More on Parsing

Haskell and Cryptocurrencies

Dr. Andres Löh, Well-Typed LLP
Dr. Lars Brünjes, IOHK

2018-01-24

Goals

- The `MonadPlus` class
- Grammar transformations
- Parsing sequences
- Operator precedence
- Parsec

Credits

This lecture is based on Johan Jeuring's lecture on "Languages and Compilers", Utrecht University, 2016–2017.

All errors are of course our own.

Alternative and MonadPlus

Reminder of the last lecture

```
newtype Parser t a = Parser
  {runParser :: [t] -> [(a, [t])]}
(<$>) :: (a -> b) -> Parser t a -> Parser t b
(<$)  :: a -> Parser t b -> Parser t a
```

```
pure  :: a -> Parser t a
(<*>) :: Parser t (a -> b) -> Parser t a
           -> Parser t b
(<*)  :: Parser t a -> Parser t b -> Parser t a
(*>)  :: Parser t a -> Parser t b -> Parser t b
```

Reminder of the last lecture

```
newtype Parser t a = Parser
  {runParser :: [t] -> [(a, [t])]}
empty    :: Parser t a
(<|>)    :: Parser t a -> Parser t a -> Parser t a
many     :: Parser t a -> Parser t [a]
some     :: Parser t a -> Parser t [a]
optional :: Parser t a -> Parser t (Maybe a)
```

Reminder of the last lecture

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

satisfy :: (t -> Bool) -> Parser t t
```

```
token :: Eq t => t -> Parser t ()
```

```
eof :: Parser t ()
```

```
digit :: Parser Char Char
```

```
letter :: Parser Char Char
```

`Alternative` for `Maybe`

Class `Alternative` is not only useful for parsing. Consider the following example:

```
type Name  = String
type Phone = String
```

```
data Person = Person
  { personName      :: Name
  , personHomePhone :: Maybe Phone
  , personWorkPhone :: Maybe Phone
  }
  deriving Show
```

`Alternative` for `Maybe` (contd.)

```
phone :: Person -> Maybe Phone
phone p = case personHomePhone p of
  Just x   -> Just x
  Nothing -> personWorkPhone p
```

If the person has a home phone, we return that. Alternatively, we try to return the work phone.

`Alternative` for `Maybe` (contd.)

Type `Maybe` is an instance of `Alternative`, too:

```
instance Alternative Maybe where
  empty :: Maybe a
  empty = Nothing
  (<|>) :: Maybe a -> Maybe a -> Maybe a
  Just a  <|> _ = Just a
  Nothing <|> b = b
```

`Alternative` for `Maybe` (contd.)

Now we can rewrite `phone`:

```
phone :: Person -> Maybe Phone
phone p = personHomePhone p <|> personWorkPhone p
```

`Alternative` for lists

Lists are `Alternative`, too:

```
instance Alternative [] where
  empty :: [a]
  empty = []
  (<|>) :: [a] -> [a] -> [a]
  (<|>) = (++)
```

guard

For instances of `Alternative`, a very useful function is defined in `Control.Monad`:

```
guard :: Alternative f => Bool -> f ()
guard False = empty
guard True  = pure ()
```

You can use it like this:

```
myFilter :: (a -> Bool) -> [a] -> [a]
myFilter p xs = do
  x <- xs
  guard (p x)
  return x
```

`MonadPlus`

There is another, similar class defined in `Control.Monad`:

```
class (Alternative m, Monad m)
   => MonadPlus m where
  mzero :: m a
  mplus :: m a -> m a -> m a
```

- `Alternative` is to `Applicative` as `MonadPlus` is to `Monad`.
- Often we have

```
mzero = empty
mplus = (<|>)
```

- `Maybe` and lists are instances of `MonadPlus`, too.

Parsers are `Monad` and `MonadPlus`

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

Parsers are monads, too!

