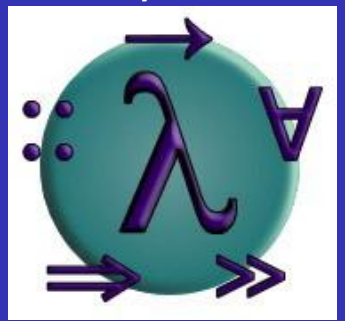
Programação Funcional



Capítulo 21

Parsers

José Romildo Malaquias

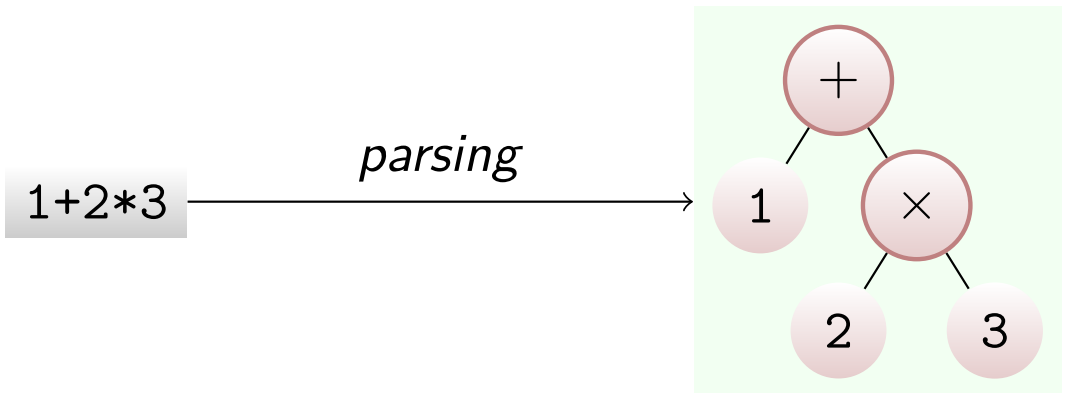
Departamento de Computação
Universidade Federal de Ouro Preto

2018.2

Tópicos

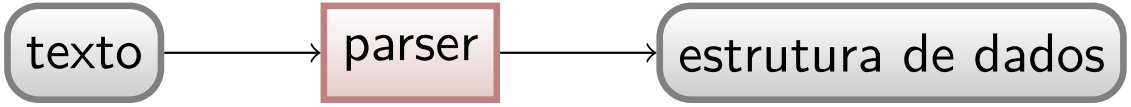
Análise sintática ou *parsing*

- **Análise sintática** (também conhecida pelo termo em inglês ***parsing***) é o processo de analisar um texto para determinar sua estrutura lógica.
- O texto pode descrever, por exemplo, uma expressão aritmética, um programa de computador, ou um banco de dados.
- A análise sintática transforma o texto em uma estrutura de dados (em geral uma árvore) que
  - captura a hierarquia implícita no texto, e
  - é conveniente para processamento posterior.
- Por exemplo, a análise sintática de uma expressão aritmética pode produzir uma árvore com constantes nas folhas, e operadores nos nós internos.



Parsers

- **Analisador sintático** ou ***parser*** é um objeto que faz análise sintática de um texto para determinar a sua estrutura lógica.
- Um parser transforma uma string em um valor de um determinado tipo.



Tópicos

Como representar um parser em Haskell?

Desejamos:

- Definir o **tipo polimórfico**

```
Parser a
```

  ou seja, o tipo dos parsers que transformam uma string em um valor do tipo `a`.

- Definir uma **função para aplicar um parser** a uma string:

```
parse :: Parser a -> String -> a
```

- Definir **parsers básicos**.
- Definir **combinadores de parsers**, que permitem construir novos parsers usando parsers mais simples.

Um parser é uma função

- Um parser pode ser representado por uma **função** que recebe uma string e retorna um valor que é construído a partir da string.

```
type Parser a = String -> a
```

- Como um parser é uma função, para aplicá-lo basta aplicar a função diretamente à string de entrada:

```
parse :: Parser a -> String -> a
parse p s = p s
```

Um parser é uma função (cont.)

