Regexp

Recap: parser combinators (1/3)

```
type Parser s a = [s] → [(a,[s])]
parseDate :: Parser Char Date
<$> ::            (a → b) → Parser s a → Parser s b
<*> :: Parser s (a → b) → Parser s a → Parser s b
=<< :: (a → Parser s b) → Parser s a → Parser s b
<$  ::            a  → Parser s b → Parser s a
<*  :: Parser s a  → Parser s b → Parser s a
```

Recap: parser combinators (2/3)

```
symbol  :: (Eq s) ⇒  s  → Parser s  s
token   :: (Eq s) ⇒ [s] → Parser s [s]
satisfy :: (s → Bool) → Parser s s
fail    ::      Parser s a
epsilon ::      Parser s ()
return  :: a → Parser s a
option  :: a → Parser s a → Parser s a
```

Recap: parser combinators (3/3)

```
<|>   :: Parser s a  → Parser s a → Parser s a
<<|>  :: Parser s a  → Parser s a → Parser s a
many     :: Parser s a → Parser s [a]
many1    :: Parser s a → Parser s [a]
greedy   :: Parser s a → Parser s [a]
greedy1  :: Parser s a → Parser s [a]
```

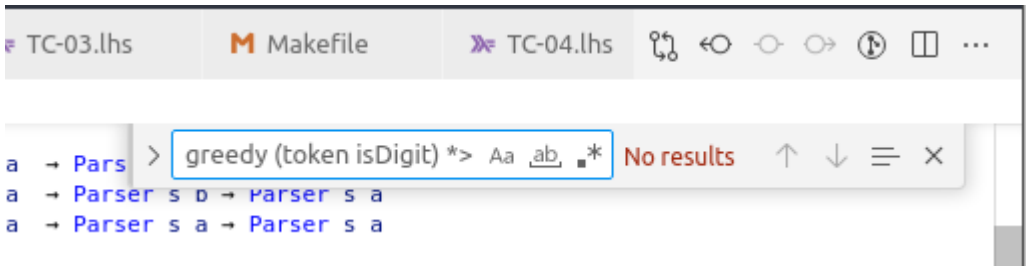
Objections to Parser Combinators

“Too many functions”

“I hate types”

“My other code is C”

“Didn’t work in search bar”



A simple subset

```
<|>  :: Parser s a          → Parser s a → Parser s a
<*>  :: Parser s (a → b) → Parser s a → Parser s b
many  :: Parser s a → Parser s [a]
many1 :: Parser s a → Parser s [a]
option  :: a → Parser s a → Parser s a
symbol  :: (Eq s) ⇒ s → Parser s s
satisfy :: (s → Bool) → Parser s s
```

A simpler subset (1/9)

```
<|>  :: Parser Char a          → Parser Char a → Parser Char a
<*>  :: Parser Char (a → b) → Parser Char a → Parser Char b
many  :: Parser Char a → Parser Char [a]
many1 :: Parser Char a → Parser Char [a]
option  :: a → Parser Char a → Parser Char a
symbol  :: Char → Parser Char Char
satisfy :: (Char → Bool) → Parser Char Char
```

A simpler subset (2/9)

```
<|>  :: P a       → P a → P a
<*>  :: P (a → b) → P a → P b
many  :: P a → P [a]
many1 :: P a → P [a]
option  :: a → P a → P a
symbol  :: Char → P Char
satisfy :: (Char → Bool) → P Char

type P a = Parser Char a
```

A simpler subset (3/9)

```
<|>  :: P a       → P a → P a
<,>  :: P a       → P b → P (a,b)
many  :: P a → P [a]
many1 :: P a → P [a]
option  :: a → P a → P a
symbol  :: Char → P Char
satisfy :: (Char → Bool) → P Char

type P a = Parser Char a
```

A simpler subset (4/9)

```
<|> :: P String  → P String → P String
<,> :: P String  → P String → P (String,String)
many  :: P String → P [String]
many1 :: P String → P [String]
option  :: String → P String → P String
symbol  :: Char → P String
satisfy :: (Char → Bool) → P String

type P a = Parser Char a
```

A simpler subset (5/9)

```
<|>  :: P String  → P String → P String
<+>  :: P String  → P String → P String
many  :: P String → P [String]
many1 :: P String → P [String]
option  :: String → P String → P String
symbol  :: Char → P String
satisfy :: (Char → Bool) → P String

type P a = Parser Char a
```

