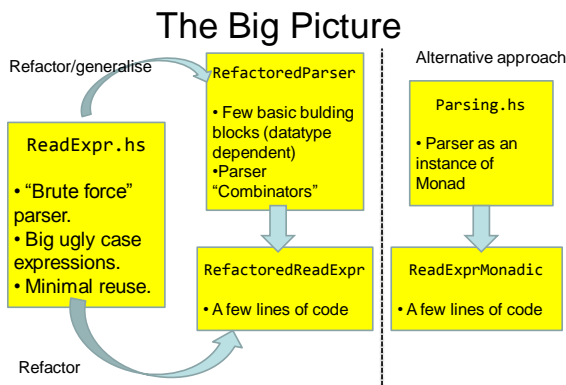


Monadic Parsing

David Sands



Basic Parsers

Some very simple building blocks

```
succeed :: a -> Parser a
succeed a = P $ \s -> Just(a,s)

failure :: Parser a
failure = P $ \s -> Nothing

item = P $ \s -> case s of
  (c:s') -> Just (c,s')
  ""      -> Nothing
```

Parsing

- So far: how to write

```
readExpr :: String -> Maybe Expr
```

- Key idea:

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

- This lecture: Building Parsers; Parsers as a new type of "instructions" – i.e. a monad.

A New Type for Parsers

Make parsers into a new type:

```
data Parser a = P (String -> Maybe (a,String))
```

need this for later

Now we need a function to apply a parser:

```
parse :: Parser a -> String -> Maybe (a,String)
parse (P p) s = p s
```

Basic Parsers

Lets define some functions to build some basic parsers

```
sat :: (Char -> Bool) -> Parser Char
-- later we will redefine this using item
sat prop = P (\s -> case s of
  (c:cs) | prop c -> Just (c,cs)
  _        -> Nothing )

number :: Parser Char
number = sat isDigit

char :: Char -> Parser Char
char x = sat (== x)
```

Building Parsers from Parsers

`p+++q` – first parse with `p`; if it fails then parse with `q`

```
(+++) :: Parser a -> Parser a -> Parser a
p +++ q = P $ \s -> case parse p s of
    Nothing      -> parse q s
    (Just (v,rest)) -> Just (v,rest)
```

```
digitOrBracket :: Parser Char
digitOrBracket = number +++ char '('
```

```
Main> digitOrBracket "12hello"
Just ('1', "2hello")
Main> digitOrBracket "xyz"
Just ('(', "xyz")
```

parse p then parse q

```
(>->) :: Parser a -> Parser b -> Parser b
p >-> q = P $ \s -> case parse p s of
    Just (a,s1) -> parse q s1
    Nothing     -> Nothing
```

```
bracketDigit :: Parser Char
bracketDigit = char '(' >-> number >-> char ')'
```

```
Main> parse bracketDigit "(9)abc"
Just (Num '9', "abc")
Main> parse bracketDigit "(9abc"
Nothing
```

`p >-> q` "throws away" what has been parsed by `p`

`p >*> f`

How we parse "b" will depend on what value we got from the first parse

```
(>*>) :: Parser a -> (a -> Parser b) -> Parser b
p >*> f = P $ \s -> case parse p s of
    Just (a,s1) -> parse (f a) s1
    Nothing     -> Nothing
```

```
wrapped :: Parser Char
wrapped = sat isAlpha >*>
    \c -> sat isSpace >-> char c
```

```
Main> parse wrapped "X Xabc"
Just ('X', "abc")
```

```
Main> wrapped "X Yabc"
Nothing
```

`p >*> f`

`>*>` can be used to define earlier operations

```
sat :: (Char -> Bool) -> Parser Char
sat p = item >*> \a -> if p a then succeed a else failure
```

```
p >-> q = p >*> \_ -> q
```

Monads and do notation

- To be an instance of class `Monad` you need (as a minimal definition) two operations: `>>=` and `return`

```
Main> :i Monad
class Monad m where
  (>>=) :: m a -> (a -> m b) -> m b
  (>>)  :: m a -> m b -> m b
  return :: a -> m a
  fail  :: String -> m a

instance Monad IO -- Defined in GHC.IOBase
```

Monad

- To be an instance of class `Monad` you need two operations: `>>=` and `return`

```
instance Monad Parser where
  return = succeed
  (>>=) = (>*>)
  -- (>->) is equivalent to (>>)
```

- Why bother?