- **Exemplo**:
  parser para datas no formato: `DD/MM/AAAA`

```haskell
data Data = MkData Int Int Int
            deriving (Show)
```

```haskell
import Data.Char (isDigit, digitToInt)

pData :: Parser Data

pData [d1, d2, '/', m1, m2, '/', a1, a2, a3, a4]
  | all isDigit [d1, d2, m1, m2, a1, a2, a3, a4] =
      MkData d m a
  where
    d = 10 * digitToInt d1 + digitToInt d2
    m = 10 * digitToInt m1 + digitToInt m2
    a = 1000 * digitToInt a1 + 100 * digitToInt a2 +
        10 * digitToInt a3 + digitToInt a4
```

Um parser é uma função (cont.)

```
parse pData "18/11/2011"       ⇝ MkData 18 11 2011
parse pData "18/11/2011 13:30" ⇝ erro
parse pData "8/1/2012"         ⇝ erro
parse pData "d5/m9/2012"       ⇝ erro
```

O tipo dos parsers deve ser um novo tipo

- A definição de tipo dada

```
type Parser a = String -> a
```

é inadequada porque queremos um novo tipo, distinto de todos os outros tipos, porém isomorfo ao tipo das funções de `String` em `a`.

- Definindo o tipo dos parsers com `newtype`:

```
newtype Parser a = MkP (String -> a)
```

Usamos `newtype` para declarar um tipo novo (único) similar a um tipo já existente, porém incompatível com ele. O compilador pode otimizar o código substituindo-o pelo tipo já existente.

- A função que aplica um parser a uma string pode ser definida como:

```
parse :: Parser a -> String -> a
parse (MkP fun) s = fun s
```

O tipo dos parsers deve ser um novo tipo (cont.)

- **Exemplo**:
  parser para datas no formato: `DD/MM/AAAA`

```
pData :: Parser Data
pData = MkP f
  where
    f [d1, d2, '/', m1, m2, '/', a1, a2, a3, a4]
      | all isDigit [d1, d2, m1, m2, a1, a2, a3, a4] =
          MkData d m a
      where
        d = 10 * digitToInt d1 + digitToInt d2
        m = 10 * digitToInt m1 + digitToInt m2
        a = 1000 * digitToInt a1 + 100 * digitToInt a2 +
            10 * digitToInt a3 + digitToInt a4
```

```
parse pData "18/11/2011"       ⇝ MkData 18 11 2011
parse pData "18/11/2011 13:30" ⇝ erro
parse pData "8/1/2012"         ⇝ erro
parse pData "d5/m9/2012"       ⇝ erro
```

Um parser pode não usar toda a entrada

- A última definição dada para o tipo dos parsers

```
newtype Parser a = MkP (String -> a)
```

  ainda não é adequada, pois ela não prevê a possibilidade de sequenciamento de parsers.

- Um parser pode usar apenas parte (um prefixo) da string dada, devendo então retornar:
  - o valor construído a partir do prefixo, e
  - o sufixo da string que não foi consumido.

  Assim o resultado da função pode ser representado por um par.

```
newtype Parser a = MkP (String -> (a, String))
```

- Isto permite a outro parser continuar a análise usando o restante da string (sufixo) que não foi utilizado.
- Adaptando o tipo da função que aplica um parser a uma string:

```
parse :: Parser a -> String -> (a, String)
parse (MkP fun) s = fun s
```

Um parser pode não usar toda a entrada (cont.)

