Parser Combinators

Before we continue ...

A Word from the Sponsor!
Don't Fear Monads

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

```
class Monad m where
  return :: a -> ma
  (>>=) :: ma -> (a -> mb) -> mb
```

---

**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...)
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
Grammars Don’t Compose!

If we have two kinds of structured objects Thingy and Whatsit.

Thingy : rule { action } ;

Whatsit : rule { action } $T_1 \quad T_2 \quad T_3 \quad T_4$
To parse sequences of Thingy and Whatsit we must *duplicate* the rules:

```plaintext
Thingies : Thingy Thingies  { ... }  
            EmptyThingy       { ... }
;
Whatsits : Whatsit Whatsits  { ... }  
            EmptyWhatsit     { ... }
;
```

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

\[(many\ thing)\]

\[\text{many video}\]

\[\text{many title}\]
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

**data** Parser a = P (String -> (a, String))

A Parser a

- Converts a prefix of a String
- Into a structured object of type a and
- 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

\[(\text{Minus (Minus 2 3) 4}), \quad \text{Minus 2 (Minus 3 4)}\]\n
So we generalize the Parser type to

```haskell
data Parser a = P (String -> [(a, String)])
```
**EXERCISE**

Given the definition

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

Implement a function

```haskell
runParser :: Parser a -> String -> [(a, String)]
runParser p s = ???
```
QUIZ

Given the definition

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

Which of the following is a valid `Parser Char`

- that returns the `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)])
Lets Run Our First Parser!

```python
>>> 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

```>>> runParser twoChar "hey!"
[[(‘h’, ‘e’), "y!"]
```

```>>> runParser twoChar "h"
[]
```
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

What must the type of foo be?

Parser Char ⇒ Parser Char  ⇒ Parser (Char, Char)
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
define f :: [a] -> (a -> [b]) -> [b]
f xs f = ???
```

such that we get the following behavior
>>> 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?
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

Lets implement the above as pairP
forEach :: [a] -> (a -> [b]) -> [b]
forEach xs f = concatMap f xs

pairP :: Parser a -> Parser b -> Parser (a, b)
pairP aP bP = P (\s -> forEach (runParser aP s) (\(a, s') ->
    forEach (runParser bP s') (\(b, s'') ->
        ((a, b), s''))
    )
)

Now we can write

twoChar = pairP oneChar oneChar

---

**QUIZ**

What does `quiz` evaluate to?
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 pair P:

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

```haskell
data ST s a = S (s -> (a, s))
```

**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
returnP :: a -> Parser a
bindP :: Parser a -> (a -> Parser b) -> Parser b

**QUIZ**

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

bindP :: Parser a -> (a -> Parser b) -> Parser b
bindP aP fbP = P (\s ->
    forEach (runParser aP s) (\(a, s') ->
        forEach (runParser (fbP a) s') (\(b, s'') ->
            [(b, s'')]
    )
)
)

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''
The *Parser* Monad

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

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