# Revisão de Haskell

Construção de compiladores I

## Objetivos

### Objetivos

- Revisar conceitos de programação funcional utilizando Haskell.
- Apresentar a aplicação dos conceitos para construir um compilador protótipo.

## Markdown

### Markdown

- Formato amplamente utilizado para representação de documentos.
- Popularizado pela adoção de plataformas como Github e ferramentas como o pandoc.

### Markdown

- Componentes do compilador.
  - Analisadores léxico e sintático: produção da AST do formato.
  - Gerador de código: produção de HTML.

### Markdown

- Primeiro passo: criar uma biblioteca para criação de HTML bem formado.
- Faremos isso criando uma EDSL para HTML

# EDSLs

## EDSLs

- Em Haskell, é bem comum desenvolvermos EDSLs
  - **E** mbedded **D** omain **S** pecific **L** anguage
- DSLs estão em toda parte
  - makefiles, sql, dot, html...

## EDSLs

- Mas porque "Embedded"?
  - EDSLs utilizam todo os recursos da linguagem de origem.
  - Vantagem: não é necessário construir um compilador do "zero".

## EDSLs

- Mas qual o problema que a EDSL de Html vai resolver?
- A princípio, podemos produzir Html usando operações sobre strings...

## EDSLs

- Problemas de uso de strings:
  - Quem garante que construímos <title><> somente dentro de um <head> usando strings?

## EDSLs

- Estratégia de projeto: Uma biblioteca de combinadores
  - Funções para elementos primitivos.
  - Funções para combinar elementos HTML

# EDSL para Html

## EDSL para Html

- Definição de tipos:
  - Tipos encapsulam strings correspondente a estrutura de HTML.

```
newtype Html
  = Html String

newtype Structure
  = Structure String

newtype Content
  = Content String

newtype Head
  = Head String
```

## EDSL para HTML

- Definindo um novo tipo
  - `newtype` permite definir tipos com um único construtor.

```
newtype Html
  = Html String
```

## EDSL para HTML

- Construtores são funções.

```
Html :: String -> Html
```

## EDSL para HTML

- Gerando o código HTML:
  - Uso de casamento de padrão

```
render :: Html -> String
render (Html str) = str
```

## EDSL para HTML

- Qual a vantagem disso? Porque não strings?
  - Vamos "ocultar" a estrutura de tipos.
  - Manipulação apenas por funções.
  - Funções garantem a geração de HTML válido.

## EDSL para Html

- Criando uma tag qualquer:

```
el :: String -> String -> String
el tag content =
  "<" <> tag <> ">" <> content <> "</" <> tag <> ">"
```

## EDSL para Html

- Mas o que é a função `<>`?

```
class Semigroup a where
  (<>) :: a -> a -> a

class Semigroup a => Monoid a where
  mempty :: a
```

## EDSL para Html

- `Semigroup` e `Monoid` são classes de tipos
  - Duas operações definidas: `(<>)` e `mempty`.
- Requisitos
  - `(<>)` deve ser associativo.
  - `mempty` deve ser elemento neutro de `(<>)`.

## EDSL para Html

- Listas são instância de `Semigroup` e `Monoid`:
  - Strings são apenas listas de caracteres.

```
instance Semigroup [a] where
  xs <> ys = xs ++ ys

instance Monoid [a] where
   mempty = []
```

### EDSL para Html

- A partir da função `el` podemos criar outras tags.
- Código para definir a tag `<b> ... </b>`:

```
getContentString :: Content -> String
getContentString (Content str) = str

b_ :: Content -> Content
b_ content =
  Content $ el "b" (getContentString content)
```

### EDSL para Html

- Demais tags seguem estrutura similar.

```
i_ :: Content -> Content
i_ content =
  Content $ el "i" (getContentString content)
```

### EDSL para HTML

- Como usar essa DSL para construir HTML?

### EDSL para HTML

- Construir esse HTML simples:

```
<html>
  <head>
    <title>My title</title>
  </head>
  <body>
    <h1>Heading</h1>
    <p>Paragraph #1</p>
```

```
    <p>Paragraph #2</p>
  </body>
</html>
```

## EDSL para HTML

- Representação na EDSL:

```
myhtml :: Html
myhtml =
  html_
    (title_ "My title")
    ((h_ 1 (txt_ "Heading"))    <>
     (p_ $ txt_ "Paragraph #1") <>
     (p_ $ txt_ "Paragraph #2"))
```

## EDSL para HTML

- Elementos da DSL são combinados utilizando `Monoid`.

```
instance Semigroup Head where
  (Head h1) <> (Head h2) = Head (h1 <> h2)

instance Monoid Head where
  mempty = Head ""
```

## EDSL para HTML

- Tipos `Structure` e `Content` também são monóides.

