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 ∪      {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)        (∘ (D c L1) L)]
    [(∘ 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))]
    [(∘ 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)        (∘ (D c L1) L)]
    [(∘ 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))]
    [(∘ 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      {tree-set})
(define-struct token  {value?})
(define-lazy-struct δ {lang})
(define-lazy-struct ∪ {this that})
(define-lazy-struct ∘ {left right})
(define-lazy-struct ★ {lang})
(define-lazy-struct → {lang reduce})

(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))]
    [(★ p1)        (∘ (D c p1) p)]
    [(→ p1 f)      (→ (D c p1) f)]
    [(∘ p1 p2)     (∪ (∘ (δ p1) (D c p2))
                      (∘ (D c p1) p2))]))

(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 (parse w p)
    (cond [(null? w) (parse-null p)]
          [else (parse (cdr w) (D (car w) p))]))
```

+ Laziness

+ Memoization

+ Fixed points

Brzozowski’s derivative?

$$D_c L$$

1. **Filter**:
   Keep every string starting with c.

2. **Chop**:
   Remove c from the start of each.

$$D_{\mathtt{f}}$$

`foo` `frak` `bar`

$$D_{\texttt{f}}$$

`foo` `frak`

$$D_{\mathtt{f}}$$

oo rak

Recognition algorithm

- Derive with respect to each character.
- Does the derived language contain ε?

$$\texttt{foo} \in (\texttt{foo})^*$$

$$\texttt{oo} \in \texttt{f}^{\texttt{(foo)*}}$$

$$\texttt{oo} \in D_{\texttt{f}}(\texttt{foo})^*$$

$$\texttt{oo} \in \texttt{oo(foo)*}$$

$$\texttt{oo} \in \texttt{oo(foo)*}$$

o ∈ o(foo)*

$\varepsilon \in$ `(foo)*`

$$\varepsilon \in (\texttt{foo})^*$$

Deriving atomic languages

$$\epsilon \equiv \{\text{""}\}$$

$$c \equiv \{c\}$$

$$\emptyset \equiv \{\}$$

```
(define-struct ∅     {})
(define-struct ε     {})
(define-struct token {value})
```

$$D_c\emptyset =$$

$$D_c\emptyset = \emptyset$$

```
(define (D c L)
```

```
(define (D c L)
  (match L
```

```
(define (D c L)
  (match L
    [(∅)                 (∅)]
```

$$D_c(\epsilon) =$$

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

```
(define (D c L)
  (match L

    [(ε)              (∅)]
```

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

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

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

```
(define (D c L)
  (match L

    [(token a)      (cond [(eqv? a c) (ε)]
                          [else       (∅)])]
```

Deriving regular languages

$$L_1 \cup L_2$$

$$L_1 \cdot L_2$$

$$L_1^\star$$

```
(define-struct ∪ {this that})
(define-struct ∘ {left right})
(define-struct ⋆ {lang})
```

$$D_c(L_1 \cup L_2)$$

$$
\begin{aligned}
D_c(L_1 \cup L_2) &= \{w : cw \in L_1 \cup L_2\} \\
&= \{w : cw \in L_1 \text{ or } cw \in L_2\} \\
&= \{w : w \in D_c L_1 \text{ or } w \in D_c L_2\} \\
&= \{w : w \in D_c L_1\} \cup \{ w : w \in D_c L_2\} \\
&= D_c L_1 \cup D_c L_2.
\end{aligned}
$$

```
(define (D c L)
  (match L


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

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

$$D_c(L^\star) = (D_c L) \cdot L^\star$$

```
(define (D c L)
  (match L




    [(★ L1)          (∘ (D c L1) (★ L1))]
```

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

    [(δ _)              (∅)]
```

$$D_c(L_1 \cdot L_2) =$$

$$D_c(L_1 \cdot L_2) = (D_c L_1 \cdot L_2)$$

$$D_c(L_1 \cdot L_2) = (D_c L_1 \cdot L_2) \cup (\delta(L_1) \cdot D_c L_2)$$

```
(define (D c L)
  (match 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

$$\delta(\epsilon) = \epsilon$$

$$\delta(c) = \emptyset$$

$$\delta(\emptyset) = \emptyset$$

$$\delta(L_1 \cup L_2) = \delta(L_1) \cup \delta(L_2)$$
$$\delta(L_1 \cdot L_2) = \delta(L_1) \cdot \delta(L_2)$$
$$\delta(L_1^\star) = \epsilon$$

```
(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))]))
```

```scheme
(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

$$L = L \cdot \mathtt{x}$$

$$\cup \ \epsilon$$

Problem

$$D_{\mathtt{x}}L = D_{\mathtt{x}}L \cdot \mathtt{x}$$
$$\cup \ \epsilon$$

(D 'x L) =

```
(D 'x L) = (D 'x (∪ (∘ L 'x)
                    ε))

         = (∪ (∪ (∘ (D 'x L) 'x)
                 (∘ (δ L) (D 'x 'x)))
              (D 'x ε))
```

(D 'x L) =

```
(D 'x L) = (D 'x (∪ (∘ 'x L)
                      ε))

