Ragax: Ragalur Expressions

Using derivatives to validate Indian Classical Music

Walter Schulze

2 May 2018

Amsterdam Functional Programming Meetup

Key Takeaways

- implement matcher
- invent operators
- **derivatives are intuitive and extendable**
- generate music

Regular Expressions

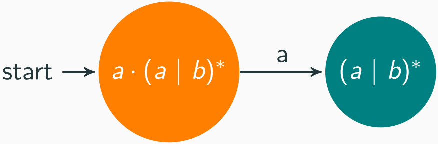
$$a(a|b)*$$

ab	✓
aabbba	✓
ac	×
ba	×

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

What is a Brzozowski Derivative

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



$$
\begin{aligned}
\partial_a a \cdot b \cdot c &= b \cdot c \\
\partial_a (a \cdot b \mid a \cdot c) &= (b \mid c) \\
\partial_c c &= \epsilon \\
\partial_a a^* &= a^*
\end{aligned}
$$

Basic Operators

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.

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

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

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

Derivative Rules

$$
\begin{array}{lcll}
\partial_a \emptyset & = & \emptyset & \\
\partial_a \epsilon & = & \emptyset & \\
\partial_a a & = & \epsilon & \\
\partial_a b & = & \emptyset \quad \text{for } b \neq a & \\
\partial_a (r \cdot s) & = & \partial_a r \cdot s & \mathit{not}(\nu(r)) \\
\partial_a (r \cdot s) & = & \partial_a r \cdot s \mid \partial_a s & \nu(r) \\
\partial_a (r^*) & = & \partial_a r \cdot r^* & \\
\partial_a (r \mid s) & = & \partial_a r \mid \partial_a s &
\end{array}
$$

Derivative Rules

```
deriv :: Expr -> Char -> Expr
deriv EmptyString c = EmptySet
deriv EmptySet c = EmptySet
deriv (Character a) c = if a == c
  then EmptyString else EmptySet
deriv (Concat r s) c =
  let left = deriv r c
      right = deriv s c
  in if nullable r
     then Or (Concat left s) right
     else Concat left s
deriv (ZeroOrMore r) c =
  Concat (deriv r c) (ZeroOrMore r)
deriv (Or r s) c =
  Or (deriv r c) (deriv s c)
```

Our regular expression matcher

$$\nu(foldl(\partial, r, str))$$

```
nullable (foldl deriv expr string)
```

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

Simplification

$$
\begin{array}{lcl}
\emptyset \cdot r & \approx & \emptyset \\
r \cdot \emptyset & \approx & \emptyset \\
\varepsilon \cdot r & \approx & r \\
r \cdot \varepsilon & \approx & r \\
\\
r \mid r & \approx & r \\
\emptyset \mid r & \approx & r \\
\\
(r^*)^* & \approx & r^* \\
\varepsilon^* & \approx & \varepsilon \\
\emptyset^* & \approx & \varepsilon
\end{array}
$$

Example: Matching a sequence of notes

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

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

ceg $\checkmark$

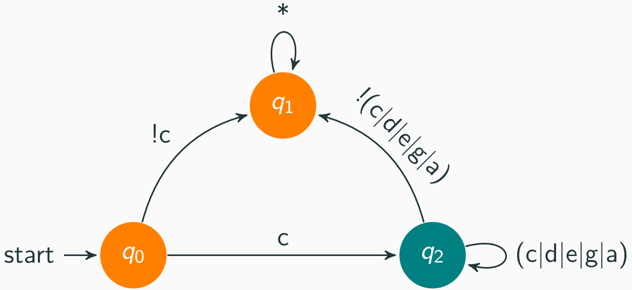
$$\begin{aligned}
\partial_c c \cdot (c|d|e|g|a)^* &= \varepsilon \cdot (c|d|e|g|a)^* \\
\partial_e \varepsilon \cdot (c|d|e|g|a)^* &= (\emptyset \cdot (c|d|e|g|a)^*) \mid (\emptyset|\emptyset|\varepsilon|\emptyset|\emptyset) \cdot (c|d|e|g|a)^* \\
&= (c|d|e|g|a)^* \\
\partial_g (c|d|e|g|a)^* &= (\emptyset|\emptyset|\emptyset|\varepsilon|\emptyset)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

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

Questions?

