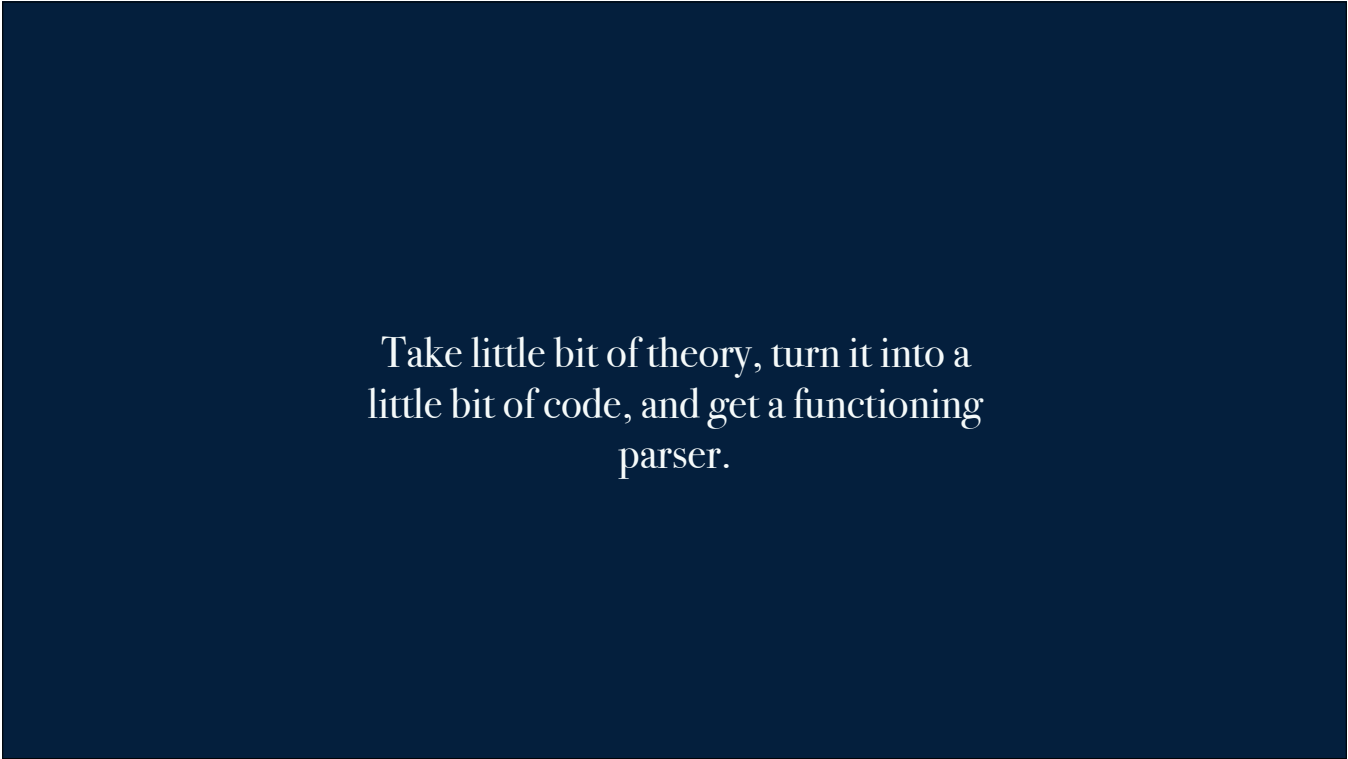


# Parsing with Derivatives

David Darais   Daniel Spiewak   Matthew Might

<http://matt.might.net/papers/might2011derivatives.pdf>



Take little bit of theory, turn it into a  
little bit of code, and get a functioning  
parser.

The important thing to the authors is that there's minimal distance between the theory, in this case, theory around formal languages, and running code that can be performant in practice.

In this talk, I'm going to be talking more about the case of recognizers. Recognizers are functions that verify whether a string belong to language, but don't provide a parse tree. Parse trees are what you use to extract semantics from a string and to generate abstract syntax tree.

