Functional Programming

Lecture 13: Parser combinators

Twan van Laarhoven

13 December 2021

Radboud University

Outline

- Parser combinators

Parsing expressions

Recall the datatype of expressions

```haskell
data Expr
  = Lit Integer      -- a literal
  | Add Expr Expr    -- addition
  | Mul Expr Expr    -- multiplication
  | Div Expr Expr    -- integer division
```

Front-end is missing: parser that turns input string (concrete syntax) into expression tree (abstract syntax)

```haskell
>>> parse "4 + 7 * 11"
Just (Add (Lit 4) (Mul (Lit 7) (Lit 11)))
>>> parse "(4 + 7) * 11"
Just (Add (Mul (Lit 4) (Lit 7)) (Lit 11))
```

Parsers as functions

A parser (for expressions) can be represented as a function of type

$$\textbf{type}\ \mathsf{Parser} = \mathsf{String} \rightarrow \mathsf{Expr}$$

Parsers as functions

A parser (for expressions) can be represented as a function of type

```
type Parser = String → Expr
```

However, parser might not always consume the entire input string

Parsers as functions

A parser (for expressions) can be represented as a function of type

```
type Parser = String → Expr
```

However, parser might not always consume the entire input string

```
type Parser = String → (Expr, String)
```

can return any unconsumed part of the argument string

Parsers as functions

A parser (for expressions) can be represented as a function of type

$$\textbf{type}\ \mathsf{Parser} = \mathsf{String} \rightarrow \mathsf{Expr}$$

However, parser might not always consume the entire input string

$$\textbf{type}\ \mathsf{Parser} = \mathsf{String} \rightarrow (\mathsf{Expr}, \mathsf{String})$$

can return any unconsumed part of the argument string

Similarly, a parser may fail

$$\textbf{type}\ \mathsf{Parser} = \mathsf{String} \rightarrow \mathsf{Maybe}\ (\mathsf{Expr}, \mathsf{String})$$

Parsers as functions

A parser (for expressions) can be represented as a function of type

$$\textbf{type}\ \mathsf{Parser} = \mathsf{String} \to \mathsf{Expr}$$

However, parser might not always consume the entire input string

$$\textbf{type}\ \mathsf{Parser} = \mathsf{String} \to (\mathsf{Expr}, \mathsf{String})$$

can return any unconsumed part of the argument string

Similarly, a parser may fail

$$\textbf{type}\ \mathsf{Parser} = \mathsf{String} \to \mathsf{Maybe}\ (\mathsf{Expr}, \mathsf{String})$$

Finally, different parsers will likely return different types of values

$$\textbf{type}\ \mathsf{Parser}\ a = \mathsf{String} \to \mathsf{Maybe}\ (a, \mathsf{String})$$

Ad hoc parsing

Suppose we have a parser for natural numbers

```
parseNat :: Parser Int
```

we can use this parser to build a parser for a non-empty list of natural numbers

```
parseNatList :: Parser [Int] -- = String → Maybe ([Int], String)
parseNatList ('[':cs0) = case parseNat cs0 of
    Nothing       → Nothing
    Just (n,cs1)  → case parseNats cs1 of
                      Just (ns,']':cs2)] → Just (n:ns,cs2)
                      Nothing            → Nothing
  where
  parseNats ""        = Just ([],"")
  parseNats (',':cs1) = case parseNat cs1 of
      Nothing       → Nothing
      Just (n,cs2)  → case parseNats cs2 of
                        Nothing         → Nothing
                        Just (ns,cs3) → Just (n:ns,cs3)
  parseNats cs        = Just ([],cs)
parseNatList _        = Nothing
```

Parser combinators

Idea: provide some basic parsers and some combinators to glue them together

Parsing a single character

```
>>> (char '0') "01"
Just ('0',"1")
>>> (char '1') "01"
Nothing
```

Returning "semantic" values

```
>>> (char '0' *> pure 0) "01"
Just (0,"1")
```

Parsing alternative choices

```
>>> let bit = (char '0' *> pure 0)
          <|> (char '1' *> pure 1)
>>> bit "01"
Just (0,"1")
>>> bit "10"
Just (1,"0")
```

The Parser type

We want the parser type to be made into an instance of type classes.

```
type Parser a = String → Maybe (a, String)
```

Can't make type synonyms an instance, so, introduce a new type:

```
newtype Parser a = P { parse :: String → Maybe (a, String) }
```

Reminder: this is the same as

```
newtype Parser a = P (String → Maybe (a, String))
parse :: Parser a → String → Maybe (a, String)
parse (P pf) = pf
```

Also: there is a small technical difference between **data** and **newtype**, but it doesn't matter here.

