Parsing with Derivatives

A Functional Pearl

Matt Might*
University of Utah
matt.might.net

David Darais
Harvard University
david.darais.com

Daniel Spiewak
Wisconsin-Milwaukee
codecommit.com

(Presented at ICFP 2011)
“I want to do parsing.”

-Me, new Grad Student
“grad-school Vietnam”

“charred remains”

“would-be Ph.D.s”

-Olin Shivers
Parsing should be simple.
Parsing should be functional.
Parsing should be fun.
It is not.
LL vs. LR
LR vs. LALR
Left-recursive?
Right-recursive?
Shift / reduce tables
Shift / reduce conflicts
Backtracking
Table management
Ambiguity?
There is a way.
Brzozowski’s derivative.
Derivatives of Regular Expressions

Janusz A. Brzozowski

Princeton University, Princeton, New Jersey†

Abstract. Kleene’s regular expressions, which can be used for describing sequential circuits, were defined using three operators (union, concatenation and iterate) on sets of sequences. Word descriptions of problems can be more easily put in the regular expression language if the language is enriched by the inclusion of other logical operations. However, in the problem of converting the regular expression description to a state diagram, the existing methods either cannot handle expressions with additional operators, or are made quite complicated by the presence of such operators. In this paper the notion of a derivative of a regular expression is introduced and the properties of derivatives are discussed. This leads, in a very natural way, to the construction of a state diagram from a regular expression containing any number of logical operators.
1964

Derivatives of Regular Expressions

Janusz A. Brzozowski

Princeton University, Princeton, New Jersey

Abstract. Kleene's regular expressions, which can be used for describing sequential circuits, were defined using three operators (union, concatenation and iterate) on sets of sequences. Word descriptions of problems can be more easily put in the regular expression language if the language is enriched by the inclusion of other logical operations. However, in the problem of converting the regular expression description to a state diagram, the existing methods either cannot handle expressions with additional operators, or are made quite complicated by the presence of such operators. In this paper the notion of a derivative of a regular expression is introduced and the properties of derivatives are discussed. This leads, in a very natural way, to the construction of a state diagram from a regular expression containing any number of logical operators.
(define-struct ∅ {})  
(define-struct ε {})  
(define-struct token {value})  
(define-struct δ {lang})  
(define-struct u {this that})  
(define-struct o {left right})  
(define-struct ★ {lang})

(define (D c L)
  (match L
    [(∅) (∅)]
    [(ε) (∅)]
    [(δ _) (∅)]
    [(token a) (if (eqv? a c) (ε) (∅))]
    [(∪ L1 L2) (∪ (D c L1) (D c L2))]
    [(★ L1) (∘ (D c L1) L)]
    [(o L1 L2) (∪ (∘ (δ L1) (D c L2)) (∘ (D c L1) L2)))]
))

(define (nullable? L)
  (match L
    [(∅) #f]
    [(ε) #t]
    [(token _) #f]
    [(★ _) #t]
    [(δ L1) (nullable? L1)]
    [(∪ L1 L2) (or (nullable? L1) (nullable? L2))]
    [(o L1 L2) (and (nullable? L1) (nullable? L2))]]
)

(define (recognizes? w p)
  (cond [(null? w) (nullable? p)]
        [else (recognizes? (cdr w) (D (car w) p))]))
(define-struct ∅ { })
(define-struct ε { })
(define-struct token {value})
(define-struct δ {lang})
(define-struct ∪ {this that})
(define-struct ∘ {left right})
(define-struct ★ {lang})

(define (D c L)
  (match L
    [(∅) (∅)]
    [(ε) (∅)]
    [(δ _) (∅)]
    [(token a) (if (eqv? a c) (ε) (∅))]
    [(∪ L1 L2) (∪ (D c L1) (D c L2))]
    [(∘ L1 L2) (∘ (D c L1) L2)]
    [(★ L1) (★ (D c L1) L)]
    [(→ L1 f) (→ (D c L1) f)]
    [(→ L1 f) (→ (D c L1) f)]
    [(null? w) (nullable? w)]
    [(else (recognizes? (cdr w) (D (car w) p)))])))

(define/memoize (D c p)
  #:order ([p #:eq] [c #:equal])
  (match p
    [(∅) (∅)]
    [(ε _) (∅)]
    [(δ _) (∅)]
    [(token p) (if (p? c) (ε (set c)) (∅))]
    [(∪ p1 p2) (∪ (D c p1) (D c p2))]
    [(δ p) (δ p)]
    [(token _) (if (p? c) (ε (set c)) (∅))]
    [(null? w) (nullable? w)]
    [(else (recognizes? (cdr w) (D (car w) p)))])))

(define/fix (parse-null p)
  #:bottom (set)
  (match p
    [(∅) (set)]
    [(ε S) S]
    [(δ p) (parse-null p)]
    [(token _) (set '())]
    [(null? w) (nullable? w)]
    [(else (recognizes? (cdr w) (D (car w) p)))])))

(define/fix (parse w p)
  (cond [(null? w) (parse-null p)]
        [else (parse (cdr w) (D (car w) p))])))
+ Laziness
+ Memoization
+ Fixed points
Brzozowski’s derivative?
A less common, yet still useful, operation on formal languages is the derivative. The derivative of a language is computable, it may be possible to test whether a string is in that language using derivatives, thanks to the following rule:

For one-character languages:

1. Compute the derivative of the language with respect to each character in the string; and
2. Test whether the resulting language accepts the empty string.

