Building the Parsing Library

Last time we saw

A library for building parsers containing:

- An abstract data type `Parser a`
- A function

```
parse ::
 Parser a -> String -> Maybe(a,String)
```

- Basic building blocks for building parsers

We also saw

A specific parser (for Expr) built from scratch, based on

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

Recap of Parsing.hs

[See course home page for API and source]

Parser implements the Monad type class

For now, that just means that we can use "do" notation to build parsers, just like for IO and Gen

IO

```
do
 s <- getLine
 c <- readFile s
 return $ s ++ c
```

Gen

```
do
 n <- elements[1..9]
 m <- vectorOf n arbitrary
 return $ n:m
```

Parser

```
do
 c <- sat (`elem` ";,:")
 ds <- chain digit (char c)
 return ds
```

IO t	**Gen t**	**Parser t**
• Instructions for interacting with operating system • Run by GHC runtime system produce value of type t	• Instructions for building random values • Run by **quickCheck** to generate random values of type t	• Instructions for parsing • Run by **parse** to parse a string and produce a **Maybe** t

Example, a CSV file

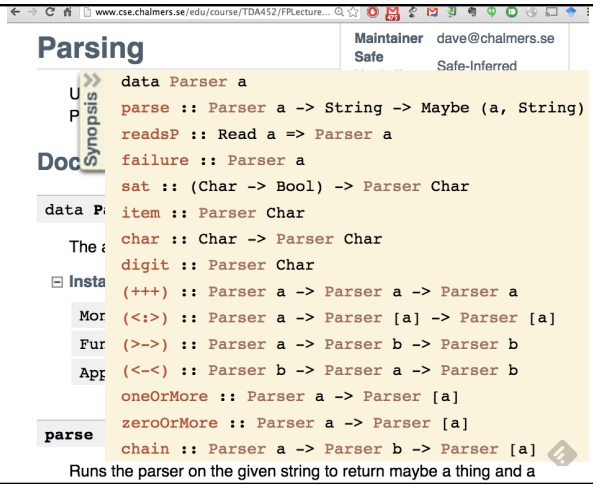
Year	Make	Model	Description	Price
1997	Ford	E350	ac, abs, moon	3000.00
1999	Chevy	Venture "Extended Edition"		4900.00
1999	Chevy	Venture "Extended Edition, Very Large"		5000.00
1996	Jeep	Grand Cherokee	MUST SELL! air, moon roof, loaded	4799.00

Example, a CSV file

The above table of data may be represented in CSV format as follows:

```
Year,Make,Model,Description,Price
1997,Ford,E350,"ac, abs, moon",3000.00
1999,Chevy,"Venture ""Extended Edition""","",4900.00
1999,Chevy,"Venture ""Extended Edition, Very
Large""",,5000.00
1996,Jeep,Grand Cherokee,"MUST SELL!
air, moon roof, loaded",4799.00
```

wikipedia



Example & Implementation

FPLectures/CSVexample.hs

FPLectures/Parsing.hs

A New Type for Parsers

Make parsers into a new type:

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

Need this for later to:
- hide inner workings
- add to class Monad

Now we need a function to apply a parser:

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

Basic parsers (1)

```
success :: a -> Parser a
success 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
```

Always succeeds in producing an a without consuming any of the input string

Always fails

parses a single Char

Not so useful on their own – but will be handy in combination with other parsers…

Basic parsers (2)

```
(+++) :: Parser a -> Parser a -> Parser a
p +++ q = P $ \s ->
   listToMaybe [x | Just x <- [p s, q s]]
```

the successful parses

return the first successful parse

try parsing both with p and with q

Basic Parsers

Lets define some functions to build some basic parsers

```
sat :: (Char -> Bool) -> Parser Char

sat prop = P $ \s ->
      case s of
        (c:cs) | prop c -> Just (c,cs)
        _               -> Nothing
digit =  sat isDigit

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

will redefine sat later from more basic parsers

Example

```
Main> parse (number +++ success 42) "123xxx"
Just (123,"xxx")
Main> parse (number +++ success 42) "xxx"
  Just (42,"xxx")
Main> map (parse $ sat isDigit +++ char '{')
         ["{hello", "8{hello", "hello"]
[Just ('{',"hello"),Just ('8',"{hello"),Nothing]
```

Basic parsers (2)

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

```
Main> pmap digitToInt (sat isDigit) "1+2"
Just (1,"+2)")
```

Parse one thing after another

Several ways to parse one thing then another, e.g.

- parse first thing, discard result then parse second thing (function **(>->)**)
- parse first thing, parse and discard a second thing, return result of the first **(<-<)**
- parse the first thing and then parse a second thing in a way which depends on the value of the first (function **(>*>)**)
- parse a sequence of as many things as possible (functions **zeroOrMore, oneOrMore**)

Parse one thing after another

```
(>->) :: Parser a -> Parser b -> Parser b

(p >-> q) s = P $ \s ->
                  case parse p s of
                         Nothing      -> Nothing
                         Just (_, s') -> q s'
```

throws awa result of first

```
Main> parse (char '['  >-> sat isDigit) "[1+2]"
Just ('1',"+2]")
```

Parse one thing after another

```
>*> :: Parser a -> (a -> Parser b) -> Parser b

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

```
Main> parse (digit >*> \a -> sat (>a))  "12xxx"
Just ('2',"xxx")
Main> parse (digit >*> \a -> sat (>a))  "10xxx"
Nothing
```

p >*> f

>*> can be used to define earlier operations

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

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

```
pmap :: (a -> b) -> Parser a -> Parser b
pmap f p = p >*> \a->success (f a)
```

Derived Parsers

```
(>->)  :: Parser a -> Parser b -> Parser b
p >-> q = p >*> \_ -> q

(<-<) :: Parser a -> Parser b -> Parser a
p <-< q = p >*> \a -> q >-> success a
```

(as before) throws away the result of first parser

throws away the result of second parser

```
Main> (sat isDigit <-< char '>' )  "2>xxx"
Just ('2',"xxx")
```

Parsing sequences to lists

```
(<:>) :: Parser a -> Parser [a] -> Parser [a]
p <:> q = p >*> \a -> pmap (a:) q

zeroOrMore,oneOrMore :: Parser a -> Parser [a]

zeroOrMore p = oneOrMore p +++ success []
oneOrMore p  =  p  <:> zeroOrMore p
```

```
Main> zeroOrMore (sat isDigit) "1234xxxx"
Just ("1234","xxxx")
Main> zeroOrMore (sat isDigit) "x1234xxx"
Just ("","x1234xxx")
Main> (char '@' <:> oneOrMore (char '+')) "@++xxx"
Just ("@++","xxx“)
```

Example: Building a Parser for Expr

```
number :: Parse Integer
number = pmap read $ oneOrMore (sat isDigit)
```

read can't fail here since it is only applied to a list of digits!

```
num :: Parse Expr
num = pmap Num number
```

Exercise: extend to include negative numbers too

Int -> Expr

Parser Integer

Building Parsers with Parsers

```
expr, term, factor :: Parser Expr

expr = foldl1 Add `pmap` chain term (char '+')
term = foldl1 Mul `pmap` chain factor (char '*')

factor = (char '(' >-> expr <-<  char ')')
         +++ num
```

```
chain :: Parser a -> Parser b -> Parser [a]
chain p q = p <:> zeroOrMore (q >-> p)
```

Terminology

- A *"monadic value"* is just an expression whose type is an instance of class Monad
- *"t is a monad"* means t is an instance of the class Monad
- We have often called a monadic value an *"instruction".* This is not standard terminology
  - but sometimes they are called "actions"