Ragax: Ragalur Expressions

Using derivatives to validate Indian Classical Music

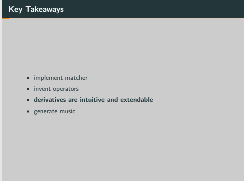
Walter Schulze

2 May 2018

Amsterdam Functional Programming Meetup

Key Takeaways

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



Hello

Welcome to my talk on Ragax or Ragalur Expressions ... pun intended.

I am going to use derivatives to show you how easy it is to implement your own regular expression matcher function. Derivatives are so trivial and extendable, that you should also be able to invent your own operators. This also means that we can extend it to Context Free Grammars, which we can use for Ragas, the whole point of this talk.

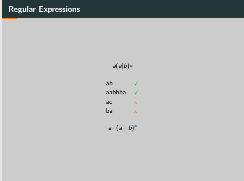
Ok, lets go.

Regular Expressions

$$a(a|b)*$$

ab	✓
aabbba	✓
ac	×
ba	×

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



Lets just do a quick review of regular expressions to make sure we are on the same page. Here we see an expression that matches a string that starts with an 'a', which is followed by any number of 'a's and 'b's

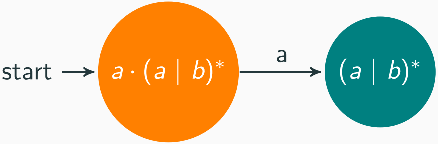
We can also write it like this (point to bottom). Which is the syntax we'll use in the rest of this talk. So the concat operator is represented with a dot.

Also notice that we are not doing substring matching and that we only match the whole string. See for example "ba" that is not matched.

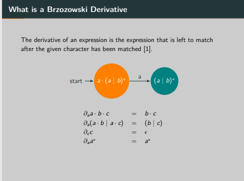
Any questions?

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].



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



What is a derivative? ... The derivative of an expression is the expression that is left to match after the given character has been matched. So the derivative of the expression a, a or b star with respect to a is a or b star.

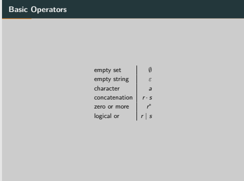
Here are some more examples:

- The derivative of abc with respect to a is ab.
- With an "or" we have to try both alternatives so we take the derivative of both.
- The derivative of c with respect to c is the empty string
- The derivative of a star with respect to a stays a star.

I will now explain the formal rules, but I hope that these examples give you an intuition.

Basic Operators

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



First we have a few basic operators.

If we think of the set of strings that matches a regular expression, then the empty set does not match any strings.

The empty string matches only the empty string.

The character a matches only the character a, not the substring a, remember we are only dealing with whole strings.

And then we have the rest: concat, zero or more and or.

Basic Operators

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

Basic Operators

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

We can represent this in haskell using an algebriac data type.

But you don't need to understand the haskell to understand most of this talk.

I just thought that for those who know it, it might make the math easier.

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.

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

Before we can explain the derivative algorithm we first need to understand the nullable function.

The nullable function returns true if the set of strings that the regular expression matches includes the empty string.

The emptyset matches no strings, so it also does not match the empty string.

the empty string, well uhm yes

the character a, no, because it only matches the character a

The concatenation of r and s, well only if r and s contain the empty string.

r star, zero or more, includes zero, which is the empty string

r or s, well r or s needs to include the empty string.

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

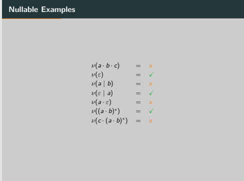
Does the expression match the empty string.

```
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
```

Or if you prefer the haskell

Nullable Examples

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



Lets look at some examples or maybe you can tell me if there is an example here that you disagree with or don't understand ... everyone happy?

Derivative Rules

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

Derivative Rules

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

Now lets do the formal derivative rules. The derivative of the emptyset is always going to be the emptyset. The derivative of the empty string is also always the emptyset. The empty string does not expect any more characters. It is done.

The derivative of a single character given the same character is the empty string, but otherwise it won't match any string, so it is the emptyset.