- **Exemplo**:
  parser para datas no formato: `DD/MM/AAAA`

```
pData :: Parser Data
pData = MkP f
  where
    f (d1:d2:'/':m1:m2:'/':a1:a2:a3:a4:resto)
      | all isDigit [d1,d2,m1,m2,a1,a2,a3,a4]
          = (MkData d m a, resto)
      where
        d = 10 * digitToInt d1 + digitToInt d2
        m = 10 * digitToInt m1 + digitToInt m2
        a = 1000 * digitToInt a1 + 100 * digitToInt a2 +
            10 * digitToInt a3 + digitToInt a4
```

```
parse pData "18/11/2011"       ⇝ (MkData 18 11 2011, "")
parse pData "18/11/2011 13:30" ⇝ (MkData 18 11 2011, " 13:30")
parse pData "8/1/2012"         ⇝ erro
parse pData "d5/m9/2012"       ⇝ erro
```

Um parser pode suceder ou falhar

- Um parser pode suceder ou falhar ao ser aplicado a uma string, uma vez que pode não ser possível construir um valor do tipo desejado a partir da string usando o parser.
- Assim é desejável que o resultado do parser indique a falha ou o sucesso, juntamente com o valor resultante quando houver sucesso.
- Para indicar sucesso ou falha podemos usar o construtor de tipo `Maybe` do prelúdio:

```
data Maybe a = Nothing | Just a
```

- Portanto o tipo dos parsers deve refletir esta possibilidade:

```
newtype Parser a = MkP (String -> Maybe (a, String))
```

- A função que aplica um parser a uma string deve ter o seu tipo adaptado a esta nova definição:

```
parse :: Parser a -> String -> Maybe (a, String)
parse (MkP fun) s = fun s
```

ou de forma mais concisa:

```
parse (MkP fun) = fun
```

Um parser pode suceder ou falhar (cont.)

- **Exemplo**:
  parser para datas no formato: `DD/MM/AAAA`

```
pData :: Parser Data
pData = MkP f
  where
    f (d1:d2:'/':m1:m2:'/':a1:a2:a3:a4:resto)
      | all isDigit [d1, d2, m1, m2, a1, a2, a3, a4]
          = Just (MkData d m a, resto)
      where
        d = 10 * digitToInt d1 + digitToInt d2
        m = 10 * digitToInt m1 + digitToInt m2
        a = 1000 * digitToInt a1 + 100 * digitToInt a2 +
            10 * digitToInt a3 + digitToInt a4
    f _ = Nothing
```

```
parse pData "18/11/2011"       ⇝ Just (MkData 18 11 2011, "")
parse pData "18/11/2011 13:30" ⇝ Just (MkData 18 11 2011, " 13:30")
parse pData "8/1/2012"         ⇝ Nothing
parse pData "d5/m9/2012"       ⇝ Nothing
```

Tópicos

Parser básico: `item`

Um parser muito simples é aquele que obtém o primeiro caracter da string de entrada.

```
item :: Parser Char
item = MkP ( \s -> case s of
                     x:xs -> Just (x, xs)
                     ""   -> Nothing
           )
```

Exemplos:

```
parse item "6123"     ⇝ Just ('6', "123")
parse item "bom dia"  ⇝ Just ('b', "om dia")
parse item "-89"      ⇝ Just ('-', "89")
parse item ""         ⇝ Nothing
```

Tópicos

Combinando de parsers

- Parsers mais simples podem ser combinados de diferentes maneiras para formar parsers mais complexos.
- **Combinadores de parsers** são funções que recebem parsers como argumentos e produzem novos parsers a partir deles.
- Nos tópicos seguintes vamos aprender alguns combinadores de parsers interessantes.

Parser com teste

Podemos fazer um parser que aplica um parser mais simples e verifica se o seu retorno satisfaz uma determinada propriedade.

```
satisfy :: (a -> Bool) -> Parser a -> Parser a
satisfy test p =
  MkP ( \s -> case parse p s of
                Just (x, s') | test x -> Just (x, xs)
                _ -> Nothing
      )
```

Exemplos:

```
parse (satisfy isDigit item) "6123"     ⇝ Just ('6', "123")
parse (satisfy isAlpha item) "bom dia"  ⇝ Just ('b', "om dia")
parse (satisfy (== '-') item) "-17"     ⇝ ('-', "17")
parse (satisfy isDigit item) "-89"      ⇝ Nothing
parse (satisfy isDigit item) ""         ⇝ Nothing
```

