# Revisão de Haskell

Construção de compiladores I

## Objetivos

### Objetivos

- Revisar conceitos sobre mônadas e separação entre funções puras e impuras em Haskell.
- Finalização da implementação do compilador.

## Markdown

### Markdown

- Iniciamos nosso curso com o projeto de um mini-compilador de Markdown.
- Objetivo: revisar conceitos da linguagem Haskell

### Markdown

- Até o momento:
  - Fazemos o parsing de markdown para uma AST.
  - Definimos uma EDSL para gerar HTML.
  - Transformamos a AST em HTML usando a EDSL.
- O que falta?

### Markdown

- Interação com usuário
  - Receber nome de arquivos de entrada / saída.
  - Ler o arquivo de entrada.
  - Escrever o arquivo de resultado.

### Markdown

- I/O é uma forma de efeito colateral
  - Resultado não depende apenas dos argumentos de entrada do código.
- A priori, Haskell impõe que todas as funções devem ser **puras**.
  - Pura = sem efeitos colaterais

### Markdown

- Mas como fazer I/O ou outro tipo de efeito colateral em Haskell?
  - Devemos utilizar **mônadas**.

## Mônadas

### Mônadas

- Haskell é uma linguagem puramente funcional.
- Por padrão, todas as funções em Haskell são puras
  - Não possuem efeitos colaterais: I/O, estado, exceções.

### Mônadas

- Funções puras são garantidas de retornar o mesmo resultado para mesmas entradas.
  - Facilita o teste.
  - Facilita o raciocínio formal.

### Mônadas

- Porém, para um compilador, precisamos de entrada e saída.

## Mônadas

- Para lidar com efeitos colaterais, Haskell utiliza o conceito de mônada.
  - Mônadas são tipos que implementam a seguinte classe:

```
class Applicative m => Monad m where
   return :: a -> m a
   (>>=)  :: m a -> (a -> m b) -> m b
```

## Mônadas

- O operador `>>=` permite compor sequencialmente computações em uma mônada.
- Exemplo: ler um nome e imprimir uma mensagem de saudação.

```
ex :: IO ()
ex = getLine >>= \ s -> putStrLn $ "Hello " ++ s
```

## Mônadas

```
getLine >>= \ s -> putStrLn $ "Hello " ++ s
```

- Entendendo o código:
  - Primeiro usamos `getLine` para ler uma string do console.
  - O resultado é passado para a função

```
\ s -> putStrLn $ "Hello " ++ s
```

## Mônadas

- Uso de `>>=` prejudica a legibilidade do código.
- Solução: notação **do**

## Mônadas

- A notação `do` pode simplificar computações em mônadas.
  - Compilador traduz o `do` para usos de `>>=`.

```
ex1 :: IO ()
ex1 = do
     s <- getLine
     putStrLn $ "Hello " ++ s
```

## Mônadas

- Então, utilizaremos mônadas apenas para realizar I/O?
  - Sim!
- Toda a lógica será implementada por funções **puras**.
  - Não dependem de fatores externos ao programa.

## Mônadas

- Padrão: Functional core / imperative shell
  - Funcionalidade central implementada por funções puras.
  - Código puro "envolvido por uma casca" de funções em mônadas

## Mônadas

- Para leitura / escrita utilizaremos a biblioteca `System.IO`
- Tipos utilizados:

```
FilePath
Handle
data IOMode
   = ReadMode
   | WriteMode
   | AppendMode
   | ReadWriteMode
```

## Mônadas

- Para leitura / escrita utilizaremos a biblioteca `System.IO`
- Funções utilizadas:

```
openFile :: FilePath -> IOMode -> IO Handle
hClose :: Handle -> IO ()
stdin :: Handle
stdout :: Handle
hGetContents :: Handle -> IO String
hPutStrLn :: Handle -> String -> IO ()
```

### Mônadas

- Utilizamos as funções anteriores para leitura / escrita de dados usando I/O.
- Composição feita utilização notação **do**

### Mônadas

- Exemplo: Confirmar a reescrita de arquivos de saída.

```
confirmOverwrite :: FilePath -> IO Bool
confirmOverwrite path
  = do
      putStrLn $ "File:\n" ++ path ++ "\nexists. Confirm overwrite?"
      answer <- getLine
      case answer of
        "y" -> return True
        "n" -> return False
        _   -> do
                 putStrLn "Invalid answer! Please type y or n."
                 confirmOverwrite path
```

### Mônadas

- Com isso a implementação está concluída?
  - Não! Falta a *interação* com o usuário.
- Devemos receber argumentos de linha de comando e processá-los de maneira adequada.

## Interação em console

### Interação em console

- Qual o formato da entrada esperado pela ferramenta?
  - Devemos especificar o arquivo de entrada e o de saída
  - Se não for especificado, a saída deverá ser o console padrão.