A simpler subset (6/9)

```
<|>  :: P String  → P String → P String
<+>  :: P String  → P String → P String
many  :: P String → P String
many1 :: P String → P String
option  :: String → P String → P String
symbol  :: Char → P String
satisfy :: (Char → Bool) → P String

type P a = Parser Char a
```

A simpler subset (7/9)

```
<|>  :: P String  → P String → P String
<+>  :: P String  → P String → P String
many  :: P String → P String
many1 :: P String → P String
option  ::          P String → P String
symbol  :: Char → P String
satisfy :: (Char → Bool) → P String

type P a = Parser Char a
```

A simpler subset (8/9)

```
<|>  :: R → R → R
<+>  :: R → R → R
many  :: R → R
many1 :: R → R
option  :: R → R
symbol  :: Char → R
satisfy :: (Char → Bool) → R

type R = Parser Char String
```

A simpler subset (9/9)

```
<|>  :: R → R → R
<+>  :: R → R → R
many  :: R → R
many1 :: R → R
option  :: R → R
symbol  :: Char → R
satisfy :: (Char → Bool) → R

type R = Parser Char String
```

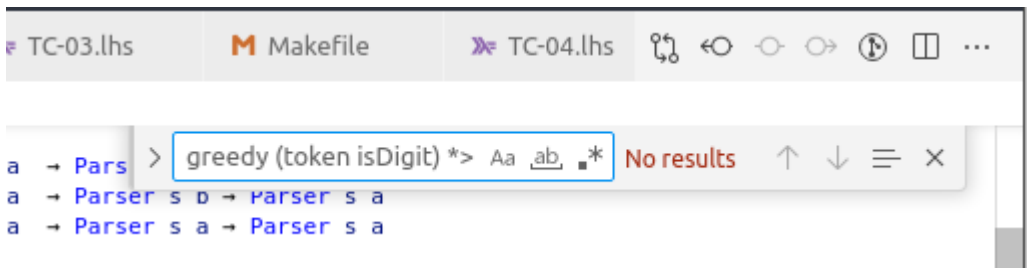
Progress on Objections

😮 *"Too many functions"* ✅

"I hate types" ✅

"My other code is C" ✅

"Didn't work in search bar" 👈



More concise (1/4)

```haskell
-- Haskell types
<|> :: R → R → R
<+> :: R → R → R
many  :: R → R
many1 :: R → R
option  :: R → R
symbol  :: Char → R
satisfy :: (Char → Bool) → R
```

More concise (2/4)

```haskell
-- Haskell
p₁ <|> p₂
p₁ <+> p₂
many p
many1 p
option p
symbol c
satisfy q
```

More concise (3/4)

```haskell
-- Haskell
p₁ <|> p₂
p₁ <+> p₂
many p
many1 p
option p
symbol c
satisfy q
```

```
-- Regular expression
r₁|r₂
r₁r₂
r*
r+
r?
c
```

