A Play on Regular Expressions

Sebastian Fischer, Frank Huch, and Thomas Wilke

Christian-Albrechts University of Kiel

ICFP 2010

A Play on Regular Expressions

in 3 acts!

Sebastian Fischer, Frank Huch, and Thomas Wilke

Christian-Albrechts University of Kiel

ICFP 2010

- intuitive method for regular expression matching
- automata construction with elegant Haskell implementation
- can be generalized in suprising ways

```
((a|b)*c(a|b)*c)*(a|b)*
```

symbols

```
((a|b)*c(a|b)*c)*(a|b)*
```

alternatives

```
((a|b)*c(a|b)*c)*(a|b)*
```

sequences

```
((a|b)*c(a|b)*c)*(a|b)*
```

repetitions

```
((a|b)*c(a|b)*c)*(a|b)*
```

```
((a|b)*c(a|b)*c)*(a|b)*
```

```
"abc"
```

preceded
by c

```
((a|b)*c(a|b)*c)*(a|b)*
```

```
"abc"
```

```
((a|b)*c(a|b)*c)*(a|b)*
```

```
"abc"
```

not at "the end" : c still to come

```
((a|b)*c(a|b)*c)*(a|b)*
```

```
"abc"
```

at "the end"

$$((a|b)*c(a|b)*\boxed{c})*\underbrace{(a|b)*}_{\text{accepts empty word}}$$

"abcc"

↑

empty

mark

symbol

```
data Reg = Eps
         | Sym Bool Char
         | Alt Reg Reg
         | Seq Reg Reg
         | Rep Reg
```

alternative

sequence

repetition

does regexp match word ?

```
match :: Reg -> String -> Bool
match r "" = empty r
...
```

predicate :
accepts empty word ?

```
empty :: Reg -> Bool
empty Eps       = True
empty (Sym _ _) = False
empty (Alt p q) = empty p || empty q
empty (Seq p q) = empty p && empty q
empty (Rep r)   = True
```

shifts marks

```
...
match r (c:cs) =
  final $ foldl (shift False)
                (shift True r c)
                cs
```

predicate:
mark at "the end"?

```
final :: Reg -> Bool

final Eps       = False
final (Sym m _) = m
final (Alt p q) = final p || final q

final (Seq p q) =
  final q || final p && empty q

final (Rep r)   = final r
```

```
match r (c:cs) =
  final $ foldl (shift False)
                (shift True r c)
                cs
```

```
shift :: Bool -> Reg -> Char -> Reg
```

preceding mark

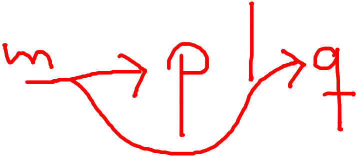
current symbol

```
shift _ Eps       _ = Eps
shift m (Sym _ x) c = Sym (m && x==c)
...
```

mark from left

correct symbol

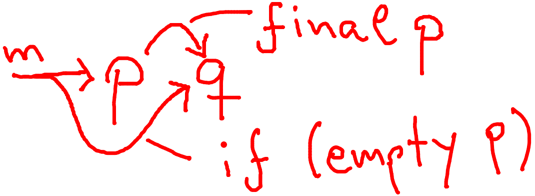
{-



-}

```
...
shift m (Alt p q) c =
  Alt (shift m p c) (shift m q c)
...
```

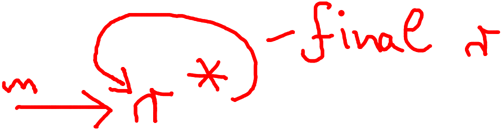
{-



-}

```
...
shift m (Seq p q) c =
  Seq (shift m p c)
      (shift (m && empty p || final p)
             q c)
...
```

{-



-}

```
...
shift m (Rep r) c =
  Rep (shift (m || final r) r c)
```

replace :

- `False` ↦ `0`
- `True` ↦ `1`
- `(||)` ↦ `(+)`
- `(&&)` ↦ `(*)`

```
match :: Reg -> String -> Int
```

↗ number of matchings

ambiguous regexps

addition

```
match (a|a*)          "a"   == 2
match ((a|a*)(b|b*))  "ab"  == 4
```

== 2*2

multiplication

algebraic structure
with 0, 1, +, *

```
match :: Semiring s
      => Reg -> String -> s
```

- position of leftmost matching
- length of longest matching
- ...

results of match depend on
specific semiring

Laziness $\rightsquigarrow$ infinite regular expressions!

non-regular languages like:

$$\{a^n b^n | n \in \mathbb{N}\}$$ context free

$$\{a^n b^n c^n | n \in \mathbb{N}\}$$ not context free

and more.

- intuitive method for regular expression matching
- automata construction with elegant Haskell implementation
- can be generalized in suprising ways

curious? read the play!

```
cabal install weighted-regexp
```

github.com/sebfisch/haskell-regexp

Thanks!