On the Complexity and Performance of Parsing with Derivatives

Michael D. Adams,[*] Celeste Hollenbeck, Matthew Might

[*] Seeking a tenure-track position in the upcoming hiring cycle

Parsing with Derivatives

can be fast

… and it’s not too hard

LR(k) ANTLR Parsec SLR

Yacc Happy LL(k)

CYK Dypgen

Isn’t parsing a solved problem?

Packrat

Early GLR LALR(k)

PEG Bison Elkhound

Compilers

Interpreters

Domain-specific langauges

Syntax highlighting

Natural languages

Command lines

Configuration files

Serialization

Query languages

Network protocols

Apache
(HTTP - RFC 2616)

lighttpd
(HTTP - RFC 2616)

Courier
(IMAP - RFC 3501)

Freeenode IRCD
(IRC - RFC 2812)

*buf++

strtok

```
$ ls /etc

/etc/fstab
/etc/crontab
/etc/hosts
/etc/passwd
/etc/protocols
/etc/services
/etc/shadow
/etc/sudoers
...
```

Yacc	`*buf++`	strtok
Create .y file		
Define %union lval		
Define tokens/types		
Define yacc rules		
Compile yacc grammar	`*buf++`	`strtok`
Create .l lex file		
#include “y.tab.h”		
Define states		
Define lex rules		
Deal with yywrap()		
Call yyparse()		

Parsing with Derivatives

(Might et al. 2011)

$$D_c(L) = \{ w \mid cw \in L \}$$

$$D_o(D_o(D_f(L))) = \{ \quad \varepsilon \quad \}$$

$D_c(\boldsymbol{\varepsilon}) = \varnothing$ $D_c(c) = \boldsymbol{\varepsilon}$

$D_c(\varnothing) = \varnothing$ $D_c(c') = \varnothing$ if $c \neq c'$

$D_c(l_1 \cup l_2) = D_c(l_1) \cup D_c(l_2)$

$D_c(l_1 >> l_2) = (D_c(l_1) >> l_2) \cup D_c(l_2)$ if $\boldsymbol{\varepsilon} \in l_1$

$D_c(l_1 >> l_2) = D_c(l_1) >> l_2$ if $\boldsymbol{\varepsilon} \notin l_1$

```
(define/memoize (derive c l)
  #:order ([l #:eq] [c #:equal])
  (match l
    [(empty) l]
    [(eps) (empty)]
    [(token pred class)
     (if (pred c) (eps) (empty))]
    [(alt l1 l2) (alt (derive c l1)
                      (derive c l2))]
    [(seq (and (nullablep) l1) l2)
     (alt (derive c l2)
          (seq (derive c l1) l2))]
    [(seq l1 l2) (seq (derive c l1) l2)]))
```

```
(define (parse l s)
  (if (null? s)
      (parse-null l)
      (parse (D (car s) l) (cdr s))))

(define/fix (nullable? l) #:bottom #f
  (match l
    [(empty)       #f]
    [(eps)         #t]
    [(token _ _)   #f]
    [(repp _)      #t]
    [(orp l1 l2)
     (or (nullable? l1) (nullable? l2))]
    [(seqp l1 l2)
     (and (nullable? l1) (nullable? l2))]
    [(redp l1 _)   (nullable? l1)]))

(define/fix (parse-null l)
  #:bottom empty-tree-set
  (match l
    [(empty)         empty-tree-set]
    [(eps* S)        S]
    [(eps)           (set l)]
    [(token _ _)     empty-tree-set]
    [(repp (nullp))
     (error "infinite parse-null")]
    [(repp _)        (set '())]
    [(orp  l1 l2)
     (set-union (parse-null l1)
                (parse-null l2))]
    [(seqp l1 l2)
     (for*/set ([t1 (parse-null l1)]
                [t2 (parse-null l2)])
       (cons t1 t2))]
    [(redp l1 f)
     (for/set ([t (parse-null l1)])
       (f t))]))
```

```
(define/memoize (D c l)
  #:order ([l #:eq] [c #:equal])
  (match l
    [(empty)     l]
    [(eps)       (empty)]
    [(token pred class)
     (if (pred c)
         (eps* (set c))
         (empty))]
    [(orp l1 l2) (alt (D c l1) (D c l2))]
    [(seqp (and (nullp) l1) l2)
     (cat (eps* (parse-null l1)) (D c l2))]
    [(seqp (and (nullablep) l1) l2)
     (alt (cat (eps* (parse-null l1))
               (D c l2))
          (cat (D c l1) l2))]
    [(seqp l1 l2) (cat (D c l1) l2)]
    [(repp l1) (cat (D c l1) l)]
    [(redp l f) (red (D c l) f)]))
```

```
(define/memoize (derive c l)
  #:order ([l #:eq] [c #:equal])
  (match l
    [(empty) l]
    [(eps) (empty)]
    [(token pred class)
     (if (pred c) (eps) (empty))]
    [(orp l1 l2) (alt (derive c l1)
                      (derive c l2))]
    [(seqp (and (nullablep) l1) l2)
     (alt (derive c l2)
          (cat (derive c l1) l2))]
    [(seqp l1 l2) (cat (derive c l1) l2)]
    [(repp l1) (cat (derive c l1) l)]
    [(redp l f) (derive c l)]))
```

$D_c(\varepsilon) = \emptyset$

$D_c(c) = \varepsilon$

$D_c(\emptyset) = \emptyset$

$D_c(c') = \emptyset$ if $c \neq c'$

$D_c(l_1 \cup l_2) = D_c(l_1) \cup D_c(l_2)$

$D_c(l_1 >> l_2) = (D_c(l_1) >> l_2) \cup D_c(l_2)$ if $\varepsilon \in l_1$

$D_c(l_1 >> l_2) = D_c(l_1) >> l_2$ if $\varepsilon \notin l_1$

Performance

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

$O(n^3)$

Parsing 31 lines of Python:

~~24,000x slower than Bison~~

~~3 minutes~~

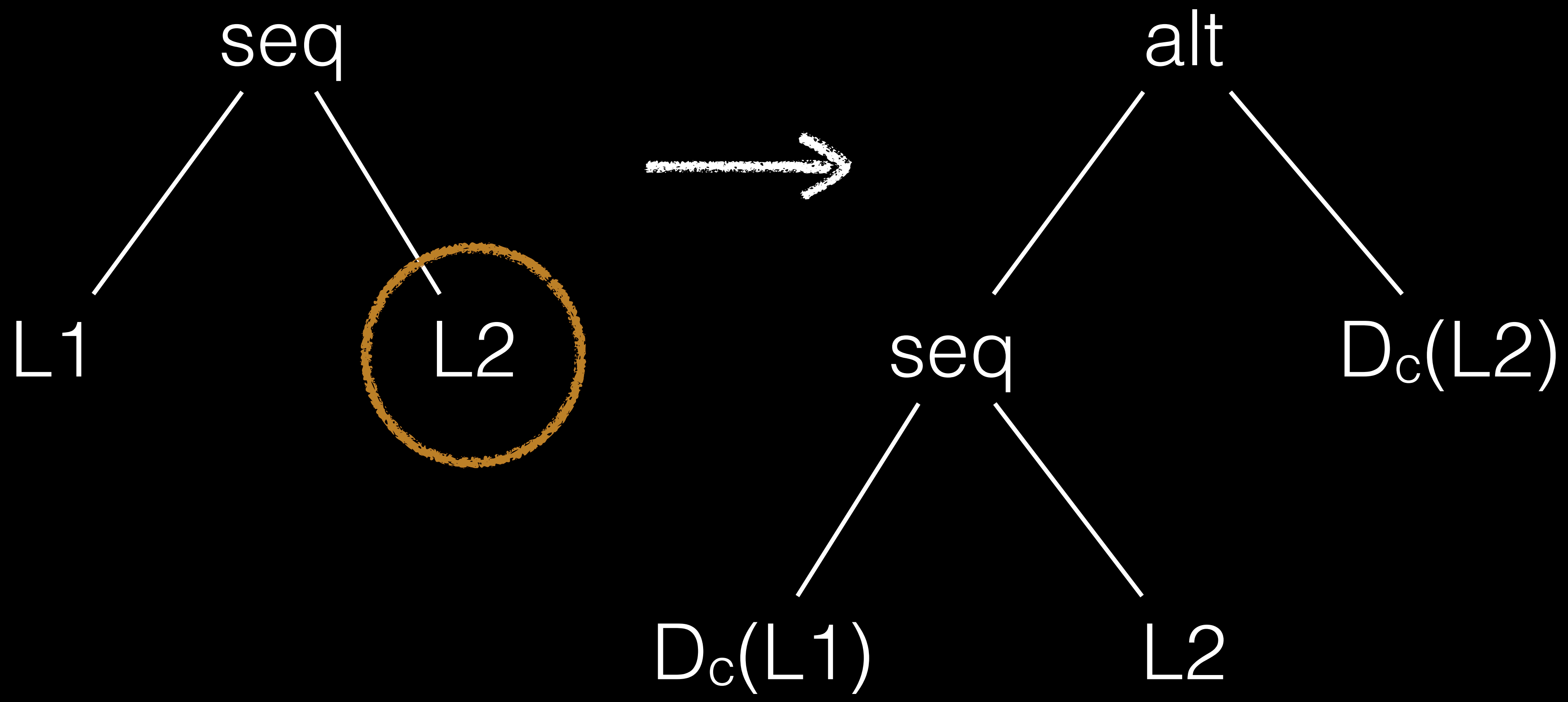
25x slower than Bison

2 seconds

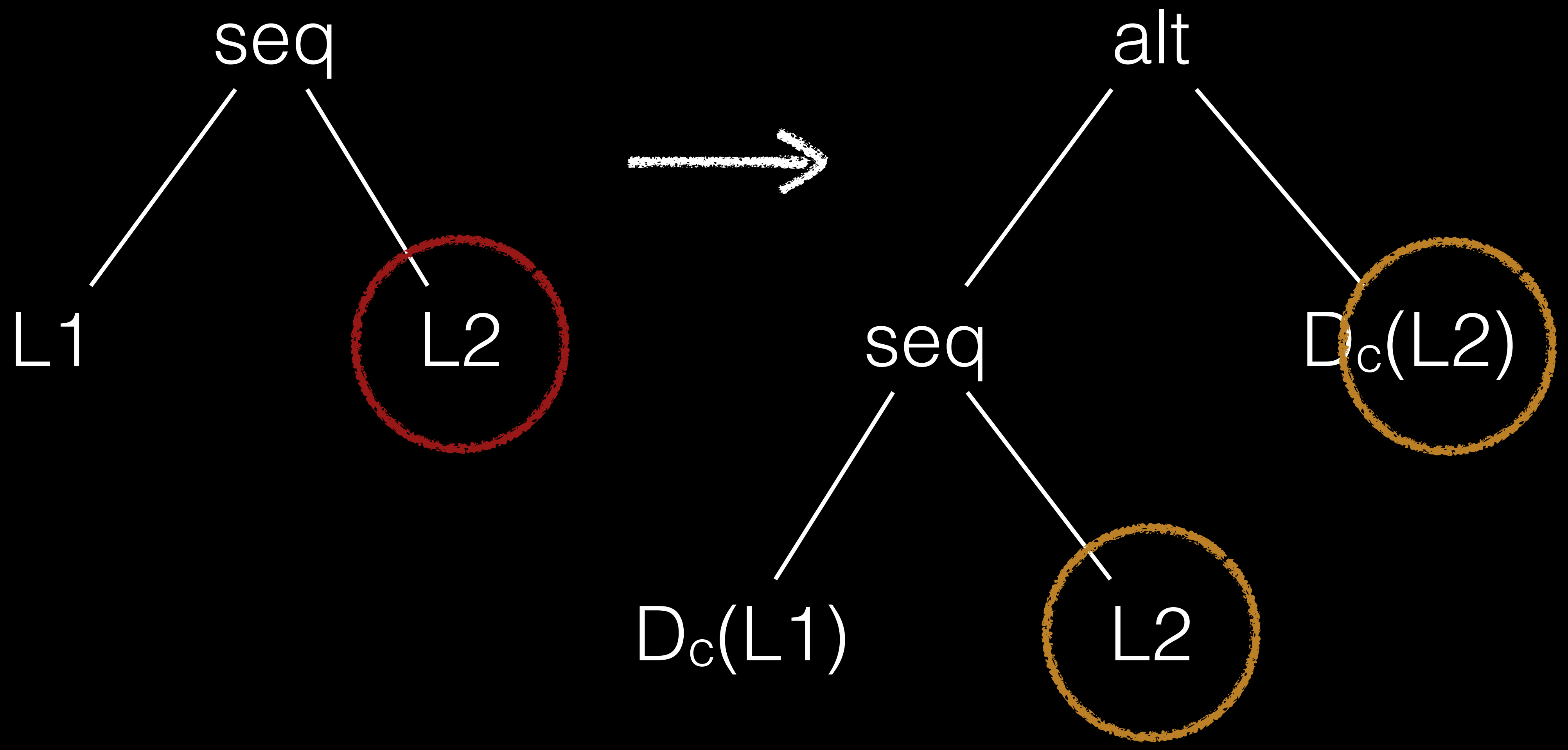
Complexity Analysis

```
(define/memoize (D c L)
  #:order ([L #:eq] [c #:equal])
  (match L
    [(empty) (empty)]
    [(eps* T) (empty)]
    [(char a) (if (equal? a c)
                (eps* (set c))
                (empty))]
    [(red L f) (red (D c L) f)]
    [(alt L1 L2) (alt (D c L1) (D c L2))]
    [(seq L1 L2)
      (if (nullable? L1)
       (alt (seq (D c L1) L2) (D c L2))
            (seq (D c L1) L2))]))
```

```
[(seq L1 L2)
  (if (nullable? L1)
   (alt (seq (D c L1) L2) (D c L2))
```



```
[(seq L1 L2)
  (if (nullable? L1)
   (alt (seq (D c L1) L2) (D c L2)))
```



