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.

STRUCTURE

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

The Derivative of a Language

a formal language is a set of strings

```
{ foo, bar }
```

/foo|bar/

the derivative

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

$$D_f \{ \texttt{foo}, \texttt{bar}, \texttt{fit} \}$$

$$=$$

$$\{ \texttt{oo}, \texttt{it} \}$$

D_f { foo, bar, fit }

D_o { oo,it }

D_o { o }

{ "" }

$$D_f \ \{ \ \texttt{foo}, \ \texttt{bar}, \ \texttt{fit} \ \}$$

$$D_o \ \{ \ \texttt{oo}, \texttt{it} \ \}$$

$$D_o \ \{ \ \texttt{o} \ \}$$

$$\texttt{""} \in \{ \ \texttt{""} \ \}$$

D_f { foo, bar, fit }

D_o { oo,it }

D_x { o }

{ }

"" $\notin$ { }

nullable = accepts the empty string

Brzozowski’s equations

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

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

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

$$D_f\ /foo|fit/ = /oo|it/$$

$$D_f\ /(foo)*/ = /oo(foo)*/$$

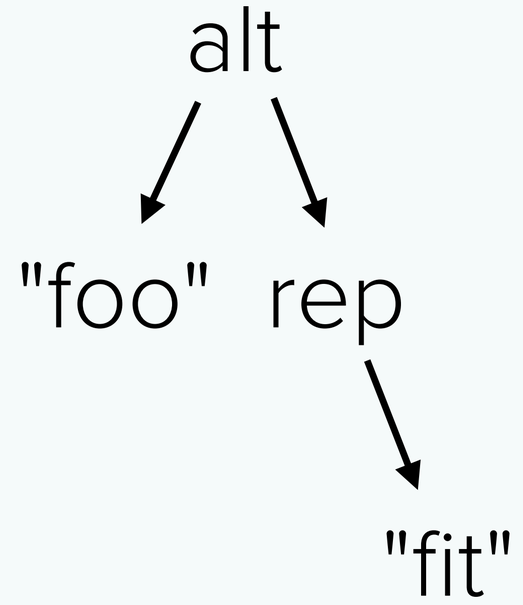
$$D_f\ /(foo)(fit)/ = /oo(fit)/$$

