo Executável

Assim como ocorre em outras linguagens de programação, o ponto de partida de um programa escrito em Haskell é a função `main`. A função `main` é a primeira função a ser executada quando o programa é iniciado e é obrigatória em todo programa Haskell.
A assinatura da função `main` é:

```
main :: IO ()
```

Isso significa que a função `main` retorna um valor do tipo `IO ()`, onde `IO` é uma mônada[2] de entrada e saída (IO significa *Input/Output*) e `()` é um tipo vazio que indica que a função não retorna valores úteis.

O código do quadro abaixo, disponível também no arquivo aula-io-exemplo-1.hs, exemplifica o uso da função `main`:

```
main :: IO ()
main = putStrLn "Hello, world!"
```

Para compilar o código, basta digitar: **`ghc aula-io-exemplo-1.hs`**

O comando de compilação gerará os seguintes arquivos:

- `aula-io-exemplo-1.hi`: arquivo de interface gerado pelo GHC. Ele contém informações sobre as interfaces de módulos e tipos definidos no programa. Este arquivo é usado pelo compilador para garantir que as referências a módulos e tipos externos estejam corretas e para acelerar a compilação de programas que dependem de

[2] Estrutura que encapsula um contexto e fornece um mecanismo para lidar com os efeitos colaterais desse contexto. São construções teóricas (matemáticas) da programação funcional que ajudam a lidar com efeitos colaterais, como a incerteza de uma resposta. Uma das aplicações mais comuns é lidar com cálculos que podem produzir resultados incertos ou que podem falhar, como uma divisão por zero. A mônada IO refere-se a efeitos colaterais de entrada e saída, que são as operações que envolvem a comunicação entre um programa e o ambiente externo, tais como ler e escrever em arquivos, interagir com o usuário via *prompt*, fazer conexões de rede etc. Essas operações podem produzir efeitos colaterais visíveis fora do programa e alterar o estado do sistema em que o programa é executado. Por exemplo, se o programa executado escreve em um arquivo, o conteúdo do arquivo será modificado mesmo depois que o programa terminar de ser executado.

módulos externos.

- `aula-io-exemplo-1.o`: arquivo que contém o código objeto[3] gerado pelo compilador a partir do programa Haskell. Esse arquivo é posteriormente ligado com outros arquivos objeto para gerar o executável final.
- `aula-io-exemplo-1`: arquivo executável gerado pelo GHC. É o arquivo que pode ser executado para rodar o programa.

Para executar o programa, basta digitar: **`./aula-io-exemplo-1`**

Neste primeiro exemplo criado, a primeira linha (assinatura da função) pode ser omitida. Neste caso, como visto anteriormente, a função `main` possuirá o tipo inferido pelo compilador Haskell. No entanto, é uma boa prática declarar explicitamente a assinatura da função para deixar claro que se trata de uma ação de E/S e garantir a coerência do tipo de saída do programa. Além disso, explicitar a assinatura da função `main` pode ajudar a detectar erros de tipagem que podem ocorrer ao compilar o programa.

O código do quadro abaixo, disponível também no arquivo aula-io-exemplo-2.hs, exemplifica o uso da função `main` sem a definição do tipo:

```
-- Sem definir o tipo da main o código também funciona!
main = putStrLn "Hello, world!"
```

Assim como ocorre com outros compiladores como o GCC, a opção `-o` permite decidir o nome do arquivo executável. Caso não seja utilizada, o nome será o mesmo do arquivo fonte, mas sem a extensão `.hs`. Os comandos abaixo exemplificam a compilação do código utilizando o parâmetro `-o` para definir o nome do arquivo executável e a execução do arquivo executável:

```
ghc aula-io-exemplo-2.hs -o executavel
./executavel
```

[3] Resultado da compilação de um programa de alto nível. É uma representação intermediária do programa, que ainda não está em um formato executável, pois ainda depende da ação do ligador (*linker*[3]).

[3] Programa que une arquivos objeto e bibliotecas compartilhadas em um único arquivo executável. Realiza a etapa final da compilação, resultando em um arquivo executável.

Até aqui a função `main` apresentada nos exemplos possui apenas uma instrução, um comando de escrita. Se a função main for composta por duas ou mais instruções, é necessário utilizar o bloco `do`, como no exemplo do quadro a seguir, também disponível no arquivo aula-io-exemplo-3.hs:

```
main :: IO ()
main = do
   putStrLn "Hello, world!"
   putStrLn "How are you?"
```

Observação: um arquivo contendo código-fonte Haskell pode ser carregado pelo GHCI (**`:l arquivo.hs`**) independentemente de ter ou não a função `main`. Isso significa que as funções definidas em um arquivo Haskell podem ser testadas no GHCI normalmente.

Entrada e Saída (*Input/Output* – IO)

Entrada/Saída (E/S) ou *Input/Output* (IO) é uma das formas de lidar com ações que envolvem interação com o mundo exterior, como ler ou escrever em um arquivo, receber um texto via teclado ou enviar dados para um servidor remoto.