         = (∪ (∪ (∘ (D 'x 'x) L)
                 (∘ (δ 'x) (D 'x L)))
              (D 'x ε))
```

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

$$\delta(L) = \delta(L) \cdot \delta(\mathtt{x})$$

$$\cup \;\; \delta(\epsilon)$$

Problem

$$\delta(L) = \delta(L) \cdot \delta(\mathtt{x}) \cup \delta(\epsilon)$$

Solution?

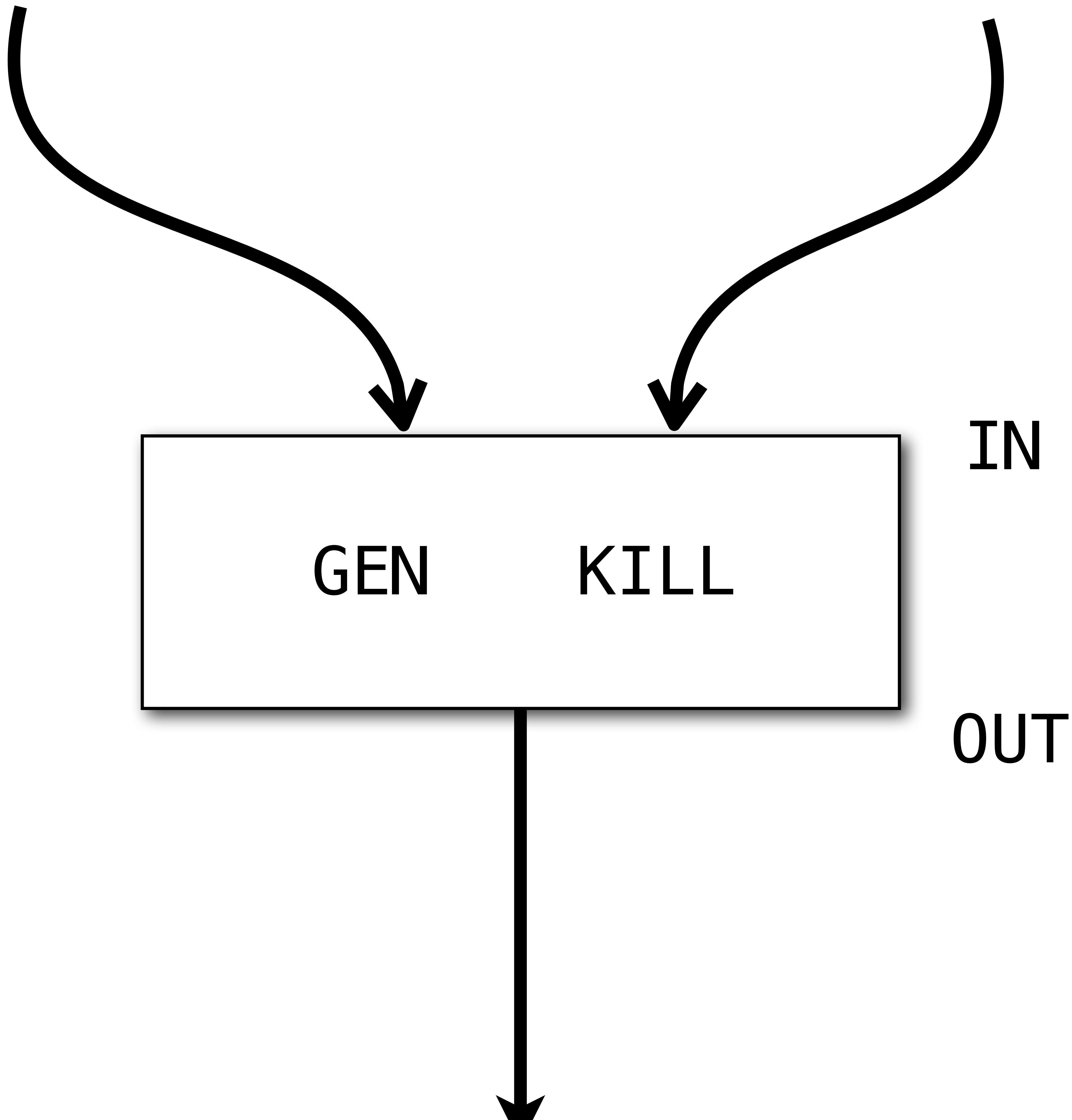
Fix it.

```
(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))]))
```


IN
GEN
KILL
OUT

```
(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?

$$D_c : \mathbb{L} \to \mathbb{L}$$

$$D_c : \mathbb{P}(A, T) \longrightarrow \mathbb{P}(A, T)$$

$$\mathbb{P}(A, T) = A^* \to \mathcal{P}(T \times A^*)$$

```
(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 _)           (∅)]

    [(∪ 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

$$
\begin{aligned}
\lfloor \emptyset \rfloor(\epsilon) &= \{\} \\
\lfloor \epsilon \downarrow T \rfloor(\epsilon) &= T \\
\lfloor \delta(p) \rfloor &= \lfloor p \rfloor(\epsilon) \\
\lfloor p \cup q \rfloor(\epsilon) &= \lfloor p \rfloor(\epsilon) \cup \lfloor q \rfloor(\epsilon) \\
\lfloor p \circ q \rfloor(\epsilon) &= \lfloor p \rfloor(\epsilon) \times \lfloor q \rfloor(\epsilon) \\
\lfloor p \rightarrow f \rfloor(\epsilon) &= \{f(t_1), \ldots, f(t_n)\} \\
&\quad \text{where } \{t_1, \ldots, t_n\} = \lfloor p \rfloor(\epsilon) \\
\lfloor p^{\star} \rfloor(\epsilon) &= (\lfloor p \rfloor(\epsilon))^*
\end{aligned}
$$

```
(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

$$\epsilon \equiv \lambda w.\{(\epsilon, w)\}$$

$$\mathbb{P}(A,T) = A^* \to \mathcal{P}(T \times A^*)$$

$$D_c(c) = \epsilon \to \lambda\epsilon.c$$

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

$$p \in \mathbb{P}(A,T)$$

$$\emptyset \equiv \lambda w.\{\}$$

$$\lfloor \mathbb{P} \rfloor(A,T) = A^* \to \mathcal{P}(T)$$

$$\lfloor p \rfloor(w) = \{t : (t,\epsilon) \in p(w)\}$$

$$f \in X \to Y$$

$$w \equiv \lambda w'. \begin{cases} \{(w, w'')\} & w' = ww'' \\ \emptyset & \text{otherwise.} \end{cases}$$

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

$$D_c : \mathbb{L} \to \mathbb{L}$$

$$D_c : \lfloor \mathbb{P} \rfloor(A,T) \to \lfloor \mathbb{P} \rfloor(A,T)$$

$$p \to f \in \mathbb{P}(A,Y)$$

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

$$p \to f = \lambda w.\{((f(x), w') : (x, w') \in p(w)\}$$

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

$$D_c : \mathbb{P}(A,T) \to \mathbb{P}(A,T)$$

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

$$p \cup q = \lambda w.p(w) \cup q(w)$$

$$D_c(p) = \lambda w.p(cw) - (\lfloor p \rfloor(\epsilon) \times \{cw\})$$

$$p(cw) = D_c(p)(w) \cup (\lfloor p \rfloor(\epsilon) \times \{cw\})$$

$$D_c(p \cup q) = D_c(p) \cup D_c(q)$$

$$D_c(p \cdot q) = \begin{cases} D_c(p) \cdot q & \epsilon \notin \mathcal{L}(p) \\ D_c(p) \cdot q \cup (\epsilon \to \lambda\epsilon.\lfloor p \rfloor(\epsilon)) \cdot D_c(q) & \text{otherwise.} \end{cases}$$

$$D_c(p \to f) = D_c(p) \to f$$

$$p \cdot q = \lambda w.\{((x,y), w'') : (x, w') \in p(w), (y, w'') \in q(w')\}$$

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.

Thanks!