## EDSL para HTML

- Definida no módulo `Markup.Printer.Html.Internal`.
- Esse módulo é encapsulado por `Markup.Printer.Html` que exporta apenas funções para manipular os tipos `Html, Structure, Content`.

## EDSL para HTML

- Definição do cabeçalho

```
module Markup.Printer.Html ( Html
                           , Head
                           , title_
                           , Structure
                           , html_
                           , h_
                           , p_
                           , ul_
                           , code_
                           , Content
                           , txt_
                           , b_
                           , i_
                           , render
                           )  where

import Markup.Printer.Html.Internal
```

### EDSL para HTML

- Seguindo esse padrão, manipulação de valores `HTML` só pode ser feito usando funções da EDSL.

## De Markdown para HTML

### De Markdown para HTML

- Agora que representamos HTML, vamos modelar a entrada do compilador.

### De Markdown para HTML

- Primeiro, devemos representar a estrutura de Markdown como um tipo Haskell

### De Markdown para HTML

- Documentos são listas de estruturas.

```
type Document = [Structure]
```

## De Markdown para HTML

- Estruturas são dos seguintes tipos:
  - Cabeçalhos
  - Parágrafos
  - Listas não ordenadas
  - Código fonte

## De Markdown para HTML

- Representação em Haskell

```
data Structure
  = Heading Natural String
  | Paragraph String
  | UnorderedList [String]
  | CodeBlock [String]
  deriving Show
```

## De Markdown para HTML

- Exemplos de representação: cabeçalhos e parágrafos.

```
 * Cabeçalho

   Revisão de Haskell
```

## De Markdown para HTML

- Representação em Haskell:

```
example1 :: Document
example1 =
  [ Heading 1 "Cabeçalho"
  , Paragraph "Revisão de Haskell"
  ]
```

## De Markdown para HTML

- Exemplos de representação: Listas e código fonte.

```
- Como compilar código Haskell:
>  ghc Main.hs
> [1 of 1] Compiling Main ( Main.hs, Main.o )
> Linking Main ...
Com isso o GHC irá criar os seguintes arquivos:
- Main.hi: Arquivo de interface.
- Main.o: Código objeto antes de link-edição.
- Main (ou Main.exe no Windows): executável
```

## De Markdown para HTML

- Representação em Haskell:

```
example2 :: Document
example2 =
  [ UnorderedList ["Como compilar código Haskell:"]
  , CodeBlock [ " ghc Main.hs"
              , "[1 of 1] Compiling Main ( Main.hs, Main.o )"
              , "Linking Main ..."]
  , Paragraph "Com isso o GHC irá criar os seguintes arquivos:"
  , UnorderedList [ "Main.hi: Arquivo de interface."
                  , "Main.o: Código objeto antes de link-edição."
                  , "Main (ou Main.exe no Windows): executável"]
  ]
```

## De Markdown para HTML

- Gerando HTML a partir de um documento.

```
translate :: AST.Document -> HTML.Structure
translate = foldMap translateStructure

foldMap :: Monoid m => (a -> m) -> [a] -> m
foldMap f = foldr ((<>) . f) mempty
```

## De Markdown para HTML

- Gerando HTML a partir de estruturas.

```
import qualified Markup.Language.Syntax as AST
import qualified Markup.Printer.Html as HTML

translateStructure :: AST.Structure -> HTML.Structure
translateStructure (AST.Heading n txt)
  = HTML.h_ n $ HTML.txt_ txt
translateStructure (AST.Paragraph p)
  = HTML.p_ $ HTML.txt_ p
translateStructure (AST.UnorderedList ds)
  = HTML.ul_ $ map (HTML.p_ . HTML.txt_) ds
translateStructure (AST.CodeBlock ds)
  = HTML.code_ (unlines ds)
```

# Concluindo

## Concluindo

- Nesta aula revisamos alguns conceitos da linguagem Haskell:
  - Definição de tipos de dados.
  - Definição de funções por casamento de padrão.
  - Funções de ordem superior

## Concluindo

- Próxima aula
  - Parsing e mônadas.