"The derivative of regular expressions [1], if gently tempered with laziness, memoization and fixed points, acts immediately as a pure, functional technique for generating parse forests from arbitrary context-free grammars. Despite—even because of—its simplicity, the derivative transparently handles ambiguity, leftrecursion, right-recursion, ill-founded recursion or any combination thereof"

## STRUCTURE

1. Turn Brzozowski's derivative into code
2. Generalize to context-free grammars (all CFGs!)
3. Make performant


- Brzozowski's equations are for regular languages. Turning them into code gives you a recognizer for regular languages / regular expressions.
- The generalization happens using normal tools like memoization and laziness
- The parser ends up having a cubic upper bound (shown in 2016, paper linked)

# The Derivative of a Language

If you skimmed this paper, the weirdest thing is probably the notion of a language or a grammar having a derivative. Brzozowski defines the derivative as function of a formal language and a character that returns a new formal language.


a formal language is a set of strings

Real quick, want to brush up on some of the language used in the paper.



{ foo, bar }

Here's an example set.



`/foo|bar/`

We can also express that as a regular expression. I'm going to start by talking about languages as sets, and then move to regular expressions, which should be a little more familiar.

# the derivative

The derivative of one of those sets is defined as an operation with two steps.



1. **Filter:** keep every string starting with  $c$
2. **Chop:** remove  $c$  from the start of each string

We take a language  $L$  and a character  $c$ . Then we filter to just those strings in  $L$  that start  $c$ , then we remove  $c$  from the start of each of those remaining strings.

$$D_f \{ \text{foo}, \text{bar}, \text{fit} \} \\ = \\ \{ \text{oo}, \text{it} \}$$

Here's an example. We're taking the derivative with respect to f. First step is to find the strings that start with "f", which here is "foo" and "fit", then chop off the "f". The derivative means where does this language go if I assume a string starts with f?

$$\begin{aligned} D_f & \{ \text{foo}, \text{bar}, \text{fit} \} \\ D_o & \{ \text{oo}, \text{it} \} \\ D_o & \{ \text{o} \} \\ & \{ "" \} \end{aligned}$$

You can use the derivative to test if strings belong to languages. You do that by repeatedly taking the derivative with each successive character in the string you're testing. If your last step produces a set that contains the empty string, then the initial string matches.

The derivative of a single character is the set containing the empty string, which is called the null language and the paper writes with a little epsilon.

$$\begin{aligned} D_f & \{ \text{foo, bar, fit} \} \\ D_o & \{ \text{oo, it} \} \\ D_o & \{ \text{o} \} \\ "" & \in \{ "" \} \end{aligned}$$

In this case, the final step does produce a nullable language, so we know that 'foo' belongs to the initial language.

$$\begin{aligned}
 D_f & \{ \text{foo}, \text{bar}, \text{fit} \} \\
 D_o & \{ \text{oo}, \text{it} \} \\
 D_x & \{ o \} \\
 & \{ \} \\
 "" & \notin \{ \}
 \end{aligned}$$

Here's an example that doesn't match. Line three, we try to match a character against another character, and they're not equal, so the derivative is the empty set. And the empty string does not belong to the empty set, so know this string doesn't belong to the language.

nullable = accepts the empty string

In the paper they talk about "nullable" languages and "nullability" a lot, which just means the language contains the empty string.

## Brzozowski's equations

This technique for determining set membership is obviously a little silly when we're working with literal explicitly listed sets like this. Normally when we work languages, we don't enumerate the member strings, we define a grammar to constrain our set of strings.

Brzozowski's equations define the derivative in terms of regular expressions.

$$D_c(L_1 \cup L_2) = D_c(L_1) \cup D_c(L_2).$$

$$D_c(L^*) = D_c(L) \circ L^*.$$

$$D_c(L_1 \circ L_2) = D_c(L_1) \circ L_2 \text{ if } \epsilon \notin L_1$$

$$D_c(L_1 \circ L_2) = (D_c(L_1) \circ L_2) \cup D_c(L_2) \text{ if } \epsilon \in L_1$$

Brzozowski defines the derivative over three operations — union, repetition, and concatenation. Each of these correspond to something familiar in regular expressions. Union is the "or" operator, or "choice" or "alternation" operator, the repetition star is the same as in regexes, and concatenation is just juxtaposition, putting one term after another in a regex.

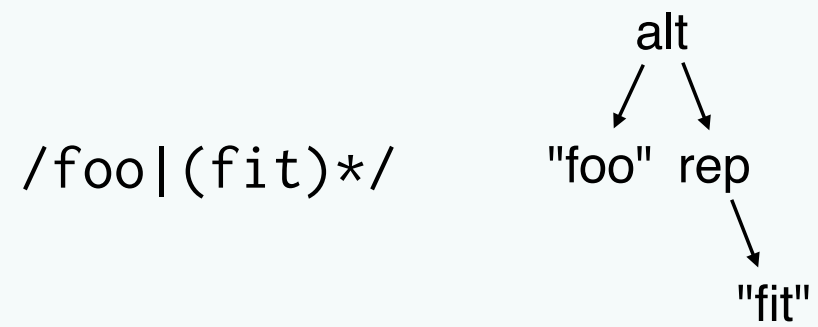


$$D_f \text{ /foo|fit/ } = \text{ /oo|it/ }$$
$$D_f \text{ /(foo)*/ } = \text{ /oo(foo)*/ }$$
$$D_f \text{ /(foo)(fit)/ } = \text{ /oo(fit)/ }$$

The first shows alternation, the second shows repetition, and the third shows concatenation.

