Ragax: Ragalur Expressions

Using derivatives to validate Indian Classical Music

Walter Schulze

27 April 2016

Cape Town Functional Programming Meetup

Key Takeaways

After this talk you should be able to:

- implement a regular expression matcher function and
- invent your own regular expression operators,

because **derivatives are that simple**.

You will also learn about:

- the most basic concept of a Raga;
- laziness, memoization and least fixed point;
- derivatives for context free grammars; and
- derivatives for trees.

Regular Expressions

Matching a string of characters

Using a regex we can validate a string.

$$a(a|b)*$$

ab	✓
aabbba	✓
ac	×
ba	×

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

Derivatives

Easy

The Brzozowski derivative is the **easiest way to evaluate a regular expression**.

Three functions:

- Nullable
- Simplification (optional)
- Derivative

What is a Derivative

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

For example:

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

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

$$
\begin{array}{lcl}
\partial_a \emptyset & = & \emptyset \\
\partial_a \epsilon & = & \emptyset \\
\partial_a a & = & \epsilon \\
\partial_a b & = & \emptyset \quad \text{for } b \neq a \\
\partial_a (r \cdot s) & = & \partial_a r \cdot s \mid \jmath(r) \cdot \partial_a s \\
\partial_a (r^*) & = & \partial_a r \cdot r^* \\
\partial_a (r \mid s) & = & \partial_a r \mid \partial_a s
\end{array}
$$

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

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)
```

Simplification

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

Execution

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

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

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

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

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 $\checkmark$

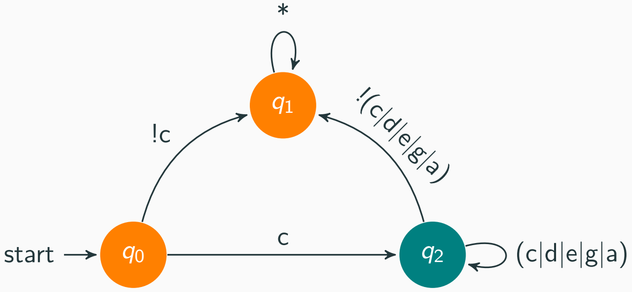
$$\begin{aligned}
\partial_c c \cdot (c|d|e|g|a)^* &= \varepsilon \cdot (c|d|e|g|a)^* \\
\partial_e \varepsilon \cdot (c|d|e|g|a)^* &= (\emptyset \cdot (c|d|e|g|a)^*) \mid (\emptyset|\emptyset|\varepsilon|\emptyset|\emptyset) \cdot (c|d|e|g|a)^* \\
&= (c|d|e|g|a)^* \\
\partial_g (c|d|e|g|a)^* &= (\emptyset|\emptyset|\emptyset|\varepsilon|\emptyset)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

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

Questions?

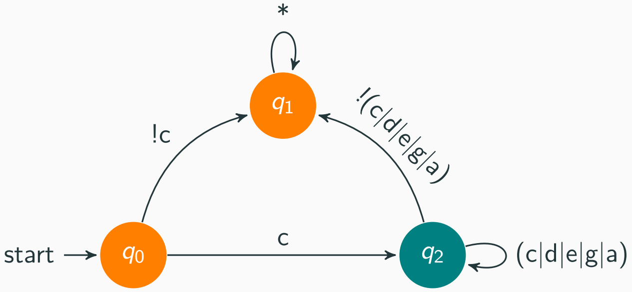
Deterministic Finite Automata

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



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]

Ragas - Indian Classical Music Thank you: Ryan Lemmer and Antoine Van Gelder

Video

https://www.youtube.com/watch?v=iEIMWziZ62A

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.

Play Raag Bhupali

Questions?

An expression for a Raga

Recursive Regular Expressions

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

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

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

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

nullable and derivative each have infinite recursion.

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

Parsing with Derivatives

This has been solved using [3] functional concepts:

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

Laziness

Instead of evaluating a function and returning a value,

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

we defer the evaluation and return a function that will return the value.

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

Memoization

A function's results are cached for the given inputs.

Least Fixed Point

When a function call results in calling the same function with the same inputs, we return a chosen fixed point.

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.

$$
\begin{array}{lcl}
\partial_a(r|s) & = & \lambda(\partial_a r) \mid \lambda(\partial_a s) \\
\partial_a(r^*) & = & \lambda(\partial_a r) \cdot r^* \\
\partial_a(r \cdot s) & = & \lambda(\lambda(\partial_a r) \cdot s) \mid \lambda(\lambda(\jmath(r)) \cdot \lambda(\partial_a s)) \\
 & & \\
\partial_n \# S & = & \lambda(\partial_n(\texttt{@}S \cdot (S \cdot (\texttt{@}R \mid \texttt{@}D)))) \mid \lambda(\partial_n \varepsilon)
\end{array}
$$

Memoization - The Handbrake

Eventually nullable is going to be called.

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

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

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

Which can result in infinite recursion.

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

Memoizing helps by closing the loop.

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

Least Fixed Point - The Gas

BUT Nullable is relentless and still needs to return a true or a false.

Laziness and memoization are not enough.

A least fixed point returns a bottom when an input is recursively revisited.

For the nullable function our bottom is false.

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

Least Fixed Point (without Laziness)

$$
\begin{aligned}
\nu(\#S) &= \nu(\texttt{@}S \cdot (S \cdot (\texttt{@}R \mid \texttt{@}D)) \mid \varepsilon) \\
&= (\nu(\texttt{@}S) \textit{ and } \nu(S \cdot (\texttt{@}R \mid \texttt{@}D))) \textit{ or } \nu(\varepsilon) \\
&= (\text{bottom and false) or true} \\
&= (\text{false and false) or true} \\
&= \text{true} \\
&= \nu((S \cdot (\texttt{@}R \mid \texttt{@}D))^*)
\end{aligned}
$$

Demo

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

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

TreeNode

```haskell
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://www.youtube.com/watch?v=SvjSP2xYZm

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

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

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/`.