Ragax: Ragalur Expressions

Using derivatives to validate Indian Classical Music

Walter Schulze

2 May 2018

Amsterdam Functional Programming Meetup

Key Takeaways

Derivatives are **Intuitive** and **Extendable**

- implement matcher
- invent operators
- listen to music

Regular Expressions
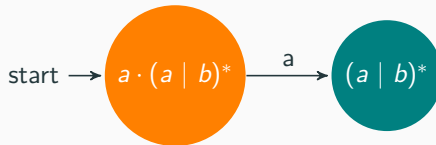
$$a(a|b)*$$

ab   ✓
aabbba   ✓
ac   ✗
ba   ✗

$$a \cdot (a \mid b)^*$$

The Brzozowski derivative (1964) [1] of an expression is the expression that is left to match after the given character has been matched.

$$\partial_a a \cdot b \cdot c \quad\quad = \quad b \cdot c$$
$$\partial_a (a \cdot b \mid a \cdot c) \quad = \quad (b \mid c)$$
$$\partial_c c \quad\quad\quad\quad = \quad \epsilon$$
$$\partial_a a^* \quad\quad\quad = \quad a^*$$

empty set	$\emptyset$
empty string	ε
character	a
concatenation	$r \cdot s$
zero or more	r^*
logical or	$r \mid s$

Basic Operators

```
data Regex = EmptySet
   | EmptyString
   | Character Char
   | Concat Regex Regex
   | ZeroOrMore Regex
   | Or Regex Regex
```

Nullable

Does the expression match the empty string.

$$
\begin{aligned}
\nu(\emptyset) &= \text{false} \\
\nu(\varepsilon) &= \text{true} \\
\nu(a) &= \text{false} \\
\nu(r \cdot s) &= \nu(r) \text{ and } \nu(s) \\
\nu(r^*) &= \text{true} \\
\nu(r \mid s) &= \nu(r) \text{ or } \nu(s)
\end{aligned}
$$

Nullable

Does the expression match the empty string.

```
nullable :: Regex -> Bool
nullable EmptySet = False
nullable EmptyString = True
nullable Character{} = False
nullable (Concat a b) = nullable a && nullable b
nullable ZeroOrMore{} = True
nullable (Or a b) = nullable a || nullable b
```

Nullable Examples

$$\begin{aligned}
\nu(a \cdot b \cdot c) &= \textcolor{orange}{\times} \\
\nu(\varepsilon) &= \textcolor{green}{\checkmark} \\
\nu(a \mid b) &= \textcolor{orange}{\times} \\
\nu(\varepsilon \mid a) &= \textcolor{green}{\checkmark} \\
\nu(a \cdot \varepsilon) &= \textcolor{orange}{\times} \\
\nu((a \cdot b)^*) &= \textcolor{green}{\checkmark} \\
\nu(c \cdot (a \cdot b)^*) &= \textcolor{orange}{\times}
\end{aligned}$$

Derivative Rules

$$\partial_a \emptyset = \emptyset$$
$$\partial_a \epsilon = \emptyset$$
$$\partial_a a = \epsilon$$
$$\partial_a b = \emptyset \qquad \text{for } b \neq a$$
$$\partial_a (r \cdot s) = \partial_a r \cdot s \qquad not(\nu(r))$$
$$\partial_a (r \cdot s) = \partial_a r \cdot s \mid \partial_a s \qquad \nu(r)$$
$$\partial_a (r^*) = \partial_a r \cdot r^*$$
$$\partial_a (r \mid s) = \partial_a r \mid \partial_a s$$

Derivative Rules

```
deriv :: Regex -> Char -> Regex
deriv EmptyString _ = EmptySet
deriv EmptySet _ = EmptySet
deriv (Character a) c = if a == c
  then EmptyString else EmptySet
deriv (Concat r s) c = if nullable r
  then (deriv r c `Concat` s) `Or` deriv s c
  else deriv r c `Concat` s
deriv (ZeroOrMore r) c =
  deriv r c `Concat` ZeroOrMore r
deriv (Or r s) c =
  deriv r c `Or` deriv s c
```

Our regular expression matcher

$$\nu(\mathit{foldl}(\partial, r, str)) \qquad \qquad \qquad \text{where}$$
$$\mathit{foldl}(\partial, r, str) \quad = \quad r \qquad \qquad \text{if } str == ""$$
$$\qquad \qquad \qquad = \quad \mathit{foldl}(\partial, \partial_{s[0]}(r), s[1:]) \qquad \text{otherwise}$$

```
match :: Regex -> String -> Bool
match r str = nullable (foldl deriv r str)

func matches(r *expr, str string) bool {
    for _, c := range str {
        r = deriv(r, c)
    }
    return nullable(r)
}
```