Time

$\leq k \cdot$ # of allocated nodes

$\leq k \cdot$ # of node names

$\leq O(n^3)$

L	L
$D_f(L)$	Lf
$D_o(D_f(L))$	Lfo
$D_o(D_o(D_f(L)))$	$Lfoo$

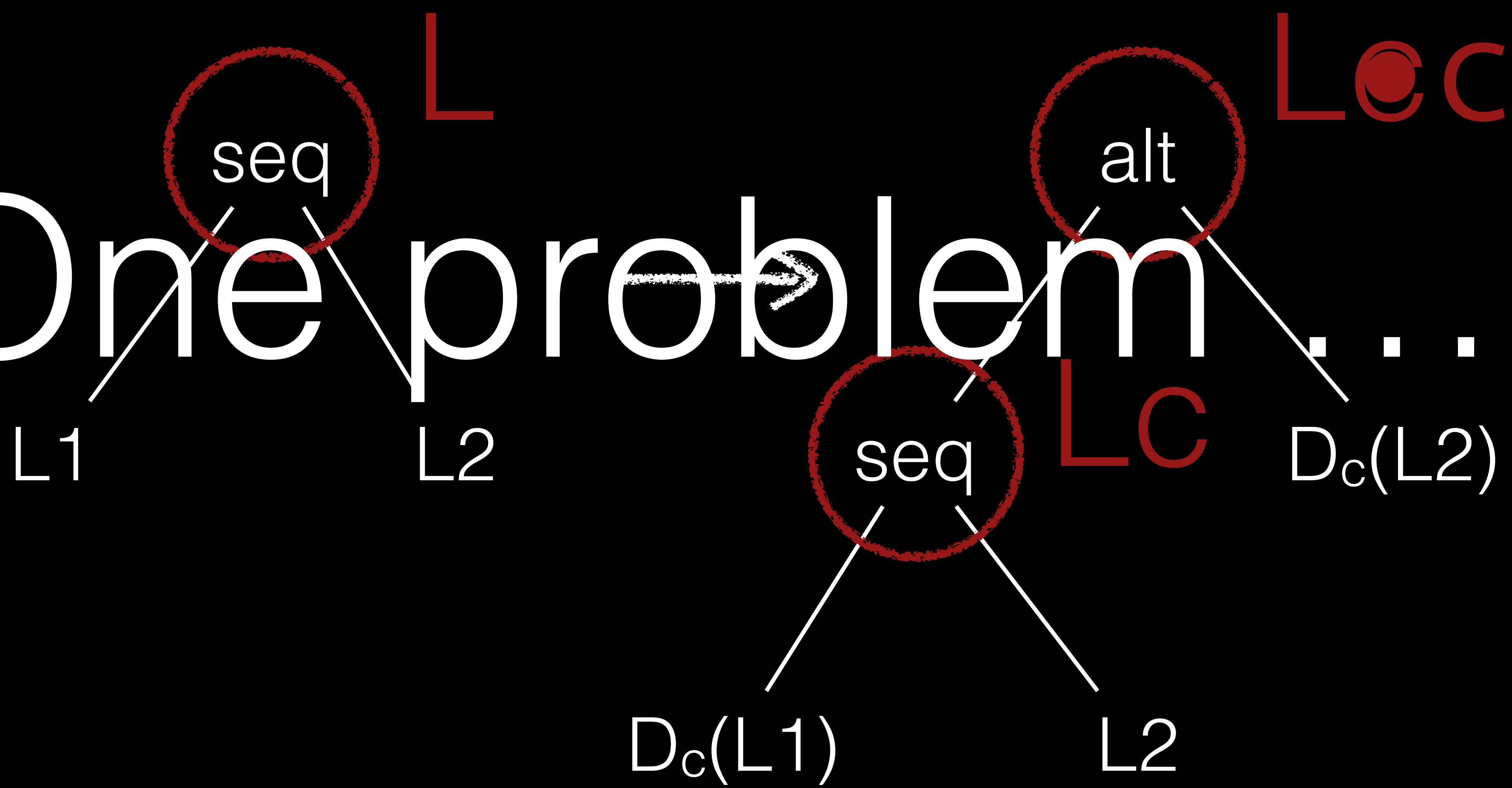
# of names $\leq |G| \cdot n^2$

# of initial grammar nodes

# of possible substrings of input

```
[(seq L1 L2)
  (if (nullable? L1)
   (alt (seq (D c L1) L2) (D c L2))
```

One problem ...



