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
“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.
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 ∘  {left right})  
(define-struct ★  {lang})  

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

(define (nullable? L)  
  (match L  
    [(∅)           #f]  
    [(ε)           #t]  
    [(token _)     #f]  
    [(★ _)         #t]  
    [(δ L1)        (nullable? L1)]  
    [(u L1 L2)     (or (nullable? L1) (nullable? L2))]  
    [(⊙ 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/memoize (D c p)
#:order ([p #:eq] [c #:equal])
(match p
[(∅) (∅)]
[(ε _) (∅)]
[(δ _) (∅)]
[(token a) (if (eqv? a c) (∅) (∅))]
[(∪ p1 p2) (∪ (D c p1) (D c p2))]
[(★ p1) (∪ (D c p1) p)]
[(→ p1 f) (→ (D c p1) f)]
[(⊙ p1 p2) (∪ (D c p1) p)]
[(⊙ p1 p2) (∪ (D c p1) p)]]
)

(define/fix (parse-null p)
#:bottom (set)
(match p
[(ε S) S]
[(∅) (set)]
[(δ p) (parse-null p)]
[(token _) (set)]
[(★ _) (set '())]
[(∪ p1 p2) (set-union (parse-null p1) (parse-null p2))]
[(⊙ p1 p2) (for*/set ([t1 (parse-null p1)]
[t2 (parse-null p2)])
(cons t1 t2))]
[(→ p1 f) (for/set ([t (parse-null p1)]
(f t)))]
)

(define (recognizes? w p)
(cond [(null? w) (nullable? p)]
[else (recognizes? (cdr w) (D (car w) p))]))

(define (parse w p)
(cond [(null? w) (parse-null p)]
[else (parse (cdr w) (D (car w) p))]))
+ Laziness
+ Memoization
+ Fixed points
Brzozowski’s derivative?
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 a string in that language using derivatives, thanks to the following rule:

\[
w D_c y \in L \) implies \( 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:

\[
D_c \varepsilon = \varepsilon.
\]

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

\[
D_c \{ \} = \varepsilon.
\]

For one-character languages:

\[
D_c \{ c \} = \{ c \}
\]

\[
D_c \{ c \} = \varepsilon
\]

if \( c \neq c = c \).

34
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 $\varepsilon$?
Deriving atomic languages
(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:

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

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:

$$w D_c y L \Rightarrow cw L.$$ 

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:

$$D_c \varepsilon = \varepsilon.$$ 

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

$$D_c \{ \} = \varepsilon.$$ 

For one-character languages:

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

$$D_c \{ c \} = \varepsilon$$ if $c \neq c$. 

$$D_c \emptyset = =$$
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 a string in that language using derivatives, thanks to the following rule:

$$w D_c y L) cw \in L.$$

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:

$$D_c \varepsilon = \varepsilon.$$

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

$$D_c \{\} = \varepsilon.$$

For one-character languages:

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

if $c \neq c \varepsilon$.

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

\[ D_c(L_1 \cdot L_2) = (D_c L_1 \cdot L_2) \triangle (L_1 \cdot D_c L_2). \]

For a union language:

\[ D_c(L_1 \triangle L_2) = \{ w : c w \downarrow \triangle L_1 \triangle L_2 \} = \{ w : c w \downarrow \triangle D_c L_1 \triangle L_2 \} = \{ w : w \downarrow \triangle D_c L_1 \triangle D_c L_2 \}. \]

For an intersection language:

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

For a complemented language:

\[ D_c L = \{ w : c w \uparrow \triangle L \} = \{ w : not c w \downarrow \triangle L \} = \{ w : c w \downarrow \triangle D_c L \} = D_c L \triangle\{ w : w \downarrow \triangle D_c L \}. \]

For a difference language:

\[ D_c(L_1 \triangleleft L_2) = D_c L_1 \triangle D_c L_2. \]

For an option language:

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

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

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

For a closure language:

\[ D_c(L \triangle\{ \}) = (D_c L) \cdot L \triangle\{ \}. \]