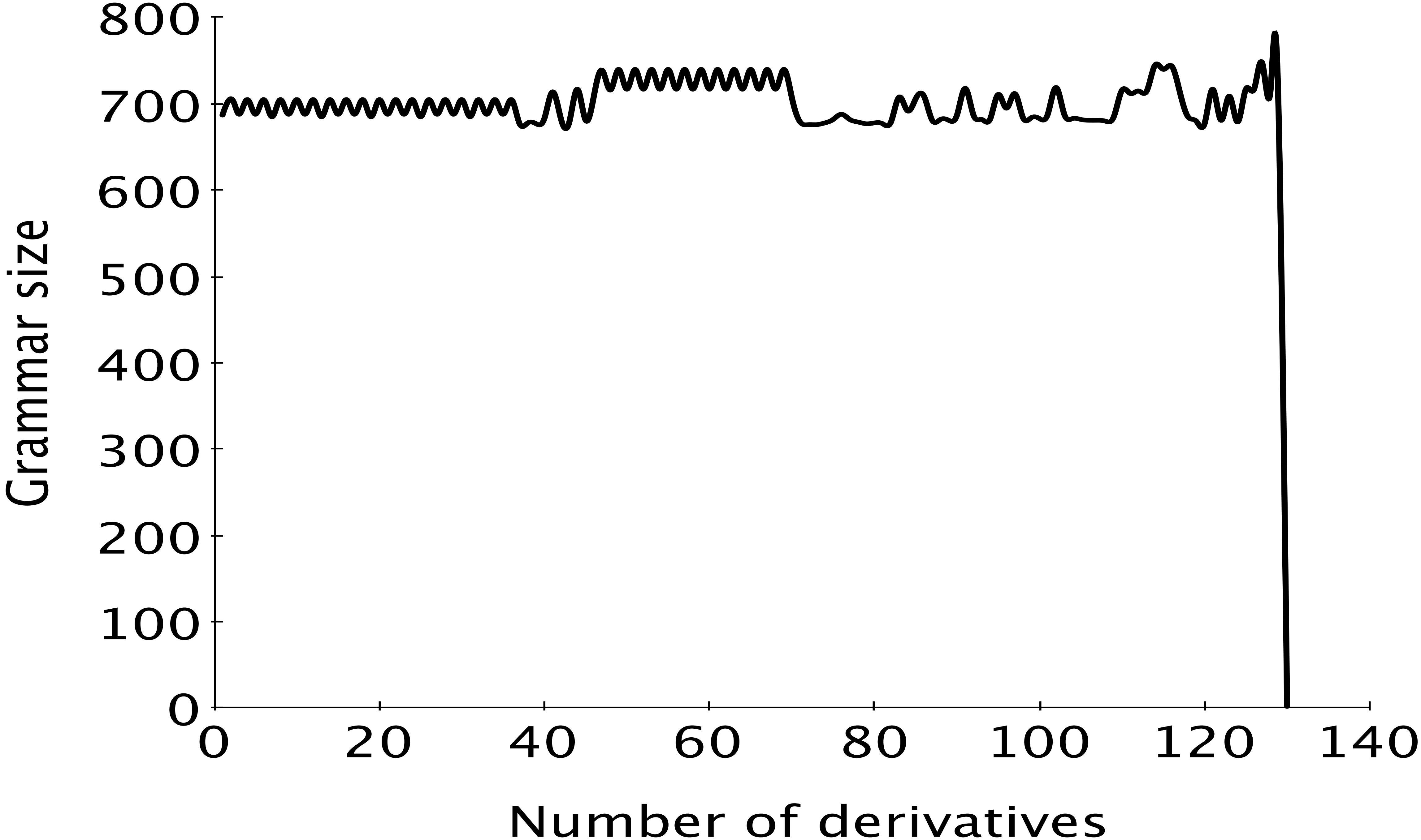
Complexity?

Theory

$$O(2^{2^n}G^2)$$

Compaction

$$
\begin{aligned}
\emptyset \circ p = p \circ \emptyset &\Rightarrow \emptyset \\
\emptyset \cup p = p \cup \emptyset &\Rightarrow p \\
(\epsilon \downarrow \{t_1\}) \circ p &\Rightarrow p \rightarrow \lambda t_2.(t_1, t_2) \\
p \circ (\epsilon \downarrow \{t_2\}) &\Rightarrow p \rightarrow \lambda t_1.(t_1, t_2) \\
(\epsilon \downarrow \{t_1, \ldots, t_n\}) \rightarrow f &\Rightarrow \epsilon \downarrow \{f(t_1), \ldots, f(t_n)\} \\
((\epsilon \downarrow \{t_1\}) \circ p) \rightarrow f &\Rightarrow p \rightarrow \lambda t_2.(t_1, t_2) \\
(p \rightarrow f) \rightarrow g &\Rightarrow p \rightarrow (g \circ f) \\
\emptyset^\star &\Rightarrow \epsilon \downarrow \{\langle\rangle\}\,.
\end{aligned}
$$



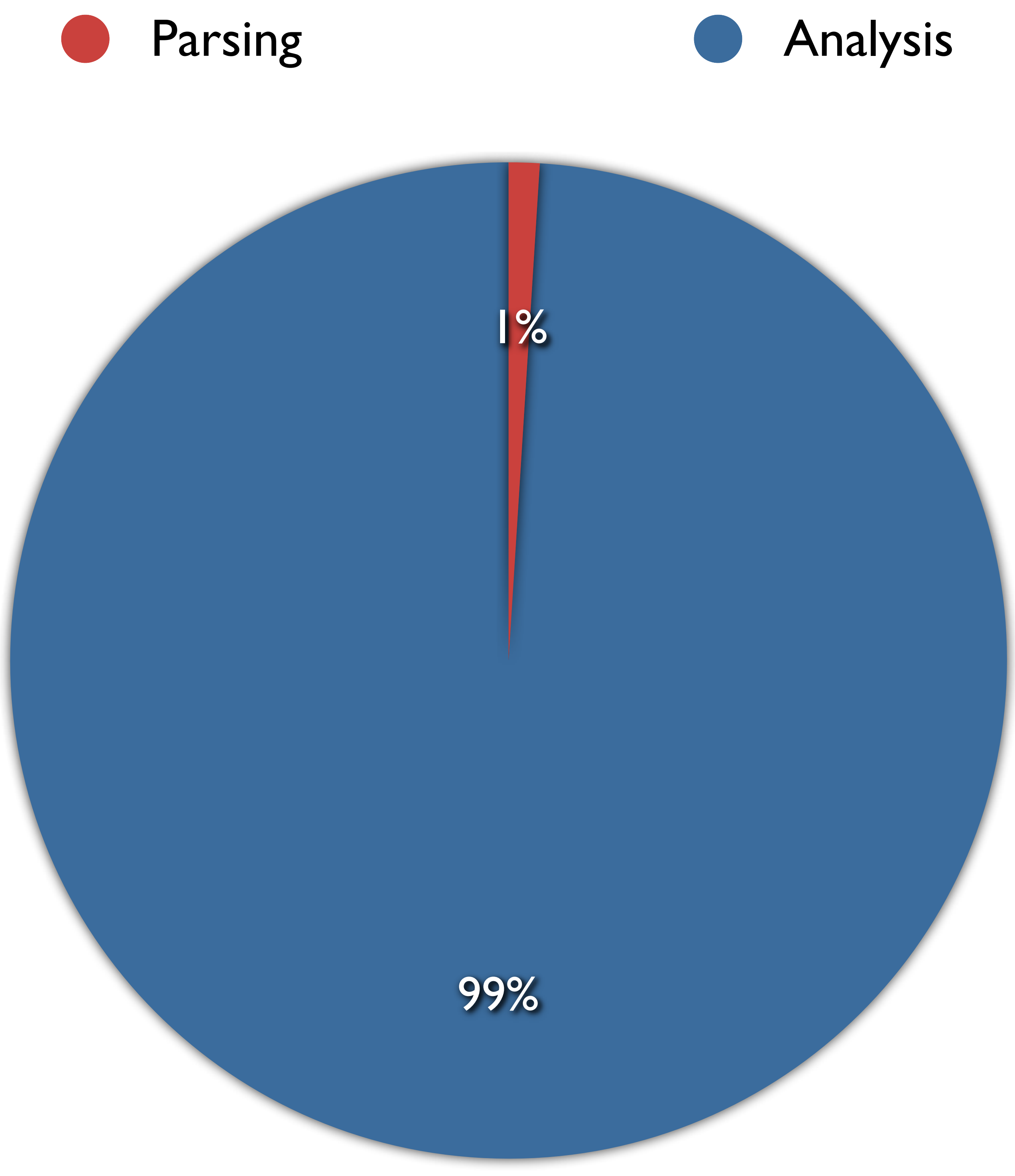
800
700
600
500
400
300
200
100
0
Grammar size
0
20
40
60
80
100
120
140
Number of derivatives