$La_1 \cdots a_k$

$La_1 \cdots a_i \bullet a_{i+1} \cdots a_k$

# of names ≤

$|G| \cdot n^2 +$

$|G| \cdot n^2 \cdot n$

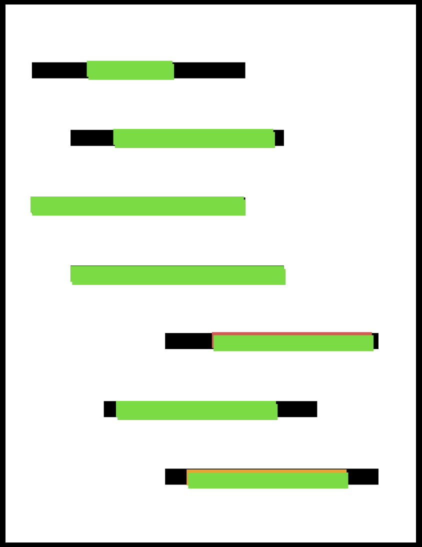
$$O(n^3)$$

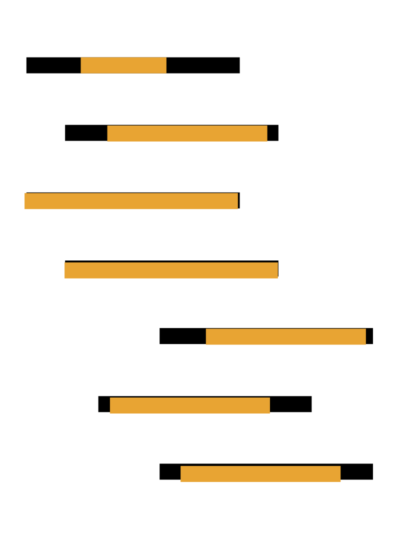
# of initial grammar nodes

# of possible substrings of input

# of possible positions for bullet

Performance





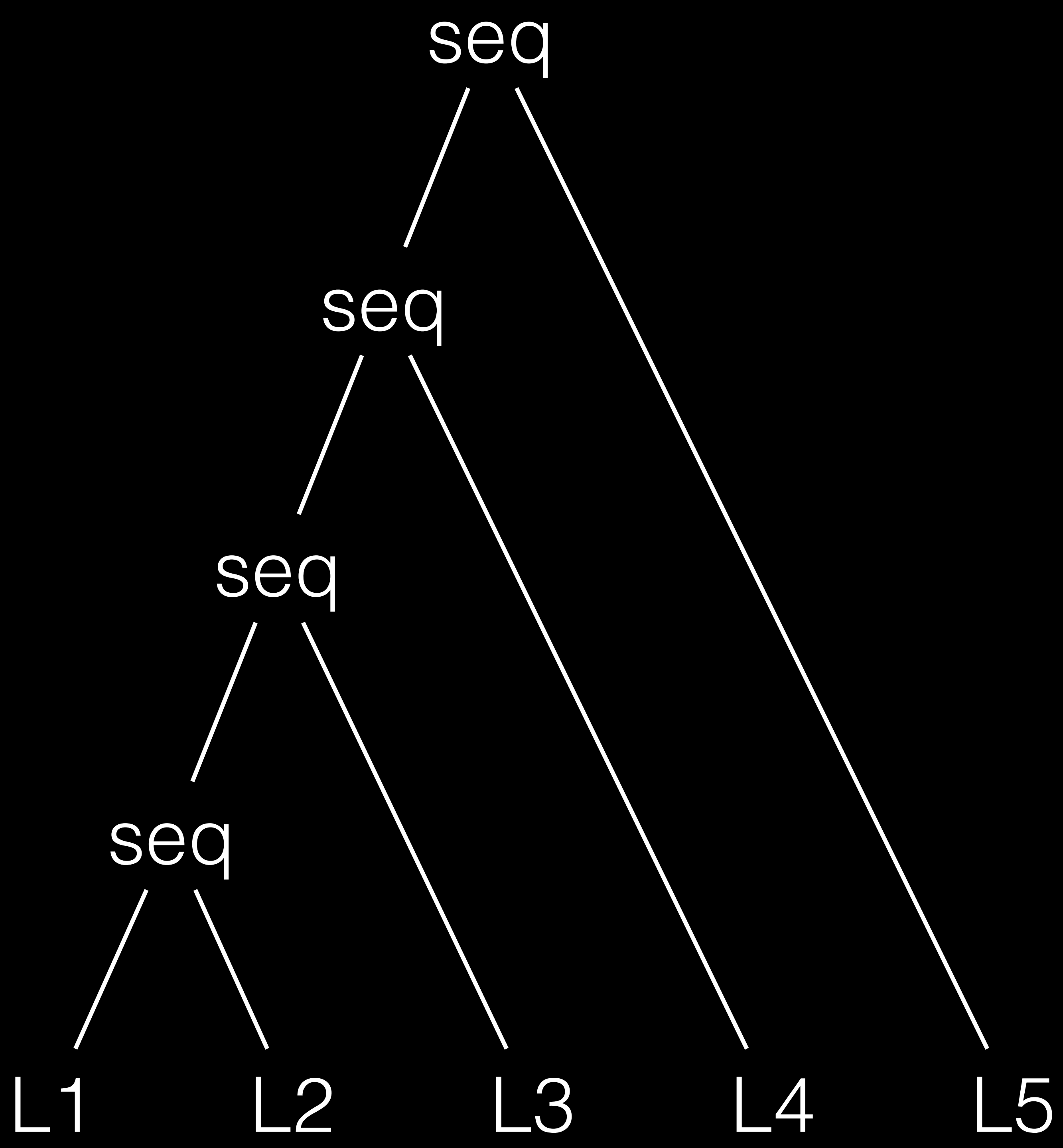
[redacted]