For a non-empty closure language:

\[ D_c(L^+ \triangle\{ \}) = (D_c L) \cdot L^\triangle\{ \}. \]

\[ D_c(L^*) = \]
$D_c(L^*) = (D_cL) \cdot L^*$
(define (D c L)
  (match L
    [(∪ L1 L2) (∪ (D c L1) (D c L2))]
    [(★ L1) (∘ (D c L1) (★ L1))]
    [(token a) (cond [(eqv? a c) (ε)] [else (∅)])]
    [(∅) (∅)]))
Concatenation?
Needs nullability operator
Nullability

\[ \delta(L) = \epsilon \text{ if } \epsilon \in L \]
\[ \delta(L) = \emptyset \text{ if } \epsilon \not\in L \]
(define-struct δ {lang})
(define (D c L)
  (match L

    [(∪ L1 L2) (∪ (D c L1) (D c L2))]
    [(★ L1) (∘ (D c L1) L)]
    [(δ _) (Ø)]])
\[ D_c(L_1 \cdot L_2) = \]
\[ D_c(L_1 \cdot L_2) = (D_cL_1 \cdot L_2) \]
\[ D_c(L_1 \cdot L_2) = (D_cL_1 \cdot L_2) \cup (\delta(L_1) \cdot D_cL_2) \]
(define (D c L)
  (match L
    [(∪ L1 L2) (∪ (⊙ (δ L1) (D c L2)) (⊙ (D c L1) L2))])
    [(∅) (∅)]
    [(ε) (∅)]
    [(token a) (eqv? a c) ε]
    [else (∅)]
  ))
(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 to *compute* nullability
\[
\delta(\epsilon) = \epsilon \\
\delta(c) = \emptyset \\
\delta(\emptyset) = \emptyset
\]
\[ \delta(L_1 \cup L_2) = \delta(L_1) \cup \delta(L_2) \]
\[ \delta(L_1 \cdot L_2) = \delta(L_1) \cdot \delta(L_2) \]
\[ \delta(L_1^*) = \epsilon \]
(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?
Recursive regular expressions.
Problem

\[ L = L \cdot x \]

\[ U \epsilon \]
Problem

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

\[ U \epsilon \]
\((D \times L) = \)
\[(D \, 'x \, L) = (D \, 'x \, (u \, (\circ \, L \, 'x) \, \varepsilon))\]

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

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

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

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

(define-lazy-struct δ {lang})
Problem

\[ \delta(L) = \delta(L) \cdot \delta(x) \]
\[ \cup \delta(\epsilon) \]
\[(L) = (L) \cdot (x) \cdot g_{\lambda_{Y}} \bigcup \bigcup_{e} g_{\varphi} \big( (L) \big) = g_{\varphi} \big( (L) \big) \]
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))))
Grammar unfolds forever
Solution?
Memoize
(define (D c L)
  (match L
    [((∅)                  (∅)]
    [((ε)                  (∅)]
    [((token a)            (cond [(eqv? a c) (ε)]
                                  [else (∅)]])]
    [((δ _)                (∅)]
    [((u L1 L2)            (u (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)]
                         [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 recognizing.)
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)    (∪ (⊙ (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 _) ) (∅)]
   [((u L1 L2) ) (u (D c L1)
                         (D c L2))]
   [((★ L1) ) (★ (D c L1) L)]
   [((⊙ L1 L2) ) (u (⊙ (δ L1) (D c L2))
                         (⊙ (D c L1) L2))]
   [((→ L1 f) ) (→ (D c L1) f)])
A partial parser resembles the derivative of a language:
The partial parser

To arrive at a framework for parsing, we can solve this equation for

If it didn't strip these, then null parses containing

that if the string

A

original parser; that is, if the original parser consumed the alphabet

ter

first, we must consider the question:

But

It is easiest to define the Kleene star of a partial parser

5.5 The repetition combinator

It should act as though the character

Intuitively, the derivative of a parser with respect to the charac-

ters of the input string within itself. Over time, as successive

reduction parser or the empty parser:

The derivative of a single-character parser is either the null

P

It at most parses the empty string:

The derivative of the empty parser is empty:

Given their similarity, it should not surprise that the derivative

•

The question of interest is how to define

code, the

the string is depleted, supply the null string to the final parser. In

has been consumed. To parse, compute successive derivatives of

modify the implementation of the derivative for languages to arrive

•

where

\[
\begin{align*}
\mathbb{[}\emptyset\mathbb{]}(\epsilon) &= \{\} \\
\mathbb{[}\epsilon \downarrow T\mathbb{]}(\epsilon) &= T \\
\mathbb{[}\delta(p)\mathbb{]} &= \mathbb{[}p\mathbb{]}(\epsilon) \\
\mathbb{[}p \cup q\mathbb{]}(\epsilon) &= \mathbb{[}p\mathbb{]}(\epsilon) \cup \mathbb{[}q\mathbb{]}(\epsilon) \\
\mathbb{[}p \circ q\mathbb{]}(\epsilon) &= \mathbb{[}p\mathbb{]}(\epsilon) \times \mathbb{[}q\mathbb{]}(\epsilon) \\
\mathbb{[}p \rightarrow f\mathbb{]}(\epsilon) &= \{f(t_1), \ldots, f(t_n)\} \\
\text{where} \quad \{t_1, \ldots, t_n\} &= \mathbb{[}p\mathbb{]}(\epsilon) \\
\mathbb{[}p^*\mathbb{]}(\epsilon) &= (\mathbb{[}p\mathbb{]}(\epsilon))^* 
\end{align*}
\]
(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
\[\epsilon \equiv \lambda w.\{(\epsilon, w)\} \quad \mathbb{P}(A, T) = A^* \rightarrow \mathcal{P}(T \times A^*) \quad D_c(c) = \epsilon \rightarrow \lambda \epsilon.c \quad D_c(c') = \emptyset \text{ if } c \neq c'\]

\[\emptyset \equiv \lambda w.\{} \quad [\mathbb{P}](A, T) = A^* \rightarrow \mathcal{P}(T)\]

\[|p|(w) = \{t : (t, \epsilon) \in p(w)\}\]

\[f \in X \rightarrow Y \quad w \equiv \lambda w'. \begin{cases} \{(w, w'')\} & w' = ww'' \\ \emptyset & \text{otherwise.} \end{cases}\]

\[p \in \mathbb{P}(A, X) \quad p \rightarrow f \in \mathbb{P}(A, Y) \quad D_c : \mathbb{L} \rightarrow \mathbb{L} \quad D_c : [\mathbb{P}](A, T) \rightarrow [\mathbb{P}](A, T)\]

\[p \rightarrow f = \lambda w.\{((f(x), w') : (x, w') \in p(w)\}\]

\[D_c : \mathbb{P}(A, T) \rightarrow \mathbb{P}(A, T)\]

\[D_c(p) = \lambda w.p(cw) - (|p|(\epsilon) \times \{cw\})\]

\[p(cw) = D_c(p)(w) \cup (|p|(\epsilon) \times \{cw\})\]

\[D_c(p \cup q) = D_c(p) \cup D_c(q)\]

\[D_c(p \cdot q) = \begin{cases} D_c(p) \cdot q & \epsilon \not\in \mathcal{L}(p) \\ D_c(p) \cdot q \cup (\epsilon \rightarrow \lambda \epsilon.|p|(\epsilon)) \cdot D_c(q) & \text{otherwise.} \end{cases}\]

\[p \cdot q = \lambda w.\{((x, y), w'') : (x, w') \in p(w), (y, w'') \in q(w')\}\]
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.
どうもありがとうございます

http://www.ucombinator.org/projects/parsing/
Complexity?
Theory

\[ O(2^{2n} G^2) \]
Compaction
\[ \emptyset \circ p = p \circ \emptyset \Rightarrow \emptyset \]
\[ \emptyset \cup p = p \cup \emptyset \Rightarrow p \]
\[ (\varepsilon \downarrow \{ t_1 \}) \circ p \Rightarrow p \rightarrow \lambda t_2.(t_1, t_2) \]
\[ p \circ (\varepsilon \downarrow \{ t_2 \}) \Rightarrow p \rightarrow \lambda t_1.(t_1, t_2) \]
\[ (\varepsilon \downarrow \{ t_1, \ldots, t_n \}) \rightarrow f \Rightarrow \varepsilon \downarrow \{ f(t_1), \ldots, f(t_n) \} \]
\[ ((\varepsilon \downarrow \{ t_1 \}) \circ p) \rightarrow f \Rightarrow p \rightarrow \lambda t_2.(t_1, t_2) \]
\[ (p \rightarrow f) \rightarrow g \Rightarrow p \rightarrow (g \circ f) \]
\[ \emptyset^* \Rightarrow \varepsilon \downarrow \{ \langle \rangle \} . \]
The grammar for unambiguous parses in the worst case. Thus, the grammar. The cost of the fixed point is quadratic in the size of the residual Python grammar with respect to each derivative. If the algorithm compacts after every derivative, then the time to parse the 31-line example finished at all. That it finished in three minutes is astonishing. Considering this complexity, it is remarkable that our example of length 100 token seq,\ldots,t \rightarrow f\rightarrow g \rightarrow \text{undefined} \rightarrow \text{a (syntactically valid) 31-line input. The culprit? The size of the grammar within the parser can grow exponentially with the number of derivatives. (The rule for concatenation is to blame.)}

