parser combinators em haskell

@o-santi

1 haskell

tudo bem. implementamos em c++ um parser porco. ele funciona? sim. mas é muito chato adicionar novas operações, em especial porque ele implementa várias coisas em lugares separados; como por exemplo a prioridade da operação (no parsing) e a ação que é executada para a ação (no push). vimos também que, por ser em c++, precisamos implementar certos hacks para que funcione (como por exemplo usando switch para os casos que queremos que ε seja aceito).

além disso, também vimos o problema de que o parser só aceita linguagens não ambíguas e não recursivas à esquerda (uma classe chamada de $LL(k)$), e por mais que eu não pretenda provar isso, é fato consumado que essa classe é estritamente menor que a classe de linguagens livres de contexto, que por sua vez é estritamente menor que a classe de linguagens sensíveis ao contexto. e nós temos exemplos de linguagens de programação que não são livres de contexto, como xml.

assim, seria conveniente se conseguíssimos construir um parser que não apenas é mais potente (consegue processar uma classe maior de linguagens) como também é mais simples e intuitivo de se trabalhar. e sim, como você leu no título, podemos utilizar haskell para tal tarefa.

não precisa falar, sim, eu sei, para a maior parte dos programadores que trabalham na mentalidade CPJR (C/Python/Java/Ruby (sim eu acabei de inventar esse acrônimo)), haskell é mais uma das linguagens estranhas que tem apenas o propósito de ser "diferente" na hora de programar. enquanto esse pode de fato ser o caso para a psique do programador médio (alienado), essa linguagem realmente apresenta inúmeras vantagens e facilidades para implementar o que queremos: parser combinators!

mas..... o que são parser combinators? e por que haskell? por que não podemos fazer em C? e o que são essas "mônades" que o artigo da wikipedia fala? muitas perguntas sem nenhuma resposta...

brincadeira. eu tenho as respostas. ou pelo menos algumas. a ideia de combinação é quebrar o problema de construir um grande parser em implementar vários pequenos (e mais simples) parsers, que posteriormente serão compostos juntos de tal maneira a processar o que queríamos. utilizamos haskell ao invés de C pois os tipos das variáveis tomarão papel essencial no programa (e o type checker de C é pífio), isto é, veremos que o tipo dos Parsers é bem *interessante*. quanto às mônades, responder essa pergunta completamente é empreitada pretenciosa demais para este artigo, mas pretendo introduzir o conceito com alguns exemplos.

antes de entrar nas especificações, vale notar que existem inúmeras bibliotecas em haskell que implementam (muito melhor que eu) o que irei fazer aqui. de fato, a maior parte deste código será

puramente para explicar o que é *monadic parser combinators* e a teoria por trás, e eu prometo que o excerto que realmente implementa o parser que queremos não terá mais de 20 linhas.

2 programação funcional

em haskell nossa principal arma são funções e composição de funções, em especial aquelas que não mudam o estado do mundo fora dela. mas o que isso significa?

vejamos o seguinte código em C:

```
int foo = 3;
void bar(){
  foo++;
  bla bla bla...
}
```

se sabemos que precisamos executar a função `bar` em diversos lugares do programa. se pegarmos um estado qualquer da execução do programa, e perguntarmos: qual o valor em `foo`? isso não é uma resposta óbvia quando o programa é marginalmente complexo, visto que precisamos analisar todas as vezes que `bar` é chamada no programa, bem como todos os possíveis lugares que assinalamos outro valor à variável.

para contornar esse problema, podemos exigir que todas as funções sejam puras; isto é, nenhuma função pode alterar o valor de uma variável que está fora do escopo dessa. isso gera impecílhos, já que no nivel do sistema operacional o *stdout* (onde geralmente printamos as coisas) também é apenas um arquivo, não podemos facilmente escrever coisas na tela.

esse estilo de programação é chamado de **programação funcional** e possui inúmeras vantagens à programação imperativa, principalmente o fato de que sempre sabemos à todo momento o estado das variáveis do nosso programa, bem como ser muito mais fácil de executar concorrentemente o programa (já que não temos *shared state*). por outro lado, existem inúmeros casos onde precisamos explicitamente de estados (por exemplo quando escrevemos algo na saída), e neles também utilizaremos estados, mas de uma maneira controlada e explicita: principalmente, utilizaremos **mônades** para isso. não pretendo me aprofundar muito sobre esse assunto, mas para os interessados: What is IO monad?.