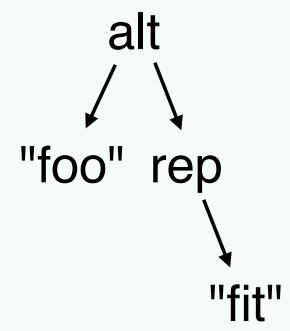
$$D_f \text{ /(foo)(fit)/ } = \text{ /oo(fit)/}$$
$$D_f \text{ /(foo)?(fit)/ } = \\ \text{ /(oo(fit))|it)/}$$

Concatenation has two cases. There's the case where the first term is not nullable, meaning it doesn't contain the empty string. That means that the first term is not optional, we have to match it. If the first term IS nullable, we can either match it, or the second term. So we get the union of those two branches.

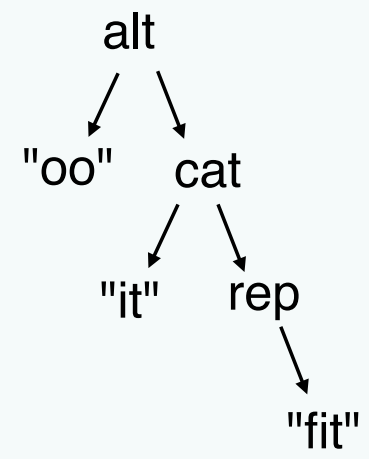


The authors represent regular expressions with a series of structs. There's an "alt" struct for alternation/choice, a "cat" struct for concatenation, and a "rep" struct for repetition. Alt and cat hold pointers to two sub trees, rep has a point to one sub tree.

$D_f$  /foo|(fit)\*/



/oo|it(fit)\*/



$$D_c(\emptyset) = \emptyset$$

$$D_c(\epsilon) = \emptyset$$

$$D_c(c) = \epsilon$$

$$D_c(c') = \emptyset \text{ if } c \neq c'.$$

$$D_c(L_1 \cup L_2) = D_c(L_1) \cup D_c(L_2).$$

$$D_c(L_1 \circ L_2) = D_c(L_1) \circ L_2 \text{ if } \epsilon \notin L_1$$

$$D_c(L_1 \circ L_2) = (D_c(L_1) \circ L_2) \cup D_c(L_2) \text{ if } \epsilon \in L_1$$

$$D_c(L^*) = D_c(L) \circ L^*.$$

```
(define (D c L)
  (match L
    [(empty)      (empty)]
    [(eps)        (empty)]
    [(char a)      (if (equal? c a)
                        (eps)
                        (empty))])

  [(alt L1 L2)    (alt (D c L1)
                       (D c L2))]
  [(cat (and (? δ) L1) L2)
   (alt (D c L2)
        (cat (D c L1) L2))]
  [(cat L1 L2)    (cat (D c L1) L2)]
  [(rep L1)       (cat (D c L1) L)]))
```

This is the implementation of the paper of the derivative of a language  $L$  with respect to  $c$ . There's a series of base cases, for handling the empty set, the empty string, and a single character. "Eps" here is short for epsilon, the symbol they're using for the set containing the empty string. The other cases are lifted straight from Brzowski's derivative.

$$D_c(\emptyset) = \emptyset$$

$$D_c(\epsilon) = \emptyset$$

$$D_c(c) = \epsilon$$

$$D_c(c') = \emptyset \text{ if } c \neq c'.$$

```
[(empty)      (empty)]  
[(eps)        (empty)]  
[(char a)     (if (equal? c a)  
                  (eps)  
                  (empty))]
```

This is the implementation of the paper of the derivative of a language  $L$  with respect to  $c$ . There's a series of base cases, for handling the empty set, the empty string, and a single character. "Eps" here is short for epsilon, the symbol they're using for the set containing the empty string. The other cases are lifted straight from Brzowski's derivative.

$$D_c(L_1 \cup L_2) = D_c(L_1) \cup D_c(L_2).$$

```
[(alt L1 L2) (alt (D c L1)
                  (D c L2))]
```