8. Compaction

A glance at run-time behavior on the left-recursive list grammar exposes the nature of the problem. The image on the right represents a (syntactically valid) 31-line input. The culprit? The size of the grammar for unambiguous parses in the worst case. Thus, the grammar. The cost of the fixed point is quadratic in the size of the residual Python grammar with respect to each derivative. If the algorithm compacts after every derivative, then the time to parse the 31-line example finished at all. That it finished in three minutes is astonishing. Considering this complexity, it is remarkable that our example of length 100 token seq,\ldots,t \rightarrow f\rightarrow g \rightarrow \text{undefined} \rightarrow \text{a (syntactically valid) 31-line input. The culprit? The size of the grammar within the parser can grow exponentially with the number of derivatives. (The rule for concatenation is to blame.)}

The size of the grammar (and the cost of each derivative) stays constant. The cost model for parsing with derivatives is:

\[
\text{cost of derivative} = n^2 \\text{cost of fixed point at the end.}
\]

The cost of the derivative is proportional to the size of the current parse trees to return. However, in terms of descriptiveness, one gains nothing by allowing the interior of a Kleene star operation to parse null. The cost of repetition can mislead. If the rule for concatenation is to blame.

A note on repetition:

We constructed a parser for Python 3.1. On one-line examples, via interpreting these equations. Thus, Kleene's fixed-point theorem, once again, the least fixed point is a sensible way of define/fix gant. So, how does this perform in practice? In brief, it is awful.

What we have at this point are mutually recursive set constraint equations. Once again, the least fixed point is a sensible way of define/fix gant. So, how does this perform in practice? In brief, it is awful. What we have at this point are mutually recursive set constraint equations. Once again, the least fixed point is a sensible way of define/fix gant. So, how does this perform in practice? In brief, it is awful.

What we have at this point are mutually recursive set constraint equations. Once again, the least fixed point is a sensible way of define/fix gant. So, how does this perform in practice? In brief, it is awful.

A note on repetition:

We constructed a parser for Python 3.1. On one-line examples, via interpreting these equations. Thus, Kleene's fixed-point theorem, once again, the least fixed point is a sensible way of define/fix gant. So, how does this perform in practice? In brief, it is awful.
Practice

\[ \approx O(nG) \]
Performance

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

or

empty

seq
L R
token
1

seq
L R
eps*
(set '())

