Parse You a JSON String using Parsing Combinators!

Irakli @Safareli

Agenda

- Basic parsers
- Parsing JSON
  - Primitive values
  - Array
  - String
  - Number
  - Object
- Some tips about purescript-parser

Basic parsers

```
data Parser a = Parser
  (String → Tuple (Either ParserError a) String)

run :: ∀ a. String → Parser a → Either ParserError a
```

```
whiteSpace :: Parser String

> run "  " P.whiteSpace
(Right "  ")

> run " 0" P.whiteSpace
(Right " ")

> run "" P.whiteSpace
(Right "")

> run "0" P.whiteSpace
(Right "")
```

```
eof :: Parser Unit

> run "" P.eof
(Right unit)

> run "0" P.eof
(Left "Expected EOF@1:1")
```

```
pure :: ∀ a . a → Parser a

> run "Whatever dude" (pure 1)
(Right 1)

> run "" (pure 1)
(Right 1)
```

```
char :: Char → Parser Char
string :: String → Parser String

> run "s uh dude" (P.char 's')
(Right 's')

> run "dude" (P.string "dude")
(Right "dude")

> run "s uh dude" (P.string "dude")
(Left "Expected \"dude\"@1:1")
```

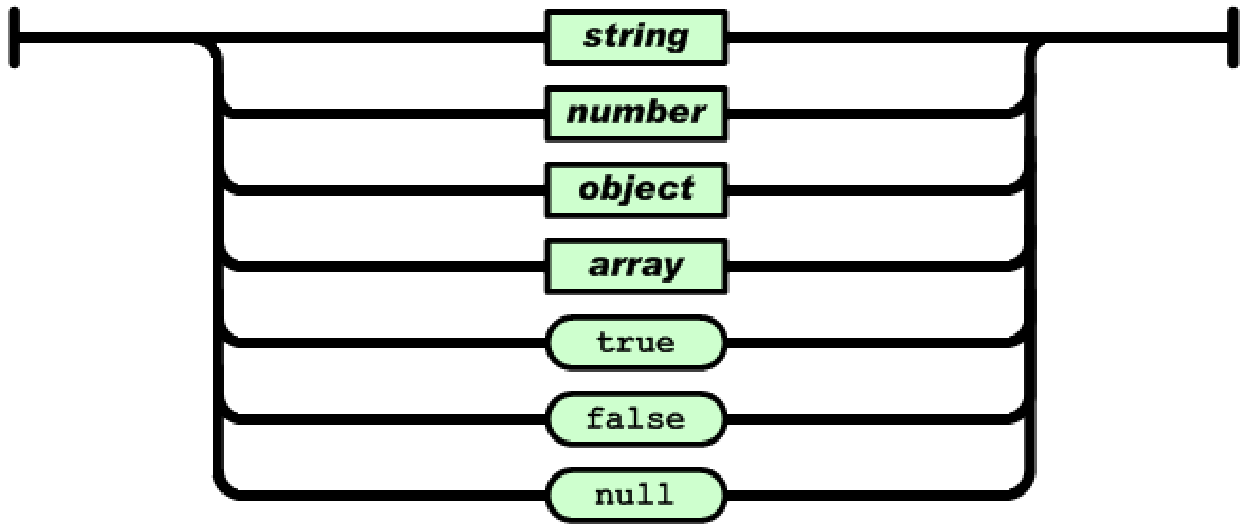
```
$> :: ∀ a b . Parser a → b → Parser b

> run "0" ((P.char '0') $> 0)
(Right 0)

> run "1" ((P.char '1') $> 1)
(Right 1)
```

```
*> :: ∀ a b . Parser a → Parser b → Parser b
<* :: ∀ a b . Parser a → Parser b → Parser a

> run "01" ((P.char '0') *> (P.char '1'))
(Right '1')

> run "01" ((P.char '0') <* (P.char '1') *> (pure "9"))
(Right "9")

> run "01" ((pure "9") <* ((P.char '0') *> (P.char '1')))
(Right "9")

> run "   0" (P.whiteSpace *> (P.char '0') <* P.eof)
(Right '0')

> run " 0  " (P.whiteSpace *> (P.char '0') <* P.eof)
(Left "Expected EOF@1:3")
```

```
tok :: Parser a → Parser a
tok p = p <* P.whiteSpace

tokC :: Char → Parser Char
tokC c = tok (P.char c)

zero : Parser Number
zero = (tokC '0') $> 0.0

one : Parser Number
one = (tokC '1') $> 1.0

zeroCommaOne :: Parser String
zeroCommaOne = zero *> (tokC ',') *> one $> "wow"

> run "  0 , 1 " (P.whiteSpace *> zeroCommaOne <* P.eof)
(Right "wow")

> run "0,1" (P.whiteSpace *> zeroCommaOne <* P.eof)
(Right "wow")
```