```haskell
instance Monad (Parser t) where
  return :: a -> Parser t a
  return a = Parser $ \ ts -> [(a, ts)]
  (>>=) :: Parser t a -> (a -> Parser t b)
        -> Parser t b
  p >>= cont = Parser $ \ ts -> do
    (a, ts') <- runParser p ts
    runParser (cont a) ts'
```

Parsers are `Monad` and `MonadPlus`

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

And they are `MonadPlus`:

```haskell
instance MonadPlus (Parser t) where

  mzero :: Parser t a
  mzero = Parser $ const []

  mplus :: Parser t a -> Parser t a -> Parser t a
  p `mplus` q = Parser $ \ ts ->
    runParser p ts ++ runParser q ts
```

Grammar Transformations

Our example from last time...

$$S \to D{+}S \mid D$$
$$D \to 0 \mid 1$$

```
data S = Plus D S | Digit D
data D = Zero | One
```

```
parseS :: Parser Char S
parseS =
      Plus  <$> parseD <* token '+' <*> parseS
  <|> Digit <$> parseD
```

```
parseD :: Parser Char D
parseD =
      Zero <$ token '0'
  <|> One  <$ token '1'
```

… slightly(?) changed

$S \rightarrow S\text{-}D \mid D$

$D \rightarrow 0 \mid 1$

```
data S = Minus S D | Digit D
data D = Zero | One
```

```
parseS :: Parser Char S
parseS =
      Minus <$> parseS <* token '-' <*> parseD
   <|> Digit <$> parseD
```

```
parseD :: Parser Char D
parseD =
      Zero <$ token '0'
   <|> One  <$ token '1'
```

… slightly(?) changed

$$S \to S\text{-}D \mid D$$

$$D \to 0 \mid 1$$

```haskell
data S = Minus S D | Digit D
data D = Zero | One
```

```
GHCi> runParser parseS "1-0-1"
… infinite loop!
```

What's going on?

Left recursion

- A production is called left-recursive if the right hand side starts with the nonterminal on the left hand side.
- Example: production $S \rightarrow S\texttt{-}D \mid D$ from the last slide.
- A grammar is called left-recursive if there is a derivation $A \Rightarrow \ldots \Rightarrow Az$ for some nonterminal A of the grammar.
- Grammars can be indirectly left-recursive, i.e. without having a left-recursive production.

Left recursion and parsers

- A left-recursive production $A \rightarrow Az$ corresponds to a parser `a = a <*> z`.
- Such a parser loops!
- Removing left-recursion from grammars is essential for combinator parsing!

Removing left recursion

- Transforming a (directly) left-recursive nonterminal A such that the left-recursion is removed is relatively simple:
- First split the productions for A into left-recursive ones and others:

$$
\begin{aligned}
A &\rightarrow Ax_1 \mid Ax_2 \mid \ldots \mid Ax_n \\
A &\rightarrow y_1 \mid y_2 \mid \ldots \mid y_m
\end{aligned}
$$

- This grammar can be transformed to:

$$
\begin{aligned}
A &\rightarrow y_1 \mid y_1Z \mid y_2 \mid y_2Z \mid \ldots \mid y_m \mid y_mZ \\
Z &\rightarrow x_1 \mid x_1Z \mid x_2 \mid x_2Z \mid \ldots \mid x_n \mid x_nZ
\end{aligned}
$$

Removing left recursion (example)

- Let's try this for our left-recursive example nonterminal S!
- There is one left-recursive production and one other (so $n = m = 1$):

$$S \rightarrow S\texttt{-}D$$
$$S \rightarrow D$$

- So as transformation we get:

$$S \rightarrow D \mid DZ$$
$$Z \rightarrow \texttt{-}D \mid \texttt{-}DZ$$

Removing left recursion (example contd.)

$$S \to D \mid DZ$$
$$Z \to \texttt{-}D \mid \texttt{-}DZ$$
$$D \to 0 \mid 1$$

```
data S = Digit D | Minus D Z
data Z = Digit' D | Minus' D Z
data D = Zero | One
```

```
parseS =
      Digit <$> parseD
  <|> Minus <$> parseD <*> parseZ
parseZ =
      Digit' <$> (token '-' *> parseD)
  <|> Minus' <$> (token '-' *> parseD) <*> parseZ
parseD = Zero <$ token '0' <|> One <$ token '1'
```

Removing left recursion (example contd.)

$$S \to D \mid DZ$$

$$Z \to \texttt{-}D \mid \texttt{-}DZ$$

$$D \to 0 \mid 1$$

```
data S = Digit D | Minus D Z
data Z = Digit' D | Minus' D Z
data D = Zero | One
```

Now it works:

```
GHCi> runParser (parseS <* eof) "1-0-1"
[(Minus One (Minus' Zero (Digit' One)), "")]
```

Left factoring

- If several productions of a nonterminal in a grammar have a common prefix, we can perform left factoring.
- The longer the common prefix and the more productions share that prefix, the more useful left factoring becomes.
- Left factoring of a grammar corresponds to optimization of the parser. Depending on the grammar and the parser combinators used, it can be absolutely essential.

Left factoring (example)

- Let's look at our example grammar again:

$$S \rightarrow D \mid DZ$$
$$Z \rightarrow \texttt{-}D \mid \texttt{-}DZ$$
$$D \rightarrow 0 \mid 1$$

- Left factoring on S (common prefix D) yields

$$S \rightarrow D\left[\epsilon \mid Z\right] = D\,Z?$$

- Left factoring on Z (common prefix $\texttt{-}D$) yields

$$Z \rightarrow \texttt{-}D\left[\epsilon \mid Z\right] = \texttt{-}D\,Z?$$

Left factoring (example contd.)

$$S \rightarrow D\ Z?$$
$$Z \rightarrow \texttt{-}D\ Z?$$
$$D \rightarrow 0 \mid 1$$

```haskell
data S = S D (Maybe Z)
data Z = Z D (Maybe Z)
data D = Zero | One
```

```haskell
parseS = S <$> parseD <*> optional parseZ
```

```haskell
parseZ = Z <$> (token '-' *> parseD)
           <*> optional parseZ
```

```haskell
parseD = Zero <$ token '0' <|> One <$ token '1'
```

Left factoring (example contd.)

$$
\begin{aligned}
S &\rightarrow D\ Z? \\
Z &\rightarrow \texttt{-}D\ Z? \\
D &\rightarrow 0 \mid 1
\end{aligned}
$$

```haskell
data S = S D (Maybe Z)
data Z = Z D (Maybe Z)
data D = Zero | One
```

```
GHCi> runParser (parseS <* eof) "1-0-1"
[(S One (Just (Z Zero (Just (Z One Nothing))))
, "")]
```

Parsing Sequences

Associative separators

- Consider the grammar

$$S \rightarrow S\,\mathtt{;}\,S \mid A$$

- The grammar is left-recursive and ambiguous.
- However, we could argue that this is no problem if the intended meaning of the different parse trees is the same, i.e. if `;` is assosiative:

$$(A\,\mathtt{;}\,A)\,\mathtt{;}\,A = A\,\mathtt{;}\,(A\,\mathtt{;}\,A)$$

- For this situation, we can define a special combinator `listOf` that simply collects all elements separated by a given separator into a list.

Associative separators (contd.)

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

```
GHCi> runParser (listOf digit (token ';') <* eof)
  "1;2;3;4"
[("1234", "")]
```

Left associative operators

- What if instead of an associative separator, we have a left associative operator (like `+` or `-`)?
- For example, we might want to parse something like `2-3+4` (as an `Int`).
- Idea: Given a `Parser t a` for the operands and a `Parser t (a -> a -> a)` for the operators, we parse the first operand and then `many` operator-operand pairs:

```
chainl :: Parser t a -> Parser t (a -> a -> a)
  -> Parser t a
chainl p s = foldl' (flip ($))
  <$> p
  <*> many (flip <$> s <*> p)
```

Left associative operators (contd.)

To try this out, we need parsers for plus/minus and integers:

```
pm :: Parser Char (Int -> Int -> Int)
pm = (+) <$ token '+' <|> (-) <$ token '-'
```

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

```
sign :: Parser Char (Int -> Int)
sign = maybe id (const negate)
  <$> optional (token '-')
```

```
int :: Parser Char Int
int = ($) <$> sign <*> nat
```

Left associative operators (contd.)

```
GHCi> runParser (int <* eof) "123"
[(123, "")]
```

```
GHCi> runParser (int <* eof) "-123"
[(- 123, "")]
```

```
GHCi> runParser (chainl int pm <* eof) "2-3+4"
[(3, "")]
```

Right associative operators

For right associative operators, we first parse *many* operand-operator pairs, then a final operand:

```
chainr :: Parser t a -> Parser t (a -> a -> a)
  -> Parser t a
chainr p s = flip (foldr ($))
  <$> many (flip ($) <$> p <*> s)
  <*> p
```