Parser instances

We will use the operations of classes Functor, Applicative and Monad (and more) as combinators.
So, we want the parser type to be made into instances of these type classes

```haskell
{-# LANGUAGE DeriveFunctor #-}
newtype Parser a = P { parse :: String -> Maybe (a,String) }
  deriving (Functor)
```

A simple parsing primitive

```haskell
item :: Parser Char
item = P (\input -> case input of
                      []     -> Nothing
                      (x:xs) -> Just (x,xs))
```

Examples

```
>>> parse item ""
Nothing
>>> parse item "abc"
Just ('a',"bc")
>>> parse (fmap toUpper item) "abc"
Just ('A',"bc")
>>> parse (fmap toUpper item) ""
Nothing
```

```haskell
newtype Parser a = P
  { parse :: String → Maybe (a, String) }
```

More instances

We make the parser type into an Applicative

```haskell
instance Applicative Parser where
  pure :: a → Parser a
  pure x = P (\inp → Just (x, inp))

  <*> :: Parser (a → b) → Parser a → Parser b
  fp <*> xp = P (\inp1 → case parse fp inp1 of
                          Nothing          → Nothing
                          Just (f, inp2)   → case parse xp of
                            Nothing          → Nothing
                            Just (x, inp3)   → Just (f x, inp3))
```

```haskell
newtype Parser a = P
  { parse :: String → Maybe (a,String) }
```

More instances

We make the parser type into an Applicative

```haskell
instance Applicative Parser where
  pure :: a → Parser a
  pure x = P (\inp → Just (x,inp))

  <*> :: Parser (a → b) → Parser a → Parser b
  fp <*> xp = P (\inp1 → do -- Maybe monad
                  (f,inp2) ← parse fp inp1
                  (x,inp3) ← parse xp inp2
                  return (f x, inp3))
```

Note: Ghc can derive Functor, but not Applicative

Examples

Trivial parser

```
>>> parse (pure 1) "abc"
Just (1,"abc")
```

Parser consuming three characters, discarding the second

```
three :: Parser (Char,Char)
three = pure (\x _y z -> (x,z)) <*> item <*> item <*> item
```

then

```
>>> parse three "abcdef"
Just (('a','c'),"def")
```

```
>>> parse three "ab"
Nothing
```

Convenience combinators

Let's (re)introduce

```
(<$) :: (Functor     f) =>   b -> f a -> f b
(<*) :: (Applicative f) => f b -> f a -> f b
(*>) :: (Applicative f) => f a -> f b -> f b
```

```
x <$ p = (\_ -> x) <$> p
p <* q = (\x _ -> x) <$> p <*> q
p *> q = (\_ y -> y) <$> p <*> q
```

We can now write

```
three :: Parser (Char, Char)
three = (,) <$> item <* item <*> item
```

Mnemonic: operators without > ignore the result of the parser after it.

```
newtype Parser a = P
  { parse :: String → Maybe (a, String) }
```

Monad instance

```
instance Monad Parser where
  (>>=) :: Parser a → (a → Parser b) → Parser b
  p >>= f = P (\inp1 → do -- Maybe monad
                  (x, inp2) ← parse p inp1
                  parse (f x) inp2)
```

Now we can use the do notation.

```
three' :: Parser (Char, Char)
three' = do x ← item
            item
            y ← item
            return (x, y)
```

Making choices

Applicative and Monad operators combine parsers in sequence

- output string from each parser in the sequence becomes input string for the next
- if one parser fails, the sequence fails

We can also combine parsers in parallel

- all parsers get the same input
- if one parser succeeds, the combintation succeeds

Making choices

Use the choice operator from class Alternative

```
class (Applicative f) ⇒ Alternative f where
  empty :: f a
  (<|>) :: f a → f a → f a
```

The operation <|> is associative, and has empty as unit element

Notice the similarity with class Monoid

```
class Monoid m where
  mempty :: m
  (<>)   :: m → m → m
```