Simplification

$$\emptyset \cdot r \quad \approx \quad \emptyset$$
$$r \cdot \emptyset \quad \approx \quad \emptyset$$
$$\varepsilon \cdot r \quad \approx \quad r$$
$$r \cdot \varepsilon \quad \approx \quad r$$

$$r \mid r \quad \approx \quad r$$
$$\emptyset \mid r \quad \approx \quad r$$

$$(r^*)^* \quad \approx \quad r^*$$
$$\varepsilon^* \quad \approx \quad \varepsilon$$
$$\emptyset^* \quad \approx \quad \varepsilon$$

Example: Matching a sequence of notes

Using a regex we can validate the C Major Pentatonic Scale.

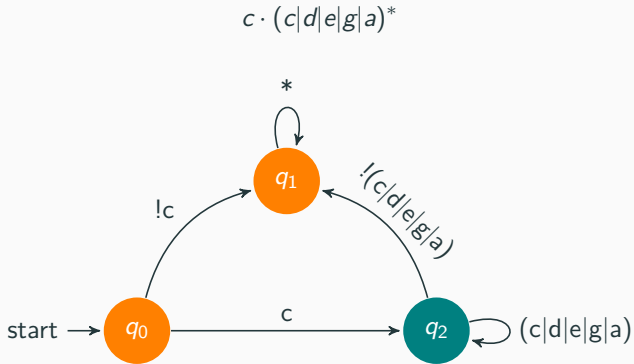$$c \cdot (c|d|e|g|a)^*$$

$$\text{ceg} \quad \checkmark$$

$$
\begin{aligned}
\partial_c c \cdot (c|d|e|g|a)^* &= & \varepsilon \cdot (c|d|e|g|a)^* \\
\partial_e \varepsilon \cdot (c|d|e|g|a)^* &= & (\emptyset \cdot (c|d|e|g|a)^*) \mid (\emptyset|\emptyset|\varepsilon|\emptyset|\emptyset) \cdot (c|d|e|g|a)^* \\
&= & \emptyset \mid (\emptyset|\emptyset|\varepsilon|\emptyset|\emptyset) \cdot (c|d|e|g|a)^* \\
&= & (\emptyset|\emptyset|\varepsilon|\emptyset|\emptyset) \cdot (c|d|e|g|a)^* \\
&= & \varepsilon \cdot (c|d|e|g|a)^* \\
&= & (c|d|e|g|a)^* \\
\partial_g (c|d|e|g|a)^* &= & (\emptyset|\emptyset|\emptyset|\varepsilon|\emptyset) \cdot (c|d|e|g|a)^* \\
&= & (c|d|e|g|a)^*
\end{aligned}
$$

$$\nu((c|d|e|g|a)^*) = \checkmark$$

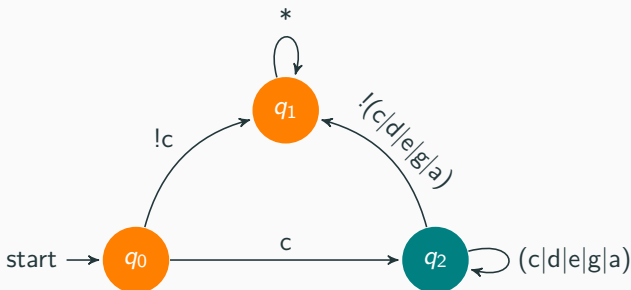**Questions?**

$$c \cdot (c|d|e|g|a)^*$$

Memoization and Simplification

$$q_0 = c \cdot (c|d|e|g|a)^*$$
$$q_1 = \emptyset$$
$$q_2 = (c|d|e|g|a)^*$$

- Memoizing deriv = transition function
- Memoizing nullable = accept function
- Simplification = minimization [4]

Recursive Regular Expressions

$$((a \cdot b)^* \mid c)$$

Two new operations:

- Define a reference: $\#myref = (a \cdot b)^*$
- Use a reference: $(@myref \mid c)$

$$
\begin{array}{rcl}
\partial_a @q & = & \partial_a \#q \\
\nu(@q) & = & \nu(\#q)
\end{array}
$$

Ragas - Indian Classical Music

https://youtu.be/iElMWziZ62A?t=136

Ragas

Ragas are indian version of western scales [5]:

- Stricter
- Next note depends on current note.
- Notes named differently and relative to root note.

