Advanced Lexical Analysis

Matthew Might
University of Utah
matt.might.net
www.ucombinator.org

Today

- Intersections: Useful?
- Derivative of a regex
- Structure of a lexer
- Mechanics of a lexer

Intersection

- Task: Regular expression to recognize dates
- Form: Mmm [D]D YYYY
- Example: Mar 20 1998, Jun 1 1995

(Left) derivatives

$$D_c L = \{w : cw \in L\}\,.$$

Brzozowski. "Derivatives of regular expressions." 1964.

Owens, *et al.* "Derivatives of regular expressions, re-examined." 2009.

Examples

- D_f {foo,bar,frak} =
- D_f (foo|bar)* =
- D_f (foo|bar)*frak =

Matching

$$\frac{w \in D_c(L)}{cw \in L.}$$

String-emptiness

$$L^{\epsilon} = \{\epsilon\} \text{ if } \epsilon \in L$$

$$L^{\epsilon} = \emptyset \text{ if } \epsilon \notin L.$$

Case: Empty set

$$D_c\emptyset =$$

Case: Empty set

$$D_c\emptyset = \emptyset.$$

Case: Empty string

$$D_c\{\epsilon\} =$$

Case: Empty string

$$D_c\{\epsilon\} = \emptyset.$$

Case: Character

$$D_c\{c\} =$$

$$D_c\{c'\} =$$

Case: Character

$$D_c\{c\} = \{\epsilon\}$$

$$D_c\{c'\} = \emptyset \text{ if } c \neq c'.$$

Case: Union

$$D_c(L_1 \cup L_2) =$$

$$=$$

$$=$$

$$=$$

$$=$$

Case: Union

$$
\begin{aligned}
D_c(L_1 \cup L_2) &= \{w : cw \in L_1 \cup L_2\} \\
&= \{w : cw \in L_1 \text{ or } cw \in L_2\} \\
&= \{w : w \in D_c L_1 \text{ or } w \in D_c L_2\} \\
&= \{w : w \in D_c L_1\} \cup \{\, w : w \in D_c L_2\} \\
&= D_c L_1 \cup D_c L_2.
\end{aligned}
$$

Case: Intersection

$$D_c(L_1 \cap L_2) =$$

Case: Intersection

$$D_c(L_1 \cap L_2) = D_c L_1 \cap D_c L_2.$$

Case: Catenation

$$D_c(L_1 \cdot L_2) =$$

Case: Catenation

$$D_c(L_1 \cdot L_2) = (D_c L_1 \cdot L_2) \cup (L_1^{\epsilon} \cdot D_c L_2).$$

Case: Complement

$$D_c\overline{L} =$$

$$=$$

$$=$$

$$=$$

$$=$$

Case: Complement

$$
\begin{aligned}
D_c\overline{L} &= \{w : cw \in \overline{L}\} \\
&= \{w : cw \notin L\} \\
&= \{w : \text{not } cw \in L\} \\
&= \overline{\{w : cw \in L\}} \\
&= \overline{\{w : w \in D_c L\}} \\
&= \overline{D_c L}.
\end{aligned}
$$

Case: Difference

$$D_c(L_1 - L_2) =$$

Case: Difference

$$D_c(L_1 - L_2) = D_c L_1 - D_c L_2.$$

Case: Option

$$D_c(L^?) =$$

Case: Option

$$D_c(L^?) = D_c L.$$

Case: Closure

$$D_c(L^\star) =$$

Case: Closure

$$D_c(L^\star) = (D_c L) \cdot L^\star.$$

Case: Non-empty closure

$$D_c(L^+) =$$

Case: Non-empty closure

$$D_c(L^+) = (D_c L) \cdot L^\star.$$

Lexer generator structure

Lexer structure

- `if in state s`
- `and pattern p matches`
- `then do a`

Actions

- `GOTO state`
- `CONTINUE`
- `YIELD token`

Lex syntax

<state> pattern { action }

Example

- <MAIN> “(“ { yield(LPAR) ; }
- <MAIN> “)” { yield(RPAR) ; }
- <MAIN> “/*” { go(COMMENT) ; }
- <COMMENT> “*/” { go(MAIN) ; }
- <COMMENT> . { }

Functional lexer

```
(define state (lexer
  [pattern action]
  ...))
```

Functional lexer

```
object state extends Lexer[C,A] {
 pattern ==> action

  ...
}
```

A problem of length

- Suppose the pattern: `f(oo)*`
- How many ways can match:

  `fooooooooooooooooo`

Options

- Shortest match
- Global match
- Longest match

Longest match

Given a string *w* and a set of regular expressions *R*, which regex can match the longest prefix of *w*?

Algorithm

```
NaïveRemoveLongestMatch(w ∈ A*,R ⊆ 2^A*)
   suffix ← w
   while (w ≠ ε)
    if ∃L ∈ R : ε ∈ L
       suffix ← w
    c:w ← w
    R ← D_c.R
  return suffix
```

Algorithm

```
RemoveLongestMatch(w ∈ A*,R ⊆ 2^A*)
  suffix ← w
  while (R ≠ ∅ or w ≠ ε)
   if ∃L ∈ R : ε ∈ L
      suffix ← w
   c:w ← w
   R ← D_c.R - {∅}
return suffix
```

Example

- RegEx: fo*
- RegEx: foobar
- RegEx: foob
- Input string: foobarbaz