3 tipos

estaremos utilizando o ghc (Glasgow Haskell Compiler) para compilar e dar typecheck no programa, então se quiser executar junto, procure antes instalar o ghc.

podemos começar o nosso programa em haskell da seguinte forma

```haskell
module Main where

main :: IO()
main = undefined
```

isso nos diz o seguinte: definimos um módulo chamado `Main` e definimos o seu *entrypoint* `main`, que é do tipo `IO` (input/output), o que significa que ele simplesmente irá escrever algo na tela, ou ler algo de algum arquivo. dizemos que `main = undefined` simplesmente para que o compilador não reclame, para poder definir outras coisas antes de definir a `main`. mas que outras coisas queremos definir? o parser!.

afinal, o que é um parser? idealmente, pensando na definição que demos inicialmente, poderiamos considerar que é uma função que recebe uma string (a expressão) e retorna um número inteiro (que representa o resultado da computação). poderíamos definir algo do tipo:

```haskell
type Parser = String -> Integer
```

essa notação de seta `a -> b` indica o tipo de uma função que recebe um argumento do tipo `a` e retorna um valor do tipo `b`.

essa definição pode ser estendida de algumas maneiras. primeiramente, para podermos combinar os parsers, podemos adicionar o resto da string que não fora processada como parte do output, para que o próximo parser possa usá-la como input:

```haskell
type Parser = String -> (Integer, String)
```

além disso, não há necessidade que todos os parsers sejam especificamente de expressões matemáticas. eles podem retornar inúmeros (ahá!) tipos. haskell nos permite construir esse tipo de tal forma que dependa em outro tipo fornecido:

```haskell
type Parser a = String -> (a, String)
```

ou seja, se criarmos um `Parser String`, teremos um parser que nos devolve String, se criarmos um `Parser Integer`, teremos um parser que devolve um inteiro. é importante perceber que `Parser` deixa de ser um tipo único e passa a ser uma família de tipos, visto que é uma função que recebe um argumento `a` e retorna um tipo (que também é uma função) associado a esse `a`.

note também que em haskell para aplicar uma função em um argumento (neste caso estamos aplicando `String` à função `Parser`) basta concatená-las (escrever uma seguida da outra): ao invés de escrevermos `f(a)`, escrevemos `f a`.

além disso, todas as funções são automaticamente *curried*, ou seja, se queremos que a função `f` receba mais de um argumento, ao invés de utilizarmos:

```
def f(a, b):
    return a + b
```

usamos:

```
def f(a):
    def h(b):
        return a + b
    return h
```

ou seja, o tipo de `f` é `Integer -> Integer -> Integer`. além de nos fornecer uma certa normalização (todas as funções sempre recebem apenas um argumento e retornam um argumento), nos permitem utilizar a concatenação: ao invés de `f(a, b)`, escrevemos `f a b`. explicitando, quando escrevemos `f a b`, primeiro aplicamos `(f a) b`, que nos dá `h b`, que pode por fim ser transformado na soma.

continuando sobre o parser, ainda podemos fazer uma última modificação. sabemos que ele pode falhar, e podemos representar isso no tipo. existem diversas formas de implementar isso (por exemplo com a mônade `Maybe`), mas escolhi a mais simples que é retornar uma lista de possíveis resultados:

```
type Parser a = String -> [(a, String)]
```

e a lista vazia `[]` representa o caso onde a expressão não pode ser processada pelo parser. poderíamos estender ainda mais esse tipo, por exemplo retornando junto a posição na string onde o erro foi encontrado (o número do índice), uma segunda String que representa a mensagem do erro, dentre muitas outras coisas. por agora, esse tipo será suficiente para o que queremos.

por fim, antes de continuar, utilizamos um pouco de magia de haskell. ao invés de simplesmente definir o tipo `Parser`, faremos o seguinte:

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

o que significa: além de definir o tipo `Parser`, definimos também uma função, chamada `runParser`, que automaticamente "desconstroi" o nosso tipo. essa função terá tipo `Parser a -> String -> [(a, String)]`, ou seja, ela recebe o nosso parser, e a string, e automaticamente **executa** ele. sim, parece magia, mas a função é automaticamente criada pelo compilador, então não precisaremos nos preocupar com isso. para entender mais sobre essa funcionalidade, leia a wiki de haskell.