```
GHCi> runParser (chainr int pm <* eof)
 "2-3+4"
[(- 5, "")]
```

Operator precedence

Operator precedence

- Consider the grammar

$$
\begin{aligned}
E &\rightarrow E\texttt{+}E \\
E &\rightarrow E\texttt{-}E \\
E &\rightarrow E\texttt{*}E \\
E &\rightarrow \texttt{(}E\texttt{)} \\
E &\rightarrow Int
\end{aligned}
$$

- This is a typical grammar for expressions with operators.
- For the same reasons as above, this grammar is ambiguous.
- Given the precedence of the operators and their associativity, we can transform the grammar so that the ambiguity is removed.

Operator precedence (contd.)

- The basic idea is to parse operators of different precedences sequentially.
- For each precedence level i we get:

$$E_i \rightarrow E_i\ Op_i\ E_{i+1} \mid E_{i+1} \qquad \text{(for left-associative operators)}$$

or

$$E_i \rightarrow E_{i+1}\ Op_i\ E_i \mid E_{i+1} \qquad \text{(for right-associative operators)}$$

or

$$E_i \rightarrow E_{i+1}\ Op_i\ E_{i+1} \mid E_{i+1} \qquad \text{(for non-associative operators)}$$

- The highest level contains the remaining productions.
- All forms of bracketing point to the lowest level of expressions.

Operator precedence (contd.)

- Applied to

$$
\begin{aligned}
E &\rightarrow E\texttt{+}E \\
E &\rightarrow E\texttt{-}E \\
E &\rightarrow E\texttt{*}E \\
E &\rightarrow \texttt{(}E\texttt{)} \\
E &\rightarrow Int
\end{aligned}
$$

- we obtain

$$
\begin{aligned}
E_1 &\rightarrow E_1\ Op_1\ E_2 \mid E_2 \\
E_2 &\rightarrow E_2\ Op_2\ E_3 \mid E_3 \\
E_3 &\rightarrow \texttt{(}E_1\texttt{)} \mid Int \\
Op_1 &\rightarrow \texttt{+} \mid \texttt{-} \\
Op_2 &\rightarrow \texttt{*}
\end{aligned}
$$

Operator precedence (contd.)

$$
\begin{aligned}
E_1 &\to E_1\ Op_1\ E_2 \mid E_2 \\
E_2 &\to E_2\ Op_2\ E_3 \mid E_3 \\
E_3 &\to (\ E_1\ ) \mid Int \\
Op_1 &\to \texttt{+} \mid \texttt{-} \\
Op_2 &\to \texttt{*}
\end{aligned}
$$

```
data E = Plus E E
       | Minus E E
       | Times E E
       | Lit Int
```

```
e1, e2, e3 :: Parser Char E
e1 = chainl e2 op1
e2 = chainl e3 op2
e3 = token '(' *> e1 <* token ')' <|> Lit <$> int
```

```
op1, op2 :: Parser Char (E -> E -> E)
op1 = Plus <$ token '+' <|> Minus <$ token '-'
op2 = Times <$ token '*'
```

A general operator parser

Using `msum` from `Data.Foldable`, we can do even better:

```
msum :: (Foldable t, MonadPlus m) => t (m a) -> m a
```

```
type Op a = (Char, a -> a -> a)
```

```
gen :: [Op a] -> Parser Char a -> Parser Char a
gen ops p = chainl p $
  msum $ map (\ (s, f) -> f <$ token s) ops
```

```
e1 = gen [('+', Plus), ('-', Minus)] e2
e2 = gen [('*', Times)] e3
```

Parsec

Parsec

- popular industrial strength parser combinator library written by Daan Leijen
- ported to OCaml, Java, C#, F#, Ruby, Erlang, C++, Python, JavaScript,...
- support arbitrary token types
- good error messages
- no multiple results: either succeeds or fails with an error message

Choice in Parsec

- The `Alternative` instance only tries the second alternative if first failed and didn't consume input.
- This is done for efficiency (*LL*(1) grammar).
- However, there is an ''escape hatch'': `try :: Parser a -> Parser a`. If `try p` *fails*, it "pretends" not to have consumed input.

Error messages in Parsec

- One of Parsec's strengths is the quality of its error messages.
- In case of a parse error, position of and reason for the error are given.
- Using the operator `(<?>) :: Parser a -> String -> Parser a`, error messages can be customized.

Parsec Demo

Parsec Demo