Alternative instance for Maybe

```
instance Alternative Maybe where
  empty :: Maybe a
  empty = Nothing

  (<|>) :: Maybe a -> Maybe a -> Maybe a
  Nothing <|> my = my
  Just x  <|> _  = Just x
```

Examples

```
>>> Just 1 <|> Just 2
Just 1
```

```
>>> Nothing <|> Just 2
Just 2
```

```
>>> Just 1 <|> Nothing
Just 1
```

```
>>> Nothing <|> Nothing
Nothing
```

```haskell
newtype Parser a = P
  { parse :: String → Maybe (a, String) }
```

A choice for Parser

```haskell
instance Alternative Parser where
  empty :: Parser a
  empty = P (\inp → Nothing)
  (<|>) :: Parser a → Parser a → Parser a
  p1 <|> p2 = P (\inp → case parse p1 inp of
                          Nothing       → parse p2 inp
                          Just result1 → Just result1)
```

```haskell
newtype Parser a = P
  { parse :: String → Maybe (a, String) }
```

A choice for Parser

```haskell
instance Alternative Parser where
  empty :: Parser a
  empty = P (\inp → empty)
  (<|>) :: Parser a → Parser a → Parser a
  p1 <|> p2 = P (\inp → parse p1 inp <|> parse p2 inp)
```

```haskell
newtype Parser a = P
  { parse :: String → Maybe (a, String) }
```

A choice for Parser

```haskell
instance Alternative Parser where
  empty :: Parser a
  empty = P (\inp → empty)

  (<|>) :: Parser a → Parser a → Parser a
  p1 <|> p2 = P (\inp → parse p1 inp <|> parse p2 inp)
```

Examples

```haskell
>>> parse empty "abc"
Nothing

>>> parse (item <|> pure 'd') "abc"
Just ('a',"bc")

>>> parse (empty <|> pure 'd') "abc"
Just ('d',"abc")
```

Derived primitives

Checking for characters that satisfy predicate p

```
sat :: (Char → Bool) → Parser Char
sat p = do x ← item
           if p x then return x else empty
```

More primitives

```
digit :: Parser Char
digit = sat isDigit

letter :: Parser Char
letter = sat isAlpha

alphanum :: Parser Char
alphanum = sat isAlphaNum

upper :: Parser Char
upper = sat isUpper

lower :: Parser Char
lower = sat isLower

char :: Char → Parser Char
char x = sat (== x)
```

Derived primitives (continued)

A parser for recognizing a complete string

```
string :: String → Parser String
string []     = pure []
string (x:xs) = pure (:) <*> char x <*> string xs
```

Applying a parser several times: many (0 or more), some (at least once)

```
many :: (Alternative f) ⇒ f a → f [a]  -- works for f = Parser
many x = some x <|> pure []
some :: (Alternative f) ⇒ f a → f [a]
some x = pure (:) <*> x <*> many x
```

More primitives

Parser for identifiers, numbers, and spacing

```haskell
ident :: Parser String
ident = (:) <$> lower <*> many alphanum

nat :: Parser Int
nat = read <$> some digit

int :: Parser Int
int = negate <$ char '-' <*> nat
  <|> nat

space :: Parser ()
space = many (sat isSpace) *> pure ()
```

Examples:

```haskell
>>> parse ident "ab4 def"
Just ("ab4"," def")
>>> parse nat "123 abc"
Just (123," abc")
>>> parse int "-123 abc"
Just (-123," abc")
>>> parse (space *> nat)
       "   123"
Just (123,"")
```

Handling spaces

Primitive that ignores spaces before and after applying a parser

```haskell
token :: Parser a → Parser a
token p = space *> p <* space
```

Primitives that ignore spaces

```haskell
natural :: Parser Int
natural = token nat
```

```haskell
symbol :: String → Parser String
symbol xs = token (string xs)
```

Handling spaces

Primitives that ignore spaces

```
natural :: Parser Int
symbol :: String → Parser String
```

Example: parsing an non-empty list of natural numbers

```
nats :: Parser [Int]
nats = do symbol "["
          n ← natural
          ns ← many (do { symbol ","; natural })
          symbol "]"
          return (n:ns)
```

A parser for expressions

Grammar in EBNF

```
expr   ::= term
         |   term "+" expr
term   ::= factor
         |   factor "*" term
factor ::= digit { digit }
         |   "(" expr ")"
```