This is the implementation of the paper of the derivative of a language L with respect to c. There's a series of base cases, for handling the empty set, the empty string, and a single character. "Eps" here is short for epsilon, the symbol they're using for the set containing the empty string. The other cases are lifted straight from Brzowski's derivative.



$$D_c(L_1 \circ L_2) = D_c(L_1) \circ L_2 \text{ if } \epsilon \notin L_1$$

$$D_c(L_1 \circ L_2) = (D_c(L_1) \circ L_2) \cup D_c(L_2) \text{ if } \epsilon \in L_1$$

```
[(cat (and (? δ) L1) L2)
  (alt (D c L2)
        (cat (D c L1) L2))]
```

This is the implementation of the paper of the derivative of a language  $L$  with respect to  $c$ . There's a series of base cases, for handling the empty set, the empty string, and a single character. "Eps" here is short for epsilon, the symbol they're using for the set containing the empty string. The other cases are lifted straight from Brzowski's derivative.

$$D_c(L^*) = D_c(L) \circ L^*$$

```
[(rep L1)      (cat (D c L1) L))])
```

This is the implementation of the paper of the derivative of a language  $L$  with respect to  $c$ . There's a series of base cases, for handling the empty set, the empty string, and a single character. "Eps" here is short for epsilon, the symbol they're using for the set containing the empty string. The other cases are lifted straight from Brzowski's derivative.

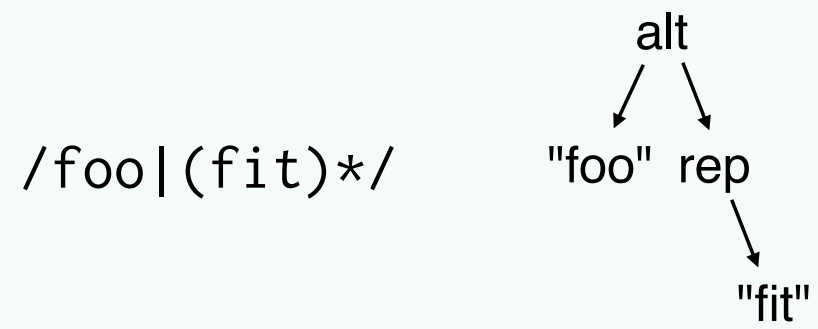
```
(define (matches? w L)
  (if (null? w)
      ( $\delta$  L)
      (matches? (cdr w) (D (car w) L))))
```

To use that function, we keep calling it on successive letters of a string, then when we're done, we check if the resulting language is nullable.

## Extending to context-free grammars

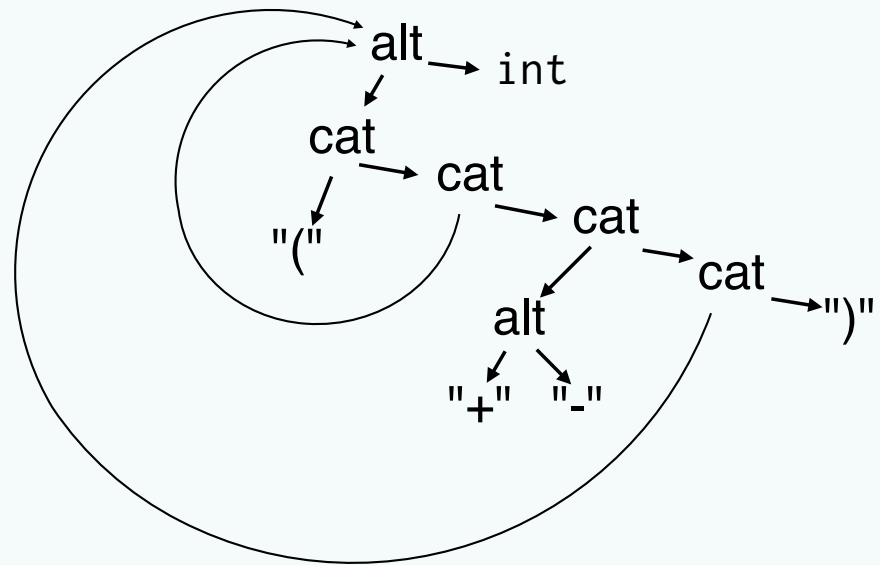
The functions before operate on language defined with just union, repetition, and concatenation. Those are regular languages. To be a useful general-purpose parser, you want to support context-free grammars. Grammars for programming languages, certain models of human language.