### Interação em console

**Em princípio, podemos utilizar a função ~getArgs** IO [String]~.

- Analisamos a lista de `String` obtida.
- Validamos se os argumentos estão corretos.
- Emitir mensagens de erro / ajuda

### Interação em console

- Muitas tarefas...
- Melhor utilizar uma biblioteca especializada para isso.
  - Utilizaremos a biblioteca `optparse-applicative`.

### Interação em console

- Para usar essa biblioteca, devemos:
  - Definir tipos de dados para opções de entrada do programa
  - Definir o parser destas opções
  - Casamento de padrão sobre as opções

### Interação em console

- Representando o argumento de entrada

```
data Input
  = Stdin
  | FileInput FilePath
```

### Interação em console

- Representando o argumento de saída

```
data Output
  = Stdout
  | FileOutput FilePath
  deriving Show
```

### Interação em console

- Agrupando entrada e saída

```
data Options
  = Single Input Output
  deriving Show
```

### Interação em console

- Criando o parser usando `optparse-applicative`.
  - Lidando apenas com entrada por arquivos.

```
pInputFile :: Parser Input
pInputFile = FileInput <$> parser
  where
    parser =
      strOption
        ( long "input"
          <> short 'i'
          <> metavar "FILE"
          <> help "Input file"
        )
```

### Interação em console

- Criando o parser para entradas
- Função `optional`
  - Retorna `Nothing` em caso de erro de parsing.
  - Retorna o resultado no construtor `Just`.

```
pSingleInput :: Parser Input
pSingleInput =
  fromMaybe Stdin <$> optional pInputFile
```

### Interação em console

- Parser para saída segue o mesmo formato.

### Interação em console

- Definição do parser.

```
pSingle :: Parser Options
pSingle =
  Single <$> pSingleInput <*> pSingleOutput
```

### Interação em console

- Para executar o parser, precisamos adicionar informações para mensagens de ajuda.
- Para isso, devemos criar um valor de tipo `ParserInfo`

```
opts :: ParserInfo Options
opts = info (pSingle <**> helper)
            (fullDesc                              <>
             header "BCC328 - Markdown compiler" <>
             progDesc "Simplified Markdown compiler")
```

### Interação em console

- Com isso, a obtenção de quais opções foram passadas é dada por:

```
parse :: IO Options
parse = execParser opts
```

### Interação em console

- O parser construído automatiza tratamento de erros em opções e mensagens de ajuda.

### Interação em console

- Valores to tipo `Option` são usados para criar `Handles`:

```
createHandles :: Options -> IO (String, Handle, Handle)
createHandles (Single inp out)
  = do
      (title, inpHandle) <- createInputHandle inp
      outHandle <- createOutputHandle out
      return (title, inpHandle, outHandle)
```

### Interação em console

- Criação de `Handle` de entrada

```
createInputHandle :: Input -> IO (String, Handle)
createInputHandle Stdin = return ("", stdin)
createInputHandle (FileInput path)
  = (,) path <$> openFile path ReadMode
```

### Interação em console

- Criação de `Handle` de saída é similar.

### Interação em console

- Criamos um parser para ler as entradas do compilador.
- O que falta?
  - Criarmos um pipeline que irá "ligar" a casca de IO com o núcleo do compilador.

### Interação em console

- Inicialização do pipeline
  - Criação de handles de entrada e de saída

```
startPipeline :: String -> Handle -> Handle -> IO ()
startPipeline title inpHandle outHandle
  = do
      content <- hGetContents inpHandle
      res <- pipeline (title_ title) content
      hPutStrLn outHandle res
```

### Interação em console

- Definição do pipeline

```
pipeline :: Head -> String -> IO String
pipeline title content
  = case frontEnd content of
      Left err -> putStrLn err >> exitFailure
      Right ast -> return $ render $ backEnd title ast
```

### Interação em console

- Definição da função `main`

```
main :: IO ()
main = do
  options <- parse
  (title,inpHandle,outHandle) <- createHandles options
  startPipeline title inpHandle outHandle
  hClose inpHandle
  hClose outHandle
```

## Concluindo

### Concluindo

- Apresentamos o projeto de um compilador de um versão simplificada de Markdown para HTML.
- Compilador segue o padrão: functional core / imperative shell

## Exercícios

### Exercícios

- Estenda a implementação do compilador desenvolvido para prover suporte a produção de slides LaTeX usando beamer.

### Exercícios

- Para isso você deverá:
  - Criar uma EDSL para slides beamer.
  - Traduzir documentos markdown para a EDSL
    - Considere que cada header de level 1 é um novo slide.
  - Modificar a "casca" de IO para incluir a opção de produção de slides.

### Exercícios

- Estender a estrutura de testes para validar a sua implementação.

### Exercícios

- Você pode considerar útil a documentação da biblioteca optparse-applicative.