JSON

value



```
data JValue
  = JNull
  | JBool   Boolean
  | JNumber Number
  | JString String
  | JArray  (List JValue)
  | JObject (List (Tuple String JValue))
```

Parsing primitive values

null

```
jNull :: Parser JValue
jNull = (tok (P.string "null")) $> JNull

> run "null      " (jNull <* P.eof)
(Right JNull)

> run "    null" jNull
(Left "Expected \"null\"@1:1")

> run "    null  " (P.whiteSpace *> jNull <* P.eof)
(Right JNull)
```

true, false

```
choice :: ∀ a . Array (Parser a) → Parser a

> run "1" (P.choice [zero, one])
(Right 1.0)

> run "9" (P.choice [zero, one])
(Left "Expected \"1\"@1:1")
```

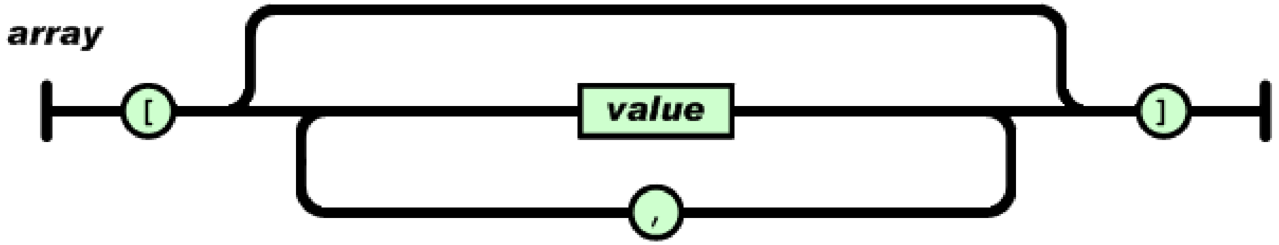
```
map :: ∀ a b . (a → b) → Parser a → Parser b

> run "0" (map (_ + 1.0) zero)
(Right 1.0)

> run "0" (map JNumber zero)
(Right (JNumber 0.0))
```

```
jBool :: Parser JValue
jBool = map JBool (tok (P.choice
  [ (P.string "true") $> true
  , (P.string "false") $> false
  ]))

> run "true" (jBool <* P.eof)
(Right (JBool true))

> run "false   " (jBool <* P.eof)
(Right (JBool false))

> run " false   " (jBool <* P.eof)
(Left "Expected \"false\"@1:1")
```

```
jValue :: Parser JValue
jValue = P.choice [jBool, jNull]

> run "false    " jValue
(Right (JBool false))

> run "null     " jValue
(Right JNull)

> run "       null     " jValue
(Left "Expected \"null\"@1:1")

> run "null     *" jValue
(Right JNull)

> run "null     *" (jValue <* P.eof)
(Left "Expected EOF@1:10")
```

```
value :: Parser JValue
value = P.whiteSpace *> jValue <* P.eof

> run "null     *" value
(Left "Expected EOF@1:10")

> run "       null     " value
(Right JNull)

> run "       false    " value
(Right (JBool false))
```