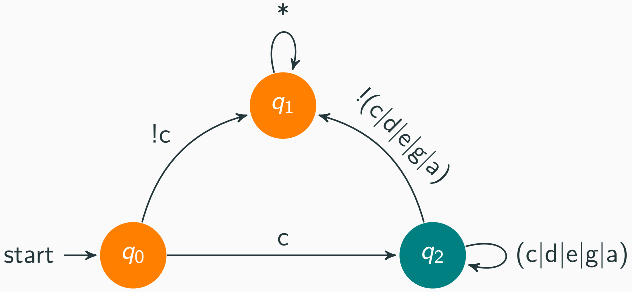
Deterministic Finite Automata

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



Memoization and Simplification

$$\begin{aligned} q_0 &= c \cdot (c|d|e|g|a)^* \\ q_1 &= \emptyset \\ q_2 &= (c|d|e|g|a)^* \end{aligned}$$



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

Recursive Regular Expressions

- One extra concept: Reference
- Two operations
- Define a reference: $\#myref = (a \cdot b)^*$
- Use a reference: $(@myref \mid c)$

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

Ragas - Indian Classical Music

https://youtu.be/iEIMWziZ62A?t=136

Ragas

- Ragas are indian version of western scales [5].
- Stricter than western scales.
- Possible next note depends on current note.
- Notes labeled 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

I will skip over microtones and all the other theory, just because its not relevant to this talk.

Example Raga

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

- Western Pentatonic scale
- Ascent: c d e g a c^1
- Descent: c^1 a g e d c

Given the current note you must choose the next ascending or descending note.

For example given G (e) you can choose P (g) if you want to ascend or R (d) if you want to descend.

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

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

Demo

Context Free Grammars

Left Recursive Raga

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

nullable and derivative each have infinite recursion.

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

Parsing with Derivatives

This has been solved using [3] functional concepts:

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

Laziness

Strict:

```
func eval(params ...) value {
    ...
    return value(...)
}
```

Lazy:

```
func lazyEval(params ...) func() value {
    return func() value {
        return eval(params)
    }
}
```

Laziness - The Brake

Instead of storing the field values of **concat**, **or** and **zero or more** we rather store functions that when called will return the value of the field.

This allows us to avoid any recursion until the value is needed.

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

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

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

Memoization - The Handbrake

Eventually nullable is going to be called.

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

Which will result in the execution of a lazy derivative function.

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

Which can result in infinite recursion.

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

Memoizing helps by closing 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

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

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

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

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

$$\text{least fixed point} = 0$$

Least Fixed Point of Derivative

$$\partial_a r = r$$

$$\partial_a \emptyset = \emptyset$$

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

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

$$\text{least fixed point} = \emptyset$$

Least Fixed Point - The Gas

Nullable is relentless:

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

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

Yacc is Dead

Now we have a fully general Context Free Grammar validator.

Yacc, Antlr, Flex, Bison, etc. definitely still perform better, especially in worst case.

But derivatives:

- are a lot easier to implement and understand than LR(1), LL(1), LALR parsers;
- have a lot in common with the functional Parser pattern;
- can validate the full set of 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.

Implementation and specification are done using derivatives.

XMLNodes instead of Characters.

New Operators: Not, Interleave and Optional

$$
\begin{array}{lcl}
\partial_a(r \mathbin{\&\&} s) & = & (\partial_a r \mathbin{\&\&} s) \mid (\partial_a s \mathbin{\&\&} r) \\
\nu(r \mathbin{\&\&} s) & = & \nu(r) \text{ and } \nu(s) \\
\emptyset \mathbin{\&\&} r & \approx & \emptyset \\
\varepsilon \mathbin{\&\&} r & \approx & r \\
 & & \\
\partial_a!(r) & = & !(\partial_a r) \\
\nu(!(r)) & = & \text{not}(\nu(r)) \\
 & & \\
(r)? & \approx & r \mid \varepsilon
\end{array}
$$

TreeNode

```
deriv :: Expr -> Tree -> Expr
deriv (TreeNode nameExpr childExpr)
      (Node name children) =
  if nameExpr == name then
     let childDeriv = foldl deriv childExpr children
     in if nullable childDeriv
        then Empty
        else EmptySet
  else EmptySet

nullable (TreeNode _ _) = False
```

Video https://youtu.be/SvjSP2xYZm8

Katydid Relapse

katydid.github.io - Relapse: Tree Validation Language.

Uses derivatives and memoization to build Visual Pushdown Automata.

JSON, Protobufs, Reflected Go Structures and XML

New Operators: And, Contains and ZAny

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

Playground and 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/`.