eps*
(set 1)
or

empty

or

seq

or

seq

eps*

eps*

seq

eps*

eps*

seq

eps*

eps*

seq

eps*

eps*

seq

eps*

eps*

seq

eps*

eps*

seq

eps*

eps*

seq

eps*

eps*

seq

eps*

eps*

seq

eps*

eps*

seq

eps*

eps*

seq

eps*

eps*

seq

eps*

eps*

seq

eps*

eps*

seq

eps*

eps*

seq

eps*

eps*

seq

eps*

eps*

seq

eps*

eps*

seq

eps*

eps*

seq

eps*

eps*

seq

eps*

eps*

seq

eps*

eps*

seq

eps*

eps*

seq

eps*

eps*

seq

eps*

eps*

seq

eps*

eps*

seq

eps*

eps*

seq

eps*

eps*

seq

eps*

eps*

seq

eps*

eps*

seq

eps*

eps*

seq

eps*

eps*

seq

eps*

eps*

seq

eps*

eps*

seq

eps*

eps*

seq

eps*

eps*

seq

eps*

eps*

seq

eps*

eps*

seq

eps*

eps*

seq

eps*

eps*

seq

eps*

eps*

seq

eps*

eps*

seq

eps*

eps*

seq

eps*

eps*

seq

eps*

eps*

seq

eps*

eps*

seq

eps*

eps*

seq

eps*

eps*

seq

eps*

eps*

seq

eps*

eps*

seq

eps*

eps*

seq

eps*

eps*

seq

eps*

eps*

seq

eps*

eps*

seq

eps*

eps*

seq

eps*

eps*

seq

eps*

eps*

seq

eps*

eps*

seq

eps*

eps*

seq

eps*

eps*

seq

eps*

eps*

seq

eps*

eps*

seq

eps*

eps*

seq

eps*

eps*

seq

eps*

eps*

seq

eps*

eps*

seq

eps*

eps*

seq

eps*

eps*

seq

eps*

eps*

seq

eps*

eps*

seq

eps*

eps*

seq

eps*

eps*

seq

eps*

eps*

seq

eps*

eps*

seq

eps*

eps*

seq

eps*

eps*

seq

eps*

eps*

seq

eps*

eps*

seq

eps*

eps*

seq

eps*

eps*

seq

eps*

eps*

seq

eps*

eps*

seq

eps*

eps*

seq

eps*

eps*

seq

eps*

eps*

seq

eps*

eps*

seq

eps*

eps*

seq

eps*

eps*

seq

eps*

eps*

seq

eps*

eps*

seq

eps*

eps*

seq

eps*

eps*

seq

eps*

eps*

seq

eps*

eps*

seq

eps*

eps*

seq

eps*

eps*

seq

eps*

eps*

seq

eps*

eps*

seq

eps*

eps*

seq

eps*

eps*

seq

eps*

eps*

seq

eps*

eps*

seq

eps*

eps*

seq

eps*

eps*

seq

eps*

eps*

seq

eps*

eps*

seq

eps*

eps*

seq

eps*

eps*

seq

eps*

eps*

seq

eps*

eps*

seq

eps*

eps*

seq

eps*

eps*

seq

eps*

eps*

seq

eps*

eps*

seq

eps*

eps*

seq

eps*

eps*

seq

eps*

eps*

seq

eps*
or

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

seq L R
token 1
eps* (set'((((((((1) . 1) . 1) . 1) . 1) . 1) . 1) . 1) . 1))
What is a parser?
\[ \mathbb{P}(A, T) = A^* \rightarrow \mathcal{P}(T \times A^*) \]
\[ \mathbb{P}(A, T) = A^* \rightarrow \mathcal{P}(T \times A^*) \]
\[ \mathcal{P}(A, T) = A^* \rightarrow \mathcal{P}(T \times A^*) \]
\[ \mathbb{P}(A, T) = A^* \rightarrow \mathcal{P}(T \times A^*) \]

- Input string
- Remaining input
- Parse tree
$[\mathcal{P}] (A, T) = A^* \rightarrow \mathcal{P} (T)$
Input string

\[
\mathcal{P}(A, T) = A^* \rightarrow \mathcal{P}(T)
\]
\[
\mathbb{P}(A, T) = A^* \rightarrow \mathcal{P}(T)
\]
\[ p \in \mathbb{P}(A, T) \]
\[ [p](w) = \{ t : (t, \epsilon) \in p(w) \} \]
Context-free parsers
\[ w \equiv \lambda w'. \begin{cases} 
\{(w, w'')\} & w' = ww'' \\
\emptyset & \text{otherwise.} 
\end{cases} \]
\[ \epsilon \equiv \lambda w. \{ (\epsilon, w) \} \]
\[ \emptyset \equiv \lambda w. \{ \} \]
\[ p \in \mathbb{P}(A, X) \]
\[ q \in \mathbb{P}(A, Y) \]
\[ p \cdot q \in \mathbb{P}(A, X \times Y) \]
\( p \cdot q = \lambda w.\{(x, y), w''\) : (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') \}\]
\[ p \cdot q = \lambda w. \{ ((x, y), w'') : (x, w') \in p(w), (y, w'') \in q(w') \}\]

Left overs

Input

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

\[ p(cw) = D_c(p)(w) \cup (\lfloor p \rfloor(\epsilon) \times \{cw\}) \]
\[ \lbrack p \rbrack (cw) = \lbrack D_c(p) \rbrack (w) \]
Calculating the derivative
\[ D_c(c) = \epsilon \rightarrow \lambda \epsilon . c \]
\[ D_c(c') = \emptyset \text{ if } c \neq c' \]
\[ D_c(p \cup q) = D_c(p) \cup D_c(q) \]
\[ D_c(p \rightarrow f) = D_c(p) \rightarrow f \]
\[ D_c(p \cdot q) = \begin{cases} 
D_c(p) \cdot q & \epsilon \not\in \mathcal{L}(p) \\
D_c(p) \cdot q \cup (\epsilon \rightarrow \lambda \epsilon. \lfloor p \rfloor(\epsilon)) \cdot D_c(q) & \text{otherwise.} 
\end{cases} \]
Further reading

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