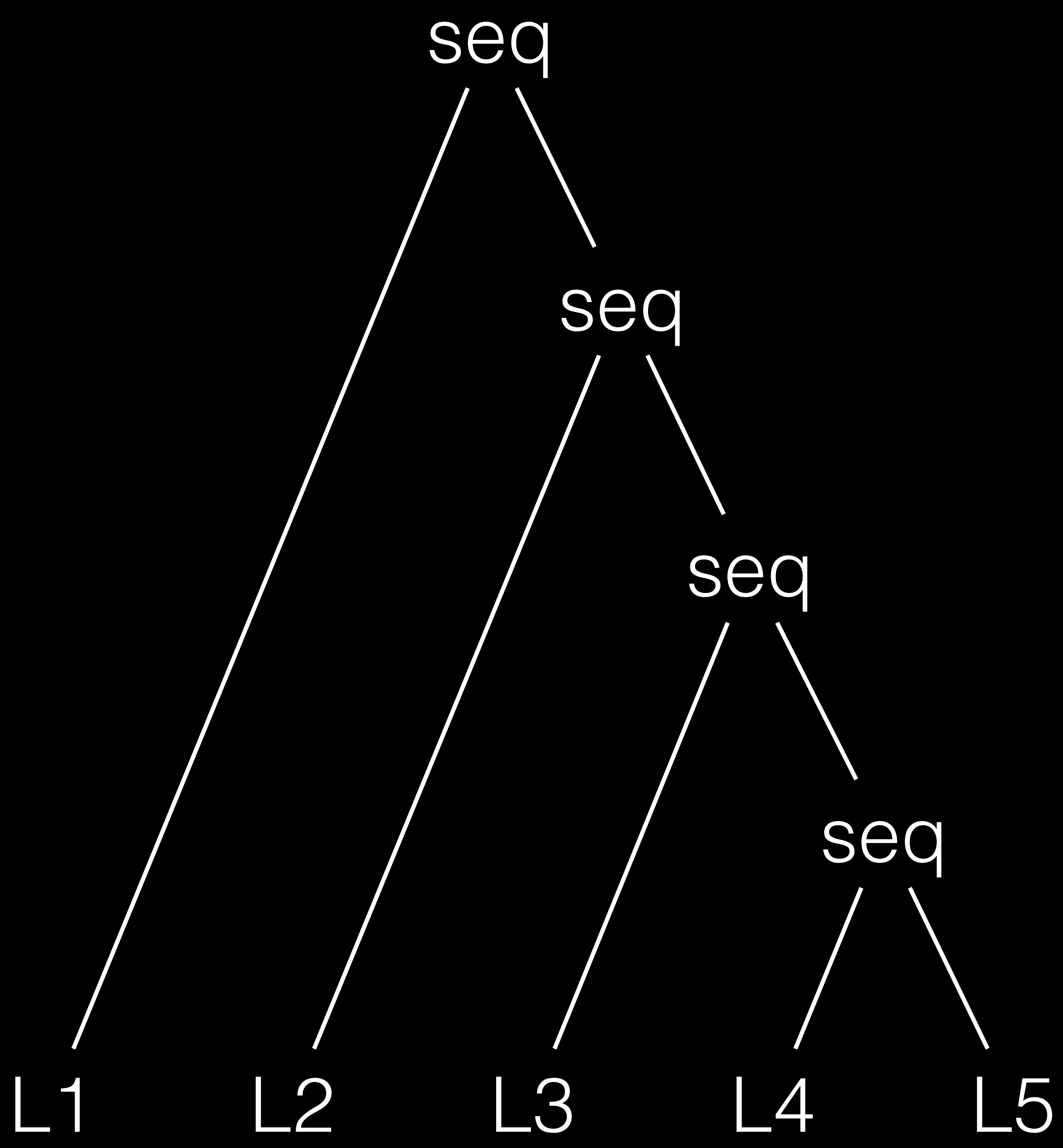
Optimizations

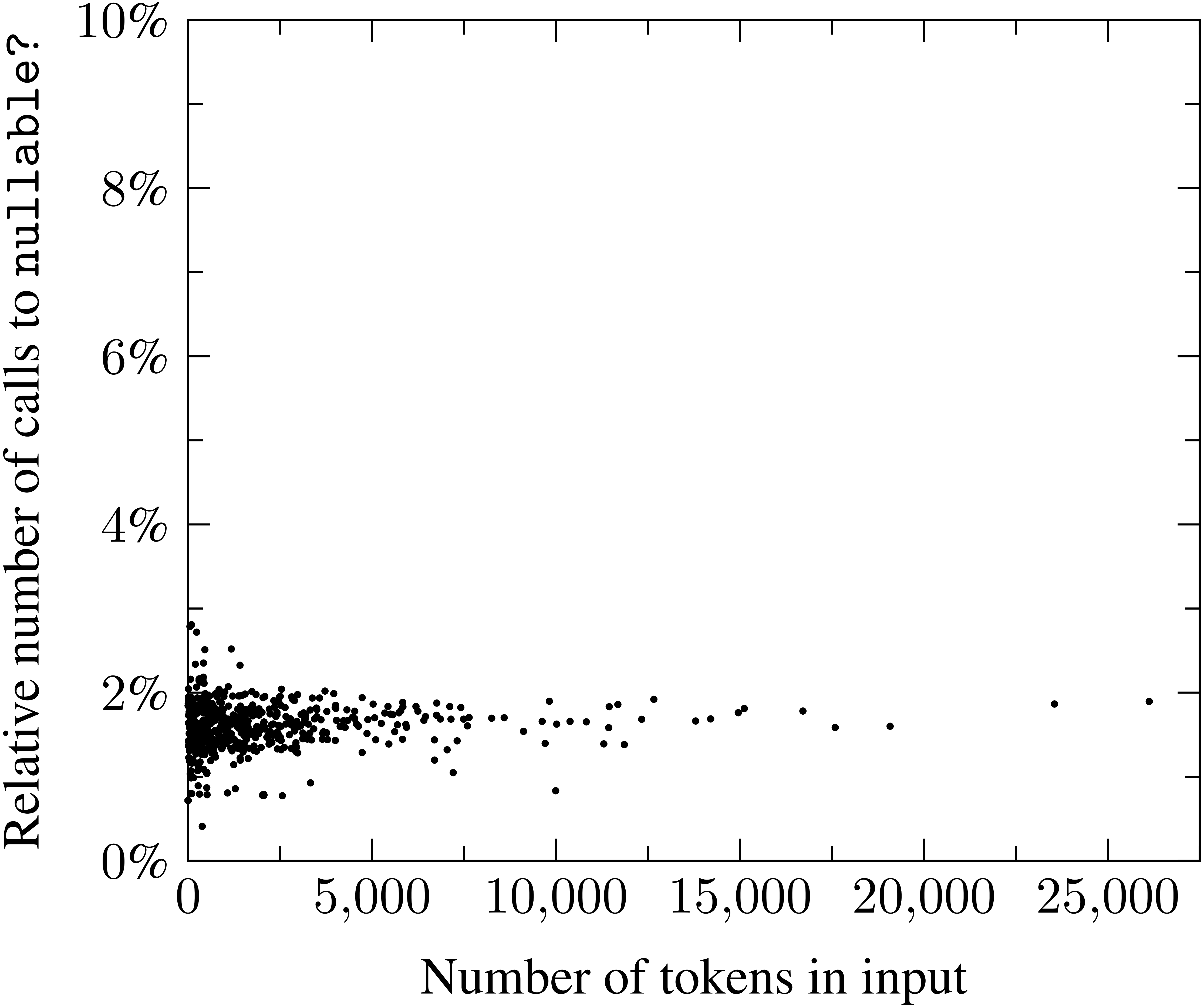
Compaction:
- Pre-processing
- Canonicalization
- Smart Constructors

