Estudo de caso - Parsing

Programação Funcional

Prof. Rodrigo Ribeiro

Setup

```
import Data.Char

main :: IO ()
main = return ()
```

Parsing

- Verificar se uma sequência de entrada possui uma estrutura de interesse
- Problema central em computação.

Objetivo

- Utilizar functores aplicativos para projetar uma biblioteca de parsing.

Parser?

- Definindo um parser:

```
newtype Parser s a
  = Parser {runParser :: [s] -> [(a,[s])]}
```

Parser

- O tipo de um parser é uma função

```
[s] -> [(a,[s])]
```

- s : Tipo do símbolo de entrada. Normalmente, `Char`.
- a : Tipo do resultado produzido ao se processar a entrada `[s]`.

Parser

- O tipo `Parser` utiliza uma abordagem conhecida como list of success
  - Falhas representadas como uma lista vazia de resultados.
  - Um resultado: determinismo.
  - Mais de um resultado: backtracking implícito (não determinismo).

Exemplo

- Um parser para um símbolo.

```
symbol :: Eq s => s -> Parser s s
symbol s
  = Parser (\ inp -> case inp of
                       [] -> []
                       (x : xs) -> if x == s
                                   then [(x,xs)]
                                   else [])
```

Parser simples

- Processando o caractere ‘a’:

```
a :: Parser Char Char
a = symbol 'a'
```

- Executando no interpretador

```
Main*> runParser a "a"
[('a',"")]
```

Strings

- Processando uma sequência de símbolos

```
token :: Eq s => [s] -> Parser s [s]
token s
  = Parser (\ inp -> if s == (take n inp)
                     then [(s, drop n inp)]
                     else [])
    where
      n = length s
```

SAT

- O parser sat processa um símbolo que atende uma certa condição.

```
sat :: (s -> Bool) -> Parser s s
sat p = Parser (\ inp -> case inp of
                           [] -> []
                           (x : xs) -> if p x
                                       then [(x,xs)]
                                       else [])
```

Digit Parser

- Podemos usar sat para construir um parser para processar um dígito da entrada.

```
digitChar :: Parser Char Char
digitChar = sat isDigit
```

Digit Parser

- Como converter o carater em seu dígito correspondente?
- Para isso, precisamos aplicar uma função sobre o resultado de um parser.
- Isto é, fazer o tipo `Parser s a` uma instância de `Functor`.

Functor

```
instance Functor (Parser s) where
   fmap f (Parser p)
      = Parser (\ inp -> [(f x, xs) | (x,xs) <- p inp])
```

Digit Parser

- Usando functor, podemos converter o caractere em dígito facilmente.

```
digit :: Parser Char Int
digit = f <$> digitChar
        where
          f c = ord c - ord '0'
```

Outros Parsers

- `succeed` é um parser que nunca falha.
- `failure` é um parser que sempre falha.

```
succeed :: a -> Parser s a
succeed v = Parser (\ inp -> [(v,inp)])

failure :: Parser s a
failure = Parser (\ _ -> [])
```

Combinadores

- Escolha entre dois parsers:

```
infixr 4 <|>

(<|>) :: Parser s a -> Parser s a -> Parser s a
(Parser p) <|> (Parser q)
   = Parser (\ inp -> p inp ++ q inp)
```

Combinadores

- Execução sequencial: Modelamos esse fato usando <*>.

```
instance Applicative (Parser s) where
   pure = succeed
   (Parser p) <*> (Parser q)
     = Parser (\ inp -> [(f x, xs) | (f, ys) <- p inp
                                   , (x, xs) <- q ys])
```

Exemplo

- Reconhecendo “ab” ou “ba”

```
ex1 :: Parser Char String
ex1 = token "ab" <|> token "ba"
```

Exemplo

- Reconhecendo parêntesis balanceados. Strings pertencentes a gramática:

$$S \rightarrow (S)\,S \mid \epsilon$$

Modelando

- Usaremos um tipo de dados para representar palavras de parêntesis balanceados

```
data Paren = Match Paren Paren | Empty

instance Show Paren where
   show Empty = ""
   show (Match p p') = "(" ++ show p ++ ")" ++ show p'
```

Parser para Paren

```
open :: Parser Char Char
open = symbol '('

close :: Parser Char Char
close = symbol ')'

parens :: Parser Char Paren
parens = (f <$> open <*> parens <*> close <*> parens)
         <|> succeed Empty
         where
           f _ p _ p' = Match p p'
```

Exercício

- Implemente um parser que calcula o número de pares de parêntesis balanceados em uma string de entrada.

```
pairs :: Parser Char Int
```

Mais combinadores

- O parser `option p d` reconhece a entrada aceita por `p` ou retorna o valor padrão `d`.

```
option :: Parser s a -> a -> Parser s a
option p d = p <|> succeed d
```

Mais combinadores

- Repetindo a execução de um parser.

```
many :: Parser s a -> Parser s [a]
many p = ((:) <$> p <*> many p) <|> succeed []

many1 :: Parser s a -> Parser s [a]
many1 p = (:) <$> p <*> many p
```

Natural

```haskell
natural :: Parser Char Int
natural = foldl f 0 <$> many digit
      where
        f ac d = ac * 10 + d
```