Parsing Array



```
sepBy :: ∀ a sep . Parser a → Parser sep → Parser (List a)

> run "0,1,0,0" (P.sepBy (P.choice [zero, one]) (P.char ','))
(Right (0.0 : 1.0 : 0.0 : 0.0 : Nil))

> run "0,1,0-0" (P.sepBy (P.choice [zero, one]) (P.char ','))
(Right (0.0 : 1.0 : 0.0 : Nil))

> run "0,1,0,-0" (P.sepBy (P.choice [zero, one]) (P.char ','))
(Left "Expected '1'@1:7")
```

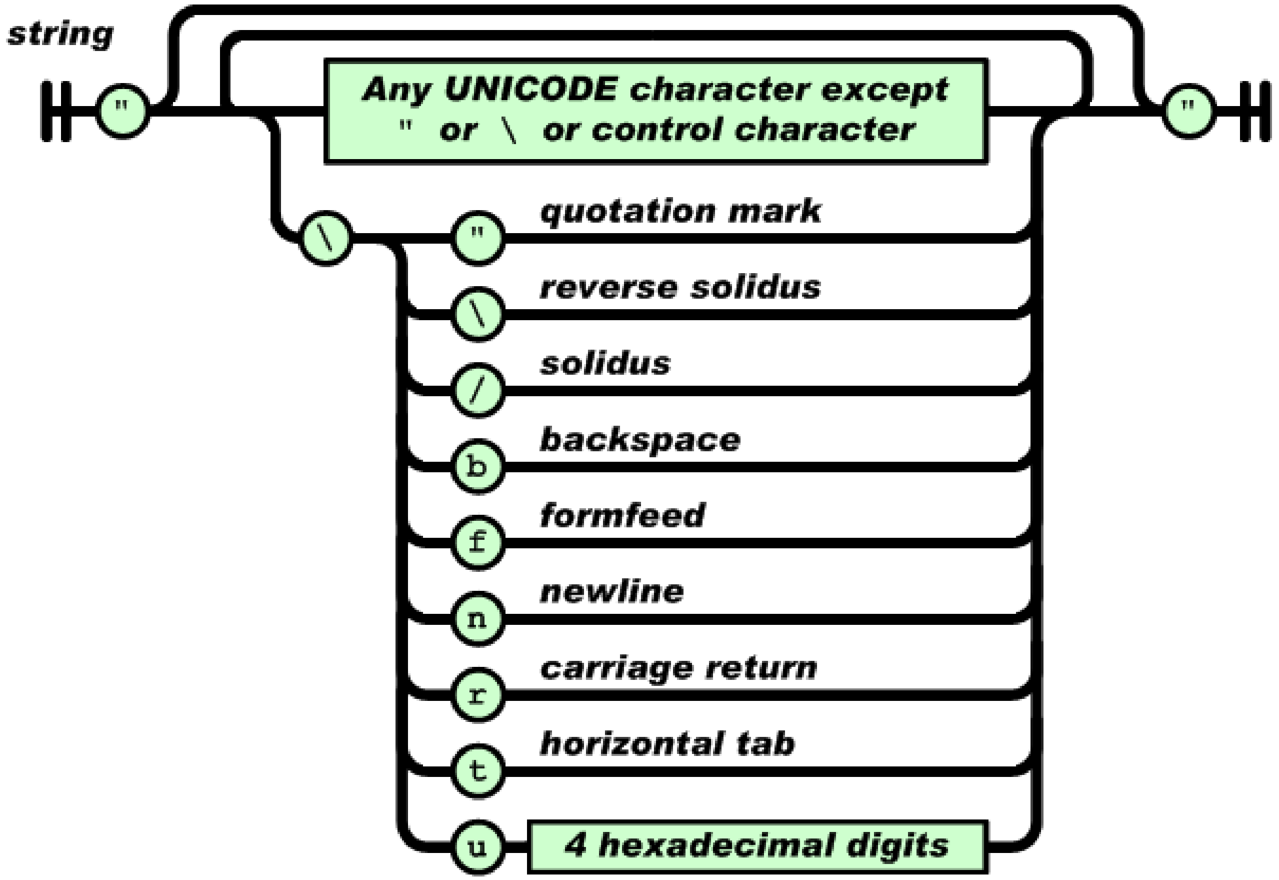
```
> run "true,false,null,null" (P.sepBy jValue (P.char ','))
(Right ((JBool true) : (JBool false) : JNull : JNull : Nil))

> run "true  ,false ,null  ,null" (P.sepBy jValue (P.char ','))
(Right ((JBool true) : (JBool false) : JNull : JNull : Nil))

> run "true  ,false ,null -,null" (P.sepBy jValue (P.char ','))
(Right ((JBool true) : (JBool false) : JNull : Nil))

> run "true  ,false ,null , null" (P.sepBy jValue (P.char ','))
(Left "Expected \"null\"@1:2"))

> run "true  , false , null ,  null" (P.sepBy jValue (tokC ','))
(Right ((JBool true) : (JBool false) : JNull : JNull : Nil))
```

```
> run "true, false, null" (map JArray (P.sepBy jValue (tok ',')))
(Right (JArray ((JBool true) : (JBool false) : JNull : Nil)))

jArray = map JArray
  ((tokC '[') *>
  (P.sepBy jValue (tokC ',')) <*
  (tokC ']'))

> run "[ true, null ]" jArray
(Right (JArray ((JBool true) : JNull : Nil)))
```

```
between :: ∀ start end a
   . Parser start
  → Parser end
  → Parser a
  → Parser a

jArray = map JArray (P.between
  (tokC '[')
  (tokC ']')
  (P.sepBy jValue (tokC ',')))

commaSeparated :: ∀ a . Char → Char → Parser a → Parser (List a)
commaSeparated open close p = P.between
  (tokC open)
  (tokC close)
  (P.sepBy p (tokC ','))

jArray = map JArray (commaSeparated '[' ']' jValue)
```

```
value :: Parser JValue
value = P.whiteSpace *> jValue <* P.eof

jValue :: Parser JValue
jValue = P.choice [(defer \_ → jArray), jBool, jNull]

jArray :: Parser JValue
jArray = map JArray (commaSeparated '[' ']' (defer \_ → jValue))

> run "    [ [ [ null ] ] ]" value
(Right (JArray ((JArray ((JArray (JNull : Nil)) : Nil)) : Nil)))
```

