Programming Obesity: A Code Health Epidemic

Aaron W. Hsu arcfide@sacrideo.us, Functional Conf 2019, Bengaluru, India

Imagine/Dream

Direct

Fast

Transmissive

Empower vs. Control

More domain knowledge,
Less systems rumination

The Main Thing

Obsolete? Hardly.

Essential
vs.
Accidental

We Are
Consumer and Producer

We are in this together

Tech Stack

GO RAILS Java Scala

Clojure python™ PostgreSQL

MariaDB RocksDB InfluxDB

hazelcast IMDG RabbitMQ redis

CHEF ANSIBLE Terraform

Kotlin TensorFlow kafka

kubernetes Swift Flink GRPC

Programming Obesity

Cascading Systemic Failure to Simplify and Tendency Towards Unsustainable Waste

Structural Complexity

Performance

Economy

Change

Change is Uncomfortable

Break Points

Simple:

Faster:

Economical:

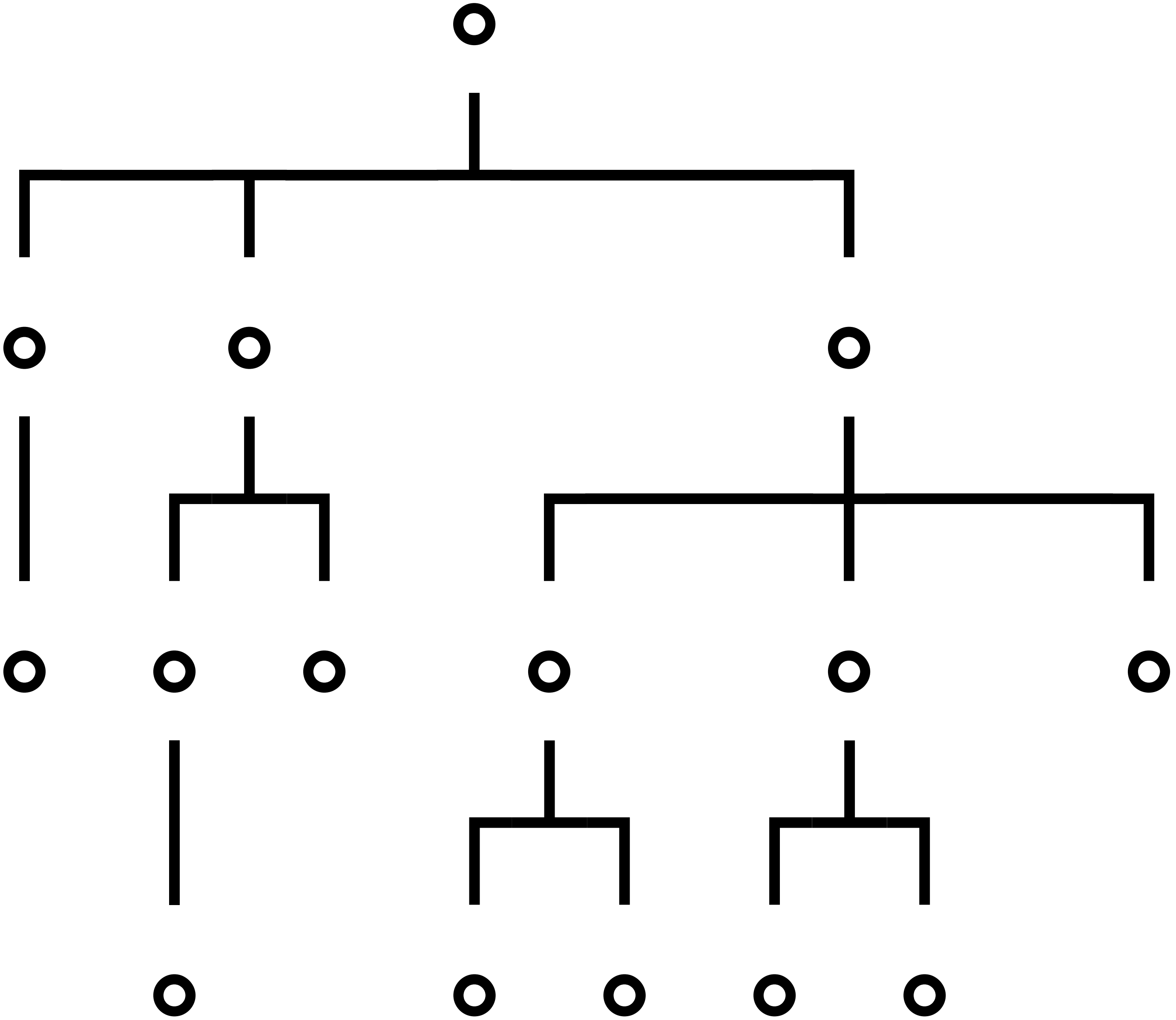