Raga	S	r	R	g	G	m	M	P	d	D	n	N
Western	c	c♯	d	d♯	e	f	f♯	g	g♯	a	a♯	b

Example Raga

- Raag Bhupali (a type of Pentatonic scale)
- Ascent: S R G P D S'
- Descent: S' D P G R S

- Western Labeling Relative to c
- Ascent: c d e g a c^1
- Descent: c^1 a g e d c

http://raag-hindustani.com/22_files/
ArohBhupali.mp3

Questions?

A Grammar for a Raga

- Raag Bhupali (a type of Pentatonic scale)
- Ascent: S R G P D S'
- Descent: S' D P G R S

$$\#S = (S \cdot (@R \mid @D))^*$$
$$\#R = R \cdot (@G \mid \varepsilon)$$
$$\#G = G \cdot (@P \mid @R)$$
$$\#P = P \cdot (@D \mid @G)$$
$$\#D = D \cdot (\varepsilon \mid @P)$$

Demo

Context Free Grammars

Left Recursive Raga

$$
\begin{aligned}
\#S &= (S \cdot (@R \mid @D))^* &= @S \cdot (S \cdot (@R \mid @D)) \mid \varepsilon \\
\#R &= R \cdot (@G \mid \varepsilon) \\
\#G &= G \cdot (@P \mid @R) \\
\#P &= P \cdot (@D \mid @G) \\
\#D &= D \cdot (\varepsilon \mid @P)
\end{aligned}
$$

nullable and derivative each have infinite recursion.

$$
\nu(\#S) = (\nu(@S) \text{ and } \nu(S \cdot (@R \mid @D))) \text{ or } \nu(\varepsilon)
$$

Parsing with Derivatives

This has been solved using [3] functional concepts:

- laziness - The Brake
- memoization - The Handbrake
- least fixed point - The Gas

Laziness

Strict:

```go
func strictPlus(a, b int) int {
    return a + b
}
```

Lazy:

```go
func lazyPlus(a, b int) func() int {
    return func() int {
        return a + b
    }
}
```

Laziness - The Brake

$$\lambda \implies \text{laziness}$$

$$
\begin{array}{rclcl}
\partial_a(r|s) & = & \partial_a r \mid \partial_a s & = & \lambda(\partial_a r) \mid \lambda(\partial_a s) \\
\partial_a(r^*) & = & \partial_a r \cdot r^* & = & \lambda(\partial_a r) \cdot r^* \\
\partial_a(r \cdot s) & = & \partial_a r \cdot s \mid \jmath(r) \cdot \partial_a s & = & \lambda(\lambda(\partial_a r) \cdot s) \mid \lambda(\lambda(\jmath(r)) \cdot \lambda(\partial_a s))
\end{array}
$$

$$
\begin{array}{rcll}
\jmath(r) & = & \epsilon & \text{if } \nu(r) \\
& = & \emptyset & \text{otherwise}
\end{array}
$$

$$\partial_n \# S = \lambda(\partial_n(@S \cdot (S \cdot (@R \mid @D)))) \mid \lambda(\partial_n \varepsilon)$$

Memoization - The Handbrake

Nullable is called:

$$\begin{aligned}
\nu(\partial_n \# S) &= \nu(\lambda(\partial_n(@S \cdot (S \cdot (@R \mid @D)))) \mid \lambda(\partial_n \varepsilon)) \\
&= \nu(\lambda(\partial_n(@S \cdot (S \cdot (@R \mid @D))))) \mid \nu(\lambda(\partial_n \varepsilon))
\end{aligned}$$

Lazy function is executed:

$$\begin{aligned}
\lambda(\partial_n(@S \cdot (S \cdot (@R \mid @D)))) &= \partial_n(@S \cdot (S \cdot (@R \mid @D))) \\
&= \lambda(\lambda(\partial_n @S) \cdot \lambda((S \cdot (@R \mid @D)))) \mid \\
&\quad \lambda(\lambda(\jmath(@S)) \cdot \lambda(\partial_n(S \cdot (@R \mid @D)))) \\
\lambda(\partial_n @S) &= \partial_n @S
\end{aligned}$$

Infinite recursion:

$$\partial_n @S = \partial_n \# S$$

Memoizing closes the loop:

$$\partial_n @S = \lambda(\partial_n(@S \cdot (S \cdot (@R \mid @D)))) \mid \lambda(\partial_n \varepsilon)$$

Memoization

```
func memoize(eval func(a) b) func(a) b {
    mem := make(map[a]b)
    return func(input a) b {
        if output, ok := mem[input]; ok {
            return output
        }
        output := eval(a)
        mem[input] = output
        return output
    }
}
```