A parser for expressions

Translation into Haskell (first (incomplete) attempt)

```
expr, term, factor :: Parser ()
expr = term
    <|> do { term; symbol "+"; expr }
term = factor
    <|> do { factor; symbol "*"; term }
factor = do { natural; return () }
       <|> do { symbol "("; expr;  symbol ")"; return () }
```

A parser for expressions – continued

Add "actions" to construct expression tree

```
expr, term, factor :: Parser Expr
expr = term
   <|> do { t ← term; symbol "+"; e ← expr; return (Add t e) }
term = factor
   <|> do { f ← factor; symbol "*"; t ← term; return (Mul f t) }
factor = do { n ← natural; return (Lit n) }
     <|> do { symbol "("; e ← expr;  symbol ")"; return e }
```

A parser for expressions – continued

Add "actions" to construct expression tree (Applicative style)

```
expr, term, factor :: Parser Expr
expr = term
    <|> Add <$> term <* symbol "+" <*> expr
term = factor
    <|> Mul <$> factor <* symbol "*" <*> term
factor = Lit <$> natural
      <|> symbol "(" *> expr <* symbol ")"
```

A parser for expressions – continued

Add "actions" to construct expression tree (Applicative style)

```
expr, term, factor :: Parser Expr
expr = term
    <|> Add <$> term <* symbol "+" <*> expr
term = factor
    <|> Mul <$> factor <* symbol "*" <*> term
factor = Lit <$> natural
      <|> symbol "(" *> expr <* symbol ")"
```

Let's try this parser

```
>>> parse expr "123"
Just (Lit 123,"")
```

A parser for expressions – continued

Add "actions" to construct expression tree (Applicative style)

```
expr, term, factor :: Parser Expr
expr = term
   <|> Add <$> term <* symbol "+" <*> expr
term = factor
   <|> Mul <$> factor <* symbol "*" <*> term
factor = Lit <$> natural
     <|> symbol "(" *> expr <* symbol ")"
```

Let's try this parser

```
>>> parse expr "123 + 5"
Just (Lit 123,"+ 5")          -- unexpected!
```

What's wrong?

$(<|>)$ is left-biased, if the left parser succeeds, then $(<|>)$ succeeds as well without trying the right parser

```
expr = term
    <|> do { t ← term; symbol "+"; e ← expr; return (Add t e) }
```

the second parser wil never succeed if the first didn't.

We first modify our grammar by factoring out common parts.

```
expr   ::= term ("+" expr | "")
term   ::= factor ("*" term | "")
factor ::= digit { digit } | "(" expr ")"
```

A parser for expressions: correct version

Translation into Haskell is straightforward

```
expr = do t ← term
          Add t <$ symbol "+" <*> expr   <|>   return t
term = do f ← factor
          Mul f <$ symbol "*" <*> term   <|>   return f
factor = Lit <$> natural
     <|> symbol "(" *> expr <* symbol ")"
```

Now

```
>>> parse expr "123 + 5"
Just (Add (Lit 123) (Lit 5),"")
```

A parser for expressions: correct version

More examples

```
>>> parse expr "123 + 5"
Just (Add (Lit 123) (Lit 5),"")
>>> parse expr "1 + 2*3"
Just (Add (Lit 1) (Mul (Lit 2) (Lit 3)),"")
>>> parse expr "123 + 5 +"
Just (Add (Lit 123) (Lit 5),"+")
```

More combinators

One or more items, separated by sep

```
sepBy1 :: Parser a → Parser b → Parser [a]
p `sepBy1` sep = (:) <$> p <*> many (sep *> p)
```

Zero or more items, separated by sep

```
sepBy :: Parser a → Parser b → Parser [a]
p `sepBy` sep = p `sepBy1` sep <|> pure []
```

Example:

```
>>> parse (int `sepBy` symbol ",") "1,2,-3"
Just ([1,2,−3],"")
```

Top level parser

Make sure the whole input is consumed

```
parseAll :: Parser a → String → Maybe a
parseAll p inp = case parse p inp of
  Just (x, []) → Just x
  _            → Nothing
```

Example

```
>>> parse expr "1 + "
Just (Lit 1, "+ ")
>>> parseAll expr "1 + "
Nothing
>>> parseAll expr "1 + 2"
Just (Add (Lit 1) (Lit 2))
```

Another pitfall: left-recursion

Consider this grammar

$$\texttt{expr} ::= \texttt{term}\ \texttt{"+"}\ \texttt{expr} \mid \texttt{term}\ \texttt{"-"}\ \texttt{expr} \mid \texttt{term}$$

We want to parse `"1 - 2 + 3"` as $(1 - 2) + 3$, not $1 - (2 + 3)$.