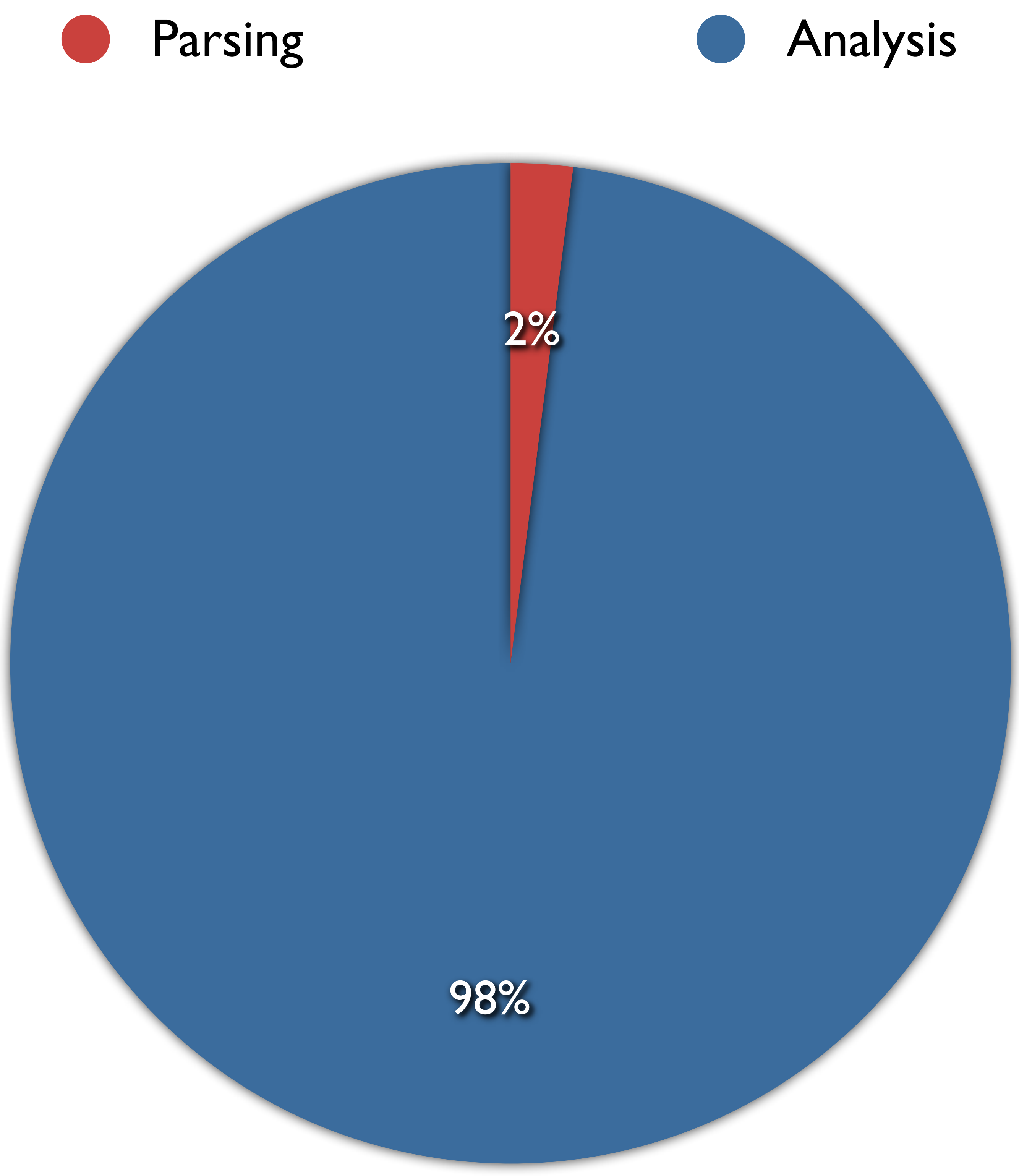
Practice

$$\approx O(nG)$$

Performance

Good enough.