Least Fixed Point

$$f(x) = x^2$$

$$f(0) = 0^2$$

$$f(1) = 1^2$$

fixed points $= \{0, 1\}$

least fixed point $= 0$

Least Fixed Point of Derivative

$$\partial_a r = r$$

$$\partial_a \emptyset = \emptyset$$

$$\partial_a a^* = a^*$$

fixed points $= \{\emptyset, a^*\}$

least fixed point $= \emptyset$

Least Fixed Point - The Gas

Nullable is relentless:

$$
\begin{aligned}
\nu(\lambda(\partial_n \# S)) &= \ldots \\
\nu(\lambda(\partial_n(@S \cdot (S \cdot (@R \mid @D))))) \mid \nu(\lambda(\partial_n \varepsilon)) &= \ldots \\
\nu(\lambda(\lambda(\partial_n @S) \cdot \lambda(\ldots))) &= \ldots \\
\nu(\lambda(\partial_n @S)) &= \nu(\text{fix}) \\
&= \nu(\emptyset) \\
&= \text{false} \\
\nu(\lambda(\lambda(\partial_n @S) \cdot \lambda(\ldots))) &= \text{false \& false} \\
\nu(\lambda(\partial_n(@S \cdot (S \cdot (@R \mid @D))))) \mid \nu(\lambda(\partial_n \varepsilon)) &= \text{false} \mid \text{false} \\
\nu(\lambda(\partial_n \# S)) &= \text{false}
\end{aligned}
$$

http://awalterschulze.github.io/ragax/

Yacc is Dead

Yacc, Antlr, Flex, Bison, etc. perform better.

But derivatives:

- more intuitive than LR and LALR parsers;
- only use functional techniques;
- recognize generalized Context Free Grammars, not just a subset.

Trees

Relaxing

http://relaxng.org/ [2] - RELAX NG is a schema language for XML, like XSchema and DTD.

Derivatives used for Implementation and Specification.

Polymorphic Regular Expressions: Characters $=>$ XMLNodes.

New Operators:

$$
\begin{array}{rcl}
\partial_a(r \,\&\&\, s) & = & (\partial_a r \,\&\&\, s) \mid (\partial_a s \,\&\&\, r) \\
\nu(r \,\&\&\, s) & = & \nu(r) \text{ and } \nu(s) \\
\emptyset \,\&\&\, r & \approx & \emptyset \\
\varepsilon \,\&\&\, r & \approx & r \\[6pt]
\partial_a !(r) & = & !(\partial_a r) \\
\nu(!(r)) & = & \text{not}(\nu(r)) \\[6pt]
(r)? & \approx & r \mid \varepsilon
\end{array}
$$

TreeNode

```
data Expr = ...
    NodeExpr String Expr
    ...

deriv :: Expr -> Tree -> Expr
deriv (NodeExpr nameExpr childExpr) (Node name children) =
  if nameExpr == name &&
        nullable (foldl deriv childExpr children)
    then Empty
    else EmptySet

nullable NodeExpr{} = False
```

https://youtu.be/SvjSP2xYZm8
https://katydid.github.io

Katydid: Relapse

Relapse: Tree Validation Language.

JSON, Protobufs, Reflected Go Structures and XML

Go, Haskell + Cross language testsuite

New Operators:

$$
\begin{array}{rcl}
\partial_a(r \, \& \, s) & = & (\partial_a r \, \& \, \partial_a s) \\
\nu(r \, \& \, s) & = & \nu(r) \text{ and } \nu(s) \\
\emptyset \, \& \, r & \approx & \emptyset \\
r \, \& \, r & \approx & r \\
* & \approx & !(\emptyset) \\
.r & \approx & * \cdot r \cdot *
\end{array}
$$

https://github.com/katydid/katydid-haskell

```
http://katydid.github.io/play/
http://katydid.github.io/tour/
```

References

J. A. Brzozowski.
Derivatives of regular expressions.
Journal of the ACM (JACM), 11(4):481–494, 1964.

M. Makoto and J. Clark.
RELAX NG home page.
http://relaxng.org.

M. Might, D. Darais, and D. Spiewak.
Parsing with derivatives: a functional pearl.
In *Acm sigplan notices*. ACM, 2011.

S. Owens, J. Reppy, and A. Turon.
Regular-expression derivatives re-examined.
Journal of Functional Programming, 19(02):173–190, 2009.

Sādhana.
Hindustani Classical Music.
http://raag-hindustani.com/.