Parser de caracter

Podemos fazer um parser que sucede se e somente se o próximo caracter for igual ao caracter dado.

```
char :: Char -> Parser Char
char x = satisfy (==x) item
```

Exemplos:

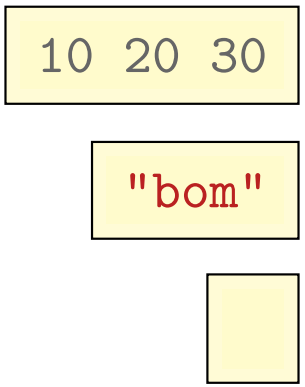
```
parse (char '6') "6123"     ⇝ Just ('6', "123")
parse (char 'b') "bom dia"  ⇝ Just ('b', "om dia")
parse (char '-') "-17"      ⇝ ('-', "17")
parse (char '+') "-89"      ⇝ Nothing
parse (char '*') ""         ⇝ Nothing
```

Valores encapsulados

- Em muitas aplicações é comum a necessidade de se realizar operações com valores *contidos* em estruturas de dados ou calculados em alguma forma de computação. Dizemos que estes valores estão **encapsulados**. Informalmente dizemos que os valores encapsulados estão *dentro de uma* ***caixinha***. Se a estrutura não contem nenhum valor dizemos que *a caixnha está vazia*.

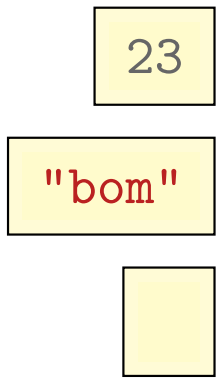
Valores encapsulados (cont.)

- São exemplos de valores encapsulados:
  - os elementos de uma lista, tais como
    - `[10, 20, 30]`
    - `["bom"]`
    - `[]`



Valores encapsulados (cont.)

- o valor opcional em uma estrutura *maybe*, tal como
  - `Just 23`
  - `Just "bom"`
  - `Nothing`



Valores encapsulados (cont.)

- o retorno de uma ação de entrada e saída
  - `getLine` (retorna a próxima linha da entrada padrão)

```
"Pedro Paulo"
```

  - `getArgs` (retorna os argumentos da linha de comando)

```
["-c", "prog.o"]
```

  - `return True` (retorna o valor `True`)

```
True
```

Vamos aprender como operar com valores encapsulados de forma generalizada e abstrata.

Aplicação de função

`f` — `x` — ⇝ — `f x`

- Já sabemos como aplicar uma função em valores simples:

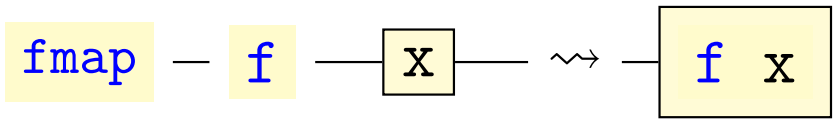
```
even 2     ⇝ True
abs (7-9)  ⇝ 2
```

- Podemos também usar o operador binário `$` (precedência 0, associativo à direita) e eliminar alguns parênteses desnecessários:

```
even $ 2   ⇝ True
abs $ 7-9  ⇝ 2
```

Funtores

- Quando queremos aplicar uma função a valores encapsulados, usamos a função `fmap`.
- O resultado da aplicação da função também será encapsulado.
- As estruturas e computações que possuem `fmap` são chamadas de **funtores**.



Funtores (cont.)

- O operador binário `<$>`, com precedência 4 e associatividade à esquerda, é idêntico à função `fmap`.

```
fmap even (Just 2)       ⇝ Just True
fmap abs [7-4, 0, 5-1]   ⇝ [3, 0, 4]
fmap length getLine      ⇝  ação que extrai e retorna o tamanho da p
fmap sqrt Nothing        ⇝ Nothing
fmap sin []              ⇝ []
even <$> return 7        ⇝  ação que retorna False
odd <$> [2, 3, 4]        ⇝ [False, True, False]
```

Funtores (cont.)