This algorithm is straightforward:

- If it is computable to take successive derivatives of a language, and it is possible to test whether or not one of those derivatives accepts the empty string, then it is possible to use the derivative to test whether a string is in a language.

Example 3.1. For the empty-string language, the derivative is also empty:

\[ D_c \{\text{empty-string}\} = \{\text{empty-string}\} \]

For the empty language, the derivative is empty:

\[ D_c \{\text{empty}\} = \{\}\]

For languages composed of other languages through the familiar formal language operations, it is frequently possible to formulate the derivative of such a language recursively.
1. **Filter:**
   Keep every string starting with \( c \).

2. **Chop:**
   Remove \( c \) from the start of each.
foo frak bar
$D_f$  
foo  frak
Recognition algorithm

- Derive with respect to each character.
- Does the derived language contain $\varepsilon$?
foo ∈ (foo)*
oo ∈ f(f(oo))∗
oo ∈ \( D_f(\text{foo})^* \)
oo ∈ oo(foo)*
oo ∈ oo(foo)*
о ∈ о(фoо)*
$\varepsilon \in (\text{foo})^*$
$\varepsilon \in (\text{foo})^*$
Deriving atomic languages
\( ε \equiv \{ "" "" \} \)

\( c \equiv \{ c \} \)

\( \emptyset \equiv \{ \} \)
(define-struct ∅ {})
(define-struct ε {})
(define-struct token {value})
3.3 Derivatives of formal languages

A less common, yet still useful, operation on formal languages is the derivative. The left derivative of a formal language $L$ with respect to character $c$, denoted $D_c L$, is the remainder of the strings in the set $L$ for which the character $c$ can be removed from the front:

$$D_c L = \{ w : cw \in L \}.$$ 

Example 3.1. The left derivative of the set $\{ \text{foo}, \text{frak}, \text{bar} \}$ with respect to the character $c$ is:

$$D_c \{ \text{foo}, \text{frak}, \text{bar} \} = \{ \text{oo}, \text{rak} \}.$$ 

Where the derivative of a language is computable, it may be possible to test whether a string is in that language using derivatives, thanks to the following rule:

$$w (D_c y L) = cw (L).$$ 

If it is computable to take successive derivatives of a language, and it is possible to test whether or not one of those derivatives accepts the empty string, then it is possible to use the derivative to test whether a string is in a language. The algorithm is straightforward:

1. compute the derivative of the language with respect to each character in the string; and
2. test whether the resulting language accepts the empty string.

3.3.1 Computing the derivative

For languages composed of other languages through the familiar formal language operations, it is frequently possible to formulate the derivative of such a language recursively.

For the empty language, the derivative is empty:

$$D_c \epsilon = \epsilon.$$

For the empty-string language, the derivative is also empty:

$$D_c \{ \} = \epsilon.$$

For one-character languages:

$$D_c \{ c \} = \{ \} \text{ if } c \notin c \text{.}$$

$$D_c \{ c \} = \epsilon \text{ if } c \in c \text{.}$$
A less common, yet still useful, operation on formal languages is the derivative. The left derivative of a formal language $L$ with respect to character $c$, denoted $D_c L$, is the remainder of the strings in the set $L$ for which the character $c$ can be removed from the front:

$$D_c L = \{ w \in L : cw \in L \}.$$

Example 3.1. The left derivative of the set $\{ \text{foo}, \text{frak}, \text{bar} \}$ with respect to the character $c$ is:

$$D_c \{ \text{foo}, \text{frak}, \text{bar} \} = \{ \text{oo}, \text{rak} \}.$$

Where the derivative of a language is computable, it may be possible to a string in that language using derivatives, thanks to the following rule:

$$w \in D_c L \Leftrightarrow cw \in L.$$

If it is computable to take successive derivatives of a language, and it is possible to test whether or not one of those derivatives accepts the empty string, then it is possible to use the derivative to test whether a string is in a language. The algorithm is straightforward:

1) compute the derivative of the language with respect to each character in the string; and
2) test whether the resulting language accepts the empty string.

3.3.1 Computing the derivative

For languages composed of other languages through the familiar formal language operations, it is frequently possible to formulate the derivative of such a language recursively.