IO em Haskell é baseado no conceito de **mônadas**[4], estruturas de dados que permitem a execução sequencial de operações com efeitos colaterais. A `IO` é uma mônada especial que representa a sequência de ações IO. Cada ação IO é representada por um valor do tipo `IO a`, que encapsula uma descrição de uma ação que, quando executada pelo sistema, produz um valor do tipo `a` e pode afetar o mundo externo.

Para executar a sequência de ações IO, é necessário utilizar a função `main`, que é o ponto de entrada do programa. A função `main` recebe uma ação IO como argumento e executa esta ação, atualizando o estado do mundo exterior a cada passo.

[4] "*Mônadas em Haskell podem ser pensadas como descrições de computação combináveis. Cada mônada representa um tipo diferente de computação. Uma mônada pode ser executada a fim de realizar a computação por ela representada e produzir um valor como resultado*" (MALAQUIAS, 2012).

Por exemplo, para imprimir uma mensagem na tela, pode-se utilizar a função `putStrLn`, como feito nos exemplos anteriores. Essa função recebe uma `String` como parâmetro e retorna uma ação IO que imprime essa `String` na saída padrão. A assinatura da função `putStrLn` é **`putStrLn :: String -> IO ()`** (ZVON, s.d.).

Funções de Escrita

Algumas funções de escrita em Haskell:

- **`putChar`**: escreve um único caractere na saída padrão.
- **`putStr`**: escreve uma `String` na saída padrão.
- **`putStrLn`**: escreve uma `String` na saída padrão e, ao final, inclui uma quebra de linha.
- **`print`**: converte um valor em `String` e escreve a `String` na saída padrão, seguida de uma nova linha.

Embora não seja uma função padrão do Haskell, também é possível utilizar a função `printf` de forma muito parecida com a que se utiliza esta função nas linguagens C e C++. Para utilizá-la, é necessário importar o módulo `Text.Printf` (`import Text.Printf`). O exemplo a seguir, disponível também no arquivo arquivo aula-io-exemplo-4.hs, faz uso do `printf` para imprimir um valor em ponto flutuante com duas casas decimais:

```
import Text.Printf (printf)


main :: IO ()
main = do
   let a = 2.0 :: Float
   let b = 6.0 :: Float
   printf "O valor total é R$ %.2f\n" (a/b)
```

Neste exemplo, foi necessário definir o tipo de `a` e `b` para `Float` para que o compilador consiga determinar o tipo da expressão `a/b`. Essa definição do tipo é chamada de *anotação* e poderia ter sido feita em apenas um dos valores que também funcionaria. Como existem vários tipos que podem ser instâncias da classe `Fractional`, o compilador não consegue

determinar qual é o tipo mais apropriado para usar na divisão `2.0 / 6.0` se a anotação não for feita.

Se for preciso utilizar mais parâmetros na função `printf`, basta separá-los por espaço, na forma como eles são enumerados em Haskell. O arquivo aula-io-exemplo-5.hs, disponível no quadro abaixo, traz um exemplo com mais parâmetros no uso da função `printf`.

```
import Text.Printf (printf)


main :: IO ()
main = do
   let a = 1.0 :: Float
   let b = 2.0 :: Float
   printf "O valor de a é %f e o valor de b é %f\n" a b
```

No código acima, será exibido para o usuário o texto "`O valor de a é 1.0 e o valor de b é 2.0`".

Funções de Leitura

Algumas funções de leitura em Haskell:

- **`getLine`**: Lê uma linha de entrada do usuário como uma `String`.
- **`readLn`**: Lê uma linha de entrada do usuário como um valor de um tipo específico.
- **`getChar`**: Lê um caractere de entrada do usuário como um `Char`.
- **`readFile`**: Lê o conteúdo de um arquivo como uma `String`.
- **`read`**: Converte uma `String` em um valor de um tipo específico.
- **`reads`**: Converte uma `String` em uma lista de valores de um tipo específico.

As principais funções de leitura são `getLine`, útil para ler uma entrada de usuário como uma `String`, e `readLn`, usada para ler uma entrada e convertê-la em um valor de um tipo específico, como `Int` ou `Double`. O exemplo a seguir, disponível no arquivo aula-io-exemplo-6.hs, utiliza essas duas funções:

```
import Text.Printf (printf)
```

```
main :: IO ()
main = do
   putStr "Digite seu nome completo: "
   nome <- getLine
   putStr "Digite sua idade: "
   idade <- readLn :: IO Int
   putStr "Digite quantos reais você tem na carteira: "
   dinheiro <- readLn :: IO Double
   putStrLn ("Seu nome é " ++ nome)
   putStrLn ("Sua idade é " ++ show idade)
   printf "Você tem R$ %.2f na carteira\n" dinheiro
```

Neste arquivo há alguns pontos interessantes. É possível notar que, ao executar, um resultado semelhante a este é obtido:

```
$ ./aula-io-exemplo-6
Joao da Silva
30
0.00
Digite seu nome completo: Digite sua idade: Digite quantos reais você tem na carteira: Seu nome é Joao da Silva
Sua idade é 30
Você tem R$ 0.00 na carteira
```

Mas como isso acontece, se a primeira instrução da função é uma escrita?