4 pequenos blocos

tudo bem. falamos muito de tipos. e quase nada de implementações. eu prometi que seria mais fácil. e será! vamos começar construindo uma função `charParser` que recebe um caracter e retorna um parser que processa **apenas** esse caracter. podemos implementá-la da seguinte forma:

```
charParser :: Char -> Parser Char
charParser x = Parser $ f
  where f (head:tail)
          | head == x    = [(head, tail)]
          | otherwise = []
        f []           = []
```

vamos analisar o código acima com calma. primeiro vemos a definição do tipo: nossa função recebe um caracter e retorna o parser desse caracter. se expandirmos `Parser Char`, vemos que é uma função do tipo `Char -> String -> [(Char, String)]`, o que é condizente com o que foi definido antes.

depois, vamos para a especificação. dizemos que `charParser` recebe um argumento, `x`, e retorna um parser aplicado à uma função `f` (o sinal $ é o de concatenação a direita). o compilador é capaz de inferir que essa função deve receber uma `String` e retornar o tipo `[(Char, String)]`. ele nos permite portanto, na hora de especificar a função `f`, analisar os casos dessa string (utilizando a notação `where`).

sabemos que uma `String` é uma lista de caractéres, e em haskell todas as listas são implementadas no estilo *linked list*, isto é, possuem um elemento head e uma lista tail. é exatamente isso que utilizamos na função: separamos a nossa string implicitamente no primeiro caracter `head` e na lista restante de caracteres `tail` (ao dizer `f(head: tail)`), o que nos permite analisar os casos:

- quando `head` for igual a `x`, definimos que a função deve retornar `[(head, tail)]`, ou seja, retornamos o próprio caracter processado e o resto da string, como havíamos prometido.
- caso contrário (`otherwise`), definimos que a função retorna a lista vazia.

além disso, sabemos que a lista vazia não pode ser desconstruida em head e tail, e portanto colocamos esse último caso: se `f` receber a lista vazia (ou seja uma string vazia), ele também retornará a lista vazia, sinalizando que a expressão não pode ser processada.

viu? é muito simples definir esse parser. apesar de a sintaxe de haskell não ser muito parecida com C e seus derivados, não doeu nada entender essas linhas. se tiver doido, leia até parar de doer, é importante, constrói caráter e desenvolve capacidade cognitiva.

podemos visualizar esse parser em ação: criamos um parser `charParser 'h'` cujo único propósito neste mundo é testar se a primeira letra da string é o caracter 'h', não se propõe a nada mais, ou nada menos. mas como rodamos esse parser? isso mesmo: usando `runParser`! utilizando o `ghci` (o REPL do compilador de haskell), podemos carregar esse arquivo e rodar a linha

```
*Main> runParser (charParser 'h') "hello"
```

que nos dá o resultado `[('h', "ello"]`. uhul!! funcionou. ele realmente viu que a primeira letra da string era 'h', retirou essa letra da string e retornou o resto. rodando com uma string que não começa com 'h', por exemplo "world", ele corretamente nos devolve a lista vazia.

tudo bem. implementamos um parser de uma letra. mas e agora? como eu uso ele pra ler uma string específica?