$$D_f\ /(foo)(fit)/ = /oo(fit)/$$

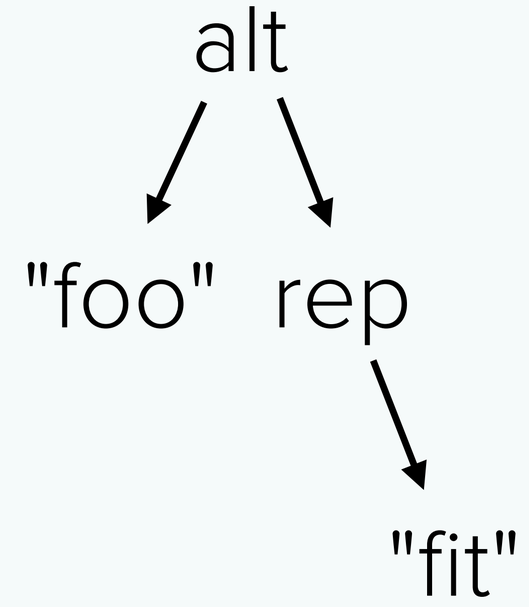
$$D_f\ /(foo)?(fit)/ =$$

$$/(oo(fit))|it)/$$

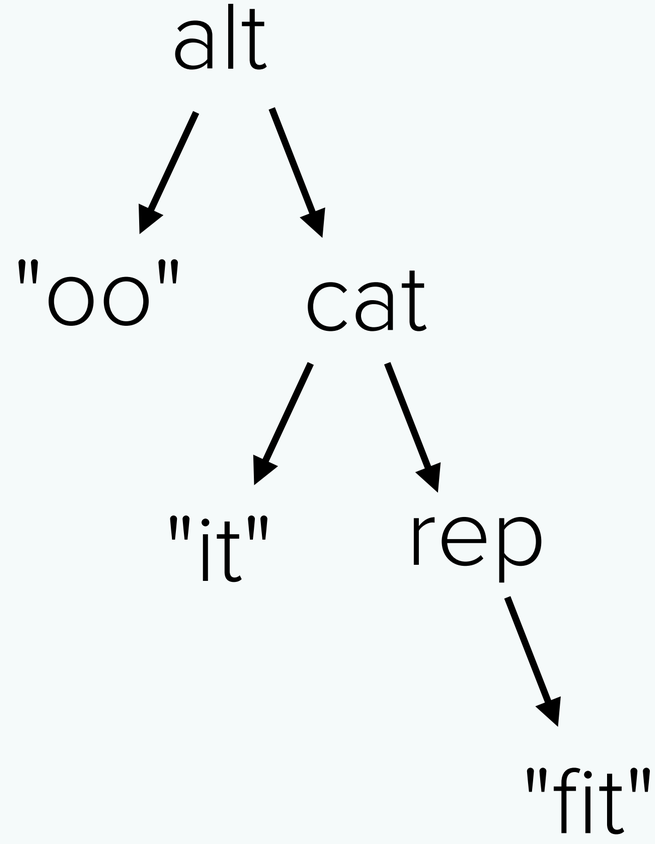
```
/foo|(fit)*/
```



D_f `/foo|(fit)*/`



```
/oo|it(fit)*/
```



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

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

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

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

```
(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)]))
```

$$
\begin{aligned}
D_c(\emptyset) &= \emptyset \\
D_c(\epsilon) &= \emptyset \\
D_c(c) &= \epsilon \\
D_c(c') &= \emptyset \text{ if } c \neq c'.
\end{aligned}
$$

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

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

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

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

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

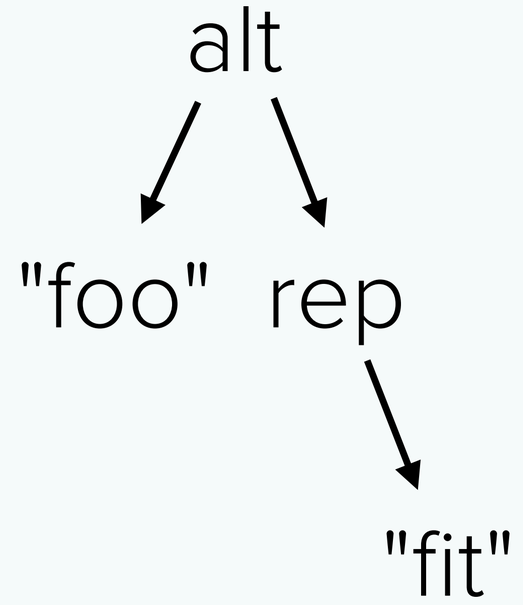
$$D_c(L^\star) = D_c(L) \circ L^\star.$$