O atraso na exibição das mensagens ocorre por causa do *buffering* da saída padrão do terminal, que armazena temporariamente o texto antes de mostrá-lo. O compilador e o *runtime* do Haskell garantem que as ações de leitura e escrita sejam executadas na ordem correta. Para que as mensagens apareçam imediatamente antes das leituras, é necessário limpar o *buffer* de saída, o que força a exibição do conteúdo. O código a seguir, disponível também no arquivo aula-io-exemplo-7.hs, garante que as mensagens sejam exibidas na ordem

desejada, corrigindo o efeito do *buffering*. Importante: isso não tem relação com a avaliação preguiçosa, mas com o funcionamento do sistema de entrada e saída.

```
import Text.Printf (printf)
import System.IO (hFlush, stdout)

main :: IO ()
main = do
   putStr "Digite seu nome completo: "
   hFlush stdout
   nome <- getLine
   putStr "Digite sua idade: "
   hFlush stdout
   idade <- readLn :: IO Int
   putStr "Digite quantos reais você tem na carteira: "
   hFlush stdout
   dinheiro <- readLn :: IO Double
   putStrLn ("Seu nome é " ++ nome)
   putStrLn ("Sua idade é " ++ show idade)
   printf "Você tem R$ %.2f na carteira\n" dinheiro
```

Outra possibilidade (arquivo aula-io-exemplo-8.hs):

```
import Text.Printf (printf)
import System.IO (hFlush, stdout)

main :: IO ()
main = do
   putStr "Digite seu nome completo: " >> hFlush stdout
   nome <- getLine
   putStr "Digite sua idade: " >> hFlush stdout
   idade <- readLn :: IO Int
    putStr "Digite quantos reais você tem na carteira: " >> hFlush
```

```
stdout
   dinheiro <- readLn :: IO Double
   putStrLn ("Seu nome é " ++ nome)
   putStrLn ("Sua idade é " ++ show idade)
   printf "Você tem R$ %.2f na carteira\n" dinheiro
```

Aqui utilizou-se o operador `>>` (chamado de `then`). Este operador realiza a composição de mônadas, combinando duas ações monádicas em uma única ação. No caso desta solução, o operador >> foi utilizado para executar duas ações monádicas em sequência, combinando a ação de imprimir uma mensagem na saída padrão (com a função `putStr`) com a ação de limpar o *buffer* da saída padrão (`hFlush stdout`), para garantir que a mensagem seja impressa antes que a ação de leitura (`getLine`) ocorra. Dessa forma, a sequência de comandos de escrita seguidos por leitura foi respeitada em todos os casos, assim como no exemplo anterior, mas utilizando o operador `>>`.

Conclusão

Haskell dispõe de um compilador que permite gerar arquivos executáveis, o que é útil para testar programas mais complexos e que sejam portáveis. Além disso, funções de entrada e saída em Haskell são modeladas com a mônada `IO`, permitindo a interação com o usuário e possibilitando a comunicação com o mundo exterior assim como feito nas principais linguagens de programação.

Exercícios

1) Crie um programa Haskell que solicite ao usuário para inserir o raio de um círculo e, em seguida, imprima a área correspondente desse círculo. Lembre-se de que a área de um círculo é calculada pela fórmula: $área = \Pi \times raio^2$. Utilize $3,14$ como o valor de Π.
2) Escreva um programa em Haskell que solicite ao usuário para digitar uma frase e, em seguida, conte o número de caracteres nessa frase, incluindo espaços. O programa

deve então imprimir o número total de caracteres. Não é necessário validar se o usuário de fato informou uma frase.

3) Escreva um programa em Haskell que solicite ao usuário para inserir uma temperatura em graus Celsius e, em seguida, converta-a para Fahrenheit. A fórmula de conversão é: $Fahrenheit = (Celsius \times 9/5) + 32$.
4) (**DESAFIO!**) Escreva um programa em Haskell que leia um arquivo de texto chamado `dados.dat` e conte o número de palavras nele. O programa deve então criar um novo arquivo chamado `palavras.txt` e escrever neste arquivo o número de palavras encontrado no arquivo de entrada, seguido pela lista de palavras únicas ordenadas alfabeticamente (com base na tabela ASCII), uma por linha. Por exemplo, se o arquivo de entrada contiver o texto "`Não acredite em duendes, eles mentem.`", o arquivo `palavras.txt` deve conter:

```
6
Não
acredite
duendes,
eles
em
mentem.
```

Referências

MALAQUIAS, José Romildo. Capítulo 13: Mônadas. Disponível em: http://www.decom.ufop.br/romildo/2012-1/bcc222/slides/13-monadas.pdf. Acesso em: 06 mai. 2024.

ZVON. putStrLn f. Disponível em: http://zvon.org/other/haskell/Outputprelude/putStrLn_f.html. Acesso em: 06 mai. 2024.