- `fmap` é introduzida na classe `Functor` da biblioteca padrão:

```
class Functor f where
  fmap :: (a -> b) -> f a -> f b
```

Funtores (cont.)

- Uma estrutura **maybe** é um funtor:

```
instance Functor Maybe where
  fmap _ Nothing  = Nothing
  fmap f (Just x) = Just (f x)
```

Funtores (cont.)

- Uma estrutura **lista** também é um funtor:

```
instance Functor [] where
  fmap _ []     = []
  fmap f (x:xs) = f x : fmap f xs
```

Funtores (cont.)

- Uma **ação de entrada e saída** também é um funtor:

```
instance Functor IO where
  fmap f acao = do x <- acao
                   return (f x)
```

Funtores (cont.)

- Um **parser** também é um funtor:

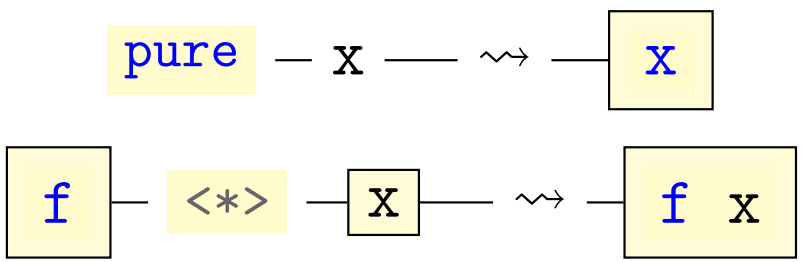
```
instance Functor Parser where
  fmap f p = MkP ( \s -> case parse p s of
                           Just (x, s') -> Just (f x, s')
                           Nothing -> Nothing
                 )
```

- Exemplos:

```
parse (fmap isDigit item) "5 + 3"     ⇝ Just (True, " + 3")
parse (fmap toUpper item) "f(x + 1)" ⇝ Just ('F', "(x + 1)")
parse (digitToInt <$> item) "2018"    ⇝ Just (2, "018")
parse (digitToInt <$> item) ""        ⇝ Nothing
```

Funtores aplicativos

- Em algumas situações queremos aplicar uma função a um valor onde ambos estão encapsulados.
- O resultado da aplicação também será encapsulado.
- A função `<*>` permite fazer tal aplicação.
- Ele é um operador binário associativo à esquerda com precedência 4.
- As estruturas e computações que possuem `<*>` são chamadas de **funtores aplicativos**.
- Elas possuem também uma função chamada `pure` para encapsular um dado valor.



Funtores aplicativos (cont.)

```
pure even <*> Just 2                ⇝ Just True
pure abs <*> [7-4, 0, 5-1]          ⇝ [3, 0, 4]
pure (==) <*> Just 2 <*> Just 3     ⇝ Just False
(==) <$> Just 2 <*> Just 3          ⇝ Just False
div <$> Just 2 <*> Nothing          ⇝ Nothing
max <$> Nothing <*> Just "belo"     ⇝ Nothing
elem <$> Nothing <*> Nothing        ⇝ Nothing
pure (abs) <*> [8, -5, -1]          ⇝ [8, 5, 1]
[(<0)] <*> [8, -5, -1]              ⇝ [False, True, True]
[(<0), even] <*> [8, -5, -1]        ⇝ [False, True, True, True, Fa
(+) <$> readLn <*> readLn           ⇝  ação que obtém duas linhas e r
```

Funtores aplicativos (cont.)

- As funções `pure` e `(<*>)` são introduzidas na classe `Applicative` da biblioteca padrão:

```
class (Functor f) => Applicative f where
  pure  :: a -> f a
  (<*>) :: f (a -> b) -> f a -> f b
```

Funtores aplicativos (cont.)