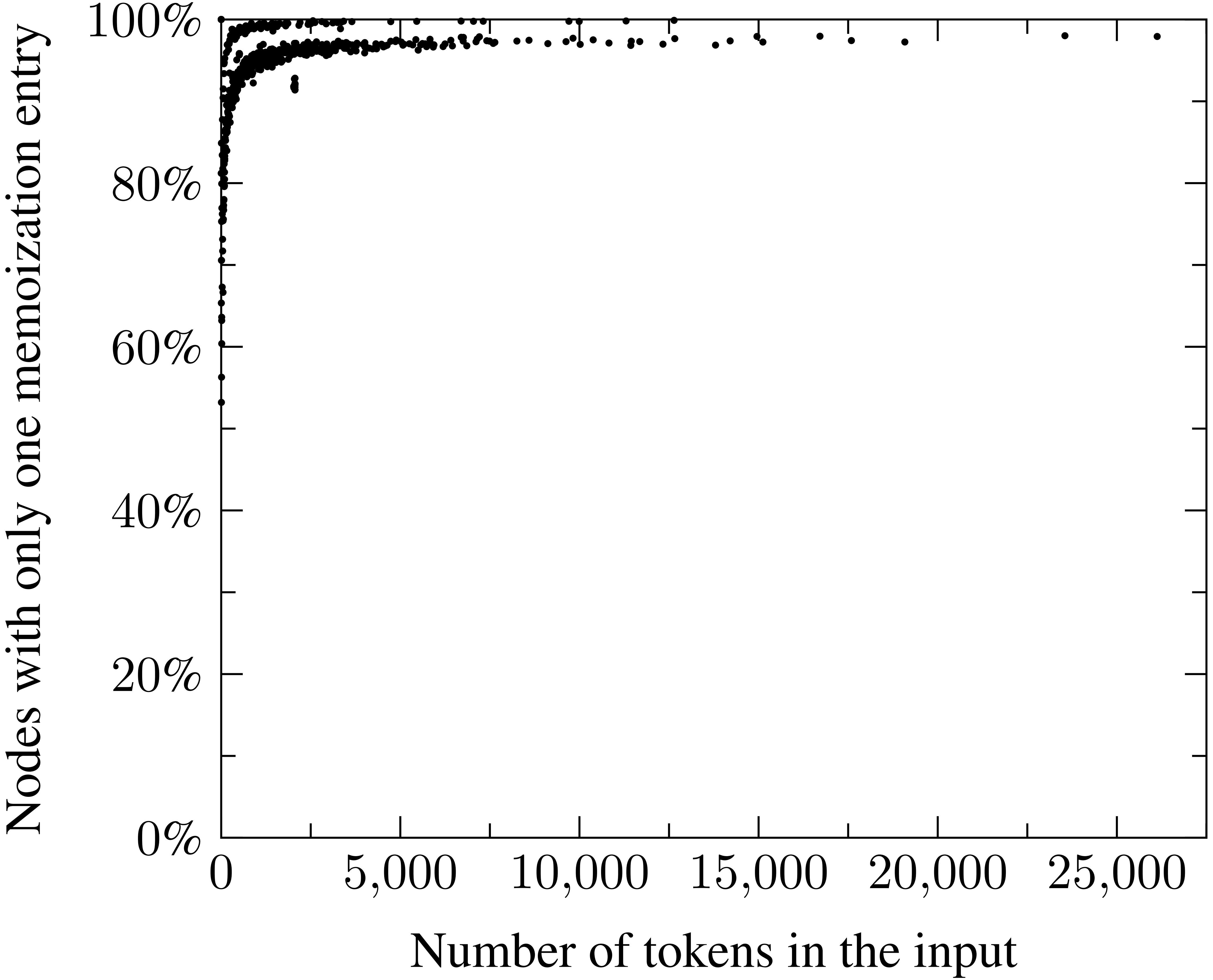
Nullability:
- Demand Based
- Field Instead of Hashtable