Parsing String



unicode

```
satisfy :: (Char → Boolean) → Parser Char

> run  "0" (P.satisfy \c → c == '0')
(Right '0')

> run  "0" (P.char '0')
(Right '0')

hexDigit :: Parser Char
hexDigit = P.satisfy isHexDigit

> run  "E" hexDigit
(Right 'E')

> run  "Z" hexDigit
(Left "Character 'Z' did not satisfy predicate@1:2")
```

```
replicateA :: ∀ a . Int → Parser a → Parser (Array a)

> run  "00" (map stringFromCharArray (replicateA 2 (P.char '0')))
(Right "00")

unicodeHexCode :: Parser String
unicodeHexCode = map stringFromCharArray (replicateA 4 hexDigit)

> run  "00" unicodeHexCode
(Left "Unexpected EOF@1:3")

> run  "0Z0" unicodeHexCode
(Left "Character 'Z' did not satisfy predicate@1:3")

> run  "265E" (unicodeHexCode <* P.eof)
(Right "265E")
```

```
hexCharsToInt :: String → Maybe Int
hexCharsToInt cs = intFromStringAs hexadecimal cs

> hexCharsToInt "265E"
(Just 9822)

hexCodeToMaybeChar :: String → Maybe Char
hexCodeToMaybeChar cs = hexCharsToInt cs >>= toEnum

> hexCodeToMaybeChar "265E"
(Just '♞')

> run "265E" (map hexCodeToMaybeChar unicodeHexCode)
(Right (Just '♞'))

hexCodeToChar :: String → Parser Char
hexCodeToChar cs = maybe empty pure (hexCodeToMaybeChar cs)

unicode :: Parser Char
unicode = unicodeHexCode >>= hexCodeToChar

> run "265E" unicode
(Right '♞')
```