For the empty language, the derivative is empty:

$$D_c \emptyset = \emptyset.$$

For the empty-string language, the derivative is also empty:

$$D_c \{ \} = \emptyset.$$

For one-character languages:

$$D_c \{ c \} = \{ c \}$$

if $c \notin L$.

$$D_c \{ c \} = \emptyset$$

if $c \in L$. 

$$D_c \{ c \} = \emptyset$$
(define (D c L)
(define (D c L)
  (match L

...
(define (D c L)
  (match L
    [(∅) (∅)]
    [(token a) (cond [(eqv? a c) (ε)] [else (∅)])]
    [(δ _) (∅)]
    [(∪ L1 L2) (∪ (D c L1) (D c L2))]
    [(★ L1) (∘ (D c L1) L)]
    [(∘ L1 L2) (∪ (∘ (δ L1) (D c L2)) (∘ (D c L1) L2))])))
\[ D_c(\epsilon) = \]
\[ D_c(\epsilon) = \emptyset \]
(define (D c L)
  (match L
    []
    [(ε) (φ)]
    ((∪ L1 L2) (∪ (D c L1) (D c L2)))
    ((★ L1) (∘ (D c L1) L))
    ((∘ L1 L2) (∘ (∘ (δ L1) (D c L2)) (D c L1) L2)))))
\[ D_c \{ c \} = \epsilon \]
\[ D_c \{c\} = \epsilon \]
\[ D_c \{c'\} = \emptyset \text{ if } c \neq c' \]
(define (D c L)
  (match L
    [(token a) (cond [(eqv? a c) (ε)] [else (∅)])])

  [(∪ L1 L2) (∪ (D c L1) (D c L2))]

  [(★ L1) (∘ (D c L1) L)]

  [ (∘ L1 L2) (∘ (∘ (δ L1) (D c L2)) (D c L1) L2))])
Deriving regular languages
\[ L_1 \cup L_2 \]
\[ L_1 \cdot L_2 \]
\[ L_1^* \]
(define-struct u {this that})
(define-struct ◦ {left right})
(define-struct ★ {lang})
\[ D_c(L_1 \cup L_2) \]
\[ D_c(L_1 \cup L_2) = \{ w : cw \in L_1 \cup L_2 \} \]
\[ = \{ w : cw \in L_1 \text{ or } cw \in L_2 \} \]
\[ = \{ w : w \in D_cL_1 \text{ or } w \in D_cL_2 \} \]
\[ = \{ w : w \in D_cL_1 \} \cup \{ w : w \in D_cL_2 \} \]
\[ = D_cL_1 \cup D_cL_2. \]
(define (D c L)
  (match L
[[∪ L1 L2) (∪ (D c L1)
  (D c L2))])}
For a concatenated language:
\[ D_c(L_1 \cdot L_2) = D_c(L_1) \cdot \overline{D_c(L_2)} \]

For a union language:
\[ D_c(L_1 \cup L_2) = \{ w : cw \downarrow L_1 \cup L_2 \} = \{ w : w \downarrow D_c(L_1) \cup \overline{D_c(L_2)} \} = D_c(L_1) \cup D_c(L_2) \]

For an intersection language:
\[ D_c(L_1 \cap L_2) = D_c(L_1) \cap D_c(L_2) \]

For a complemented language:
\[ D_c(L) = \{ w : \overline{cw} \downarrow L \} = \{ w : w \downarrow D_c(L) \} \]

For a difference language:
\[ D_c(L_1 \setminus L_2) = D_c(L_1) \setminus D_c(L_2) \]

For an exponentiated language, where \( n \geq 1 \):
\[ D_c(L^n) = D_c(L) \cdot L^n \]

For a closure language:
\[ D_c(L^\ast) = D_c(L) \cdot L^\ast \]

For a non-empty closure language:
\[ D_c(L^+) = D_c(L) \cdot L^\ast \]
For a concatenated language:
\[ D_c(L_1 \cdot L_2) = (D_c L_1 \cdot L_2) \]

For a union language:
\[ D_c(L_1 \cup L_2) = \{ w : c w \downarrow \cup L_1 \cup L_2 \} = \{ w : c w \downarrow \cup L_1 \text{ or } c w \downarrow \cup L_2 \} = \{ w : w \downarrow D_c L_1 \cup \cup w \downarrow D_c L_2 \} = D_c L_1 \cup D_c L_2. \]

For an intersection language:
\[ D_c(L_1 \cap L_2) = D_c L_1 \cap D_c L_2. \]

For a complemented language:
\[ D_c(L) = \{ w : cw \uparrow \cup L \} = \{ w : not cw \downarrow \cup L \} = \{ w : cw \downarrow \cup L \} = D_c L. \]

For a di\-ference language:
\[ D_c(L_1 - L_2) = D_c L_1 - D_c L_2. \]

For an option language:
\[ D_c(L?) = D_c L. \]

For an exponentiated language, where \( n \geq 1 \):
\[ D_c(L^n) = (D_c L)^n \cdot L^* \]

For a closure language:
\[ D_c(L^\cup) = (D_c L)^\cup \cdot L^\cup. \]

For a non-empty closure language:
\[ D_c(L^+) = (D_c L)^+ \cdot L^+. \]
(define (D c L)
  (match L
    [(∪ L1 L2) (∪ (D c L1) (D c L2))]
    [(★ L1) (★ (D c L1) (★ L1))]
    [(else (D c L))])))
Concatenation?
Needs nullability operator
\[\delta(L) = \epsilon \text{ if } \epsilon \in L\]
\[\delta(L) = \emptyset \text{ if } \epsilon \notin L\]
(define-struct δ {lang})
\( D_c(\delta(L)) = \emptyset \)
(define (D c L)
  (match L