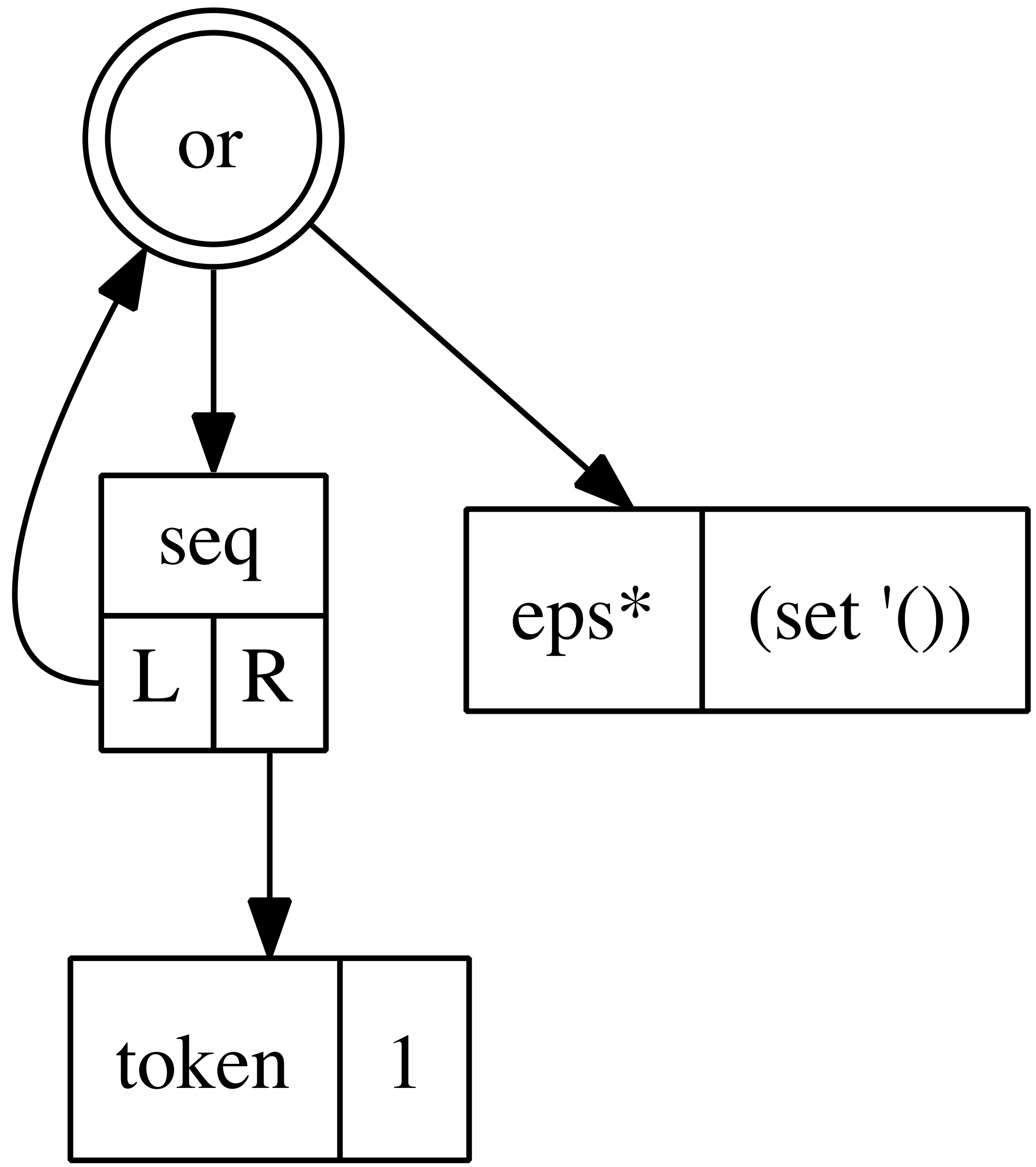

Parsing
Analysis
1%
99%

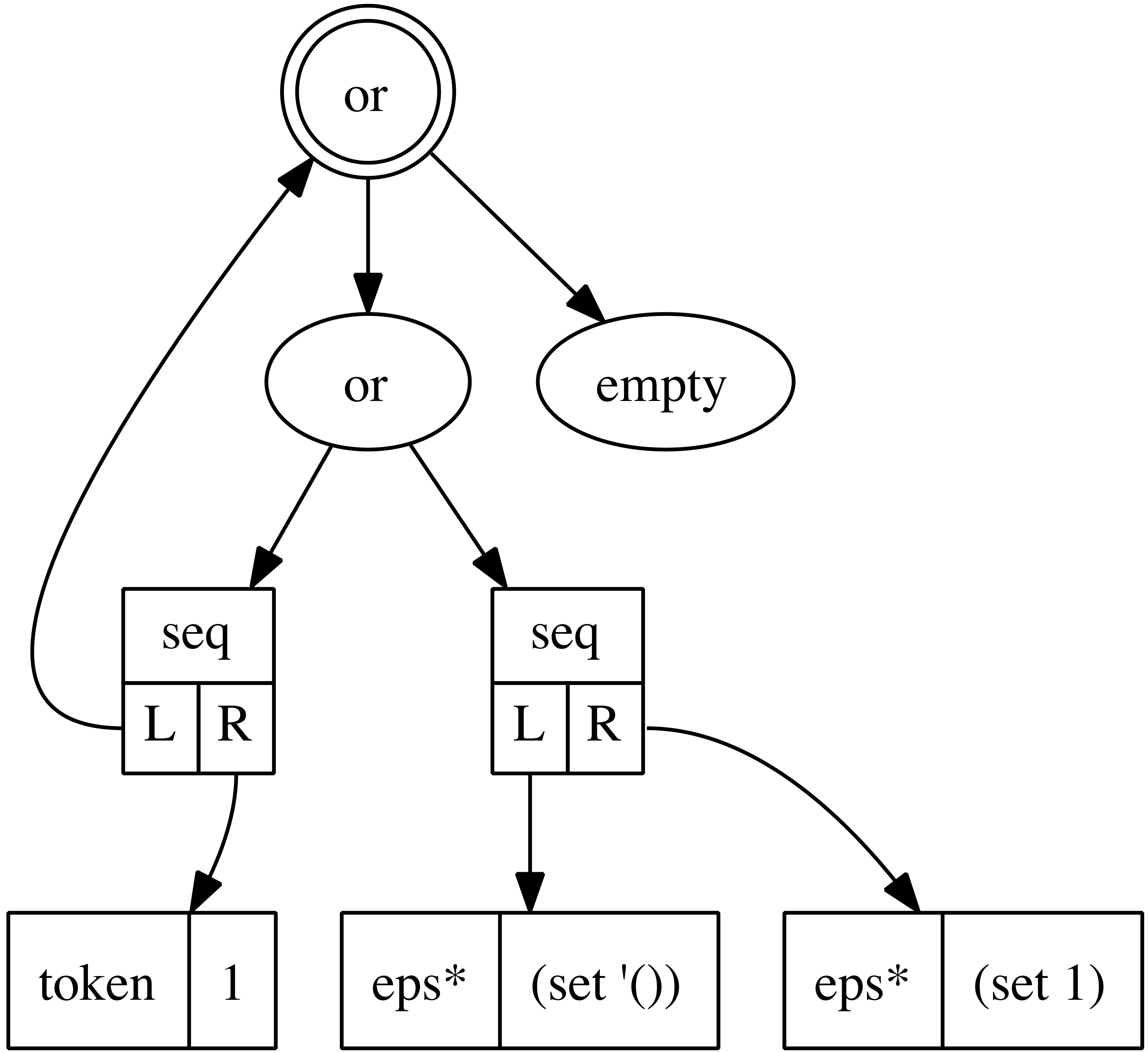

Parsing
Analysis
2%
98%

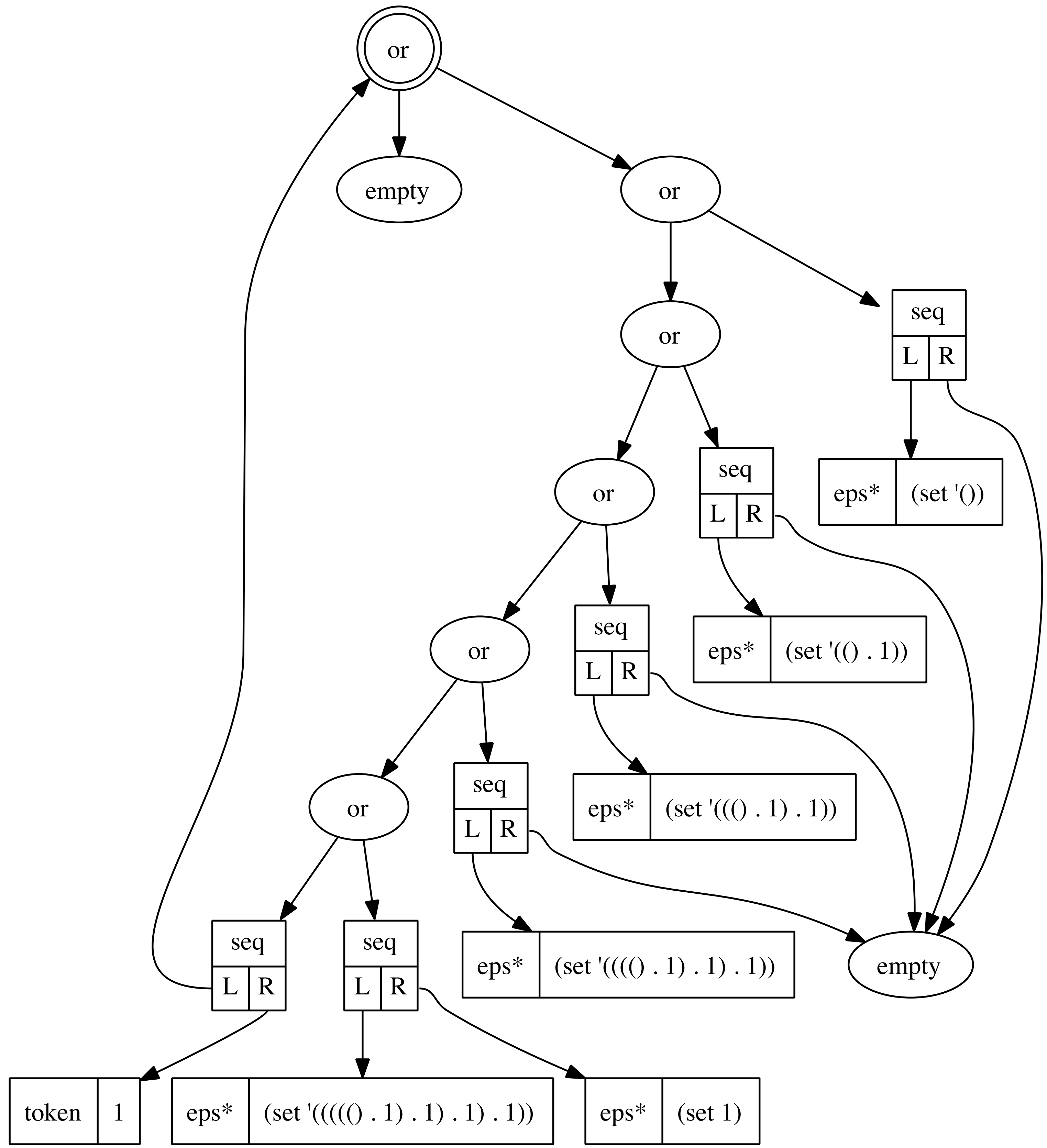
Compaction

$$p \cdot \emptyset = \emptyset$$

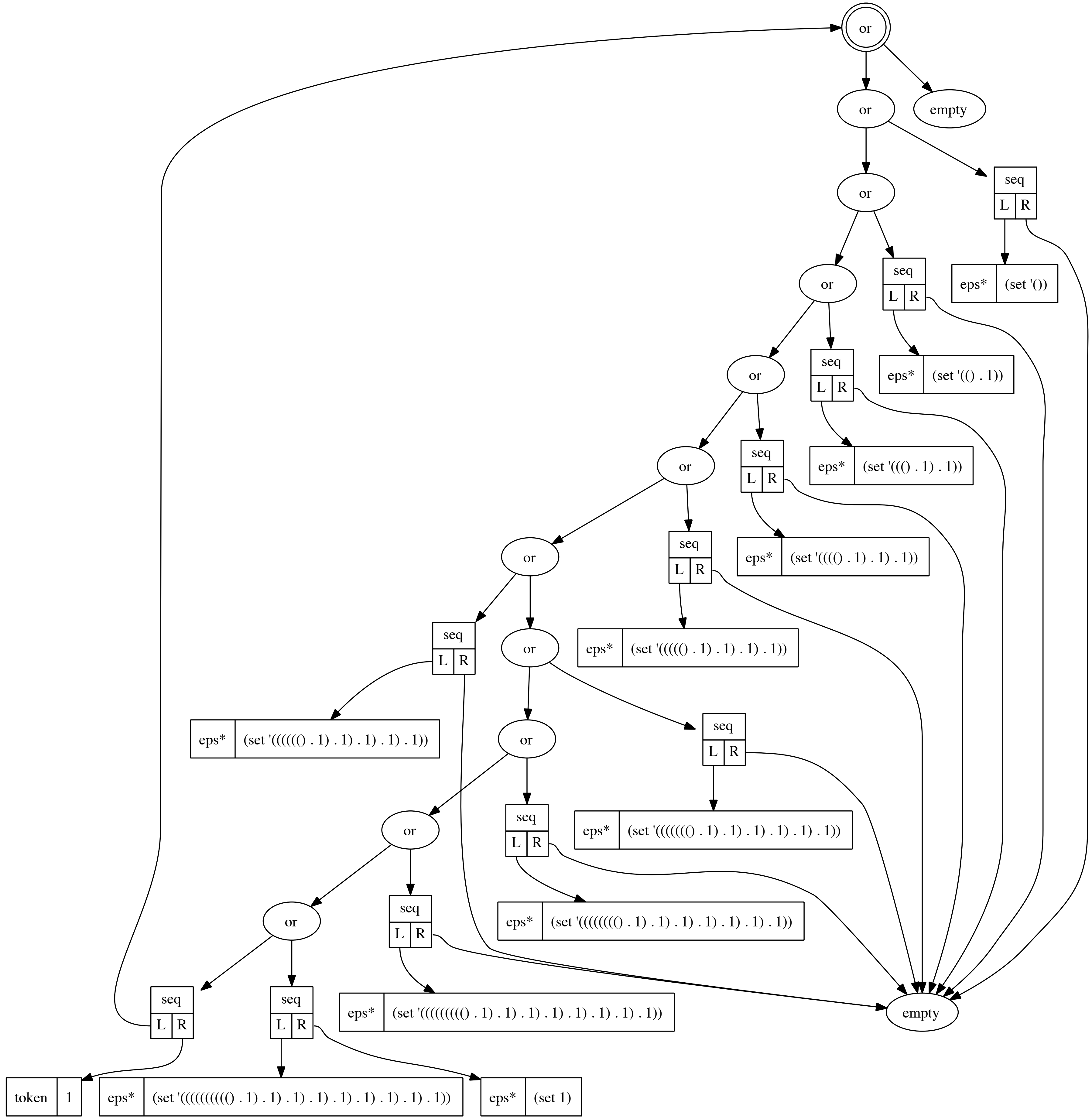
$$
\begin{aligned}
\emptyset \circ p = p \circ \emptyset &\Rightarrow \emptyset \\
\emptyset \cup p = p \cup \emptyset &\Rightarrow p \\
(\epsilon \downarrow \{t_1\}) \circ p &\Rightarrow p \to \lambda t_2.(t_1, t_2) \\
p \circ (\epsilon \downarrow \{t_2\}) &\Rightarrow p \to \lambda t_1.(t_1, t_2) \\
(\epsilon \downarrow \{t_1, \ldots, t_n\}) \to f &\Rightarrow \epsilon \downarrow \{f(t_1), \ldots, f(t_n)\} \\
((\epsilon \downarrow \{t_1\}) \circ p) \to f &\Rightarrow p \to \lambda t_2.(t_1, t_2) \\
(p \to f) \to g &\Rightarrow p \to (g \circ f) \\
\emptyset^\star &\Rightarrow \epsilon \downarrow \{\langle\rangle\}\,.
\end{aligned}
$$