- Uma estrutura **maybe** é um funtor aplicativo:

```
instance Applicative Maybe where
  pure x = Just x

  Nothing <*> _                 = Nothing
  Just f  <*> maybeSomething = fmap f maybeSomething
```

Funtores aplicativos (cont.)

- Uma estrutura **lista** também é um funtor aplicativo:

```
instance Applicative [] where
  pure x = [x]

  fs <*> xs = [ f x | f <- fs, x <- xs ]
```

Funtores aplicativos (cont.)

- Uma **ação de entrada e saída** também é um funtor:

```
instance Applicative IO where
  pure x = return x

  acao1 <*> acao2 = do f <- acao1
                       x <- acao2
                       return (f x)
```

Funtores aplicativos (cont.)

- Um **parser** também é um funtor aplicativo:

```
instance Applicative Parser where
  pure x = MkP ( \s -> Just (x, s) )

  pf <*> px = MkP ( \s -> case parse pf s of
                            Just (f, s') -> fmap f px
                            Nothing -> Nothing
                  )
```

- Exemplos:

```
parse (pure (+) <*> fmap digitToInt <*> fmap digitToInt)
      "53fim"
⇝ Just (8, "fim")
```

Mônadas

- Podemos compor duas estruturas ou comptações que são funtores aplicativos sequencialmente, de forma que o valor encapsulado na primeira pode ser usado para montar a segunda do sequenciamento.
- Ou seja, a segunda estrutura ou computação depende dos valores encapsulados na primeira.
- Tais estruturas ou computações são chamadas de **mônadas**.

Mônadas (cont.)

- O sequenciamento monádico pode ser obtido através de funções definidas na classe `Monad` da biblioteca padrão:
  - `>>=` é um operador binário infixo associativo à esquerda e com precedência 1, que combina sequencialmente duas estruturas ou computações
    - o primeiro operando é a primeira estrutura ou computação
    - o segundo operando é *uma função* que, quando aplicada a um valor, resulta na segunda estrutura ou computação da sequência
    - o resultado da operaçao é a segunda estrutura ou computação, sendo que a segunda estrutura ou computação é obtida pela aplicação do segundo operando no valor encapsulado pelo primeiro operando

Mônadas (cont.)

- `>>` é um operador binário infixo associativo à esquerda e com precedência 1, que combina sequencialmente duas estruturas ou computações
  - o primeiro operando é a primeira estrutura ou computação
  - o segundo operando é a segunda estrutura ou computação da sequência
  - o resultado da operaçao é a segudna estrutura ou computação

  Observe que o valor encapsulado na primeira estrutura ou computação é completamente ignorado.

Mônadas (cont.)

- `return` é uma função que encapsula um valor. Na grande maioria dos casos coincide com a função `pure`.

Mônadas (cont.)

- Exemplos:

```
return 5 :: [Int]
⇝ [5]

putStr "nome: " >> getLine
⇝ ação que exibe uma mensagem e retorna a próxima linha

getLine >>= \x -> print (length x)
⇝ ação que lê a próxima linha e exibe o seu tamanho

getLine >>= (print .  length)
⇝ ação que lê a próxima linha e exibe o seu tamanho

readLn >>= \x -> readLn >>= \y -> return (x+y)
⇝ ação que lê a próxima linha e exibe o seu tamanho
```

Mônadas (cont.)

- As funções `return`, `(>>=)` e `(>>)` são introduzidas na classe `Monad` da biblioteca padrão:

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

  return = pure

  p >> q = p >>= (\_ -> q)
```

Mônadas (cont.)

- Uma estrutura *maybe* é uma mônada

```
instance Monad Maybe where
  Nothing >>= _ = Nothing
  Just x  >>= f = f x
```

Mônadas (cont.)

- Uma estrutura lista também é uma mônada:

```
instance Monad [] where
  xs >>= f = [ y | x <- xs, y <- f x ]
```

Mônadas (cont.)

- Uma ação de entrada e saída também é uma mônada

```
instance Monad IO where
  return = operação de um tipo abstrato

  acao1 >>= acao2 operação de um tipo abstrato
```

Fim