    [(\(\emptyset\)) (\(\emptyset\))]

    [((\(\delta\) _)) (\(\emptyset\))])

  (\(\cup\) L1 L2 (\(\cup\) (D c L1) (D c L2)))

  (\(\ast\) L1 (\(\ast\) (D c L1) L))

  ((D c L1) L2))}
For a concatenated language:

\[ D_c(L_1 \cdot L_2) = (D_c L_1 \cdot L_2) \cup (\delta(L_1) \cdot D_c L_2). \]

For a union language:

\[ D_c(L_1 \cup L_2) = \{ w : cw \in L_1 \cup L_2 \} = \{ w : w \in D_c L_1 \text{ or } w \in D_c L_2 \} = D_c L_1 \cup D_c L_2. \]

For an intersection language:

\[ D_c(L_1 \cap L_2) = D_c L_1 \cap D_c L_2. \]

For a complemented language:

\[ D_c L = \{ w : cw \in L \} = \{ w : w \notin D_c L \} = \{ w : \neg cw \in L \} = \{ w : w \in D_c L \}. \]

For a difference language:

\[ D_c(L_1 - L_2) = D_c L_1 - D_c L_2. \]

For an option language:

\[ D_c(L_1?) = D_c L_1. \]

For an exponentiated language, where \( n \geq 1 \):

\[ D_c(L^n) = (D_c L) \cdot L^{n-1}. \]

For a closure language:

\[ D_c(L^\star) = (D_c L) \cdot L^\star. \]

For a non-empty closure language:

\[ D_c(L^+) = (D_c L) \cdot L^\star. \]
$D_c(L_1 \cdot L_2) = (D_cL_1 \cdot L_2)$
\[ D_c(L_1 \cdot L_2) = (D_cL_1 \cdot L_2) \cup (\delta(L_1) \cdot D_cL_2) \]
(define (D c L)
  (match L
    [(∅ ∅) (ε ∅)]
    [(token a) (cond [(eqv? a c) (ε ∅)] [else (∅ ∅)])]
    [(δ _) (∅ ∅)]
    [(∪ L1 L2) (∪ (D c L1) (D c L2))]
    [(★ L1) (∘ (D c L1) L)]
    [(∘ L1 L2) (∘ (∘ (δ L1) (D c L2)) (∘ (D c L1) L2))])))
(define (D c L)
  (match L
    [(∅) (∅)]
    [(ε) (∅)]
    [(token a) (cond [(eqv? a c) (ε)] [else (∅)])]
    [(δ _) (∅)]
    [(∪ L1 L2) (∪ (D c L1) (D c L2))]
    [(★ L1) (∗ (D c L1) L)]
    [(⊙ L1 L2) (∪ (∗ (δ L1) (D c L2)) (∗ (D c L1) L2))])))
To recognize?
Need nullability
Need
nullability
Need to compute nullability
\[\delta (\epsilon) = \epsilon\]
\[\delta (c) = \emptyset\]
\[\delta (\emptyset) = \emptyset\]
\[ \delta(L_1 \cup L_2) = \delta(L_1) \cup \delta(L_2) \]
\[ \delta(L_1 \cdot L_2) = \delta(L_1) \cdot \delta(L_2) \]
\[ \delta(L_1^\ast) = \epsilon \]
(define (nullable? L)
  (match L
    [(∅) #f]
    [(ε) #t]
    [(token _) #f]
    [(δ L1) (nullable? L1)]
    [(★ _) #t]
    [(∪ L1 L2) (or (nullable? L1) (nullable? L2))]
    [(⊙ L1 L2) (and (nullable? L1) (nullable? L2))])))
(define (recognizes? w L)
  (if (null? w)
      (nullable? L)
      (recognizes? (cdr w) (D (car w) L)))))
How about context-free grammars?
context-free grammars
Recursive regular expressions
Problem

\[ L = L \cdot x \]

\[ U \cup \epsilon \]
Problem

\[ D_x L = D_x L \cdot x \]

\[ U \cup \epsilon \]
(D '× L) =
\[(D \ 'x \ L) = (D \ 'x (\cup (\circ L \ 'x) \ \varepsilon))
= (\cup (\cup (\circ (D \ 'x L) \ 'x)
(\circ (\delta L) (D \ 'x \ 'x)))
(D \ 'x \ \varepsilon))\]
(D 'x L) =
\[(D \ 'x L) = (D \ 'x (\cup (\circ \ 'x L) \ \varepsilon)) \]

\[= (\cup (\cup (\circ (D \ 'x 'x) L) (\circ (\delta \ 'x) (D \ 'x L))) (D \ 'x \varepsilon)) \]
Solution?
(define-struct φ {})
(define-struct ε {})
(define-struct token {value})

(define-struct ∪ {this that})
(define-struct ◦ {left right})
(define-struct ★ {lang})

