Introduction
000000

Choices and Recursion
00
00000

Repeating and Composing
00000

Combinator Parsing

Dr. Mattox Beckman

University of Illinois at Urbana-Champaign
Department of Computer Science

Introduction
●○○○○○

Choices and Recursion
○○
○○○○○

Repeating and Composing
○○○○○

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.

Introduction
○●○○○○

Choices and Recursion
○○
○○○○○

Repeating and Composing
○○○○○

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.

Introduction
○○●○○○

Choices and Recursion
○○
○○○○○

Repeating and Composing
○○○○○

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.

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

Introduction
000●00

Choices and Recursion
00
00000

Repeating and Composing
00000

Our First Parser: Parsing a Character

The char Parser

```
1 char s =
2   Parser (\inp -> case inp of
3                   (x:xs) | s == x    -> [(x,xs)]
4                   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"
[]
```

Introduction
○○○○●○

Choices and Recursion
○○
○○○○○

Repeating and Composing
○○○○○

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.

```
1 oneOf xx =
2   Parser (\inp -> case inp of
3                     (s:ss) | s `elem` xx -> [(s,ss)]
4                     otherwise           -> [])
5
6 digit = oneOf ['0'..'9']
```

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

Introduction
○○○○○●

Choices and Recursion
○○
○○○○○

Repeating and Composing
○○○○○

Making It a Higher Order Function

▶ sat takes a predicate that it can run on the character.

▶ Compare with oneOf.

```
1 oneOf xx =
2   Parser (\inp -> case inp of
3                   (s:ss) | s `elem` xx -> [(s,ss)]
4                   otherwise            -> [])
5
6 sat pred =
7   Parser (\inp -> case inp of
8                   (s:ss) | pred s   -> [(s,ss)]
9                   otherwise         -> [])
10
11 digit = sat (\x -> x >= '0' && x <= '9')
```

Introduction
000000

Choices and Recursion
●○
○○○○○

Repeating and Composing
○○○○○

Adding a Choice Operator

▶ We want to compose two parsers together.

▶ If the first fails, we will try the second.

```
1 (Parser p1) <|> (Parser p2) =
2    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"
[]
```

Introduction
000000

Choices and Recursion
0●
00000

Repeating and Composing
00000

Recursion

▶ Come and see the plumbing inherent in the system!

```
1 rstring [] = Parser (\inp -> [([],inp)])
2 rstring (s:ss) = Parser (\inp ->
3     case run (char s) inp of
4       [(c,r1)] -> case run (rstring ss) r1 of
5                   [(cs,rr)] -> [(c:cs,rr)]
6                   _ -> []
7       _         -> [])
```

```
> 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?

Introduction
000000

Choices and Recursion
00
●0000

Repeating and Composing
00000

Enter the Monad – Functor

```
1 instance Functor Parser where
2   fmap f (Parser  p1) =
3       Parser (\inp -> [(f t, s) |
4                       (t,s) <- p1 inp])
5
6 sdi :: Parser Integer
7 sdi = Parser (\inp -> case run digit inp of
8                       [(d, dd)] -> [(read [d], dd)]
9                       otherwise -> [])
```

▶ sdi = "single digit integer," not "strategic defense initiative"

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

Introduction
000000

Choices and Recursion
00
0●000

Repeating and Composing
00000

Enter the Monad – Applicative

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

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

Introduction
000000

Choices and Recursion
00
00●00

Repeating and Composing
00000

Enter the Monad

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

```
1 instance Monad Parser where
2   (Parser p) >>= f =
3       Parser (\inp -> concat [run (f v) inp'
4                               | (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")]
```

Introduction
000000

Choices and Recursion
00
00000

Repeating and Composing
00000

Recursion, Revisited

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

Before

```
1 rstring [] = Parser (\inp -> [([],inp)])
2 rstring (s:ss) = Parser (\inp ->
3    case run (char s) inp of
4      [(c,r1)] -> case run (rstring ss) r1 of
5                    [(cs,rr)] -> [(c:cs,rr)]
6                    _ -> []
7      _         -> [])
```

Introduction
000000

Choices and Recursion
00
0000●

Repeating and Composing
00000

Recursion, Revisited

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

After

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

Introduction
000000

Choices and Recursion
00
00000

Repeating and Composing
●0000

Many and Many1

```
1 many p = next <|> return ""
2   where next = do v <- p
3                   vv <- many p
4                   return (v:vv)
5
6 many1 p = do v <- p
7              vv <- many p
8              return (v:vv)
9
10 spaces = many (oneOf " ")
```

Introduction
000000

Choices and Recursion
00
00000

Repeating and Composing
0●000

Returning a Type

```
1 data Exp = IntExp Integer
2          | OpExp String Exp Exp
3     deriving Show
4
5 int :: Parser Exp
6 int = do digits <- many1 digit
7          spaces
8          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],"")]
```

Introduction
000000

Choices and Recursion
00
00000

Repeating and Composing
00●00

Operators

```
1 oper o = do v <- string o
2            spaces
3            return $ OpExp v
4 chainl1 p op = p >>= rest
5   where rest x = do o <- op
6                     v <- p
7                     rest (o x v)
8                 <|> return x
9 expr = chainl1 term (oper "+")
10 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)),"")]
```

Introduction
000000

Choices and Recursion
00
00000

Repeating and Composing
000●0

Longer Example

```
1 expr = disj `chainl1` orOp
2 disj = conj `chainl1` andOp
3 conj = arith `chainl1` compOp
4 arith = term `chainl1` addOp
5 term = factor `chainl1` mulOp
6 factor = atom
```

```
7  atom = intExp
8     <|> ifExp
9     <|> try boolExp
10    <|> funExp
11    <|> appExp
12    <|> letExp
13    <|> varExp
14    <|> parens expr
```

Introduction
000000

Choices and Recursion
00
00000

Repeating and Composing
0000●

Longer Example, II

```
1 letExp = do try $ symbol "let"
2             symbol "["
3             params <- many $ do v <- var
4                                 e <- expr
5                                 return (v,e)
6             symbol "]"
7             body <- expr
8             symbol "end"
9             return $ LetExp params body
```

▶ The try allows for backtracking.

▶ There are many packages: parsec, attoparsec, and megaparsec.