Impact

Don’t wait

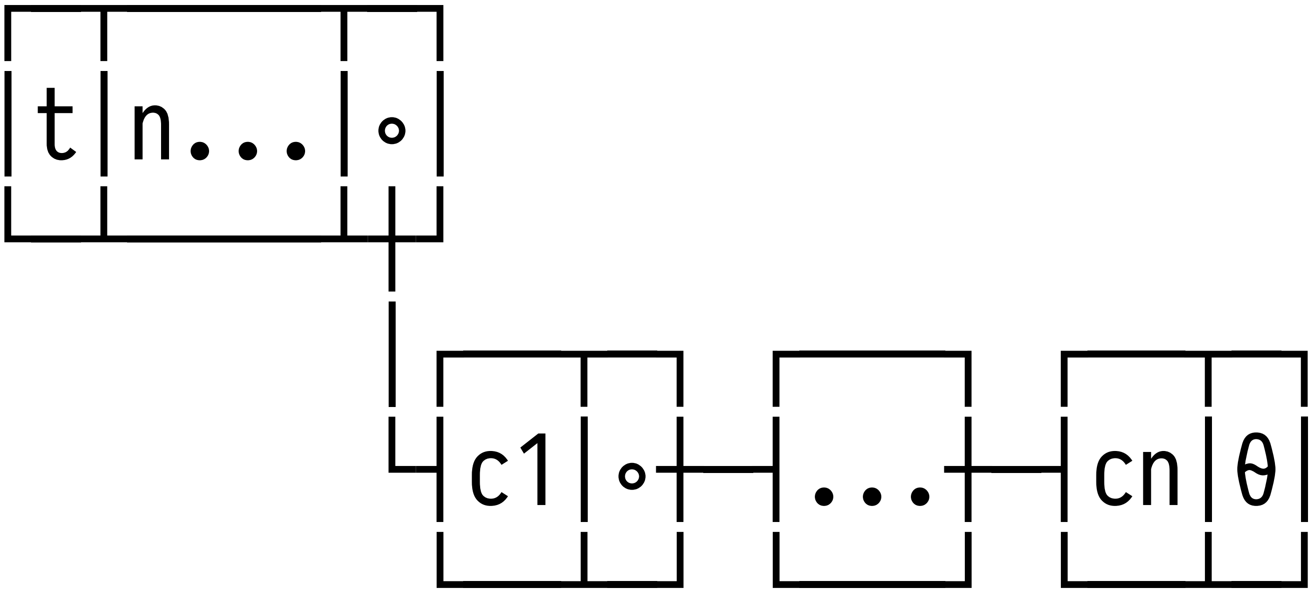
Given Up

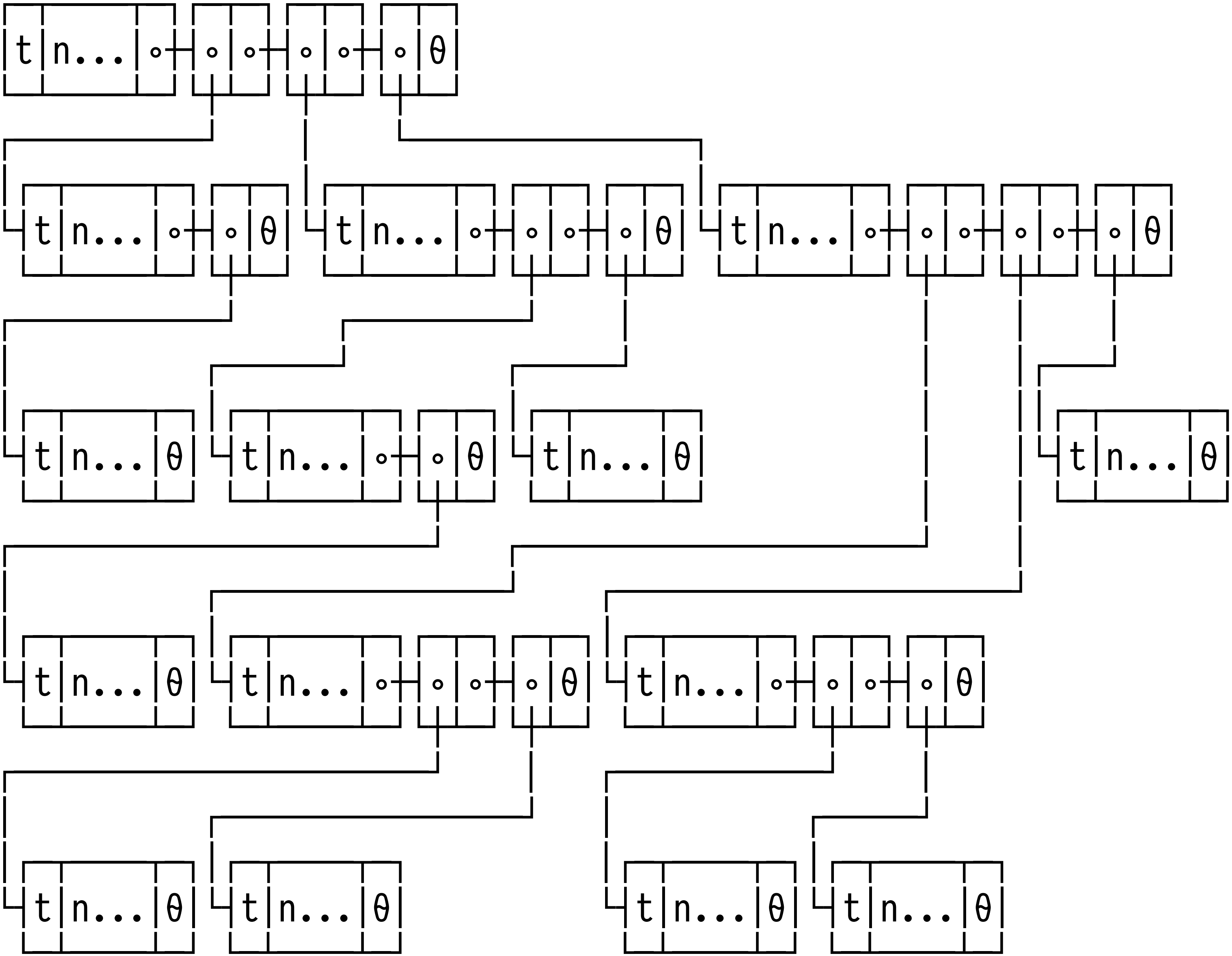
Where do we start?

Generalized pointers
are the **Refined Sugar**
of Programming



```
(Expr (e)
   v
   (B.0 ([name ns]) e)
   (A.0 ([name ns]) num* ...)
   (A.3 ([name ns]) v* ...)
   (E.1 ([name ns]) f e)
   (E.2 ([name ns]) e f ebrk)
   (E.4 ([name ns]) v ebrk e))
```