(define-struct δ {lang})
(define-struct ø {})
(define-struct ε {})
(define-struct token {value})

(define-lazy-struct u {this that})
(define-lazy-struct ⨿ {left right})
(define-lazy-struct ★ {lang})

(define-lazy-struct δ {lang})
Problem

$$\delta(L) = \delta(L) \cdot \delta(x) \cup \delta(\epsilon)$$
Problem

\[ \delta(L) = \delta(L) \cdot \delta(x) \cup \delta(\epsilon) \]
Solution?
Fix it.
(define (nullable? L)
  (match L
    [(∅) #f]
    [(ε) #t]
    [(token _) #f]
    [(δ L1) (nullable? L1)]
    [(★ _) #t]
    [(∪ L1 L2) (or (nullable? L1) (nullable? L2))]
    [(⊙ L1 L2) (and (nullable? L1) (nullable? L2))]))
(define/fix (nullable? L)
    #:bottom #f

    (match L
      [(∅)            #f]
      [(ε)            #t]
      [(token _)      #f]
      [(δ L1)         (nullable? L1)]

      [(★ _)          #t]
      [(∪ L1 L2)      (or (nullable? L1)
                                    (nullable? L2))]
      [(⊙ L1 L2)      (and (nullable? L1)
                                    (nullable? L2))])))
(define/fix (OUT stmt)
    #:bottom ∅
    (¬ (∪ (IN stmt) (GEN stmt)))
    (KILL stmt)))

(define/fix (IN stmt)
    #:bottom ∅
    (apply ∪ (map OUT (preds stmt))))
Final problem
Grammar unfolds forever
Solution?
Memoize
(define (D c L)
  (match L
      [(∅)               (∅)]
      [(ε)               (∅)]
      [(token a)         (cond [(eqv? a c) (ε)]
                                 [else             (∅)]))]

  [(δ _)            (∅)]

  [(∪ L1 L2)       (∪ (D c L1)
                         (D c L2))]

  [(★ L1)          (⊙ (D c L1) L)]

  [(⊙ L1 L2)       (∪ (⊙ (δ L1) (D c L2))
                         (⊙ (D c L1) L2))])))
(define/memoize (D c L)
  #:order [([L #:eq] [c #:equal]])]
(match L
  [(∅) (∅)]
  [(ε) (∅)]
  [(token a) (cond [(eqv? a c) (ε)]
                   [else (∅)])]
  [(δ _) (∅)]
  [(∪ L1 L2) (∪ (D c L1)
                 (D c L2))]
  [(★ L1) (⊙ (D c L1) L)]
  [(⊙ L1 L2) (∪ (⊙ (δ L1) (D c L2))
               (⊙ (D c L1) L2))])))
It works!
(for recognition)
What about parsing?
$D_c : \mathbb{L} \rightarrow \mathbb{L}$
$D_c : \mathbb{P}(A, T) \rightarrow \mathbb{P}(A, T)$
\( \mathcal{P}(A, T) = A^* \rightarrow \mathcal{P}(T \times A^*) \)
(define/memoize (D c L)
  #:order [([L #:eq] [c #:equal]))]
(match L
  [(∅) (∅)]
  [(ε) (∅)]
  [(token a) (cond [(eqv? a c) (ε)]
                     [else (∅)])]
  [(δ _) (∅)]
  [(∪ L1 L2) (∪ (D c L1) (D c L2))]
  [(★ L1)    (∗ (D c L1) L)]
  [(⊙ L1 L2) (u (∗ (δ L1) (D c L2)) (∗ (D c L1) L2))])}
(define/memoize (D c L)
  #:order [([L #:eq] [c #:equal]]))
(match L
  [(∅) (∅)]
  [(ε _) (∅)]
  [(token a) (cond [(eqv? a c) (ε (set c))] [else (∅)])]
  [(δ _) (∅)]
  [(∪ L1 L2) (∪ (D c L1) (D c L2))]
  [(★ L1) (∘ (D c L1) L)]
  [(⊙ L1 L2) (∪ (∘ (δ L1) (D c L2)) (∘ (D c L1) L2))]
  [(→ L1 f) (→ (D c L1) f)])
Computing nullability
Computing null parses
6. Derivatives of parser combinators

If we can generalize the derivative into a parse forest with nodes like this. However, it also needs to strip away any null parses that come back. But, how should the derived parser behave? 

Given their similarity, it should not surprise that the derivative of the null parser is also empty:

\[ \text{parse-null } p \]

The derivative of the empty parser is empty:

\[ (\text{empty}) = \text{empty} \]

The colon operator is the head of the input string within itself. Over time, as successive characters of the input string are depleted, supply the null string to the final parser. In other words:

\[ \text{parse-null } p \]

This rule is important to define the Kleene star of a partial parser:

\[ (\text{rep } L) = (\text{alt } (\text{empty}) (\text{rep } L)) \]