5 combinando

já vimos anteriormente que uma String é simplesmente uma lista de caracteres, ou seja `[Char]`. utilizando a função `map` (que recebe uma função `f` e uma lista, e aplica `f` à cada elemento da lista), podemos aplicar a função `charParser` à uma string, e nos daria o tipo `[Parser Char]`, ou seja, uma lista de parsers de um caracter. porém, contudo, todavia, sabemos que na verdade um parser de string é do tipo `Parser [Char]`.

precisamos então de uma função que receba `[Parser Char]` e nos devolva `Parser [Char]`. felizmente, inúmeros nerds inteligentes já pesquisaram programação (e teoria matemática também) o suficiente para implementar essa função para nós! de fato, na biblioteca padrão de haskell, temos uma função chamada `sequenceA`, que recebe `t (f a)` e nos devolve `f (t a)`!! isso é exatamente o que queremos, visto que precisamos transformar `List (Parser Char)` em `Parser (List Char)`.

porém, temos alguns problemas: se tentarmos aplicar essa função diretamente, veremos que o compilador reclama que o tipo `Parser` não é `Applicative`. mas o que isso significa?

bom, a maneira que os nerds ancestrais implementaram essa função (e várias outras) foi super inteligente. ao invés de simplesmente nos dar a função e torcer para funcionar no tipo específico, eles fizeram da seguinte forma: para utilizar essa função, você precisa primeiro fornecer uma prova de que `f` é `Applicative` e `t` é `Traversable`. enquanto a biblioteca padrão de haskell já possui uma prova de que `List` (nosso `t`) é `Traversable`, nós não temos essa prova de que `Parser` é `Applicative`.

mas o que significa provar isso? para provar que algo é `Applicative`, precisamos implementar duas funções especiais (chamadas `pure` e `<*>`), e também provar que esse mesmo algo é `Functor`. ok, nós prolongamos o problema, o que é um `Functor`?

a interface `Functor` representa a classe de funções que são capazes de implementar a função `fmap`, isto é:

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

como assim? parece que esses nerds estão apenas complicando mais as coisas. imagine o seguinte cenário: eu te dou uma função `tradutor` que, dado um número inteiro, retorna uma string que repre-

senta esse número em português. então `tradutor 2` retorna a string `"dois"` e `tradutor 13` retorna a string `"treze"`. naturalmente, o tipo dessa função é `Integer -> String`.

utilizando `fmap`, podemos "penetrar" o nosso parser com essa função e aplicar diretamente ao resultado! isso significa que se eu te fornecer um `intParser` (um parser de inteiros), nós podemos traduzir os números que esse parser retorna simplesmente penetrando o resultado com `tradutor`. esse parser resultante leria por exemplo a string `"2"` e retornaria o resultado `[("dois", "")]`. informalmente, ele seria um `Parser (tradutor Integer)`, ou uma função que retorna `[tradutor(Integer), String]`.

super interessante. mas como nós fazemos isso? basta... pasme, implementar `fmap`!

```
instance Functor Parser where
  fmap f (Parser p) = Parser $ g
    where g input = case p input of
            []           -> []
            [(x, resto)] -> [(f x, resto)]
```

vamos analisar o que acabamos de escrever: a função `fmap` recebe uma função e o parser que queremos "penetrar" a função em. sabemos que o segundo argumento é da forma `Parser p`, onde `p` é a nossa função `String -> [(a, String)]`, então utilizamos pattern matching pra automaticamente ganhar acesso a essa função.

depois, sabemos que o resultado deve ser do tipo `Parser g` onde `g` recebe uma string e transforma-a usando `p`. de novo, utilizamos pattern matching para pegar o input da função, e simplesmente aplicamos o parser ao input. sabemos que o resultado deve ser da forma `[(x, resto)]`, e é exatamente nesse `x` que queremos aplicar a função `f` (que passamos inicialmente). portanto, o que retornamos é a lista com a aplicação da função `f` em `x` e o resto do input a ser consumido. também precisamos inserir o caso em que `p` não processa o input, que deverá ser a lista vazia.

pronto! provamos que o nosso tipo `Parser` na verdade é um `Functor`. agora podemos utilizar `fmap` junto do nosso parser.

agora podemos provar que ele o tipo `Parser` também é um `Applicative`, implementando as duas outras funções: `pure` e `<*>`. primeiro observamos que o tipo de `pure` é `a -> f a`. isso significa que, independente do input em `f`, ele deve sempre retornar `f a` (o elemento inicial que passamos). mas isso é muito fácil, basta implementar o seguinte parser:

```
instance Applicative Parser where
  pure x = Parser $ \input -> [(x, input)]
  p1 <*> p2 = undefined
```

ou seja, se chamarmos `pure 2`, receberemos um `Parser` que simplesmente ignora o input e retorna `[(2, input)]` (por isso o nome *pure*, ele não liga para o que passam como argumento, ele será um parser "constante"). a sintaxe `\a -> b` representa uma função que recebe `a` e retorna `b`.