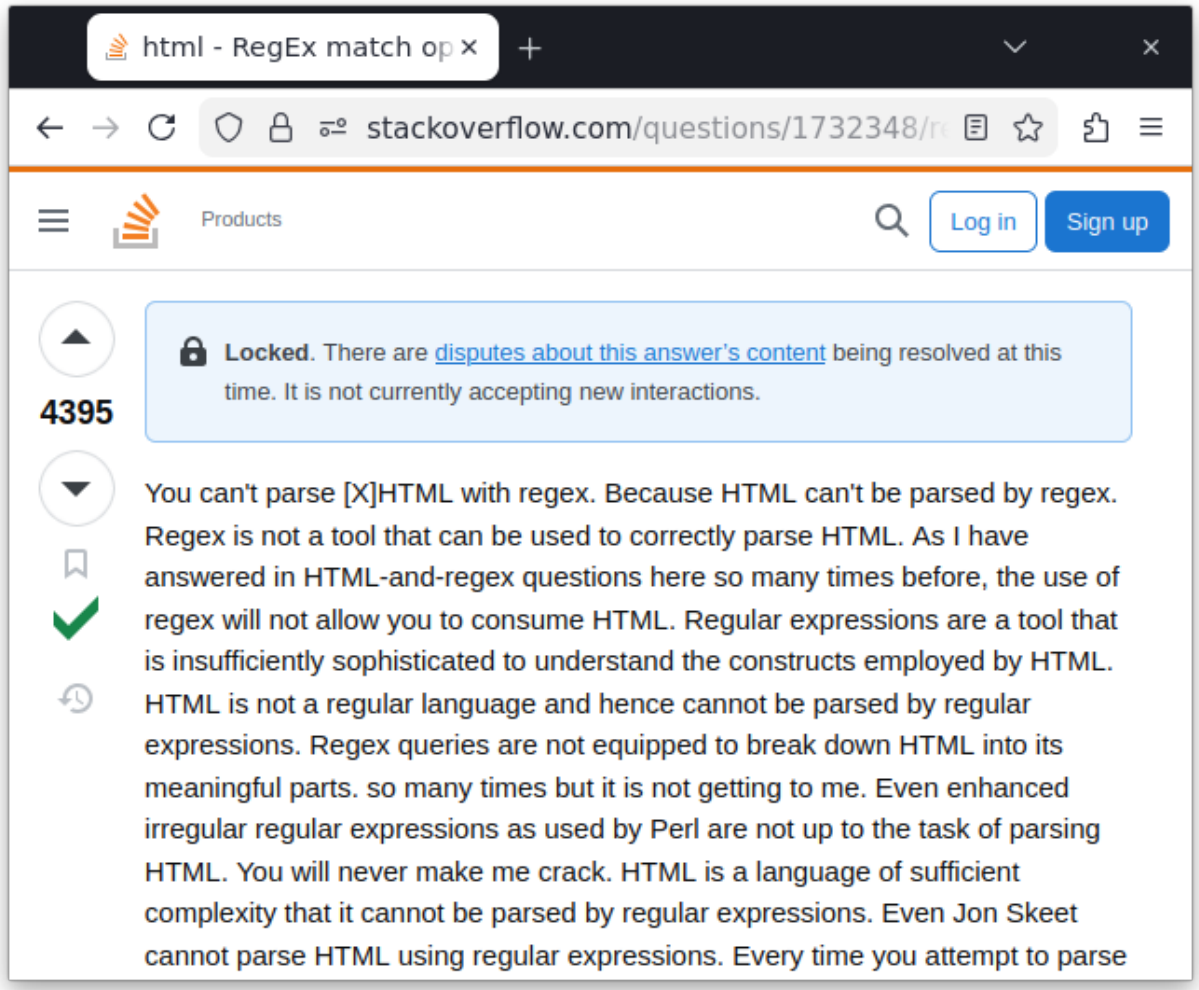
More concise (4/4)

-- *Haskell*	-- Regular expression
p_1 `<|>` p_2	$r_1 \mid r_2$
p_1 `<+>` p_2	$r_1 r_2$
`many p`	r^*
`many1 p`	r^+
`option p`	$r?$
`symbol c`	`c`
`satisfy isDigit`	`\d`
`satisfy isWhitespace`	`\s`
`satisfy (`elem`` `['a'..'z'])`	`[a-z]`
`satisfy (const True)`	`.`

Quiz: which regexp?

APDPDO

Don’t parse HTML with regexp



Regular Languages

> *A **regular** language is matched by some regular expression*

Regular languages ⊂ Context-Free languages

See board

Regular Grammar

A **right-regular grammar** (also called right-linear grammar) is a formal grammar (N, Σ, P, S) in which all production rules in P are of one of the following forms:

1. $A \to a$
2. $A \to aB$
3. $A \to \varepsilon$

where A, B, $S \in N$ are non-terminal symbols, $a \in \Sigma$ is a terminal symbol, and ε denotes the empty string, i.e. the string of length 0. S is called the start symbol.

Regular grammars ⊂ Context-Free grammars

See board

Regular ⊂ Context-Free ⊂ …

See board

Quiz: classify languages

APDPDO

Why not regexp?

- Unreadability
- Expressivity
- Performance 👈

Regexp parser performance (1/5)

```
ghci> matchRegExp "a*a*" "aaaaaaaaaaaaaaa"
```

Regexp parser performance (2/5)

```
ghci> matchRegExp "a*a*" "aaaaaaaaaaaaaaa"
  [ ("aaaaaaaaaaaaaaa","")
  , ("aaaaaaaaaaaaaa","a")
  , ("aaaaaaaaaaaaa","aa")
  , ("aaaaaaaaaaaa","aaa")
  , ("aaaaaaaaaaa","aaaa")
  , ("aaaaaaaaaa","aaaaa")
  , ("aaaaaaaaa","aaaaaa")
  , ("aaaaaaaa","aaaaaaa")
  , …
  ]
```

Regexp parser performance (3/5)

```
ghci> matchRegExp "a*a*$" "aaaaaaaaaaaaaaa"
```

