Before we continue ...

A Word from the Sponsor!

Don't Fear Monads

They are just a versatile abstraction, like map or fold.
**Parsers**

A *parser* is a function that

- converts *unstructured data* (e.g. String, array of Byte,...)
- into *structured data* (e.g. JSON object, Markdown, Video...)

**type** Parser = String -> StructuredObject

*Every large software system contains a Parser*
<table>
<thead>
<tr>
<th>System</th>
<th>Parses</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shell Scripts</td>
<td>Command-line options</td>
</tr>
<tr>
<td>Browsers</td>
<td>HTML</td>
</tr>
<tr>
<td>Games</td>
<td>Level descriptors</td>
</tr>
<tr>
<td>Routers</td>
<td>Packets</td>
</tr>
<tr>
<td>Netflix</td>
<td>Video</td>
</tr>
<tr>
<td>Spotify</td>
<td>Audio, Playlists...</td>
</tr>
</tbody>
</table>

**How to build Parsers?**

Two standard methods

**Regular Expressions**

- Doesn’t really scale beyond simple things
- No nesting, recursion

**Parser Generators**

1. Specify grammar via rules
2. Tools like `yacc`, `bison`, `antlr`, `happy`  
  - convert grammar into executable function

**Grammars Don’t Compose!**

If we have two kinds of structured objects `Thingy` and `Whatsit`.
Thingy: rule { action } ;

Whatsit: rule { action } ;

To parse sequences of Thingy and Whatsit we must duplicate the rules:

\[
\text{Thingies} : \text{Thingy Thingies} \{ \ldots \} \\
\quad \text{EmptyThingy} \{ \ldots \} \\
\]

\[
\text{Whatsits} : \text{Whatsit Whatsits} \{ \ldots \} \\
\quad \text{EmptyWhatsit} \{ \ldots \} \\
\]

No nice way to reuse the sub-parsers for Whatsit and Thingy :-(

\[\text{A New Hope: Parsers as Functions}\]
Let's think of parsers directly as functions that

- **Take as input a String**
- **Convert** a part of the input into a `StructuredObject`
- Return the **remainder** unconsumed to be parsed **later**

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

A Parser `a`

- Converts a **prefix** of a String
- Into a **structured** object of type `a`
- Returns the **suffix** String **unchanged**

---

**Parsers Can Produce Many Results**

Sometimes we want to parse a String like

```
"2 - 3 - 4"
```

into a list of possible results
[(\text{Minus} (\text{Minus} 2 3) 4), \text{Minus} 2 (\text{Minus} 3 4)]

So we generalize the \texttt{Parser} type to

\begin{verbatim}
data Parser a = P (String -> [(a, String)])
\end{verbatim}

\textit{EXERCISE}

Given the definition

\begin{verbatim}
data Parser a = P (String -> [(a, String)])
\end{verbatim}

Implement a function

\begin{verbatim}
runParser :: Parser a -> String -> [(a, String)]
runParser p s = ???
\end{verbatim}
QUIZ

Given the definition

\[
\text{data } \text{Parser } a = P (\text{String } \to [\text{a, String}])
\]

Which of the following is a valid \text{Parser Char}

- that returns the \text{first Char} from a string (if one exists)
-- A
oneChar = P (\cs -> head cs)

-- B
oneChar = P (\cs -> case cs of
    []    -> [('', [])
    c:cs -> (c, cs))

-- C
oneChar = P (\cs -> (head cs, tail cs))

-- D
oneChar = P (\cs -> [(head cs, tail cs)])

-- E
oneChar = P (\cs -> case cs of
    []    -> []
    cs    -> [(head cs, tail cs)])

Let's Run Our First Parser!
>> runParser oneChar "hey!"
[['h', 'ey']]

>> runParser oneChar "yippee"
[['y', 'ippee']]

>> runParser oneChar ""
[]

Failure to parse means result is an empty list!

**EXERCISE**

Your turn: Write a parser to grab first two chars

twoChar :: Parser (Char, Char)
twoChar = P (\cs -> ???)

When you are done, we should get
QUIZ

Ok, so recall

twoChar :: Parser (Char, Char)
twoChar = P (\cs -> case cs of
  c1:c2:cs' -> [((c1, c2), cs')]
  _ -> [])

Suppose we had some foo such that twoChar' was equivalent to twoChar

twoChar' :: Parser (Char, Char)
twoChar' = foo oneChar oneChar

oneChar :: Parser Char
What must the type of foo be?