para a segunda operação `<*>`, precisamos olhar analisar o tipo dela: `f(a ->b) -> f a -> f b`. ok, o que isso significa? bom, se olharmos de perto, vemos que é bem parecido com o que implementamos em `fmap`, com a diferença de que a função inicial está "envolta" em `Parser` (`f (a->b)` ao invés de `(a ->b)`). de fato, se imaginarmos que `fmap` faz a "penetração" incial da função, então o operador `<*>` "penetra" mais fundo, pegando 2 operadores que já estão envoltos e aplicando a função que queremos no que eles estão envolvendo.

por exemplo, podemos somar o resultado de dois `Parser`'s sem ter que "desencapá-los" utilizando fmap junto da operação soma. se fizermos `fmap (+) pure 5 <*> pure 6`, o `fmap` fará a penetração inicial no primeiro parser, que será agora um parser do tipo `pure (+) 5`; isto é, o elemento que ele segura é a soma com o elemento 5 já aplicado, usando *currying*. ao usar o operador `<*>`, aplicamos a função `(+) 5` dentro do parser, no elemento `6`, e obtemos o resultado `pure 11`.

a implementação disso é exatamente como descrito acima:

```
instance Applicative Parser where
  pure x = Parser $ \input -> [(x, input)]
  (Parser p1) <*> (Parser p2) = Parser $ \input ->
    case p1 input of
      [] -> []
      [(g, resto)] -> case p2 resto of
                        [] -> []
                        [(a, resto')] -> [(g a, resto')]
```

primeiro usamos pattern matching, nos dois `Parser`'s, depois aplicamos o primeiro ao input, pegamos a função `g` resultante, aplicamos o resto ao segundo parser, e retornamos a aplicação de `g` no que foi processado em `p2` junto do novo resto. precisamos também adicionar os casos em que os parsers falham, e nos dois simplesmente retornamos as listas vazias.

tome um tempo para compreender o que acabamos de implementar; não é simples mas é bem poderoso. isso significa que podemos "injetar" (ou aplicar, dai o nome `Applicative`) qualquer função que quisermos dentro do resultado de um parser e aplicar subsequentemente outras funções, sem ter que explicitamente "desencapar" (*unwrap*) o nosso tipo. temos uma maneira clara de **combinar** dois parsers: através do operador `<*>` e `fmap`!

finalmente podemos implementar o parser de strings proposto inicialmente:

```
stringParser :: String -> Parser String
stringParser input = sequenceA $ map charParser input
```

que nos permite fazer:

```
*Main> runParser (stringParser "hello world") "hello world blablabla"
[("hello world"," blablabla")]
```

eureka! o parser `stringParser "hello world"` procura especificamente pela string `"hello world"`, e qualquer outra coisa que ele ache no inicio fará com que ele retorne a lista vazia.

6 mais combinações

tudo bem, nós sabemos como procurar por um caracter e por uma string, mas e se nós quiséssemos procurar por duas strings diferentes? por exemplo, poderíamos estar procurando por um `Bool`, que pode ser tanto a string "true" quanto a string "false".

para fazer isso, precisamos de alguma maneira dizer para o parser "por favor (o por favor é muito importante), tente processar primeiro a string "true" e se não conseguir, processe a string "false"".

novamente, os nerds ancestrais vêm ao resgate. podemos implementar outra interface em haskell chamada `Alternative`, que exige que a nossa função também seja `Applicative` (por isso provamos isso primeiro). essa interface implementa diversas funções que nós usaremos.

a principal delas é o operador `<|>`, que pode ser interpretado como o "ou" lógico. para que ele funcione, precisamos implementar junto uma função chamada `empty`, que será exatamente o parser que sempre retorna vazio. o operador `<|>` nos permitirá então juntar diversos parsers e retornará o primeiro da lista que não tiver um resultado vazio. simples, não?