t
n...
∘
c1
∘
...
cn
0


t
n...
θ

Eliminate

expedient complexity

and waste

Achieve
real simplicity

Holistic Economy

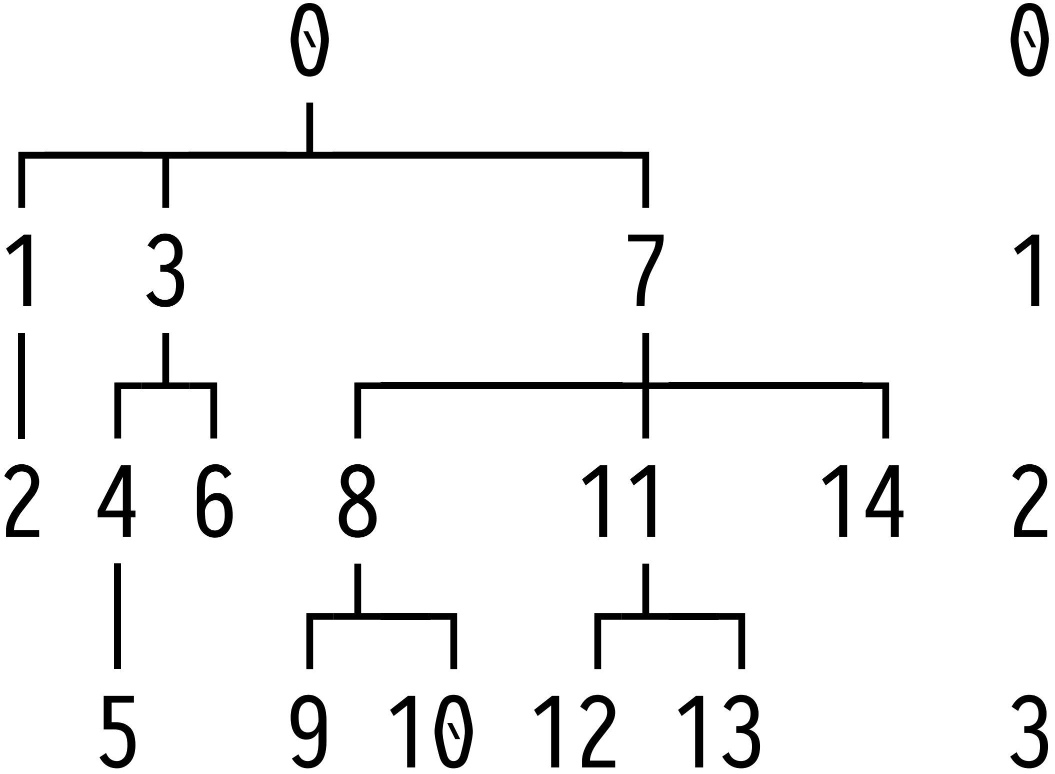
Can a compiler be:
GPU hosted, fast,
portable, and simple?