A. Parser (Char, Char)

B. Parser Char -> Parser (Char, Char)

C. Parser a -> Parser a -> Parser (a, a)

D. Parser a -> Parser b -> Parser (a, b)

E. Parser a -> Parser (a, a)

**EXERCISE: A foreach Loop**

Lets write a function

```haskell
foreach :: [a] -> (a -> [b]) -> [b]
foreach xs f = ???
```

such that we get the following behavior

```haskell
>> foreach [] (\i -> [i, i + 1])
[]

>> foreach [10,20,30] (\i -> [show i, show (i+1)])
["10", "11", "20", "21", "30", "31"]
```
QUIZ

What does `quiz` evaluate to?

```javascript
quiz = forEach [10, 20, 30] (\i ->
    forEach [0, 1, 2] (\j ->
        [i + j]
    )
)  
```

A. `[10,20,30,0,1,2]`

B. `[10,0,20,1,30,2]`

C. `[[10,11,12], [20,21,22] [30,31,32]]`

D. `[10,11,12,20,21,22,30,31,32]`

E. `[32]`
A pairP Combinator

Let's implement the above as pairP

\[
\text{forEach} :: [a] \to (a \to [b]) \to [b]
\]
\[
\text{forEach } xs \ f = \text{concatMap } f \ xs
\]

\[
\text{pairP} :: \text{Parser } a \to \text{Parser } b \to \text{Parser } (a, b)
\]
\[
\text{pairP } aP \ bP = \text{P } (s \to \text{forEach } (\text{runParser } aP \ s) ((a, s') \to

\[
\text{forEach } (\text{runParser } bP \ s') ((b, s'') \to

\[
((a, b), s'')
\]

)
\]

Now we can write

\[
\text{twoChar } = \text{pairP } \text{oneChar } \text{oneChar}
\]

**QUIZ**
What does `quiz` evaluate to?

```haskell
twoChar = pairP oneChar oneChar

quiz = runParser twoChar "h"
```

A. `[((h,h), "")]

B. `[(h, "]

C. `[("", "]

D. `[]

E. Run-time exception

---

Does the `Parser` a type remind you of something?

Let's implement the above as `pairP`
Parser is a Monad!

Like a state transformer, Parser is a monad! (http://homepages.inf.ed.ac.uk/wadler/papers/marktoberdorf/baastad.pdf)

We need to implement two functions

\[
\text{returnP :: } a \rightarrow \text{Parser } a
\]

\[
\text{bindP :: Parser } a \rightarrow (a \rightarrow \text{Parser } b) \rightarrow \text{Parser } b
\]
Which of the following is a valid implementation of `returnP`:

- `data Parser a = P (String -> [(a, String)])`
- `returnP :: a -> Parser a`
- `returnP a = P (\s -> [])` --- A
- `returnP a = P (\s -> [(a, s)])` --- B
- `returnP a = P (\s -> (a, s))` --- C
- `returnP a = P (\s -> [(a, "")])` --- D
- `returnP a = P (\s -> [(s, a)])` --- E
HINT: return a should just
- “produce” the parse result a and
- leave the string unconsumed.

Bind

Next, let’s implement bindP

- we almost saw it as pairP

\[
\text{bindP} :: \text{Parser } a \rightarrow (a \rightarrow \text{Parser } b) \rightarrow \text{Parser } b \\
\text{bindP } aP \text{ fbP } = P (\s \rightarrow \\
\quad \text{forEach (runParser } aP \text{ s) } ((a, s') \rightarrow \\
\quad \quad \text{forEach (runParser (fbP a) s') } ((b, s'') \rightarrow \\
\quad \quad \quad [(b, s'')] \\
\quad ) \\
\) \\
\)
The function

- Builds the `a` values out of `aP` (using `runParser`)
- Builds the `b` values by calling `fbP a` on the `remainder` string `s'`
- Returns `b` values and the remainder string `s''`

![Diagram](https://ucsd-cse230.github.io/fa20/lectures/12-parsers.html)

```
aa \Rightarrow (\va \rightarrow eb)
```

**The Parser Monad**

We can now make `Parser` an instance of `Monad`

```
instance Monad Parser where
    (>>=) = bindP
    return = returnP
```
And now, let the *wild rumpus start!*

**Parser Combinators**

Let's write lots of *high-level* operators to **combine** parsers!

Here's a cleaned up `pairP`
pairP :: Parser a -> Parser b -> Parser (a, b)
pairP aP bP = do
  a <- aP
  b <- bP
  return (a, b)

Failures are the Pillars of Success!

Surprisingly useful, always fails
  • i.e. returns [] no successful parses

failP :: Parser a
failP = P (_ -> [])
QUIZ

Consider the parser

\[
\text{satP} :: (\text{Char} \rightarrow \text{Bool}) \rightarrow \text{Parser Char}
\]

\[
\text{satP } p = \text{do} \quad \text{c <- oneChar} \\
\text{if } p \text{ c then return c else failP}
\]

What is the value of

\[
\begin{align*}
\text{quiz1} &= \text{runParser (satP (c \rightarrow c == 'h')) "hellow"} \\
\text{quiz2} &= \text{runParser (satP (c \rightarrow c == 'h')) "yellow"}
\end{align*}
\]

