Lexical Analysis

Matthew Might
University of Utah
matt.might.net

Derivatives

flex

Matthew Might
University of Utah
matt.might.net

www.hacknightslc.com

Next week: Racket

Today

- Derivatives of regular expressions
- How to match regular expressions
- How to lexically analyze indentation

1964

Derivatives of Regular Expressions

JANUSZ A. BRZOZOWSKI

Princeton University, Princeton, New Jersey†

Abstract. Kleene's regular expressions, which can be used for describing sequential circuits, were defined using three operators (union, concatenation and iterate) on sets of sequences. Word descriptions of problems can be more easily put in the regular expression language if the language is enriched by the inclusion of other logical operations. However, in the problem of converting the regular expression description to a state diagram, the existing methods either cannot handle expressions with additional operators, or are made quite complicated by the presence of such operators. In this paper the notion of a derivative of a regular expression is introduced and the properties of derivatives are discussed. This leads, in a very natural way, to the construction of a state diagram from a regular expression containing any number of logical operators.

Remember: L is a set of strings.

$$D_c L$$

1. **Filter**:
   Keep every string starting with c.

2. **Chop**:
   Remove c from the start of each.

$$D_{\mathtt{f}}$$

`foo` `frak` `bar`

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

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

Recognition algorithm

- Derive with respect to each character.
- Does the derived language contain ε?

$\texttt{foo} \in (\texttt{foo})^*$

$$\texttt{o o} \in D_{\texttt{f}}(\texttt{foo})^*$$

oε ∈ oo(foo)*

$$\varepsilon \in (\texttt{foo})^*$$

```
class RegEx:

  def isNullable(self): raise Exception()
  def derive(self,c): raise Exception()

  def matches(self, string):
    if (len(string) == 0):
      return self.isNullable()
    else:
      return self.derive(string[0]).matches(string[1:])
```

Deriving atomic languages

$$\epsilon \equiv \{\text{""}\}$$

$$c \equiv \{c\}$$

$$\emptyset \equiv \{\}$$

```
class Blank(RegEx):
  pass

class Empty(RegEx):
  pass

class Primitive(RegEx):

  def __init__(self,c): self.c = c

empty = Empty()
blank = Blank()
```

$$D_c\emptyset =$$

```
class Empty(RegEx):

  def derive(self,c): return empty
```

$$D_c(\epsilon) =$$

```
class Blank(RegEx):

  def derive(self,c): return empty
```

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

```
class Primitive(RegEx):

  def derive(self,c):
    if self.c == c:
       return blank
    else:
       return empty
```

Deriving regular languages

$$L_1 \cup L_2$$

$$L_1 \cdot L_2$$

$$L_1^\star$$

```
class Choice(RegEx):

  def __init__(self,this,that):
    self.this = this
    self.that = that

class Repetition(RegEx):

  def __init__(self,base):
    self.base = base

class Sequence(RegEx):

  def __init__(self,left,right):
    self.left = left
    self.right = right
```

$$D_c(L_1 \cup L_2)$$

```
class Choice(RegEx):

  def derive(self,c):
    return Choice(self.this.derive(c),
                  self.that.derive(c))
```

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

```
class Repetition(RegEx):

  def derive(self,c):
    return Sequence(self.base.derive(c),
                    self)
```

Concatenation?

WRONG

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

Needs nullability

$$\delta(L) = \epsilon \text{ if } \epsilon \in L$$

$$\delta(L) = \emptyset \text{ if } \epsilon \notin L$$

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

```
class Sequence(RegEx):

  def derive(self,c):
    if self.left.isNullable():
      return Choice(Sequence(self.left.derive(c),self.right),
                    self.right.derive(c))
    else:
      return Sequence(self.left.derive(c), self.right)
```

To recognize?

Need nullability

Need to *compute* nullability

$$\delta(\epsilon) = \epsilon$$

$$\delta(c) = \emptyset$$

$$\delta(\emptyset) = \emptyset$$

```
class Blank(RegEx):

  def isNullable(self): return True

class Empty(RegEx):

  def isNullable(self): return False

class Primitive(RegEx):

  def isNullable(self): return False
```

$$
\begin{aligned}
\delta(L_1 \cup L_2) &= \delta(L_1) \cup \delta(L_2) \\
\delta(L_1 \cdot L_2) &= \delta(L_1) \cdot \delta(L_2) \\
\delta(L_1^\star) &= \epsilon
\end{aligned}
$$

```
class Choice(RegEx):

  def isNullable(self):
    return self.this.isNullable() or self.that.isNullable()


class Repetition(RegEx):

  def isNullable(self): return True


class Sequence(RegEx):

  def isNullable(self):
    return self.left.isNullable() and self.right.isNullable()
```

Code

But, there's more!

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

$$D_c(L_1 - L_2) =$$

$$D_c(\overline{L}) =$$

What about nullability?

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

$$\delta(L_1 - L_2) =$$

$$\delta(\overline{L}) =$$

Code

Lexing Python

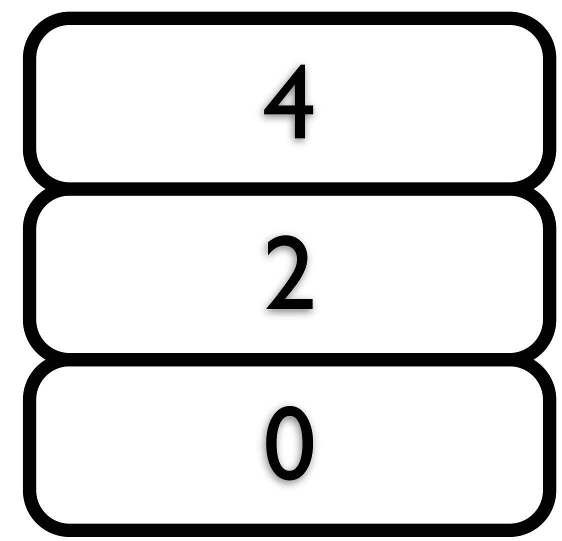
How do you lexically analyze indentation?

Off-side rule

```
foo
  bar
  baz
    qux
quux
```

```
foo { bar baz { qux } } quux
```

```
foo
  bar
  baz
    qux
quux
```




```
foo { bar baz { qux } } quux
```