The derivative of a concatenation has two cases. If the left expression is not nullable then we simply take the derivative of the left expression concatenated to the right. But if the left expression is nullable, then we have to consider the case where we skip over it and take the derivative of the right expression.

The derivative of the zero or more expression is the concatenation of the derivative of the contained expression and the original zero or more expression. The or expression simply pushes its problems down to its children.

Derivative Rules

```haskell
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)
```

Derivative Rules

```haskell
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)
```

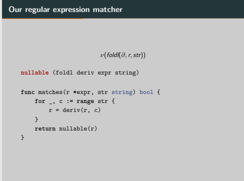
Maybe the haskell is easier to understand.

Our regular expression matcher

$$\nu(\mathit{foldl}(\partial, r, \mathit{str}))$$

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

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



Now lets put our two functions together to create a regular expression matcher.

We simply apply the derivative function to each character in the string starting with our regular expression.

If the regular expression that we end up with matches the empty string, then the regular expression matches the input string.

Simplification

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

Simplification

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

Finally we have simplification which is optional, but is great for optimization and will make our examples much more readable. Here are some of the rules, which should be intuitive:

- Any expression concatenated with the emptyset is equivalent to the emptyset.
- Any expression concatenated with the empty string is equivalent to the expression.
- Any expression ored with itself is equivalent to that expression.
- Any expression ored with the emptyset is equivalent to that expression.
- Zero or more, zero or more times, is still just zero or more.

Simplification can be part of the derivative function or happen in the constructors.

Example: Matching a sequence of notes

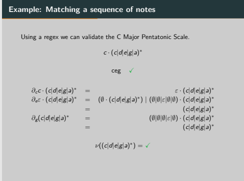
Using a regex we can validate the C Major Pentatonic Scale.

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

ceg ✓

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

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



Ok finally we have another example. Lets say our characters are musical notes. And we have an expression for a c major pentatonic scale.

We can take the derivative of the regular expression with respective to each input note to get a resulting regular expression. It matches if the resulting regular expression is nullable.

Lets walk through it. The derivative of the initial expression with respect to c is the empty string concatenated with the zero or more expression. We could simplify that, but lets first try taking the next derivative, for the sake of example.

So the derivative of the empty string concatenated with the zero or more expression is the emptyset concatenated with the zero or more expression, but since the empty string is nullabe we also have to take the derivative of the right expression as an alternative. Here we can see that every musical note has become the emptyset, except the matching character which has become the

Questions?

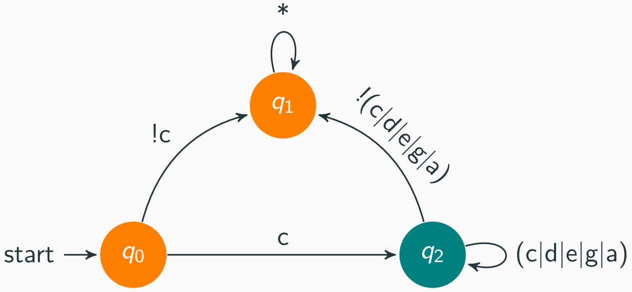
Questions?

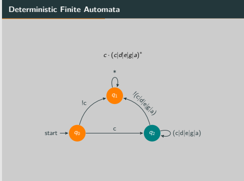
Questions

This is the part of the talk that everything builds on so if you don't understand something lets quickly take some time to try and get it right.

Deterministic Finite Automata

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





Lets just take a quick tangent.

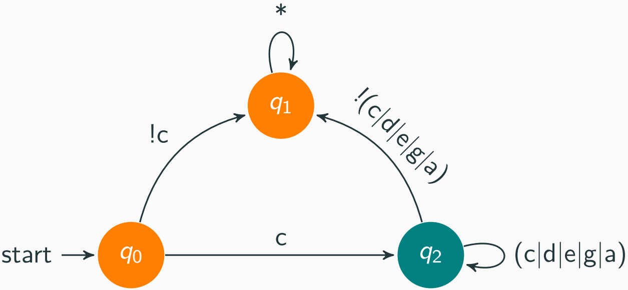
Do you remember deterministic finite automata.

