Teoria de Categorias & Programação Funcional
2022.2

Lista de Exercícios 5[†]

Entregar as soluções das questões assinaladas com *
até **14/1 às 23:59**.

A entrega deve ser feita por email para `hugonobrega@ic.ufrj.br`

As questões podem ser resolvidas em dupla, mas as duplas não podem ser repetidas de listas anteriores.

Questão 1. Implemente cada um dentre

```
>>=  :: Monad m => m a -> (a -> m b) -> m b
>=>  :: Monad m => (a -> m b) -> (b -> m c) -> a -> m c
join :: Monad m => m (m a) -> m a
```

usando cada um dos outros. Em outras palavras, implemente `(>>=)` usando `(>=>)`, implemente `(>>=)` usando `join`, implemente `(>=>)` usando `(>>=)`, etc. Os casos "usando `join`" foram feitos em sala; você não precisa fazer esses.

***Questão 2.** Lembrete: as leis de `Applicative` são:

Identidade	`pure id <*> x = x`
Composta	`pure (.) <*> u <*> v <*> w = u <*> (v <*> w)`
Homomorfismo	`pure g <*> pure x = pure (g x)`
Intercâmbio	`u <*> pure y = pure ($ y) <*> u`

Assuma que o seguinte teorema é válido:

Teorema (Teorema de Graça para Funtores)**.** *Pra todo `Functor f`, pra toda função (polimórfica) `fun :: (a -> b) -> f a -> f b`, e pra toda função (polimórfica) `g :: a -> b`, temos:*

```
fun g = (g <$>) . (fun id)
```

Seja `f` um `Applicative`. Considerando o teorema acima para o caso

```
fun :: (a -> b) -> f a -> f b
fun g = pure g <*>
```

[†]Publicada em 3/1

prove que a seguinte igualdade é uma consequência das leis de `Applicative`:

```
pure g <*> = g <$>
```

Questão 3 (*Parsers* com `Applicative`)**.** Um *parser* é uma função que recebe como entrada um dado menos estruturado (ou com estrutura apenas implícita) e dá como saída um dado mais estruturado (ou com a estrutura explicitada). Por exemplo, um parser poderia receber uma `String` formada por números inteiros separados por espaços, e retornar uma `[Integer]` (a lista dos números lidos).

Nessa questão, veremos como `Applicative` e `Alternative` são classes de tipos bastante apropriadas para trabalharmos com parsers por permitirem *combinarmos* parsers "simples" para formar parsers mais complexos (no espírito da abstração que guiou essa disciplina).

Para nós, parsers serão implementados usando o seguinte construtor de tipos:

```
newtype Parser a = P {rodaParser :: String -> Maybe (a, String)}
```

(`newtype` permite que o compilador faça otimizações que não são possíveis com `data`: usamos `newtype` quando temos um tipo "record" com apenas um construtor (no nosso caso, `P`) e apenas um campo (no nosso caso, `rodaParser`); falando matematicamente, nesse caso o novo tipo é "produto de um tipo só", portanto é isomorfo ao tipo original! Assim, o compilador do Haskell pode simplesmente eliminar o novo tipo e usar apenas o original.)

Assim, um valor de tipo `Parser a` tem a forma `P f`, onde `f` é a função que vai de fato "parsear" o dado: recebe uma `String` a ser consumida e retorna um `Maybe (a, String)`, correspondendo a `Just` um dado de tipo `a` parseado e o restante da `String` que não foi consumida (no caso de sucesso), ou `Nothing` no caso de fracasso.

Por exemplo, podemos construir um parser simples que tenta ler apenas um caracter satisfazendo uma dada condição:

```
satisfaz :: (Char -> Bool) -> Parser Char
satisfaz condição = P f
  where
    f (x:xs) = if condição x then Just (x, xs) else Nothing
    f []     = Nothing
```

Um parser é "rodado" de fato usando a função `rodaParser`:

```
> rodaParser (satisfaz isUpper) "Hugo"
Just ('H', "ugo")
> rodaParser (satisfaz isUpper) "hugo"
Nothing
```

(`isUpper` é uma função do módulo `Data.Char` que faz o que se espera dela.)