It should act as though the character has been consumed, so

\[ (\text{char } a) = \begin{cases} \text{empty} & \text{if } (a \neq c) \\ \text{parse-null } p & \text{if } (a = c) \end{cases} \]

\[ (\text{eps* } T) = \text{empty} \]

\[ (\text{empty}) = \text{empty} \]

\[ (\text{char } a) = \begin{cases} \text{empty} & \text{if } (a \neq c) \\ \text{parse-null } p & \text{if } (a = c) \end{cases} \]

\[ (\text{eps* } T) = \text{empty} \]

The rules are so similar to the derivative for languages that we can modify the implementation of the derivative for languages to arrive at a derivative suitable for parsers:

\[ (\text{rep } L) = (\text{alt } (\text{empty}) (\text{rep } L)) \]

\[ (\text{char } a) = \begin{cases} \text{empty} & \text{if } (a \neq c) \\ \text{parse-null } p & \text{if } (a = c) \end{cases} \]

\[ (\text{eps* } T) = \text{empty} \]

\[ (\text{empty}) = \text{empty} \]

\[ (\text{char } a) = \begin{cases} \text{empty} & \text{if } (a \neq c) \\ \text{parse-null } p & \text{if } (a = c) \end{cases} \]

\[ (\text{eps* } T) = \text{empty} \]

The rules are so similar to the derivative for languages that we can modify the implementation of the derivative for languages to arrive at a derivative suitable for parsers:

\[ (\text{rep } L) = (\text{alt } (\text{empty}) (\text{rep } L)) \]

\[ (\text{char } a) = \begin{cases} \text{empty} & \text{if } (a \neq c) \\ \text{parse-null } p & \text{if } (a = c) \end{cases} \]

\[ (\text{eps* } T) = \text{empty} \]

\[ (\text{empty}) = \text{empty} \]

\[ (\text{char } a) = \begin{cases} \text{empty} & \text{if } (a \neq c) \\ \text{parse-null } p & \text{if } (a = c) \end{cases} \]

\[ (\text{eps* } T) = \text{empty} \]

The rules are so similar to the derivative for languages that we can modify the implementation of the derivative for languages to arrive at a derivative suitable for parsers:

\[ (\text{rep } L) = (\text{alt } (\text{empty}) (\text{rep } L)) \]

\[ (\text{char } a) = \begin{cases} \text{empty} & \text{if } (a \neq c) \\ \text{parse-null } p & \text{if } (a = c) \end{cases} \]

\[ (\text{eps* } T) = \text{empty} \]

\[ (\text{empty}) = \text{empty} \]

\[ (\text{char } a) = \begin{cases} \text{empty} & \text{if } (a \neq c) \\ \text{parse-null } p & \text{if } (a = c) \end{cases} \]

\[ (\text{eps* } T) = \text{empty} \]

The rules are so similar to the derivative for languages that we can modify the implementation of the derivative for languages to arrive at a derivative suitable for parsers:

\[ (\text{rep } L) = (\text{alt } (\text{empty}) (\text{rep } L)) \]

\[ (\text{char } a) = \begin{cases} \text{empty} & \text{if } (a \neq c) \\ \text{parse-null } p & \text{if } (a = c) \end{cases} \]

\[ (\text{eps* } T) = \text{empty} \]

\[ (\text{empty}) = \text{empty} \]

\[ (\text{char } a) = \begin{cases} \text{empty} & \text{if } (a \neq c) \\ \text{parse-null } p & \text{if } (a = c) \end{cases} \]

\[ (\text{eps* } T) = \text{empty} \]
(define/fix (parse-ε p)
   #:bottom (set)
   (match p
      [(ε S) S]
      [(∅) (set)]
      [(δ p) (parse-ε p)]
      [(token _) (set)]
      [(★ _) (set '())]
      [(∪ p1 p2) (set-union (parse-ε p1) (parse-ε p2))]
      [(⊙ p1 p2) (for*/set ([t1 (parse-ε p1)] [t2 (parse-ε p2)]) (cons t1 t2))]
      [(→ p1 f) (for/set ([t (parse-ε p1)]) (f t))])))
(define (recognizes? w L)
  (if (null? w)
      (nullable? L)
      (recognizes? (cdr w) (D (car w) L)))))
(define (parse w L)
  (if (null? w)
      (parse-ε L)
      (parse (cdr w) (D (car w) L)))))
Demo
\[
\begin{align*}
\epsilon & \equiv \lambda w.\{(\epsilon, w)\} & \mathbb{P}(A, T) &= A^* \rightarrow \mathcal{P}(T \times A^*) \\
p \in \mathbb{P}(A, T') & & \emptyset & \equiv \lambda w.\{} & \mathbb{P}(T) \equiv \lambda w.\{} \\
[p](w) &= \{t : (t, \epsilon) \in p(w)\} & \mathbb{P}(A, T) &= A^* \rightarrow \mathcal{P}(T) \\
f \in X \rightarrow Y & & w & \equiv \lambda w'.\left\{\begin{array}{ll}
\{(w, ww'')\} & w' = w w'' \\
\emptyset & \text{otherwise.}
\end{array}\right.
\\
p \in \mathbb{P}(A, X) & & D_c : \mathbb{L} \rightarrow \mathbb{L} & D_c : \mathbb{P}(A, T) \rightarrow \mathbb{P}(A, T) \\
p \rightarrow f \in \mathbb{P}(A, Y) & & D_c : \mathbb{P}(A, T) \rightarrow \mathbb{P}(A, T) & \mathbb{P}(A, X) \\
p \rightarrow f = \lambda w.\{(f(x), w') : (x, w') \in p(w)\} & & \mathbb{P}(A, X) \\
D_c : \mathbb{P}(A, T) \rightarrow \mathbb{P}(A, T) & & p \cup q \in \mathbb{P}(A, X) \\
D_c(p) = \lambda w.\mathbb{P}(cw) - (\mathbb{P}(\epsilon)(w) \times \{cw\}) & & p \cup q = \lambda w.\mathbb{P}(w) \cup q(w) \\
p(cw) = D_c(p)(w) \cup (\mathbb{P}(\epsilon)(w) \times \{cw\}) & & D_c(p \cup q) = D_c(p) \cup D_c(q) \\
D_c(p \cdot q) = \begin{cases} 
D_c(p) \cdot q & \text{if } \epsilon \notin \mathcal{L}(p) \\
D_c(p) \cdot q \cup (\epsilon \rightarrow \lambda \epsilon.\mathbb{P}(\epsilon)) \cdot D_c(q) & \text{otherwise.}
\end{cases}
\\
p \cdot q = \lambda w.\{(x, y, w''') : (x, w') \in p(w), (y, w''') \in q(w')\} & & D_c(p \rightarrow f) = D_c(p) \rightarrow f
\end{align*}
\]
More in paper