or
seq
L
R
eps*
(set '())
token
1

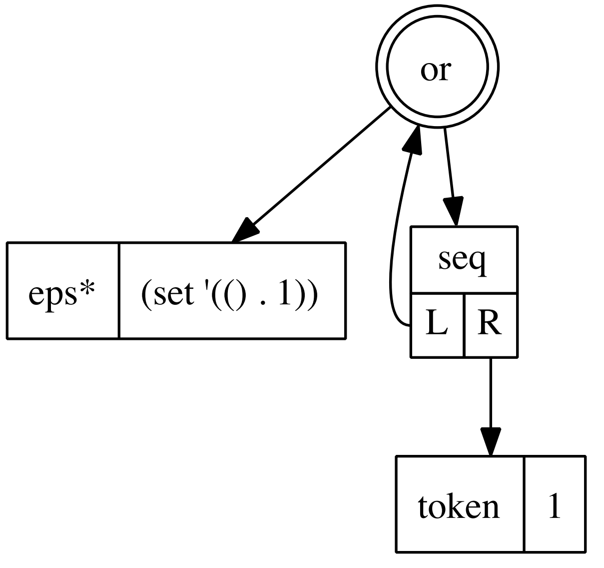

or
or
empty
seq
L
R
seq
L
R
token
1
eps*
(set '())
eps*
(set 1)



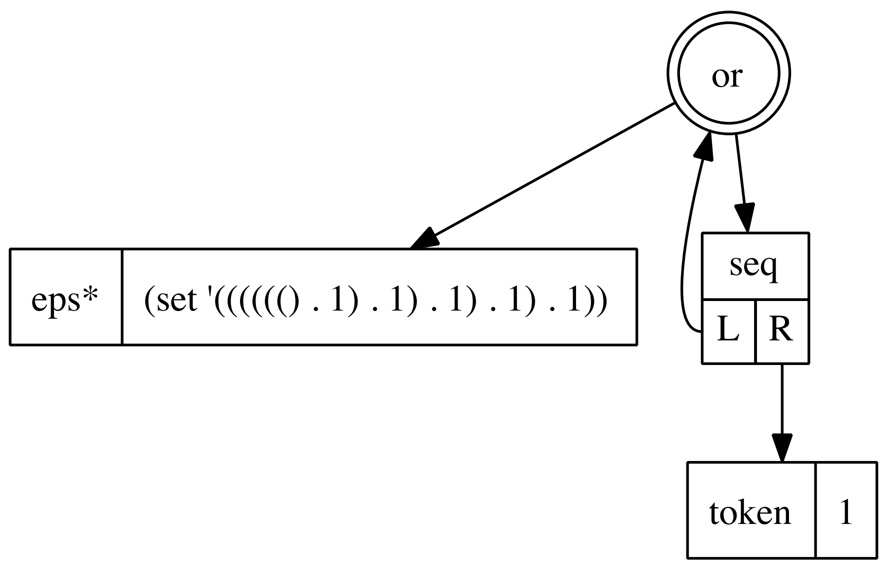
or
empty
or
or
seq
L
R
or
seq
L
R
eps*
(set '())
or
seq
L
R
eps*
(set '(() . 1))
or
seq
L
R
eps*
(set '((() . 1) . 1))
seq
L
R
seq
L
R
eps*
(set '(((() . 1) . 1) . 1))
empty
token
1
eps*
(set '((((() . 1) . 1) . 1) . 1))
eps*
(set 1)