Regexp parser performance (4/5)

```
ghci> matchRegExp "a*a*$" "aaaaaaaaaaaaaaa"
  [ ("aaaaaaaaaaaaaaa","")
  , ("aaaaaaaaaaaaaaa","")
  , ("aaaaaaaaaaaaaaa","")
  , ("aaaaaaaaaaaaaaa","")
  , ("aaaaaaaaaaaaaaa","")
  , ("aaaaaaaaaaaaaaa","")
  , ("aaaaaaaaaaaaaaa","")
  , ...
  ]
```

Regexp parser performance (5/5)

```
ghci> matchRegExp "a*a*$" "aaaaaaaaaaaaaaa"
  [ ("aaaaaaaaaaaaaaa"++"","")
  , ("aaaaaaaaaaaaaa"++"a","")
  , ("aaaaaaaaaaaaa"++"aa","")
  , ("aaaaaaaaaaaa"++"aaa","")
  , ("aaaaaaaaaaa"++"aaaa","")
  , ("aaaaaaaaaa"++"aaaaa","")
  , ("aaaaaaaaa"++"aaaaaa","")
  , ...
  ]
```

Take the first? (1/3)

```
ghci> head $ matchRegExp "a*a*$" "aaaaaaaaaaaaaaa"
  ("aaaaaaaaaaaaaaa"++"","")
```

Take the first? (2/3)

```
ghci> head $ matchRegExp "a*a*$" "aaaaaaaaaaaaaaa"
  ("aaaaaaaaaaaaaaa"++"","")
ghci> head $ matchRegExp "a*(abcdefg)?$" "aaabcdefg"
```

Take the first? (3/3)

```
ghci> head $ matchRegExp "a*a*$" "aaaaaaaaaaaaaaa"
  ("aaaaaaaaaaaaaaa"++"","")
ghci> head $ matchRegExp "a*(abcdefg)?" "aaabcdefg"
  ("aaa"++"","bcdefg")
```

Take the longest? (1/3)

```
ghci> longest $ matchRegExp "a*a*$" "aaaaaaaaaaaaaaa"
  ("aaaaaaaaaaaaaaa"++"","")
ghci> longest $ matchRegExp "a*(abcdefg)?" "aaabcdefg"
  ("aa"++"abcdefg","")
```

Take the longest? (2/3)

```
ghci> longest $ matchRegExp "a*a*$" "aaaaaaaaaaaaaaa"
  ("aaaaaaaaaaaaaaa"++"","")
ghci> longest $ matchRegExp "a*(abcdefg)?" "aaabcdefg"
  ("aa"++"abcdefg","")
ghci> longest $ matchRegExp "a*(aab)?" "aaab"
  ("a"++"aab","")
```

Take the longest? (3/3)

```
ghci> longest $ matchRegExp "a*a*$" "aaaaaaaaaaaaaaa"
  ("aaaaaaaaaaaaaaa"++"","")
ghci> longest $ matchRegExp "a*(abcdefg)?" "aaabcdefg"
  ("aa"+"abcdefg","")
ghci> matchRegExp "a*(aab)?" "aaab"
  [("aaaa"++"","b")
  ,("aaa"++"","aab")
  ,("aa"++"","aab")
  ,("a"++"","aaab")
  ,(""++"","aaaab")
  ,("a"++"aaaab","")
  ]
```

Use greedy? (1/3)

```
ghci> matchRegExp' "a*a*$" "aaaaaaaaaaaaaaa"
   [ ("aaaaaaaaaaaaaaa","")]
```

Use greedy? (2/3)

```
ghci> matchRegExp' "a*a*$" "aaaaaaaaaaaaaaa"
   [ ("aaaaaaaaaaaaaaa","")]
ghci> matchRegExp' "a*a$" "aaaaaaaaaaaaaaa"
   [ ]
```

Use greedy? (3/3)

```
ghci> matchRegExp' "a*a*$" "aaaaaaaaaaaaaaa"
   [ ("aaaaaaaaaaaaaaa","")]
ghci> matchRegExp' "a*a$" "aaaaaaaaaaaaaaa"
   [ ]
```

_* must match as much as possible
without making the match fail

Performance is hard

Next Thursday:
O(length * regexp_complexity) matching time

Challenges

- Craft regex with O(length^2) matching time
- Craft regex with O(2^length) matching time
- Hint: swtch.com/~rsc/regexp/regexp1.html