Impossible

Eliminate Everything

A New (Old) Hope

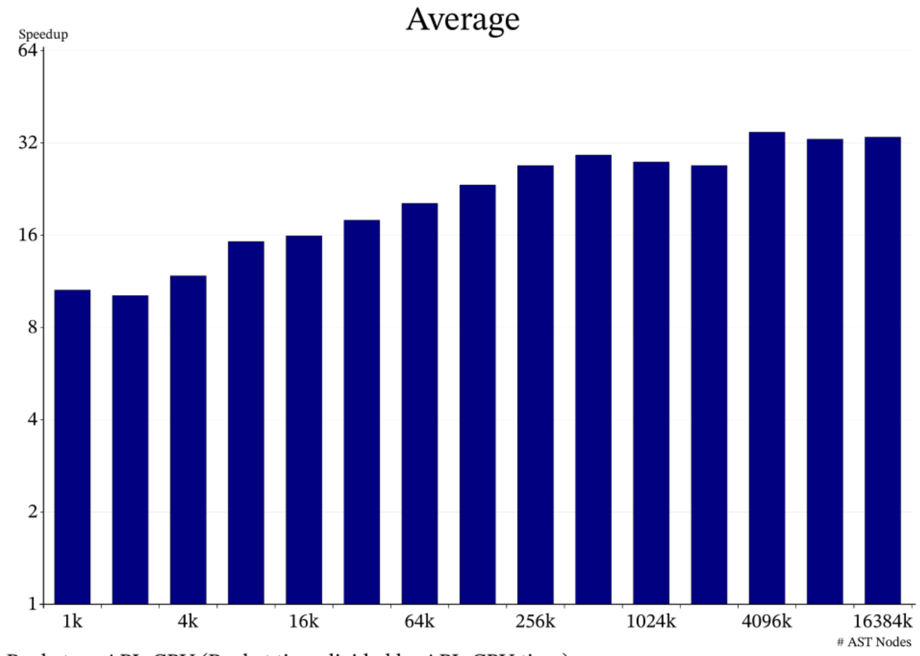
APL



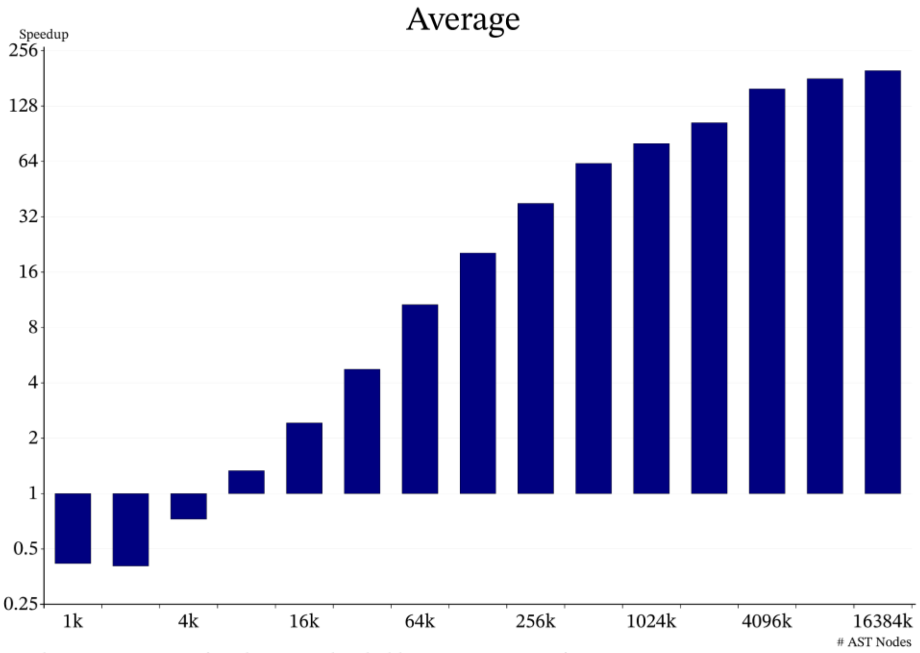
```
 ⍉(⍳≢d),⍪d
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14
0 1 2 1 2 3 2 1 2 3  3  2  3  3  2
```

```
Simple:      94% 15×
Faster:  CPU 89%  9×
         GPU 98% 56×
Economy: Raw 70%  3×
       Ratio 88%  8×
```

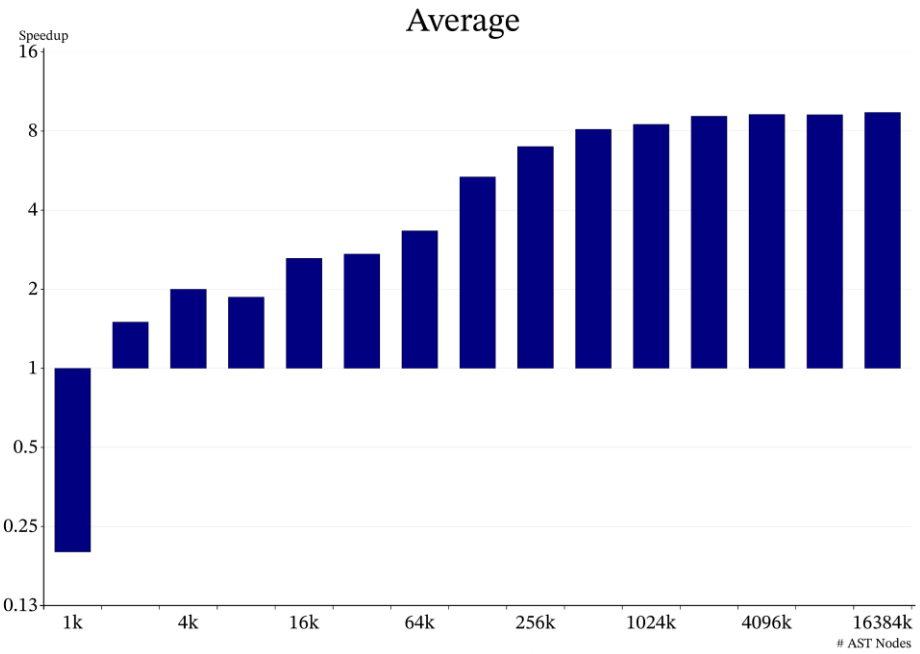
	LoC	**Tokens**	**Names**	**Nodes**
Nanopass	1012	20947	248	14680
Co-dfns	17	760	74	948