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

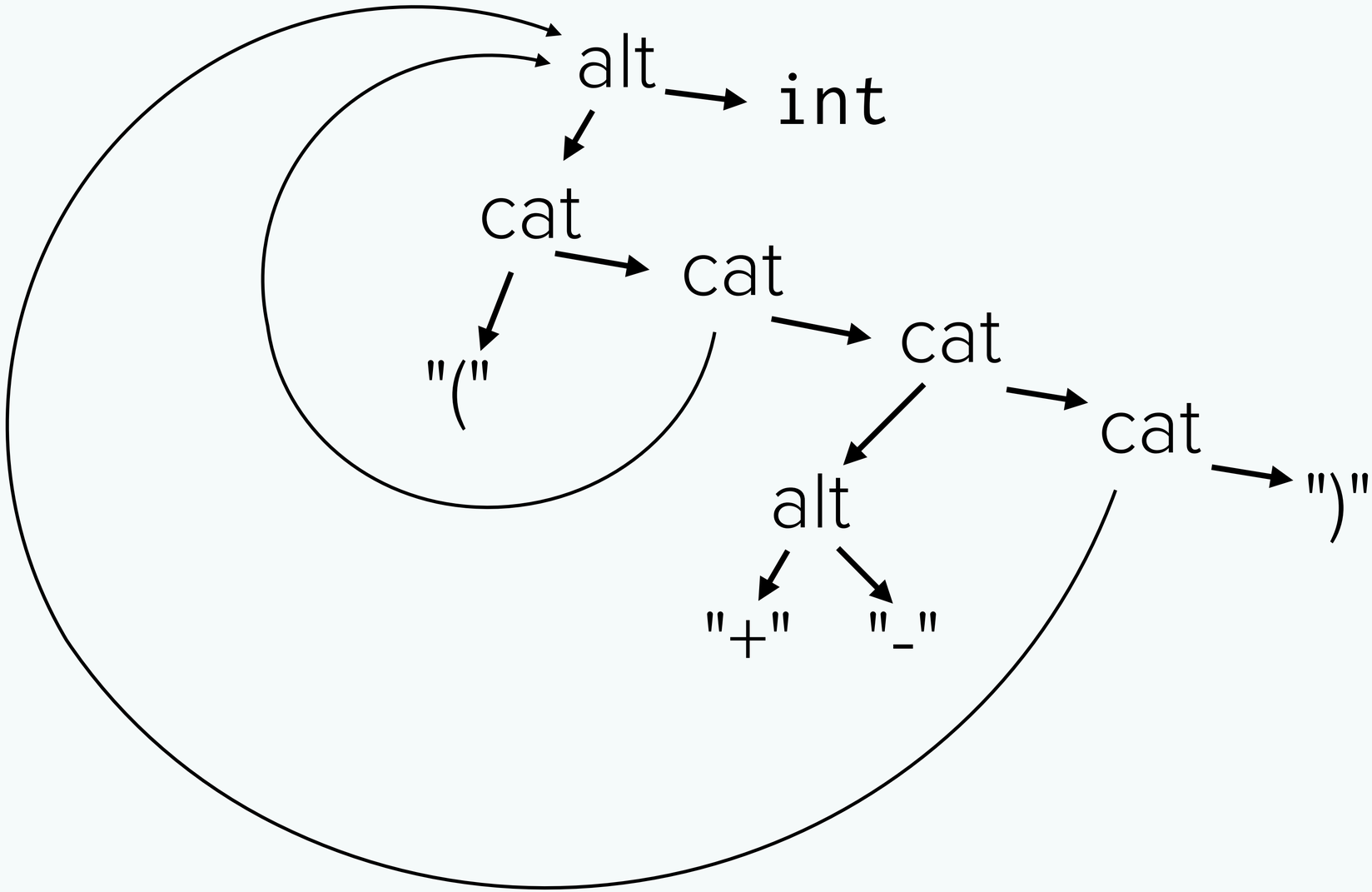
```
(define (matches? w L)
  (if (null? w)
      (δ L)
      (matches? (cdr w) (D (car w) L))))
```

Extending to context-free grammars

```
/foo|(fit)*/
```



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



```
(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)]))
```

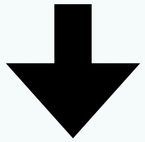
```
(define (δ L)
  (match L
    [(empty)       #f]
    [(eps)         #t]
    [(char _)      #f]

    [(rep _)       #t]
    [(alt L1 L2)   (or  (δ L1) (δ L2))]
    [(cat L1 L2)   (and (δ L1) (δ L2))]))
```

```
(define/memoize (D c L)
  #:order ([L #:eq] [c #:equal])
  (match L
    [(empty)       (empty)]
    [(eps* T)      (empty)]
    [(δ _)         (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 (δ L1) (D c L2))]
    [(rep L1)      (cat (D c L1) L)]
    [(red L f)     (red (D c L) f)]))
```

```
(define/fix (δ L)
  #:bottom #f
  (match L
    [(empty)       #f]
    [(eps)         #t]
    [(char _)      #f]

    [(rep _)       #t]
    [(alt L1 L2)   (or  (δ L1) (δ L2))]
    [(cat L1 L2)   (and (δ L1) (δ L2))]))
```

Performance

$$D_b\, D_a \quad a \circ b$$



$$((\varnothing \circ \varnothing) \cup (\varepsilon \circ \varepsilon)) \cup ((\varnothing \circ \varnothing) \cup (\varnothing \circ \varnothing))$$

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

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

What I didn't talk about:

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

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