basta então implementar as duas funções. primeiro precisamos importar a interface `Alternative`. depois, basta fazermos:

```haskell
import Control.Applicative

instance Alternative Parser where
  empty = Parser $ \_ -> []
  (Parser p1) <|> (Parser p2) = Parser $ f
    where f = case p1 input of
            []           -> p2 input
            [(x, resto)] -> [(x, resto)]
```

o que acabamos de construir é muito simples, mas muito útil. a função `empty` é uma função que implementa o parser mais básico de todos: independente do input, ele sempre retorna a lista vazia. para implementar o "ou lógico" do nosso parser, simplesmente fazemos uma análise de casos: aplicamos o `input` a `p1`, se isso for vazio, devolvemos o resultado de `p2` (vazio ou não), e se o resultado for um parse bem sucedido, retornamos ele. fácil, não?

com essa função, podemos implementar o nosso parser de `Bool` bem facilmente:

```
boolParser = stringParser "true" <|> stringParser "false"
```

que é capaz de ler ambas as strings:

```
*Main> runParser boolParser "true"
[("true","")]
*Main> runParser boolParser "false"
[("false","")]
```

e utilizando a penetração que implementamos anteriormente (`fmap`), podemos transformar automaticamente o que esse parser acha na representação de `True` e `False` nativos de haskell (fica de exercício para o leitor).

uma função ainda mais importante que ganhamos acesso é `many`, que utiliza a noção de "vazio" para procurar dentro de uma lista o maior número possível de coisas em seguida não-vazias. podemos compará-la com a estrela de kleene, e podemos utilizar ela para implementar um parser que procura por classes de caracteres (por exemplo, dígitos, ou letras maiúsculas) em sequência. chamaremos essa função de `spanParser`:

```
spanParser :: (Char -> Bool) -> Parser String
spanParser f = many $ Parser $ g
  where g (head: tail)
          | f head == True  = [(head, tail)]
          | f head == False = []
        g []                = []
```

é uma função simples. primeiro implementamos o parser, que utiliza a função predicativa (que diz `true` ou `false` para todo caracter) para comparar o primeiro caracter do input. se ele for verdadeiro, retornamos o resultado processado, se for falso retornamos a lista vazia. por fim, aplicamos `many` a esse parser.

podemos então usar esse parser para implementar outro extremamente importante (e muitas vezes esquecido):

```
import Data.Char -- isSpace está nesse módulo
whitespaceParser = spanParser isSpace
```

uma função que simplesmente consome os espaços vazios.

e o parser mais essencial (que ainda não implementamos), o parser de números, pode ser implementado com a mesma ideia:

```haskell
intParser :: Parser Integer
intParser = read <$> spanParser isDigit -- isDigit também é definida
 ↪  em Data.Char
```

penetramos esse parser com `read` para transformar os números de `String` para a representação nativa `Integer`. note que o operador `<$>` é simplesmente outro nome para a função `fmap`, mas ao invés de ser uma função prefix (vem antes dos argumentos), ele age infix (vem no meio dos argumentos).

entretanto, temos um problema. assim como a estrela de kleene, a função `many` procura por zero ou mais objetos, e se não houver um número, ela irá retornar a string vazia processada (ao invés de retornar um erro), e quando passarmos essa string vazia, a função read levantará um erro, já que vazio não é um número.

o que precisamos fazer então é uma função que quebra o parser se o nosso parser retornar uma lista vazia. chamaremos de `notNull`:

```haskell
notNull :: Parser [a] -> Parser [a]
notNull (Parser p) = Parser $ \input ->
  case p input of
    []     -> []
    [(x, resto)] -> case x of
                      []  -> []
                      otherwise -> [(x, resto)]
```

ele simplesmente aplica o parser dado na string de input, e só devolve `x` se ele não for a lista vazia.

assim, implementamos `intParser`:

```haskell
intParser :: Parser Integer
intParser = read <$> notNull (spanParser isDigit)
```

ok. vimos isso tudo. mas ainda sim a sintaxe é um pouco complexa. `fmap`? `<*>`? e o que esse cifrão quer dizer? muitas funções não intuitivas e eu não sei exatamente como juntar todas elas, e ainda precisamos analisar os casos? as coisas estão ficando cada vez mais complexas...

não se preocupe, ainda temos um último truque na manga.