- Theory: From languages to parsers
- Optimization: Grammar compaction
- Discussion: Complexity & performance
Implementation

www.ucombinator.org/projects/parsing/

Reference implementations, test cases, test grammars.
Thanks!
Complexity?
Theory

\[ O(2^{2n} G^2) \]
Compaction
\[ \emptyset \circ p = p \circ \emptyset \Rightarrow \emptyset \]
\[ \emptyset \cup p = p \cup \emptyset \Rightarrow p \]
\[ (\epsilon \downarrow \{t_1\}) \circ p \Rightarrow p \rightarrow \lambda t_2.(t_1, t_2) \]
\[ p \circ (\epsilon \downarrow \{t_2\}) \Rightarrow p \rightarrow \lambda t_1.(t_1, t_2) \]
\[ (\epsilon \downarrow \{t_1, \ldots, t_n\}) \rightarrow f \Rightarrow \epsilon \downarrow \{f(t_1), \ldots, f(t_n)\} \]
\[ ((\epsilon \downarrow \{t_1\}) \circ p) \rightarrow f \Rightarrow p \rightarrow \lambda t_2.(t_1, t_2) \]
\[ (p \rightarrow f) \rightarrow g \Rightarrow p \rightarrow (g \circ f) \]
\[ \emptyset^* \Rightarrow \epsilon \downarrow \{\langle\rangle\} . \]
A glance at run-time behavior on the left-recursive list grammar.

7.1 Example: Growth in the grammar

Considering this complexity, it is remarkable that our example of length worst-case complexity of parsing a grammar of size grammar. The cost of the fixed point is quadratic in the size of the grammar within the parser can grow exponentially with the number of derivatives. (The rule for concatenation is to blame.)

Specifically, the grammar can double in size under the derivative. The implementation is brief. The code is pure. The theory is ele-

A note on repetition

We constructed a parser for Python 3.1. On one-line examples, we can implement these simplification rules in a memoized, constant.

The size of the grammar (and the cost of each derivative) stays compacts after every derivative, then the time to parse the 31-line recursive and memoized, we term it recursive simplification function. When simplification is deeply

Another equational theory shows how to eliminate unnecessary nation nodes—all of these nodes can be discarded.

A graph of the size of the residual Python grammar with respect to each derivative explodes from two seconds to one minute.

If one were to zoom in on image on the right, the node on the bot-

Warning

It is possible to aggressively perform reductions as pieces of

number of derivatives. (The rule for concatenation is to blame.)

gains nothing by allowing the interior of a Kleene star operation of parse trees to return. However, in terms of descriptiveness, one interior parser can parse null, then there are an infinite number of parse trees emerging. Our implementation utilizes the following sim-

interpreting these equations. Thus, Kleene's fixed-point theorem,

languages. Once again, the least fixed point is a sensible way of

equations that mimic the structure of the nullability function for practice, we can replace that last rule by:

to parse null—Kleene star already parses null by definition. So, in

A graph of the size of the residual Python grammar with respect to each derivative explodes from two seconds to one minute.

It assumes that the null parse of each node is initially empty.