The authors do that by allowing the input grammar data structures to be recursive.



With a regular language, recursion isn't possible. You always end up with strict trees.

```
term = "(" term op term ")" | int  
op = "+" | "-"
```



With a regular language, recursion isn't possible. You always end up with strict trees.

```

(define (D c L)
  (match L
    [(empty)      (empty)]
    [(eps)        (empty)]
    [(char a)      (if (equal? c a)
                        (eps)
                        (empty))])

  [(alt L1 L2)    (alt (D c L1)
                       (D c L2))]
  [(cat (and (?  $\delta$ ) L1) L2)
   (alt (D c L2)
        (cat (D c L1) L2))]
  [(cat L1 L2)    (cat (D c L1) L2)]
  [(rep L1)       (cat (D c L1) L)])])

(define ( $\delta$  L)
  (match L
    [(empty)      #f]
    [(eps)        #t]
    [(char _)      #f]

    [(rep _)       #t]
    [(alt L1 L2)   (or ( $\delta$  L1) ( $\delta$  L2))]
    [(cat L1 L2)   (and ( $\delta$  L1) ( $\delta$  L2))]))

```



```

(define/memoize (D c L)
  #:order ([L #:eq] [c #:equal])
  (match L
    [(empty)      (empty)]
    [(eps* T)      (empty)]
    [( $\delta$  _)      (empty)]
    [(char a)      (if (equal? a c)
                        (eps* (set c))
                        (empty))])

  [(alt L1 L2)    (alt (D c L1) (D c L2))]
  [(cat L1 L2)    (alt (cat (D c L1) L2)
                       (cat ( $\delta$  L1) (D c L2)))]
  [(rep L1)       (cat (D c L1) L)]
  [(red L f)      (red (D c L) f)])])

(define/fix ( $\delta$  L)
  #:bottom #f
  (match L
    [(empty)      #f]
    [(eps)        #t]
    [(char _)      #f]

    [(rep _)       #t]
    [(alt L1 L2)   (or ( $\delta$  L1) ( $\delta$  L2))]
    [(cat L1 L2)   (and ( $\delta$  L1) ( $\delta$  L2))]))

```

The really interesting thing about this paper is the extension of the parser from regular languages to context-free languages, and from a recognizer into a parser. And they do that by allowing the components of the grammar to be mutually recursive.

You have to add three things to prevent infinite recursion —

the ``alt``, ``cat``, structs become lazy. That means that what looks immediate recursion in the alt and cat cases does happen immediately, but on demand.

Second, the derivative is memoized, so you don't follow cycles forever

Third, the nullability test function is wrapped in a fixed point combinator, so self-referential nullability tests terminate with "false".

# Performance

An important part of the paper is they start approach tractable performance. There's a follow up paper for 2016, that I'll provide a link for, that gets performance ACTUALLY tractable, with similar performance to other parsing libraries in Racket.



$$D_b D_a \ a \circ b$$

$$((\emptyset \circ \emptyset) \cup (\epsilon \circ \epsilon)) \cup ((\emptyset \circ \emptyset) \cup (\emptyset \circ \emptyset))$$

The reason you have to put some effort into performance is that if you just follow the derivative rules, you generate these gigantic grammars that are 95% garbage. For example, this is the end grammar from parsing (a concat b). If you look, you can see that this grammar is actually nullable, since you can simplify it to just epsilon.

$$\emptyset \circ p = p \circ \emptyset \Rightarrow \emptyset$$

$$\emptyset \cup p = p \cup \emptyset \Rightarrow p$$

To prevent exploding grammars, the authors provide a set of patterns

## What I didn't talk about:

- the nullability function  $\delta$
- parser combinators vs. trees of structs
- partial parsers
- null parses to extract parse trees
- reduction nodes

An important part of the paper is they start approach tractable performance. There's a follow up paper for 2016, that I'll provide a link for, that gets performance ACTUALLY tractable, with similar performance to other parsing libraries in Racket.

## Resources

"Parsing with Derivatives"

<http://matt.might.net/papers/might2011derivatives.pdf>

"On the Complexity and Performance of Parsing with Derivatives"

<https://pdfs.semanticscholar.org/528c/5dfcc650c99c4376f7373f84dee664c93779.pdf>

"parseback: A Scala implementation of parsing with derivatives"

<https://github.com/djspiewak/parseback>

An important part of the paper is they start approach tractable performance. There's a follow up paper for 2016, that I'll provide a link for, that gets performance ACTUALLY tractable, with similar performance to other parsing libraries in Racket.