unescaped

```
many :: ∀ a . Parser a → Parser (Array a )

> run "111" (many one)
(Right [1.0, 1.0, 1.0])

> run "" (many one)
(Right [])

> run "0" (many one)
(Right [])
```

```
isUnescaped :: Char → Boolean
isUnescaped x = not (x == '"' || x == '\\' || isControl x)

unescaped :: Parser Char
unescaped = P.satisfy isUnescaped

> run "foo" $ many unescaped <* P.eof
(Right ['f','o','o'])

> run "foo\\" $ many unescaped <* P.eof
(Left "Expected EOF@1:4")

> run "foo\\z" $ many unescaped <* P.eof
(Left "Expected EOF@1:4")

> run "foo\n" $ many unescaped <* P.eof
(Left "Expected EOF@1:4")
```

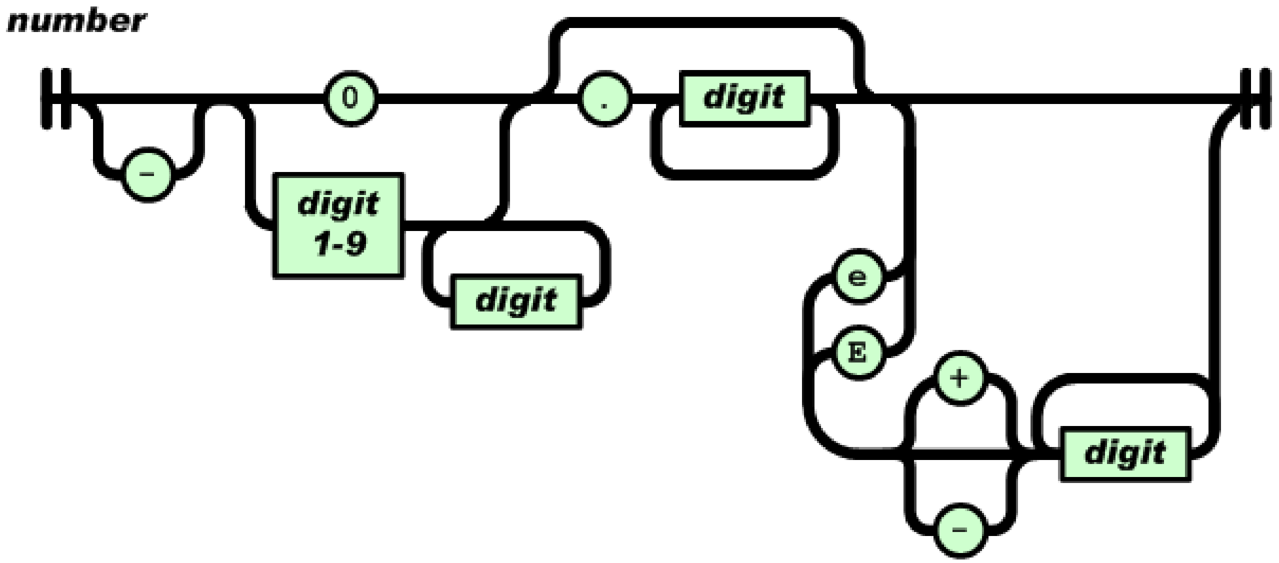
escaped

```haskell
escaped :: Parser Char
escaped = (P.char '\\') *> (P.choice
  [ P.char '"'
  , P.char '\\'
  , P.char '/'
  , P.char 'b'  $> '\b'
  , P.char 'f'  $> '\f'
  , P.char 'n'  $> '\n'
  , P.char 'r'  $> '\r'
  , P.char 't'  $> '\t'
  , P.char 'u'  *> unicode
  ])

> run "\\u265E" escaped
(Right '♞')

> run "\\r" escaped
(Right '\r')

> run "\\\\" escaped
(Right '\\')
```