O arquivo `Parser.hs` traz mais dois exemplos de parsers: `caracter :: Char -> Parser Char`, que recebe um caracter e retorna um parser que aceita apenas esse caracter, e `intPos :: Parser Integer` que lê um número inteiro positivo no início da `String`:

```
> rodaParser intPos "32412bla"
Just (32412,"bla")
> rodaParser intPos "-32412bla"
```

```
Nothing
> rodaParser intPos "ble32412bla"
Nothing
```

O arquivo `Parser.hs` tem várias ocorrências de `undefined`, correspondentes aos itens abaixo. Portanto, sua tarefa é substituir cada uma dessas ocorrências pelo que se pede em cada item, e submeter o arquivo `Parser.hs` resultante.

*** a.** Escreva um instância de `Functor` para `Parser`. Em outras palavras, dados uma função `a -> b` e um `Parser a`, como produzir um `Parser b`?

*** b.** Escreva um instância de `Applicative` para `Parser`.

A ideia, como em geral para `Applicative`, é "encadear" parsers, com os "novos" parsers na cadeia consumindo a parte da `String` que os "anteriores" deixaram de consumir.

Para `parser1 <*> parser2`, temos que `parser1` é um parser de "função `a -> b`", e `parser2` é um parser de `a`; como usar isso para fazer um parser de `b`? A ideia é que um fracasso em qualquer etapa implique em fracasso no geral; o que fazer no caso de sucesso?

*** c.** Escreva um instância de `Alternative` para `Parser`. A ideia é que `parser1 <|> parser2` seja um parser que só "tenta" `parser2` no caso de `parser1` não ter sido bem sucedido.

Exemplos:

```
> rodaParser (caracter 'a' <|> caracter 'b') "abc"
Just ('a', "bc")
> rodaParser (caracter 'a' <|> caracter 'b') "babc"
Just ('b', "abc")
> rodaParser (caracter 'a' <|> caracter 'b') "cbabc"
Nothing
```

*** d.** Agora vamos fazer "repetidores gulosos" de parsers: implemente `zeroOuMais :: Parser a -> Parser [a]` e `umOuMais :: Parser a -> Parser [a]`, que recebem um `Parser a` como entrada e produzem parsers que tentam aplicá-lo quantas vezes forem possíveis, gerando uma lista dos resultados de acordo com a seguinte semântica:

- `zeroOuMais parser` sempre é bem sucedido, retornando `Just` a lista de todos os sucessos (que pode ser vazia) e a `String` não consumida;
- `umOuMais parser` só é bem sucedido se `parser` for bem sucedido ao menos uma vez, e neste caso o retorno é como no caso de `zeroOuMais`.

Exemplos:

```
> rodaParser (zeroOuMais (caracter 'x')) "abcd"
Just ("", "abcd")
> rodaParser (umOuMais (caracter 'x')) "abcd"
Nothing
> rodaParser (zeroOuMais (caracter 'x')) "xxxabcd"
Just ("xxx", "abcd")
> rodaParser (umOuMais (caracter 'x')) "xxxabcd"
Just ("xxx", "abcd")
```

A partir do próximo item, faremos um parser para uma linguagem com uma sintaxe estilo "Lisp" (porém bastante simplificada).

*** e.** Na nossa linguagem, um *identificador* por caracteres alfanuméricos, mas necessariamente começando por um caracter alfabético. Para simplificar, vamos declarar um tipo sinônimo (`type` em `Haskell` funciona como `typedef` em `C`):

```
type Identif = String
```

Faça um parser `parseIdentif :: Parser Identif`
Dica: o módulo `Data.Char` tem funções `isAlpha` e `isAlphaNum`.

*** f.** Um *átomo* é um número inteiro positivo ou um identificador:

```
data Átomo = N Integer | I Identif
  deriving (Eq, Show)
```

Faça um parser `parseÁtomo :: Parser Átomo`

*** g.** Faça um parser `parseEspaços :: Parser String` que leia (gulosamente) caracteres de espaço em branco. *Dica*: o módulo `Data.Char` tem uma função `isSpace`.