Here is one for the regular expression we just evaluated.

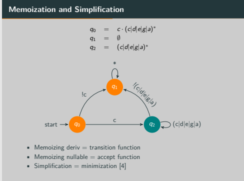
Given a c we go the accepting state. Then we accept any note in the pentatonic scale zero or more times. Otherwise the song is rejected.

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]



We can create the same DFA with derivatives by simply memoizing the derivative function for all possible inputs.

The memoized derivative function is the transition function where the regular expressions themselves are states. The memoized nullable function is the accept function. Our simplification rules can be used for minimization.

Ragas - Indian Classical Music

Ok now onto Ragas. First I have to thank Ryan Lemmer and Antoine Van Gelder. They introduced me to Ragas and came up with the idea to combine it with derivatives.

Video https://youtu.be/iElMWziZ62A

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$\sharp$	d	d$\sharp$	e	f	f$\sharp$	g	g$\sharp$	a	a$\sharp$	b

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

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.

Ragas are basically the indian version of western scales.

They are stricter than western scales.

The possible next note is not simply chosen from a set, but rather depends on the current note.

Also notes are labeled a bit differently.

They are labeled relatively to the start note. So this is simply how they are labeled if you start at c.

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.

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.

Here is Raag Bhupali a type of Pentatonic scale

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

For example given G you can choose to play P or R next

Play simple song

Play Raag Bhupali

Play Raag Bhupali

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

Questions?

Questions

Is everyone still with us. After this things start to speed up a little. It starts to become less of a lesson and more a presentation. So its good to have this part solid.

An expression for a Raga

Ok, lets write an expression for a Raga

Recursive Regular Expressions

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

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

Recursive Regular Expressions

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

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

Wait, before we start lets just quickly add one more concept.

We need references for recursion and to help us split our expression into parts.

It has two operators

The definition of a reference

and the use of the referenace

The derivative of @q with respect to a is the derivative of whatever q's definition was with respect to a.

Same with nullable.

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

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

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

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

Ok so here finally we have a raga expressed as a recursive regular expression.

Our first note S is followed by its ascent or descent note and the whole expression is repeated zero or more times.

The ascending note R is ascended by G or descended back to S which is just the empty string, since the termination of the expression results in the end or the repetition of the whole expression which starts with S.

Same goes for the rest.

G is followed by P or R

P is followed by D or G

and D is followed by P or a termination, since it can be followed by S.

Demo

Lets see it in action

This program basically does not allow me to play a note that will make the expression go into an emptyset state.

We can even add random input to create a generated piece that satisfies the raga rules.

Context Free Grammars

Ok back to the theory, now its going to get a little tough.

Don't worry if you don't understand it all, because it took me quite a while.

I think if you just absorb the overview then you are doing it right.

Oh and by the way if you are not going to be pendantic then Context free grammars are just another name for recursive regular expressions.

Our example from before was already a context free grammar.

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.

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

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.

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

Here is our expression from before, but just written a bit differently. We cannot really control how a user uses our expression language, so we have to think of all cases.

The definition of S is either the empty string or it is the S again followed by the expression contained in the original zero or more expression.

You can see how these are equivalent.

Unfortunately this causes infinite recursion for the nullable and derivative function.

We can see that calculating nullable for S requires the calculation of the nullability of S.

Parsing with Derivatives

This has been solved using [3] functional concepts:

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

This has been solved using [3] functional concepts:

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

We are going to solve this problem with 3 functional concepts.

laziness

memoization

least fix points

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

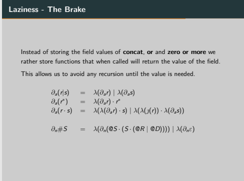
When a function call results in calling the same function with the same inputs, we return a chosen fixed point.

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.

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



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.

I have used the lambda symbol here to represent a function that will return a value when called.

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 - 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.

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

Nullability is relentless and won't be stopped by laziness.