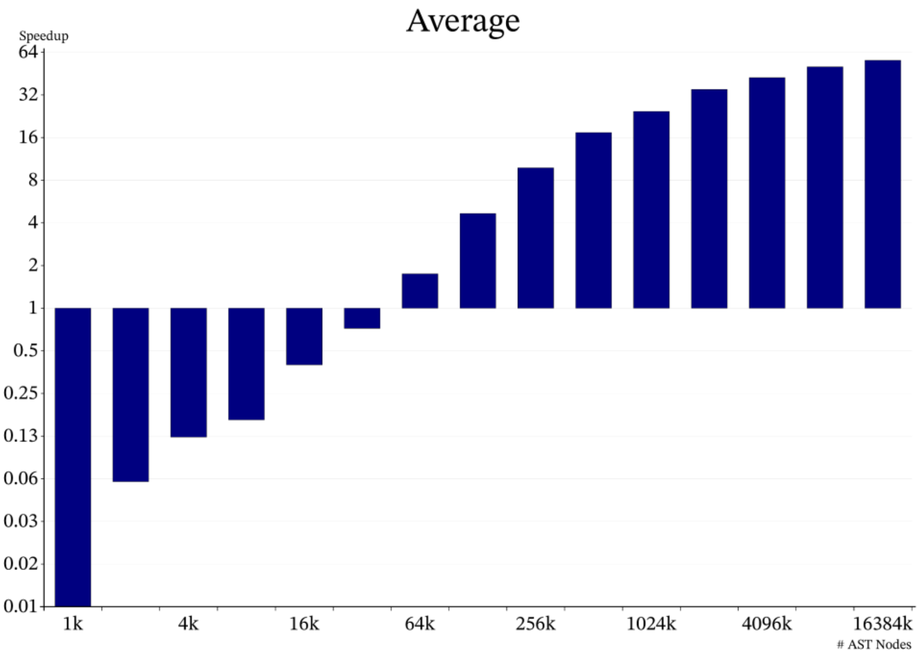
Racket vs. APL-CPU (Racket time divided by APL-CPU time)



Racket vs. APL-GPU (Racket time divided by APL-GPU time)



Chez vs. APL-CPU (Chez time divided by APL-CPU time)



Chez vs. APL-GPU (Chez time divided by APL-GPU time)

Memory Usage

Size	Dya	Rack	Chez	Size	Dya	Rack	Chez	Size	Dya	Rac	Chez
0	0.009	0.065	2.027	**5**	0.129	2.056	4.055	**10**	3.985	65.721	93.258
1	0.013	0.13	2.027	**6**	0.254	4.11	6.082	**11**	7.966	131.44	186.516
2	0.021	0.259	2.027	**7**	0.503	8.217	10.137	**12**	15.927	262.877	371.508
3	0.036	0.516	2.027	**8**	1	16.432	22.301	**13**	31.849	525.752	739.980
4	0.067	1.029	2.027	**9**	1.995	32.862	44.602	**14**	63.692	1051.502	1485.023

Feeling is Believing

Positive Cascading Effects

Conclusion

Start with APL

Thank You.

Aaron Hsu – arcfide@sacrideo.us

A data parallel compiler hosted on the GPU

http://www.dyalog.com

```
i←(⍸(~t∊3 4)∧t[p]=3),{⍵⌿⍨2|⍳≢⍵}⍸t[p]=4 ⋄ p t k n r⌿⍨←⊂m←2@i⊢1⍴⍨≢p
p r i I⍨←⊂j←(+⍀m)-1 ⋄ n←j I@(0≤⊢)n ⋄ p[i]←j←i-1
k[j]←-(k[r[j]]=0)∨0@({⊃⌽⍵}⌸p[j])⊢t[j]=1 ⋄ t[j]←2
```

```
(define i
  (catenate
    (where
      (and
        (not (members-of t `(3 4)))
        (pequal? 3 (indexing t p))))
    ((lambda (w)
       (replicate
         (modulo
           (index-gen (tally w))
           2)
         w))
     (where (pequal? 4 (indexing t p))))))
(define m ((at 2 i) (reshape (tally p) 1)))
(define-values (p t k n r)
  (let ([m^ (enclose m)])
    (apply values
      (map (lambda (x) (replicate m^ x))
           p t k n r))))
(define j (pminus (scan-first + m) 1)
(define-values (p r i)
  (let ([j^ (enclose j)])
    (apply values
      (map (lambda (x) (indexing j^ x)) p r i))))
```

```
(define n
  ((at indexing (lambda (x)
                  (pless-equal? 0 x)))
   j n))
(define j (pminus i 1))
(array-set! p i j)
(array-set! k j
  (negate
    (or
      (pequal? 0
        (indexing k
          (indexing r j)))
      ((at 0
         ((key (lambda (x)
                 (disclose (reverse x))))
               (indexing p j)))
        (pequal 1 (indexing t j))))))
(array-set! t j 2)
```