The null parser is an annihilator under concatenation.
Practice

\approx O(nG)
Performance

Good enough.
Compaction
\[ p \cdot \emptyset = \emptyset \]
\emptyset \circ p = p \circ \emptyset \Rightarrow \emptyset
\emptyset \cup p = p \cup \emptyset \Rightarrow p
(\varepsilon \downarrow \{t_1\}) \circ p \Rightarrow p \rightarrow \lambda t_2.(t_1, t_2)
p \circ (\varepsilon \downarrow \{t_2\}) \Rightarrow p \rightarrow \lambda t_1.(t_1, t_2)
(\varepsilon \downarrow \{t_1, \ldots, t_n\}) \rightarrow f \Rightarrow \varepsilon \downarrow \{f(t_1), \ldots, f(t_n)\}
((\varepsilon \downarrow \{t_1\}) \circ p) \rightarrow f \Rightarrow p \rightarrow \lambda t_2.(t_1, t_2)
(p \rightarrow f) \rightarrow g \Rightarrow p \rightarrow (g \circ f)
\emptyset^* \Rightarrow \varepsilon \downarrow \{\langle\rangle\}.
or

seq

eps*

(set '())

token 1
or

eps * (set'(((((((((() . 1) . 1) . 1) . 1) . 1) . 1) . 1) . 1) . 1))

seq

L

R

token

1
What is a parser?
\[ \mathcal{P}(A, T) = A^* \rightarrow \mathcal{P}(T \times A^*) \]
\( \mathcal{P}(A, T) = A^* \rightarrow \mathcal{P}(T \times A^*) \)
\[ P(A, T) = A^* \rightarrow \mathcal{P}(T \times A^*) \]
\( \mathcal{P}(A, T) = A^* \rightarrow \mathcal{P}(T \times A^*) \)
\[ \mathcal{P}(A, T) = A^* \rightarrow \mathcal{P}(T) \]
$$[\mathcal{P}](A, T) = A^* \rightarrow \mathcal{P}(T)$$
\[ \mathcal{P}(A, T) = A^* \rightarrow \mathcal{P}(T) \]
\( p \in \mathbb{P}(A, T) \)

\[ \llbracket p \rrbracket(w) = \{ t : (t, \epsilon) \in p(w) \} \]
Context-free parsers
\( w \equiv \lambda w'. \begin{cases} (w, w'') \\ \emptyset \end{cases} \quad w' = ww'' \text{ otherwise.} \)
\[ \epsilon \equiv \lambda w. \{ (\epsilon, w) \} \]
\[ \emptyset \equiv \lambda w. \{ \} \]
\( p \in \mathbb{P}(A, X) \)

\( q \in \mathbb{P}(A, Y) \)

\( p \cdot q \in \mathbb{P}(A, X \times Y) \)
\[ p \cdot q = \lambda w.\{((x, y), w''): (x, w') \in p(w), (y, w'') \in q(w')\} \]
\[ p \cdot q = \lambda w.\{((x, y), w'') : (x, w') \in p(w), (y, w'') \in q(w')\} \]
\[ p \cdot q = \lambda w.\{((x, y), w'') : (x, w') \in p(w), (y, w'') \in q(w')\} \]
\[ p \cdot q = \lambda w.\{((x, y), w'') : (x, w') \in p(w), (y, w'') \in q(w')\} \]
\[ p \cdot q = \lambda w.\{((x, y), w'') : (x, w') \in p(w), (y, w'') \in q(w')\} \]
\[ p \cdot q = \lambda w.\{((x, y), w'') : (x, w') \in p(w), (y, w'') \in q(w')\} \]
$p \in \mathbb{P}(A, X)$

$q \in \mathbb{P}(A, X)$

$p \cup q \in \mathbb{P}(A, X)$

$p \cup q = \lambda w. p(w) \cup q(w)$
\( f \in X \rightarrow Y \)
\( p \in \mathbb{P}(A, X) \)
\( p \rightarrow f \in \mathbb{P}(A, Y) \)
\( p \rightarrow f = \lambda w.\{(f(x), w') : (x, w') \in p(w)\} \)
Defining the derivative
\( D_c : \mathbb{L} \rightarrow \mathbb{L} \)
\[ D_c : \mathbb{P}(A, T) \rightarrow \mathbb{P}(A, T) \]
\[ D_c(p) = \lambda w. p(cw) - ([p](\epsilon) \times \{cw\}) \]
\[ D_c(p) = \lambda w. p(cw) - ([p](\epsilon) \times \{cw\}) \]
\[ p(cw) = D_c(p)(w) \cup ([p](\epsilon) \times \{cw\}) \]
\[ [p](cw) = [D_c(p)](w) \]
Calculating the derivative
\[ D_c(c) = \epsilon \rightarrow \lambda \epsilon.c \]

\[ D_c(c') = \emptyset \text{ if } c \neq c' \]
\[ D_c(p \cup q) = D_c(p) \cup D_c(q) \]
\[ D_c(p \to f) = D_c(p) \to f \]
\[ D_c(p \cdot q) = \begin{cases} 
D_c(p) \cdot q & \text{if } \epsilon \not\in \mathcal{L}(p) \\
D_c(p) \cdot q \cup (\epsilon \rightarrow \lambda \epsilon. [p](\epsilon)) \cdot D_c(q) & \text{otherwise.}
\end{cases} \]
Further reading

• Brzozowski. JACM 1964.
• Owens, Reppy, Turon. JFP 2010.
• Danielsson. ICFP 2010.