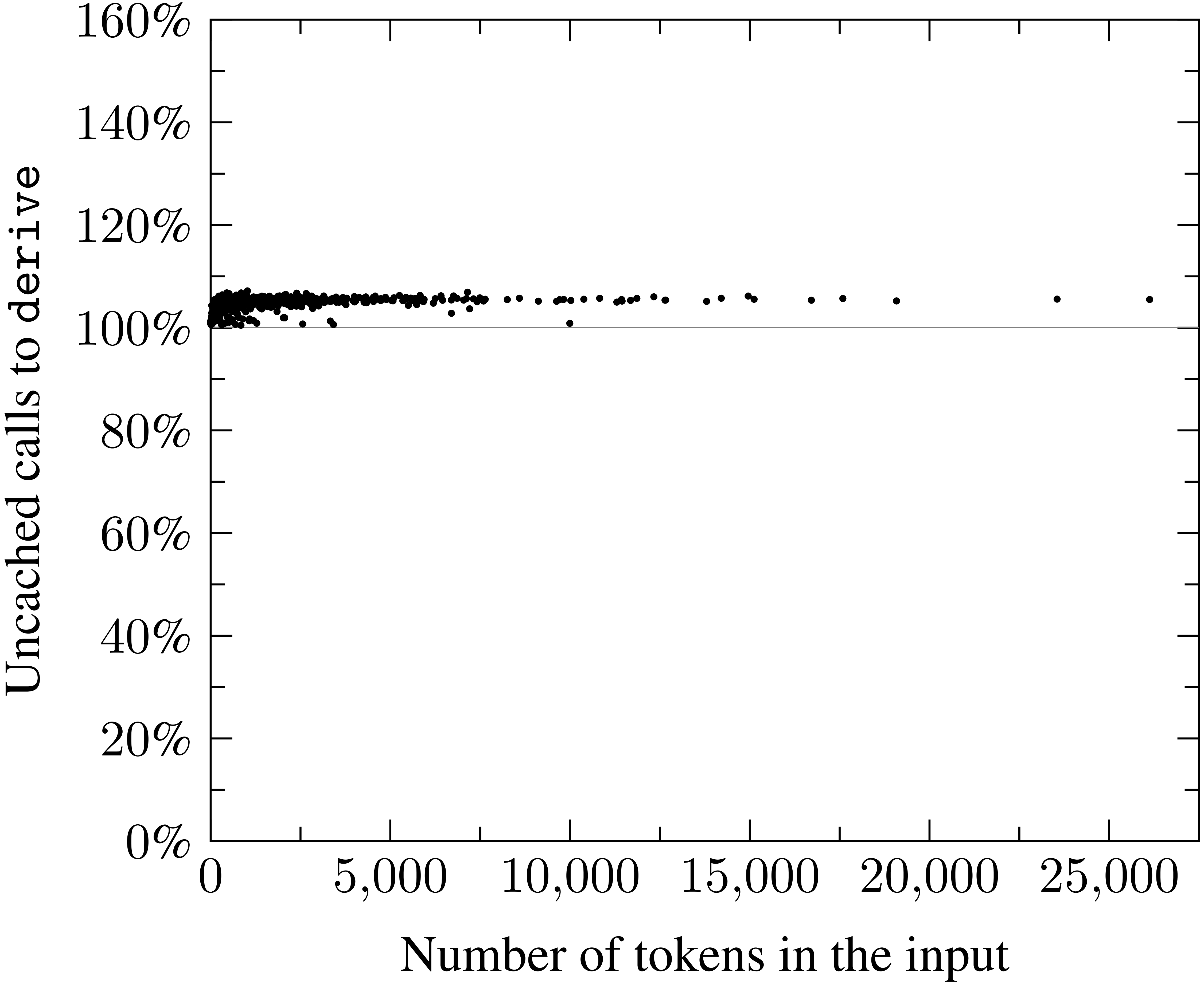
Memoization:
- One-entry/Forgetful Hashtables
- Field Instead of Hashtable

