Combinator Parsing

Dr. Mattox Beckman

University of Illinois at Urbana-Champaign
Department of Computer Science

Objectives

- Show how to build complex parsers by composing simpler parsers.
- Use monads to hide the mechanics of plumbing the input.
- Build a small parser library similar to the `Parsec` combinator parser library in Haskell.

The Problem

- Recursive descent parsers are easy to write.
  - But plumbing the input is a bit tedious.
  - And sometimes the common prefix problem is a real problem.
  - And we can't really *compose* them.
- So we'll build a parser *combinator* library instead.

A Parser

- We begin by defining a type.
- The `newtype` is like `data` but with only one constructor.
  - Compiler can handle this more efficiently.
- The `run` function unboxes a parser so we can run it.

```
newtype Parser t = Parser (String -> [(t,String)])
run (Parser p) = p
```

Our First Parser: Parsing a Character

The char Parser

```
char s =
  Parser (\inp -> case inp of
                  (x:xs) | s == x    -> [(x,xs)]
                  otherwise          -> [])
```

- Single quotes are for single characters.
- Double quotes are for strings (lists of characters).

```
Main> run (char 'a') "asdf"
[('a',"sdf")]
Main> run (char 'a') "qwert"
[]
```


Predicates and Parsers

- `oneOf` takes a list of characters and succeeds if the input is one of them.
- In real life you might want to build a lookup table.

```
oneOf xx =
  Parser (\inp -> case inp of
                  (s:ss) | s `elem` xx -> [(s,ss)]
                  otherwise            -> [])

digit = oneOf ['0'..'9']
```

```
Main> run (oneOf "asb") "sb"
[('s',"b")]
Main> run (oneOf "asb") "xsb"
[]
Main> run digit "42"
[('4',"2")]
```


Making It a Higher Order Function

- `sat` takes a predicate that it can run on the character.
- Compare with `oneOf`.

```
oneOf xx =
  Parser (\inp -> case inp of
                  (s:ss) | s `elem` xx -> [(s,ss)]
                  otherwise            -> [])

sat pred =
  Parser (\inp -> case inp of
                  (s:ss) | pred s    -> [(s,ss)]
                  otherwise          -> [])

digit = sat (\x -> x >= '0' && x <= '9')
```


Adding a Choice Operator

- We want to compose two parsers together.
- If the first fails, we will try the second.

```
(Parser p1) <|> (Parser p2) =
  Parser (\inp -> take 1 $ p1 inp ++ p2 inp)
```

```
Main> run (digit <|> (char 'a')) "12ab"
[('1',"2ab")]
Main> run (digit <|> (char 'a')) "a2ab"
[('a',"2ab")]
Main> run (digit <|> (char 'a')) "xa2ab"
[]
```

Recursion

- Come and see the plumbing inherent in the system!

```
rstring [] = Parser (\inp -> [([],inp)])
rstring (s:ss) = Parser (\inp ->
    case run (char s) inp of
      [(c,r1)] -> case run (rstring ss) r1 of
                     [(cs,rr)] -> [(c:cs,rr)]
                     _ -> []
      _        -> [])
```

```
> run (rstring "Arthur Dent") "Arthur Dent"
[("Arthur Dent","")]
```

- We have created a parser using recursion, but this is painful.
- What operation do you know that unpacks a data structure, propagates success cases, and aborts computation after a failure?

Enter the Monad – Functor

```
instance Functor Parser where
  fmap f (Parser  p1) =
      Parser (\inp -> [(f t, s) |
                         (t,s) <- p1 inp])

sdi :: Parser Integer
sdi = Parser (\inp -> case run digit inp of
                         [(d, dd)] -> [(read [d], dd)]
                         otherwise -> [])
```

- sdi = “single digit integer,” not “strategic defense initiative”

```
Main> run sdi "123"
[(1,"23")]
Main> run (fmap (+1) sdi) "123"
[(2,"23")]
```


Enter the Monad – Applicative

```
instance Applicative Parser where
  pure a = Parser (\inp -> [(a,inp)])
  (Parser p1) <*> (Parser p2) =
      Parser (\inp -> [(v1 v2, ss2) |
                         (v1,ss1) <- p1 inp,
                         (v2,ss2) <- p2 ss1])
```

```
Main> run ( (+) <$> sdi <*> sdi) "456"
[(9,"6")]
```


Enter the Monad

- Remember that `f` takes data from the first parser and returns a new parser.

```
instance Monad Parser where
  (Parser p) >>= f =
      Parser (\inp -> concat [run (f v) inp'
                              | (v,inp') <- p inp])
```

```
Main> run (sdi >>= (\x -> sdi >>= (\y -> return $ x + y)))
          "8675309"
[(14,"75309")]
Main> run (do x <- sdi
              y <- sdi
              return $ x + y }) "123"
[(3,"3")]
```

Recursion, Revisited

- Using do notation, we can really clean up our code.

Before

```
rstring [] = Parser (\inp -> [([],inp)])
rstring (s:ss) = Parser (\inp ->
    case run (char s) inp of
      [(c,r1)] -> case run (rstring ss) r1 of
                    [(cs,rr)] -> [(c:cs,rr)]
                    _ -> []
      _         -> [])
```


Recursion, Revisited

- Using do notation, we can really clean up our code.

After

```
string [] = Parser (\inp -> [([],inp)])
string (s:ss) = do v <- char s
                   vv <- string ss
                   return $ v:vv
```


Many and Many1

```
many p = next <|> return ""
  where next = do v <- p
                  vv <- many p
                  return (v:vv)

many1 p = do v <- p
             vv <- many p
             return (v:vv)

spaces = many (oneOf " ")
```


Returning a Type

```
data Exp = IntExp Integer
         | OpExp String Exp Exp
    deriving Show

int :: Parser Exp
int = do digits <- many1 digit
         spaces
         return (IntExp $ read digits)
```

```
Main> run int "1234 567"
[(IntExp 1234,"567")]
Main> run (many int) "10 20 30 40"
[([IntExp 10,IntExp 20,IntExp 30,IntExp 40],"")]
```

Operators

```
oper o = do v <- string o
            spaces
            return $ OpExp v
chainl1 p op = p >>= rest
  where rest x = do o <- op
                    v <- p
                    rest (o x v)
                  <|> return x
expr = chainl1 term (oper "+")
term = int <|> parens expr
```

```
Main> run expr "10 + 20 + 30"
[(OpExp "+" (OpExp "+" (IntExp 10) (IntExp 20)) (IntExp 30),"")]
Main> run expr "10 + (20 + 30)"
[(OpExp "+" (IntExp 10) (OpExp "+" (IntExp 20) (IntExp 30)),"")]
```


Longer Example

```
expr = disj `chainl1` orOp
disj = conj `chainl1` andOp
conj = arith `chainl1` compOp
arith = term `chainl1` addOp
term = factor `chainl1` mulOp
factor = atom
```

```
atom = intExp
   <|> ifExp
   <|> try boolExp
   <|> funExp
   <|> appExp
   <|> letExp
   <|> varExp
   <|> parens expr
```


Longer Example, II

```
letExp = do try $ symbol "let"
            symbol "["
            params <- many $ do v <- var
                                e <- expr
                                return (v,e)
            symbol "]"
            body <- expr
            symbol "end"
            return $ LetExp params body
```

- The `try` allows for backtracking.
- There are many packages: `parsec`, `attoparsec`, and `megaparsec`.