7 magia (mônades)

inicialmente o conceito de mônades pode parecer um pouco mágico, mas a ideia é que elas nos fornecem uma interface para retirar e colocar novamente valores dentro de tipos. como assim?

a principal força das mônades é uma operação chamada `bind`, em haskell denotada por:

```
(>>=) :: m a -> (a -> m b) -> m b
```

o ponto dessa função é a seguinte: ela recebe um valor envolto em uma mônade (`m a`), uma função que recebe esse valor e transforma em um novo valor envolto em uma mônade (`a -> m b`) e nos devolve esse novo valor (`m b`). de uma certa forma, a operação `bind` empurra goela abaixo um valor envolto numa mônade em uma função que inicialmente não aceita valores envoltos em mônades.

por exemplo, podemos utilizar `bind` para somar o resultado de dois `Parser`'s de números. a ideia é a seguinte: sabemos que a função que `bind` espera uma função que aceita um argumento e devolve um parser. portanto, nós podemos encadear os parsers usando funções lambdas; isto é: para cada parser, nós podemos guardar o valor dele num argumento que será passado para a próxima função que devolverá um novo parser. mais ou menos da seguinte forma:

```
soma = intParser >>= \x  ->
  (charParser '+' >>= \_ ->
      (intParser >>= \y  -> return (x + y)))
```

ok, parece complexo mas é simples. o primeiro parser nos dá um `Parser Integer`. utilizando `bind`, nós passamos o inteiro que foi achado como argumento para a próxima função (e guardamos o valor em `x`). essa nova função le o caracter '+' e não passa nenhum argumento a frente (por isso o buraco `\_`, que significa que a função não passa nenhum argumento), já que só queremos que o caracter esteja lá, não precisamos de nenhum valor. por fim, pegamos outro inteiro, guardamos o valor lido em `y` e passamos para a função final.

a função `return` é simplesmente outro nome para a função `pure` implementada anteriormente; isto é, ela nos dá um `Parser` que retornará o valor `x+y` independente da string que for passada. assim, a função toda nos dará outro `Parser`, tendo exatamente o tipo esperado pela mônade.

ainda sim, essa sintaxe não é a ideal. muitas funções, muito parênteses, muitas coisas acontecendo na tela. felizmente, os nerds que inventaram haskell tiveram *yet another* ideia brilhante: **do notation**. o que é isso? é uma notação em haskell que é açúcar sintático (isto é, puramente frufru) que esconde todas as aplicações de `bind` e de funções lambdas, e **magicamente** conecta tudo. não acredita? vejam o código a seguir.

```
soma = do
  x <- intParser
  charParser '+'
  y <- intParser
  return x+y
```

viram? é só isso. nenhum operador »=, nenhum parênteses, nenhuma função lambda, simplesmente guardamos valores e utilizamos eles depois, tal qual programação imperativa em C. guardamos o

valor do primeiro parser em `x`, o valor do segundo parser em `y`, e retornamos `x+y`. internamente, cada linha do tipo `a <- expressao` é transformada em uma chamada de função `expressao >>=\a ->`, e cada linha que não guarda variável em uma chamada de função `expressao >>= \_ ->`. por trás dos panos, esse código e o bloco anterior são compilados para exatamente a mesma coisa.

se essa notação é tão boa? então porque nós simplesmente não usamos? bom, precisamos primeiro implementar a interface `Monad` para poder usá-la. e eu queria primeiro mostrar como funcionam as coisas mais simples antes de mostrar isso. para mostrar que `Parser` é uma `Monad`, basta implementar a função `>>=` (visto que já mostramos que é `Applicative`). mas isso é relativamente simples:

```
instance Monad Parser where
  (Parser p) >>= f = Parser $ \input ->
    case p input of
      []    -> []
      [(a, resto)] -> runParser (f a) resto
```

pegamos o parser que foi passado à função e aplicamos ao input. se for vazio, retornamos vazio e acabamos com a operação, se não, aplicamos `f` a `a`, que nos dará um novo parser. utilizamos então esse novo parser para processar o resto do input.