string

```
char :: Parser Char
char = P.choice [unescaped, escaped]

> run "foo\\u265E\\n" ((many char) <* P.eof)
(Right ['f','o','o','♞','\n'])

string :: Parser String
string = P.between (P.char '"') (tokC '"')
  (map stringFromCharArray (many char))

jString :: Parser JValue
jString = map JString string

> run "\"foo\\u265E\"" $ jString <* P.eof
(Right (JString "foo♞"))

jValue :: Parser JValue
jValue = P.choice [jString, (defer \_ → jArray), jBool, jNull]

> run "[null, [\"\"] ]" jValue
(Right (JArray (JNull : (JArray ((JString "") : Nil)) : Nil)))
```

Parsing Number



```
<|> :: ∀ a . Parser a → Parser a → Parser a
P.choice [p1, p2] == p1 <|> p2
P.choice [p1, p2, p3] == p1 <|> p2 <|> p3

lift2 :: ∀ a b c . (a → b → c) → Parser a → Parser b → Parser c

bit :: Parser Number
bit = P.choice [zero, one]

> run "11" (lift2 (+) bit bit)
(Right 2.0)

> run "1011" (lift2 (+)
  (lift2 (+) bit bit)
  (lift2 (+) bit bit))
(Right 3.0)

fold :: Array (Parser String) → Parser String

fold [ P.string "foo" , P.string "bar"]
==
lift2 append (P.string "foo") (P.string "bar")
```

sign

```
sign :: Parser String
sign = (P.string "-") <|> (pure "")

-- JSON.parse("+1")
-- SyntaxError: Unexpected token + in JSON at position 0

> run "1" (sign <* one)
(Right "")

> run "-1" (sign <* one)
(Right "-")
```

digit

```
int :: Int → Parser String
int n = P.string (show n)

-- JSON.parse("01")
-- SyntaxError: Unexpected number in JSON at position 1

digit1_9 :: Parser String
digit1_9 = P.choice (map int (range 1 9))

> run "0" digit1_9
(Left "Expected \"9\"@1:1")

> run "9" digit1_9
(Right "9")

digit :: Parser String
digit = int 0 <|> digit1_9

> run "0" digit
(Right "0")
```

integer

```
fold :: (Array String) → String

manyDigits :: Parser String
manyDigits = map fold (many digit)

> run "000123" manyDigits
(Right "000123")

integer :: Parser String
integer = int 0 <|> fold [ digit1_9, manyDigits ]

> run "0100123" integer
(Right "0")

> run "0100123" (integer <* P.eof)
(Left "Expected EOF@1:2")

> run "100123" integer
(Right "100123")

> run "-100123" (fold [sign, integer])
(Right "-100123")

> run "100123" (fold [sign, integer])
(Right "100123")
```

```
some :: ∀ a . Parser a → Parser (Array a )

someDigits :: Parser String
someDigits = map fold (some digit)

> run "123" someDigits
(Right "123")

> run "" someDigits
(Left "Expected \"9\"@1:1")
```

fractional

```
fractional :: Parser String
fractional = fold [ P.string ".",  someDigits] <|> (pure "")

> run "-100123" (fold [sign, integer, fractional])
(Right "-100123")

> run "-100123.213" (fold [sign, integer, fractional])
(Right "-100123.213")

> run "-100123." (fold [sign, integer, fractional])
(Left "Expected \"9\"@1:9")
```