Combinadores

- Descartando resultados intermediários.

```
first :: Parser s a -> Parser s a
first (Parser p)
   = Parser (\ inp -> let r = p inp
                      in if null r then []
                         else [head r])
```

Combinadores

```haskell
greedy :: Parser s a -> Parser s [a]
greedy = first . many

greedy1 :: Parser s a -> Parser s [a]
greedy1 = first . many1
```

Identificadores

```
identifier :: Parser Char String
identifier
    = (:) <$> letter <*> greedy (sat isAlphaNum)
      where
        letter = sat isLetter
```

Separadores

- Parser `listOf p sep` processa elementos usando `p` usando como separador `sep`.

```
listOf :: Parser s a -> Parser s b -> Parser s [a]
listOf p sep
   = (:) <$> p <*> many ((\ x y -> y) <$> sep <*> p)
```

Separadores

- Parser `pack p q r` processa o parser `q` usando os separadores `p` e `r`, descartando-os.

```
pack :: Parser s a -> Parser s b ->
        Parser s c -> Parser s b
pack p q r = (\ _ x _ -> x) <$> p <*> q <*> r
```

Exemplo

- Processando conteúdo entre parêntesis.

```
parenthesized :: Parser Char a -> Parser Char a
parenthesized p = pack (symbol '(') p (symbol ')')
```

Separadores

- O parser `endBy p q` processa listas de elementos reconhecidos por `p` usando como separadores o que é processado por `q`.

```
endBy :: Parser s a -> Parser s b -> Parser s [a]
endBy p sep = greedy ((\ x _ -> x) <$> p <*> sep)
```

Exemplo

- Processando arquivos CSV
- Arquivos CSV: representação textual de dados em tabelas (planilhas)
  - Dados representados como strings.
  - Separadores para dividir colunas.
  - Linhas no arquivo denotam linhas na tabela.

Modelagem

- Tipos para representar dados CSV.

```
type CSV = [Line]
type Line = [Cell]
type Cell = String
```

Células

- Qualquer string sem `\n` e `,`.

```
cellParser :: Parser Char Cell
cellParser = greedy valid
    where
        valid = sat (\ c -> notElem c ",\n")
```

Linhas

- Usaremos o parser `listOf`, seperando colunas

```
lineParser :: Parser Char Line
lineParser = listOf cellParser (symbol ',')
```

CSV

```
csvParser :: Parser Char CSV
csvParser = endBy lineParser eol
   where
     eol = symbol '\n'
```

Exemplo

```
parseCSV :: FilePath -> IO ()
parseCSV file
    = do
        content <- readFile file
        print (runParser csvParser content)
```

Separadores

- No que vimos, separadores não possuem significado.
- Mas, seperadores podem possuir significado?
  - Sim! Expressões com operadores binários.

Chainr

- Operador associativo a direita

```
chainr :: Parser s a ->                  -- expressão
          Parser s (a -> a -> a) -> -- operador
          Parser s a
chainr pe po
   = h <$> many (j <$> pe <*> po) <*> pe
     where
       j x op = op x
       h fs x = foldr ($) x fs
```

Chainl

- Operador associativo a esquerda

```
chainl :: Parser s a ->                  -- expressão
          Parser s (a -> a -> a) ->      -- operador
          Parser s a
chainl pe po
   = h <$> pe <*> many (j <$> po <*> pe)
     where
       j op x = \ y -> op y x
       h x fs = foldl (flip ($)) x fs
```

Exemplo

- Processando expressões.
- Sintaxe de expressões:

$$
\begin{array}{lcl}
e & \rightarrow & t + e \\
 & | & t \\
t & \rightarrow & f * t \\
 & | & f \\
f & \rightarrow & n \\
 & | & (e)
\end{array}
$$

Sintaxe

- Tipo de dados para representar a árvore de sintaxe de expressões.

```
data Exp
   = Const Int
   | Exp :+: Exp
   | Exp :*: Exp
   deriving (Eq, Ord, Show)
```

Parser

- Regras $f \rightarrow n \mid (e)$:

```
factorParser :: Parser Char Exp
factorParser
   =  parenthesized (expParser) <|>
      (Const <$> natural)
```

Parser

- Regras $t \rightarrow f + t \mid t$:

```
termParser :: Parser Char Exp
termParser
    = chainr factorParser pmult
      where
        pmult = const (:*:) <$> symbol '*'
```

Parser

- Regras $e \rightarrow t * e \mid t$:

```
expParser :: Parser Char Exp
expParser
  = chainr termParser pplus
    where
      pplus = const (:+:) <$> symbol '+'
```

Exercício

- Implemente um parser para strings da forma:

$$(id_1, ..., id_n)$$

em que cada id_i é um identificador. Note que a lista de identificadores usa como separador o caractere ‘,’.

Exercício

- Modifique o tipo de dados `Exp` para incluir a operação de subtração. Altere o parser de forma a reconhecer expressões evolvendo subtração.

Exercício

- O parser de expressões utiliza a função `natural` que reconhece um número natural qualquer. Implemente um parser `integer :: Parser Char Int` que seja capaz de reconhecer números negativos. *Dica*: Use o combinador `option`.