- First example of a home-grown monad
- Can understand and use `do` notation

The truth about Do

- Do syntax is just a shorthand:

```
do act1
  act2 == act1 >> act2 == act1 >>= \_ -> act2
```

```
do v <- act1
  act2 == act1 >>= \v -> act2
```

Can you figure out the general case for the translation?

Example

- recall doTwice

```
doTwice :: Monad m => m a -> m (a,a)
doTwice cmd =
  do a <- cmd
     b <- cmd
     return (a,b)
```

```
Main> parse (doTwice number) '9876"
Just (('9','8'), "76")
```

Example revisited: Parsing Expressions

```
expr :: Parser Expr
expr s1 = case parse num s1 of
  Just (a,s2) -> case s2 of
    '+' : s3 -> case parse expr s3 of
      Just (b,s4) -> Just (Add a b, s4)
      Nothing -> Just (a,s2)
    _ -> Just (a,s2)
  Nothing -> Nothing
```

modified to use the new version of Parser type. Otherwise as before

Monadic style abstracts away from implementation of the Parser type

```
expr :: Parser Expr
expr = do a <- num
       do char '+'
         b <- expr
         return (Add a b)
       +++ return a
```

Parser Combinators

```
zeroOrMore, oneOrMore :: Parser a -> Parser [a]
zeroOrMore p = oneOrMore p +++ return []
oneOrMore p = do v <- p
                vs <- zeroOrMore p
                return (v:vs)
```

```
Main> parse (oneOrMore number) '9876+"
Just ("9876","+")
```

Combinator: a function which take functions as arguments and produces a function as a result

Parser Combinators

```
nat :: Parser Int -- Parses a non negative integer
nat = do xs <- oneOrMore number
      return (read xs)

int :: Parser Int
int = nat +++
      do char '-'
        n <- nat
        return (-n)
```

Chain

```
chain p op f = P $ \s1 ->
  case parse p s1 of
    Just (a,s2) -> case s2 of
      c:s3 | c == op -> case chain p op f s3 of
        Just (b,s4) -> Just (f a b, s4)
        Nothing -> Just (a,s2)
      _ -> Just (a,s2)
    Nothing -> Nothing
```

Old definition (modified to work with the new type)

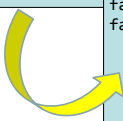
```
chain p op f = do v <- p
                vs <- zeroOrMore (char op >> p)
                return (foldr1 f (v:vs))
```

Prelude.foldr1: fold operation for lists with at least one element (no "nil" case)

Factor

```
factor :: Parser Expr
factor (' ':s) =
  case expr s of
    Just (a, '):s1 -> Just (a, s1)
    _               -> Nothing
factor s = num s
```

```
factor :: Parser Expr
factor = num +++
  do char '('
    e <- expr
    char ')'
  return e
```



Summary

- We can use higher-order functions to build Parsers from other more basic Parsers.
- Parsers can be viewed as an instance of Monad
- We can build our own Monads!
 - A lot of "plumbing" is nicely hidden away
 - The implementation of the Monad is not visible and can thus be changed or extended

IO t

- Instructions for interacting with operating system
- Run by GHC runtime system produce value of type t

Gen t

- Instructions for building random values
- Run by **quickCheck** to generate random values of type t

Parser t

- Instructions for parsing
- Run by **parse** to parse a string and **Maybe** produce a value of type t

Three Monads

Code

- Parsing.hs
 - module containing the parser monad and simple parser combinators.
- ReadExprMonadic.hs
 - A reworking of Read

See course home page