The point of these equatiosn are that we still need the nullability of the derivative of S to calculate the nullability of the derivative of S.

Memoizing helps to close the loop.

It stops the recursive execution of the derivative by only calculating the derivative once and returning the same lazy function for following executions.

But its not enough.

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.

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

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.

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

Eventually nullable still needs to return a true or a false.

Laziness and memoization are not enough.

Here we calcuate the least fix point of nullable given a bottom of false. Basically, when the same lazy expression is recursively revisited, while calculating nullability, a false is returned.

But lets first look at a simpler example

Least Fixed Point (without Laziness)

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

Least Fixed Point (without Laziness)

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

Here is an example without laziness

Basically we set a bottom of false, meaning that if we revisit an expression we return false.

So nullable of $@S$ revisits $\#S$ which returns the bottom and allows us to evaluate the rest of the expression and stop the recursion.

The concat returns false, since it starts with a character.

The empty string returns true.

And then we can see that our equation evaluates to true.

We can also see that this nullability is equivalent to the nullability of the equivalent expression.

– Go back

Demo

Now if we do the same to our lazy expressions we get false, since this is the derivative of S and the expression always expects more than one note.

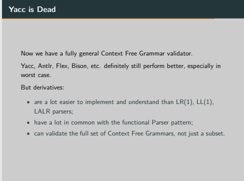
Yacc is Dead

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.



Now we have a fully general Context Free Grammar validator.

Yacc, Antlr, Flex, Bison, etc. definitely still performs 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 and

can validate the full set of Context Free Grammars, not just a subset.

Trees

Finally lets quickly just brush over trees

Relaxing

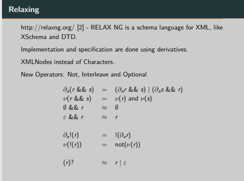
http://relaxng.org/ [2] - RELAX NG is a schema language for XML, like XSchema and DTD.

Implementation and specification are done using derivatives.

XMLNodes instead of Characters.

New Operators: Not, Interleave and Optional

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



RELAX NG is a schema language for XML, like XSchema and DTD.

The implementation specification is done using derivatives.

Some differences are that:

Instead of a character as an input to the derivative function we have an XML Node.

They have also included some new operators: Not or Compliment, Interleave and Optional

With interleave the derivative can take the derivative of any one of the interleaving patterns. The other pattern needs to keep its original form.

Nullability is easy.

Not, like all the logical operators just passes its problem down.

And optional is just syntactic sugar.

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
```

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
```

So here is the haskell code for a derivative of a Tree which is pretty close to an XML Node.

The TreeNode pattern has a name expression and a child expression. To keep it basic I have made the name expression just a plain string.

The Node has a label string and a list of child nodes.

if the name expression equals the label string we have to take the derivative of the children.

otherwise we return the emptyset just like with a character.

if the result of taking the derivative of the children is nullable we return the empty pattern else we return the emptyset.

And nullability of a treenode is always false, just like a character.

Video https://youtu.be/SvjSP2xYZm8

Katydid Relapse

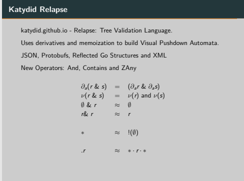
katydid.github.io - Relapse: Tree Validation Language.

Uses derivatives and memoization to build Visual Pushdown Automata.

JSON, Protobufs, Reflected Go Structures and XML

New Operators: And, Contains and ZAny

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



Finally the thing I am actually working on.

Katydid is a tree toolkit, which includes relapse: a validation language based on relaxng

I use derivatives and memoization to build a visual pushdown automata. This allows me to do matching with zero memory allocations once the automata has been compiled.

I support. Json, Protobuf, XML and reflected structures, but its easy to add your own parser.

I added some new operators.

And, Contains and ZAny.

Zany is just zero or more of anything, like .* in a regular expression. Which is just syntatic sugar for the compliment of the emptyset.

Playground and Tour

I can now show you the playground.

And I also have a tour.

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/`.

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/`.

Thank you