<table>
<thead>
<tr>
<th></th>
<th>quiz1</th>
<th>quiz2</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>[]</td>
<td>[]</td>
</tr>
<tr>
<td>B</td>
<td>[('h', &quot;ellow&quot;)], ['y', &quot;ellow&quot;])</td>
<td>['y', &quot;ellow&quot;]</td>
</tr>
<tr>
<td>C</td>
<td>[('h', &quot;ellow&quot;)], []</td>
<td>[]</td>
</tr>
<tr>
<td>D</td>
<td>[]</td>
<td>[('y', &quot;ellow&quot;)],</td>
</tr>
</tbody>
</table>
Parsing Alphabets and Numerics

We can now use satP to write

```haskell
-- parse ONLY the Char c
char :: Parser Char
c char c = satP (\c' -> c == c')

-- parse ANY ALPHABET
alphaCharP :: Parser Char
alphaCharP = satP isAlpha

-- parse ANY NUMERIC DIGIT
digitChar :: Parser Char
digitChar = satP isDigit
```
**QUIZ**

We can parse a single Int digit

```
digitInt :: Parser Int
digitInt = do
  c <- digitChar        -- parse the Char c
  return (read [c])    -- convert Char to Int
```

What is the result of

```
quiz1 = runParser digitInt "92"
quiz2 = runParser digitInt "cat"
```

<table>
<thead>
<tr>
<th></th>
<th>quiz1</th>
<th>quiz2</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>[]</td>
<td>[]</td>
</tr>
<tr>
<td>B</td>
<td>[('9', '2')]</td>
<td>('c', 'at')]</td>
</tr>
<tr>
<td>C</td>
<td>[(9, '2')]</td>
<td>[]</td>
</tr>
<tr>
<td>D</td>
<td>[]</td>
<td>[('c', 'at')]</td>
</tr>
</tbody>
</table>
Write a function

```haskell
strP :: String -> Parser String
strP s = -- parses EXACTLY the String s and nothing else
```

when you are done, we should get the following behavior

```haskell
>>> dogeP = strP "doge"

>>> runParser dogeP "dogerel"
[("doge", "rel")]

>>> runParser dogeP "doggoneit"
[]
```

## A Choice Combinator

Let's write a combinator `orElse p1 p2` such that

- returns the results of `p1`
or, else if those are empty

- returns the results of p2

\[
\text{orElse p1 p2} = \text{-- produce results of `p1` if non-empty}
\]

\[
\text{-- OR-ELSE results of `p2`}
\]

e.g. orElseP lets us build a parser that produces an alphabet OR a numeric character

\[
\text{alphaNumChar :: Parser Char}
\]

\[
\text{alphaNumChar} = \text{alphaChar `orElse` digitChar}
\]

Which should produce

>>> runParser alphaNumChar "cat"
[['c', "at"]]

>>> runParser alphaNumChar "2cat"
[['2', "cat"]]

>>> runParser alphaNumChar "230"
[['2', "30"]]

https://ucsd-cse230.github.io/fa20/lectures/12-parsers.html
QUIZ

orElse :: Parser a -> Parser a -> Parser a
orElse p1 p2 = -- produce results of `p1` if non-empty
    -- OR-ELSE results of `p2`

Which of the following implements `orElse`?
-- a
orElse p1 p2 = do
  r1s <- p1
  r2s <- p2
  return (r1s ++ r2s)

-- b
orElse p1 p2 = do
  r1s <- p1
  case r1s of
    [] -> p2
    _  -> return r1s

-- c
orElse p1 p2 = P (\cs ->
  runParser p1 cs ++ runParser p2 cs)

-- d
orElse p1 p2 = P (\cs ->
  case runParser p1 cs of
    [] -> runParser p2 cs
    r1s -> r1s)
An “Operator” for `orElse`

It will be convenient to have a short “operator” for `orElse`

\[ p1 \lor p2 = \text{orElse} \; p1 \; p2 \]

A Simple Expression Parser

Now, let's write a tiny calculator!
-- 1. First, parse the operator
intOp :: Parser (Int -> Int -> Int)
intOp = plus <|> minus <|> times <|> divide

where
  plus  = do { _ <- char '+'; return (+) }
  minus = do { _ <- char '-'; return (-) }
  times = do { _ <- char '*'; return (*) }
  divide = do { _ <- char '/'; return div }

-- 2. Now parse the expression!
calc :: Parser Int
calc = do x <- digitInt
         op <- intOp
         y <- digitInt
         return (x `op` y)

When calc is run, it will both parse and calculate

>>> runParser calc "8/2"
[(4,"" )]

>>> runParser calc "8+2cat"
[(10,"cat")]

>>> runParser calc "8/2cat"
[(4,"cat")]

>>> runParser calc "8-2cat"
[(6,"cat")]

>>> runParser calc "8*2cat"
[(16,"cat")]

https://ucsd-cse230.github.io/fa20/lectures/12-parsers.html
The `calc0` parser implicitly forces *all operators* to be *right associative*

- doesn’t matter for +, *
- but is incorrect for -

*Does not respect precedence!*

---

**Simple Fix: Parentheses!**

Lets write a combinator that parses something within (...