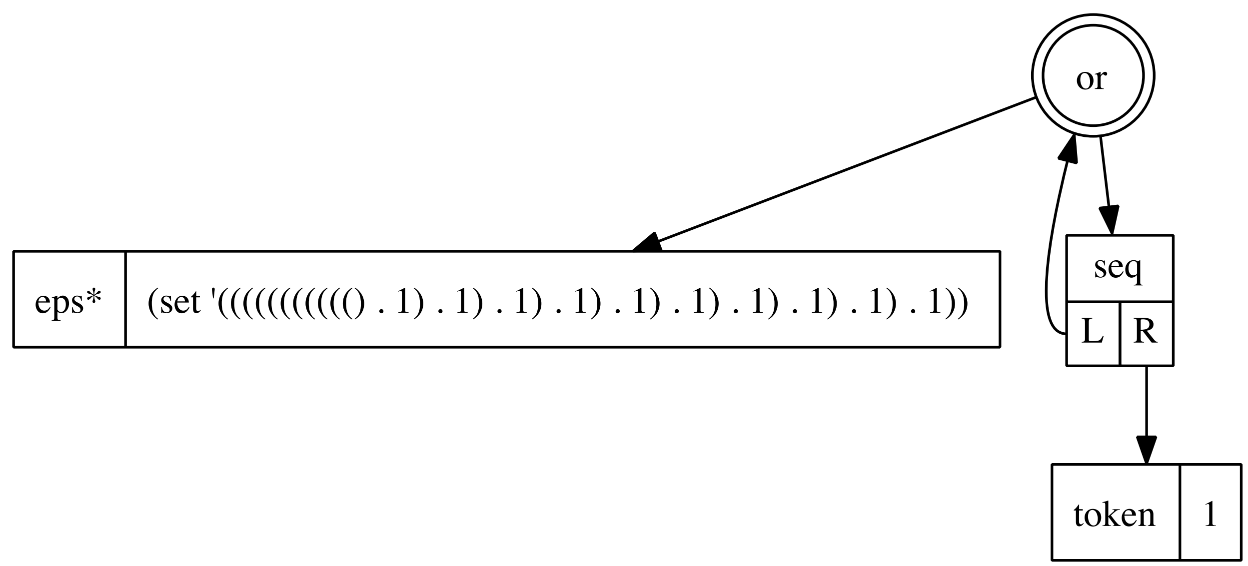


or
empty
or
seq
L
R
eps*
(set '())
or
seq
eps*
(set '(() . 1))
or
seq
eps*
(set '((() . 1) . 1))
or
seq
eps*
(set '(((() . 1) . 1) . 1))
or
seq
eps*
(set '((((() . 1) . 1) . 1) . 1))
or
seq
eps*
(set '(((((() . 1) . 1) . 1) . 1) . 1))
or
seq
eps*
(set '((((((() . 1) . 1) . 1) . 1) . 1) . 1))
or
seq
eps*
(set '(((((((() . 1) . 1) . 1) . 1) . 1) . 1) . 1))
or
seq
eps*
(set '((((((((() . 1) . 1) . 1) . 1) . 1) . 1) . 1) . 1))
or
seq
seq
empty
token
1
eps*
(set '(((((((((() . 1) . 1) . 1) . 1) . 1) . 1) . 1) . 1) . 1))
eps*
(set 1)



or
eps*
(set '(() . 1))
seq
L
R
token
1


or
eps*
(set '((((((() . 1) . 1) . 1) . 1) . 1))
seq
L
R
token
1


or
eps*
(set '((((((((((() . 1) . 1) . 1) . 1) . 1) . 1) . 1) . 1) . 1) . 1))
seq
L
R
token
1

What is a parser?

$$\mathbb{P}(A, T) = A^* \rightarrow \mathcal{P}(T \times A^*)$$

Input string

↓

$$\mathbb{P}(A, T) = A^* \rightarrow \mathcal{P}(T \times A^*)$$

Input string

$$\mathbb{P}(A, T) = A^* \to \mathcal{P}(T \times A^*)$$

Parse tree

Input string

Remaining input

$$\mathbb{P}(A, T) = A^* \to \mathcal{P}(T \times A^*)$$

Parse tree

$$\lfloor \mathbb{P} \rfloor (A, T) = A^* \to \mathcal{P}(T)$$

Input string

↓

$$\lfloor \mathbb{P} \rfloor (A, T) = A^* \rightarrow \mathcal{P}(T)$$

Input string

$$\lfloor \mathbb{P} \rfloor (A, T) = A^* \to \mathcal{P}(T)$$

Parse tree

$$p \in \mathbb{P}(A, T)$$

$$\lfloor p \rfloor (w) = \{t : (t, \epsilon) \in p(w)\}$$

Context-free parsers

$$w \equiv \lambda w'. \begin{cases} \{(w, w'')\} & w' = ww'' \\ \emptyset & \text{otherwise.} \end{cases}$$

$$\epsilon \equiv \lambda w.\{(\epsilon, w)\}$$

$$\emptyset \equiv \lambda w.\{\}$$

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

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

$$p \cdot q \in \mathbb{P}(A, \boldsymbol{X} \times \boldsymbol{Y})$$

$$p \cdot q = \lambda w.\{((x, y), w'') : (x, w') \in p(w), (y, w'') \in q(w')\}$$

$$p \cdot q = \lambda w.\{((x, y), w'') : (x, w') \in p(w), (y, w'') \in q(w')\}$$

Input

$$p \cdot q = \lambda w.\{((x, y), w'') : (x, w') \in p(w), (y, w'') \in q(w')\}$$

Input

First parse

Left overs

$$p \cdot q = \lambda w.\{((x, y), w'') : (x, w') \in p(w), (y, w'') \in q(w')\}$$

Input

First parse

Left overs

$$p \cdot q = \lambda w.\{((x, y), w'') : (x, w') \in p(w), (y, w'') \in q(w')\}$$

Input

First parse

Second parse

Left overs

Remainder

$$p \cdot q = \lambda w.\{((x, y), w'') : (x, w') \in p(w), (y, w'') \in q(w')\}$$

Input

First parse

Second parse

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

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

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

$$p \cup q = \lambda w.p(w) \cup q(w)$$

$$
\begin{aligned}
f &\in X \to Y \\
p &\in \mathbb{P}(A, X) \\
p \to f &\in \mathbb{P}(A, Y) \\
p \to f &= \lambda w. \{((f(x), w') : (x, w') \in p(w)\}
\end{aligned}
$$

Defining the derivative

$$D_c : \mathbb{L} \rightarrow \mathbb{L}$$

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

$$D_c(p) = \lambda w.p(cw) - (\lfloor p \rfloor(\epsilon) \times \{cw\})$$

$$D_c(p) = \lambda w.p(cw) - (\lfloor p \rfloor(\epsilon) \times \{cw\})$$
$$p(cw) = D_c(p)(w) \cup (\lfloor p \rfloor(\epsilon) \times \{cw\})$$

$$\lfloor p \rfloor (cw) = \lfloor D_c(p) \rfloor (w)$$

Calculating the derivative

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

$$D_c(p \cup q) = D_c(p) \cup D_c(q)$$

$$D_c(p \to f) = D_c(p) \to f$$

$$D_c(p \cdot q) = \begin{cases} D_c(p) \cdot q & \epsilon \notin \mathcal{L}(p) \\ D_c(p) \cdot q \cup (\epsilon \rightarrow \lambda\epsilon.\lfloor p \rfloor(\epsilon)) \cdot D_c(q) & \text{otherwise.} \end{cases}$$

Further reading

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