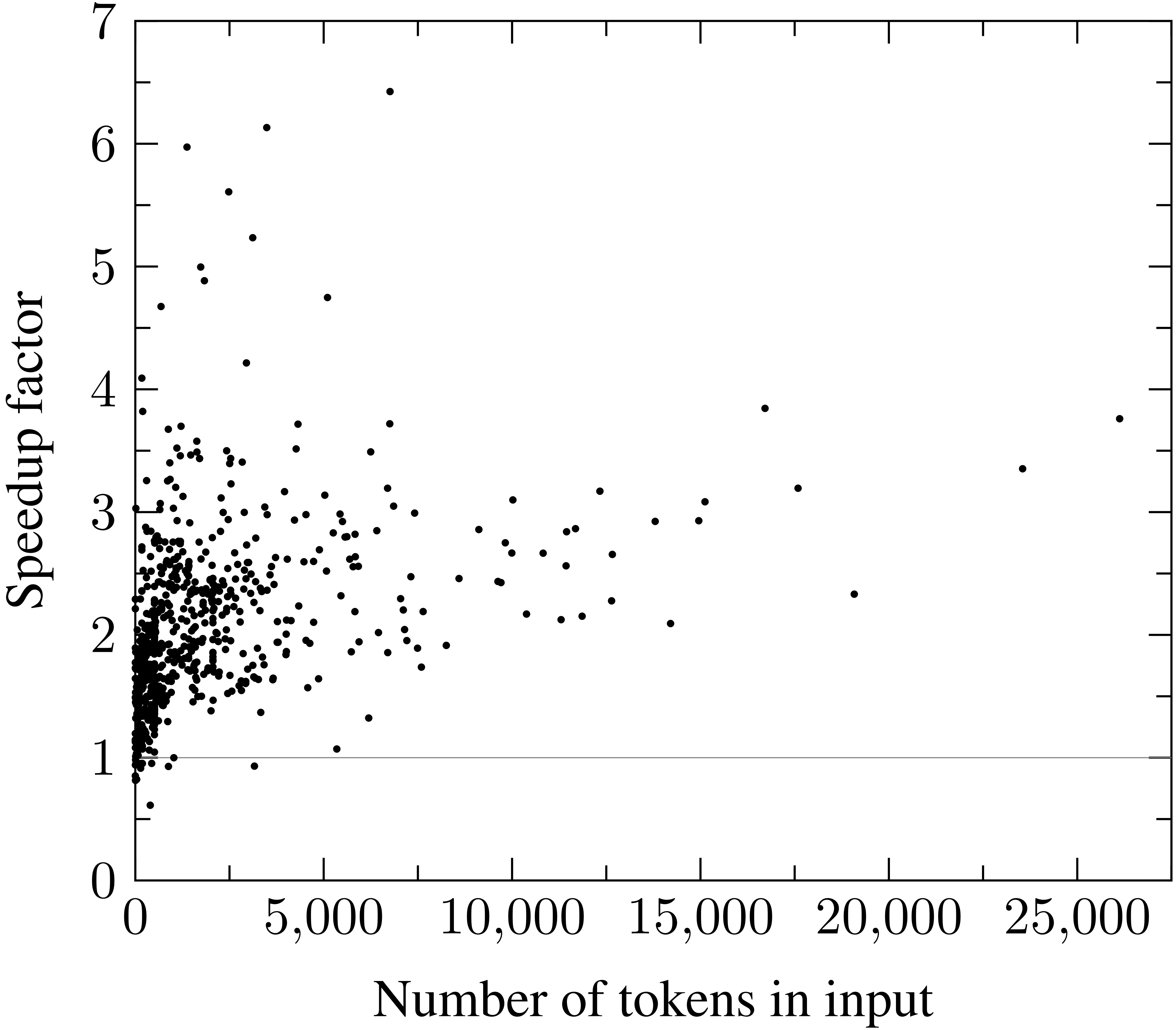

seq
seq
seq
seq
L1
L2
L3
L4
L5

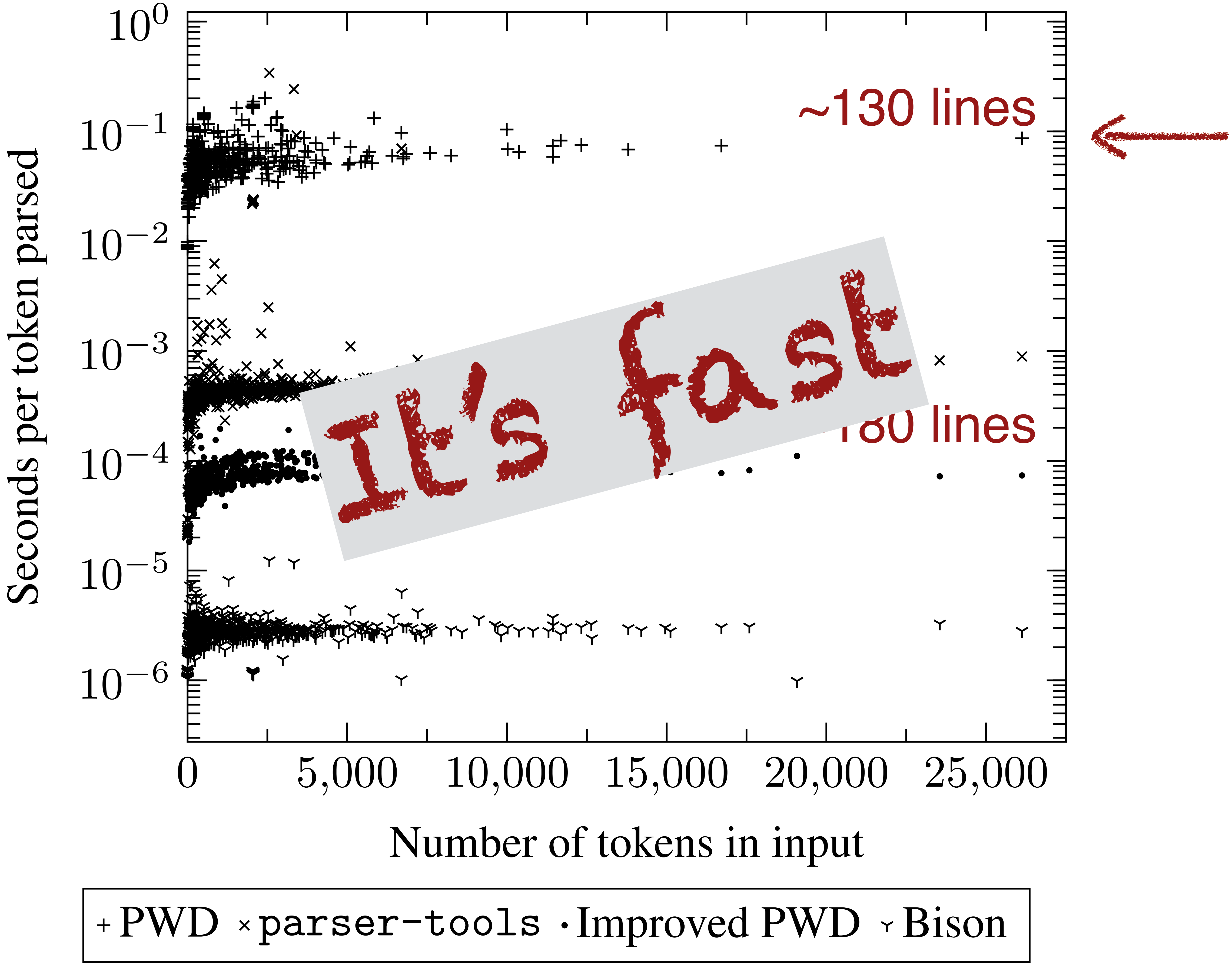

seq
seq
seq
seq
L1
L2
L3
L4
L5


Relative number of calls to nullable?
10%
8%
6%
4%
2%
0%
0
5,000
10,000
15,000
20,000
25,000
Number of tokens in input


Nodes with only one memoization entry
100%
80%
60%
40%
20%
0%
0
5,000
10,000
15,000
20,000
25,000
Number of tokens in the input


160%
140%
120%
100%
80%
60%
40%
20%
0%
Uncached calls to derive
0
5,000
10,000
15,000
20,000
25,000
Number of tokens in the input


Speedup factor
0
1
2
3
4
5
6
7
0
5,000
10,000
15,000
20,000
25,000
Number of tokens in input



~130 lines

180 lines

It's fast

On the Complexity and Performance of Parsing with Derivatives

~~$O(2^{2n})$~~ $O(n^3)$

Michael D. Adams [*], Celeste Hollenbeck, Matthew Might

~~24,000x~~ 25x

Implementation available at
http://michaeldadams.org/papers/derivatives2/

[*] Seeking a tenure-track position in the upcoming hiring cycle