*** h.** Finalmente, uma *s-expressão* é um átomo, ou (recursivamente) uma lista de s-expressões:

```
data SExpr = A Átomo | C [SExpr]
  deriving (Eq, Show)
```

Na representação "menos estruturada" em `String`, o caso "lista" é denotado envolvendo por parênteses e separando as s-expressões internas por pelo menos um espaço cada. Além disso, cada s-expressão pode começar ou terminar com uma quantidade qualquer de espaços.

Exemplos de `String` representando s-expressões:

- `"oi"`
- `"  ( tud0  (b3m com  (vo6 100 )) )"`
- `"(lambda (f) ((lambda (x) (f x x)) (lambda (x) (f x x))))"`

Faça um parser `parseSExpr :: Parser SExpr`.

Exemplos:

```
> rodaParser parseSExpr "oi"
Just (A (I "oi"),"")
> rodaParser parseSExpr "  ( tud0  (b3m com  (vo6  100 )) )"
Just (C [A (I "tud0"),C [A (I "b3m"),A (I "com"),C [A (I "vo6"),
                                                    A (N 100)]]],"")
> rodaParser parseSExpr "(lambda (f) ((lambda (x) (f x x))
                                      (lambda (x) (f x x))))"
Just (C [A (I "lambda"),C [A (I "f")],C [C [A (I "lambda"),
     C [A (I "x")],C [A (I "f"),A (I "x"),A (I "x")]],C [A
     (I "lambda"),C [A (I "x")],C [A (I "f"),A (I "x"),A (I "x")
                                                        ]]]],"")
> rodaParser parseSExpr " esse nao vai todo  "
Just (A (I "esse"),"nao vai todo  ")
```

(bastante liberdade poética foi usada nos espaços em branco aqui para que tudo ficasse mais legível)

Dica: para essa questão, vai ser bastante útil você usar `(*>) :: f a -> f b -> f b` e `(<*) :: f a -> f b -> f a` de `Applicative` ("de graça", i.e., você não precisa implementar pois eles são derivados automaticamente de `(<*>)`), que como `(<*>)` executam as operações "em sequência", mas `(*>)` "joga fora" o primeiro resultado e fica apenas com o segundo, e `(<*)` faz o oposto.

***Questão 4.** Considere o seguinte construtor de tipos:

```
data Cont r a = C ((a -> r) -> r)
```

(`Cont` vem de "continuação"). "Abanando as mãos" e saindo um pouco do `Haskell`, poderíamos pensar nos casos[1]

- `a` $= [0, 1]$ e `r` $= \mathbb{R}$. Considerando apenas funções contínuas teríamos max, min e "integral definida de 0 a 1" como exemplos de continuações.
- `a` um conjunto qualquer `r` $= \{\mathsf{Verdadeiro}, \mathsf{Falso}\}$. Aqui valores de tipo `a -> r` são predicados a respeito dos elementos de `a`, e os quantificadores $\forall x \in$ `a` e $\exists x \in$ `a` são continuações.
- `a` é o conjunto de estratégias em alguma classe de jogos, `r` o conjunto de resultados possíveis dos jogos dessa classe. Nesse caso, cada valor de tipo `a -> r` pode ser visto como um jogo (associando cada possível estratégia ao resultado correspondente quando os jogadores seguem aquela estratégia). Um exemplo de continuação nesse contexto é: para cada jogo, calcular o "resultado ótimo", i.e., quando os jogadores seguem a melhor estratégia possível.
- `a` $=$ `r` $= [0, 1]$. Nesse caso, novamente considerando apenas funções contínuas, uma continuação poderia ser "calcular o ponto fixo da função contínua $f : [0, 1] \to [0, 1]$ dada" (tal ponto fixo sempre existe, pelo teorema do ponto fixo de Brouwer).

Continuações também são muito importantes no contexto da teoria de linguagens de programação como `Scheme`.[2]

Implemente instâncias de `Functor`, `Applicative` e `Monad` para `Cont r`.

[1] Esses exemplos não são importantes para essa questão e podem ser ignorados.
[2] Isso também não é importante para essa questão.