exponential

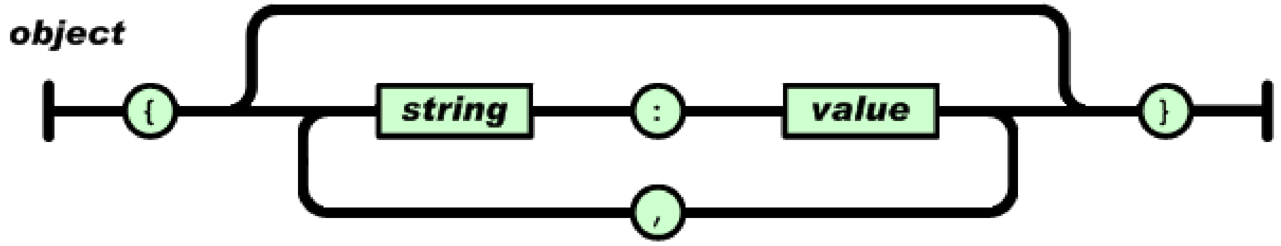
```
expE :: Parser String
expE = (P.string "e") <|> (P.string "E")

expSign :: Parser String
expSign = (P.string "+") <|> sign

exponential :: Parser String
exponential = (fold [ expE , expSign , someDigits ]) <|> (pure "")

> run "e+10" exponential
(Right "e+10")

> run "e+" exponential
(Left "Expected \"9\"@1:3")
```

number

```
number :: Parser String
number = fold [sign, integer, fractional, exponential]

> run "-100.1e-10" number
(Right "-100.1e-10")

> run "-100.1e-" number
(Left "Expected \"9\"@1:9")

> run "-100." number
(Left "Expected \"9\"@1:6")

> run "-" number
(Left "Expected \"9\"@1:2")
```

```
stringToJNumber :: String → JValue
stringToJNumber n = JNumber (readFloat n)

jNumber :: Parser JValue
jNumber = map stringToJNumber (tok number)

jValue :: Parser JValue
jValue = P.choice
  [ jString
  , jNumber
  , (defer \_ → jArray)
  , jBool
  , jNull ]

> run "-100.1e-10" jValue
(Right (JNumber -1.001e-8))

> run "[-100.1e-10]" jValue
(Right (JArray ((JNumber -1.001e-8) : Nil)))

> run "[-100.1e-10, -2]" jValue
(Right (JArray ((JNumber -1.001e-8) : (JNumber -2.0) : Nil)))
```

Parsing Object



```
field :: Parser (Tuple String JValue)
field = lift2 Tuple (string <* (tokC ':')) (defer \_ → jValue)

jObject :: Parser JValue
jObject = map JObject (commaSeparated '{' '}' (defer \_ → field))

jValue :: Parser JValue
jValue = P.choice
[ jString
, jNumber
, (defer \_ → jObject)
, (defer \_ → jArray)
, jBool
, jNull
]

> run "{ \"foo\": 1 }" value
(Right (JObject ((Tuple "foo" (JNumber 1.0)) : Nil)))
```

Some tips about purescript-parser

try

```
try :: Parser a → Parser a

> run "foo" ((P.string "f" *> P.string "zz") <|> (P.string "foo"))
(Left "Expected \"zz\"@1:2")

> run "foo" ((P.string "fzz") <|> (P.string "foo"))
(Right "foo")

> run "foo" ((try (P.string "f" *> P.string "zz")) <|> (P.string "foo"))
(Right "foo")
```

<?>

```
<?> :: Parser a → String -> Parser a

> run "foo" ((P.string "fiz") <|> (P.string "faz"))
(Left "Expected \"faz\"@1:2")

> run "foo" (((P.string "fiz") <|> (P.string "faz")) <?> "fiz or faz")
(Left "Expected fiz or faz@1:1")
```

Questions?

@safareli

github.com/safareli/talks