Another pitfall: left-recursion

Consider this grammar

$$\text{expr} ::= \text{term}\ \texttt{"+"}\ \text{expr} \mid \text{term}\ \texttt{"-"}\ \text{expr} \mid \text{term}$$

We want to parse `"1 - 2 + 3"` as $(1 - 2) + 3$, not $1 - (2 + 3)$.

Need a left-recursive grammar

$$\text{expr} ::= \text{expr}\ \texttt{"+"}\ \text{term} \mid \text{expr}\ \texttt{"-"}\ \text{term} \mid \text{term}$$

Another pitfall: left-recursion

Consider this grammar

```
expr ::= term "+" expr | term "-" expr | term
```

We want to parse "1 - 2 + 3" as $(1 - 2) + 3$, not $1 - (2 + 3)$.

Need a left-recursive grammar

```
expr ::= expr "+" term | expr "-" term | term
```

In Haskell:

```haskell
expr = do { t1 ← expr; symbol "+"; t2 ← term; return (Add t1 t2) }
   <|> do { t1 ← expr; symbol "-"; t2 ← term; return (Sub t1 t2) }
   <|> term
```

Another pitfall: left-recursion

Consider this grammar

```
expr ::= term "+" expr | term "-" expr | term
```

We want to parse "1 - 2 + 3" as $(1 - 2) + 3$, not $1 - (2 + 3)$.

Need a left-recursive grammar

```
expr ::= expr "+" term | expr "-" term | term
```

In Haskell:

```
expr = do { t1 ← expr; symbol "+"; t2 ← term; return (Add t1 t2) }
   <|> do { t1 ← expr; symbol "-"; t2 ← term; return (Sub t1 t2) }
   <|> term
```

To parse an expr, first parse an expr...

Left-recursion – solution

All leaves are terms, so

$$\text{expr} ::= \text{term}\ \{\texttt{"+"}\ \text{term}\ \mid\ \texttt{"-"}\ \text{term}\}$$

Left-recursion – solution

All leaves are terms, so

$$\text{expr} ::= \text{term} \ \{\text{"+"} \ \text{term} \ | \ \text{"-"} \ \text{term}\}$$

In Haskell

```
expr :: Parser Expr
expr = foldl (\t suffix → suffix t) <$> term <*> many exprSuffix
exprSuffix :: Parser (Expr → Expr)
exprSuffix = (\t₂ → \t₁ → Add t₁ t₂) <$ symbol "+" <*> term
          <|> (\t₂ → \t₁ → Sub t₁ t₂) <$ symbol "-" <*> term
```

Left-recursion – solution

All leaves are terms, so

```
expr       ::= term exprSuffix
exprSuffix ::= "+" term exprSuffix | "-" term exprSuffix | ""
```

In Haskell

```
expr :: Parser Expr
expr = do { t1 ← term; exprSuffix t1 }
  where
  exprSuffix t1
       = do { symbol "+"; t2 ← term; exprSuffix (Add t1 t2) }
     <|> do { symbol "-"; t2 ← term; exprSuffix (Sub t1 t2) }
     <|> return t1
```

Left-recursion – solution

All leaves are terms, so

```
expr ::= term exprSuffix
exprSuffix ::= "+" term exprSuffix | "-" term exprSuffix | ""
```

In Haskell

```haskell
expr :: Parser Expr
expr = manyInfixL term (Add <$ symbol "+" <|> Sub <$ symbol "-")

manyInfixL :: Parser a -> Parser (a -> a -> a) -> Parser a
manyInfixL px po = px >>= go
  where
  go x1 = do { f <- po; x2 <- px; go (f x1 x2) }
        <|> pure x1
```

Take away

Summary

- Specify a parser with combinators
- Close to EBNF grammar
- Two styles
  - Applicative: Add <$>term <*symbol "+"<*>term
  - Monadic: **do** $\{t_1 \leftarrow$ term; symbol "+"; $t_2 \leftarrow$ term; return (Add t_1 t_2)$\}$
- Libraries for more combinators, error handling, etc.