outra vantagem de utilizar a **do notation** é que não precisamos mais nos importar com o caso em que a lista é vazia: a operação `bind` automaticamente quebra toda a sequência e retorna vazio caso um dos parsers retorne vazio.

podemos finalmente, com todos esses operadores implementados, criar a nossa gramática de expressões

8 expressões matemáticas

nosso parser de expressões, diferentemente do parser em c++, não tem problema com inicio comum (devido à lazy evaluation em haskell). nossa gramática fica mais simples:

$$expr \rightarrow term + expr \mid term$$
$$term \rightarrow factor * term \mid factor$$
$$factor \rightarrow (expr) \mid int$$

isso é implementado por:

```
expr = do x <- term
          charParser '+'
          y <- expr
          return (x+y)
       <|> term -- note o uso do operador "ou"
```

```
term = do x <- factor
          charParser '*'
          y <- term
          return (x*y)
        <|> factor -- e de novo
factor = do charParser '('
            x <- expr
            charParser ')'
            return x
          <|> intParser -- e de novo²
```

é só isso. sem brincadeira. é só isso. leia e releia. esse é o parser inteiro. não precisamos de um lexer, não precisamos de quarenta funções, nem de *switch* cases nem da função `push` com inúmeros casos pra cada operador. vemos que adicionar outro operador (por exemplo, subtração ou divisão) torna-se adicionar duas ou três novas linhas em um desses parsers. além disso, pessoalmente acho isso muito mais legível do que um código em C.

para fazer com que o nosso parser não ligue para espaços em branco, só precisamos adicionar o parser `whitespace` entre o parser dos operadores. todo o resto é feito magicamente. vejamos alguns exemplos:

```
*Main> runParser expr "3+4*5"
[(23,"")]
*Main> runParser expr "5 * (6 + 7) + 1"
[(66,"")]
*Main> runParser expr "1 + 2 * 3"
[(7,"")]
*Main> runParser expr "1 + 2 +* 3"
[(3," +* 3")]
```

note no último caso ele só conseguiu ler a primeira soma, e a segunda foi deixada não-processada, assim como esperado.

de fato, implementar parsers *combinators* monádicos é, pelo menos pra mim, muito mais intuitivo e fácil do que utilizando outros tipos de parsers. existem inúmeras implementações de parsers monádicos em haskell, muito mais profissionais, corretas e seguras do que a minha, mas as ideias e os conceitos são exatamente os mesmos. por exemplo, uma das ferramentas mais famosas de conversão de arquivos (chamada *pandoc*) é implementada em haskell justamente por causa dos parsers combinators.

talvez um dos principais motivos para usarmos *parser combinators* monádicos (necessariamente os monádicos) é que eles são capazes de processar gramáticas sensíveis ao contexto. se não utilizarmos as mônades, e ao invés disso combinarmos tudo a partir das ferramentas oferecidas pela interface `Applicative` (como `<*>` e `fmap`), então o parser resultante só é capaz de processar gramáticas livre de contexto. essa diferença se dá pois através das mônades (mais especificamente do `bind`),

o parser é capaz de tomar decisões sobre qual parser ele deve rodar em seguida utilizando o valor lido anteriormente.

tome tempo para entender tudo que foi dito aqui, pois é uma ferramenta extremamente poderosa (e é até hoje um assunto pesquisado), especialmente as noções mais avançadas de teoria dos tipos e de programação funcional.

podemos por fim completar o nosso programa adicionando a função main

```
main :: IO()
main = do input <- getLine
          putStr $ show $ runParser expr input
```

note novamente a do notation ao utilizarmos a mônade `IO`. o que ela quer dizer é simples, primeiro lemos uma linha diretamente do *stdin*. depois, passamos essa string para o parser, que irá retornar uma lista com os possíveis resultados (no nosso caso, como não é ambígua, será apenas um possível resultado), depois passamos para a função show, que transforma o resultado em uma string, e por fim escrevemos essa string no *stdout*.

compilamos com `ghc Main.hs` e *voilà*, temos um executavel compilado (exatamente igual o do C) que é capaz de processar as expressões que queríamos. para ter acesso ao arquivo completo, clique aqui (ou procure no repositório do github deste site).