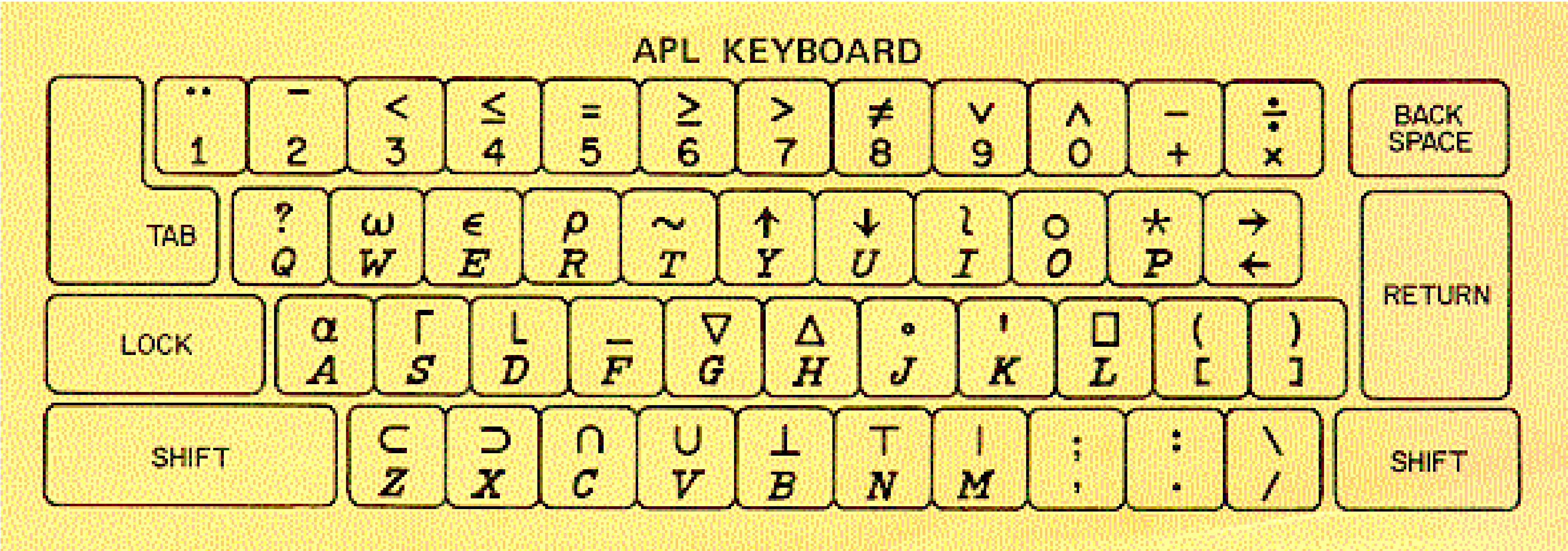
APL: The Greatest Programming Language You Never Heard Of



Ken Iverson

Prime Numbers

$$(2 = +\neq 0 = (\iota X) \circ.| \ \iota X) \ / \ \iota X$$

Iota

ιX

1 2 3 4 5 6 7 8 9 10

Rho

3 3 ρ ι 9

1	2	3
4	5	6
7	9	9

Select

0 1 0 1 0 1 0 1 0 / ι 8

1 3 5 7

Outer Product

3 4 5 ∘.+ 1 2 3 4

4	5	6	7
5	6	7	8
6	7	8	9

Identity Matrix

$(\iota X) \circ .= \iota X$

1	0	0	0	0	0	0	0	0	0
0	1	0	0	0	0	0	0	0	0
0	0	1	0	0	0	0	0	0	0
0	0	0	1	0	0	0	0	0	0
0	0	0	0	1	0	0	0	0	0
0	0	0	0	0	1	0	0	0	0
0	0	0	0	0	0	1	0	0	0
0	0	0	0	0	0	0	1	0	0
0	0	0	0	0	0	0	0	1	0
0	0	0	0	0	0	0	0	0	1

Residue Matrix

```
(ιX) ∘.| ιX
```

0	0	0	0	0	0	0	0	0	0
1	0	1	0	1	0	1	0	1	0
1	2	0	1	2	0	1	2	0	1
1	2	3	0	1	2	3	0	1	2
1	2	3	4	0	1	2	3	4	0
1	2	3	4	5	0	1	2	3	4
1	2	3	4	5	6	0	1	2	3
1	2	3	4	5	6	7	0	1	2
1	2	3	4	5	6	7	8	0	1
1	2	3	4	5	6	7	8	9	0

Integral Divisibility Matrix

```
0 = (ιX) ∘.| ιX
1 1 1 1 1 1 1 1 1 1
0 1 0 1 0 1 0 1 0 1
0 0 1 0 0 1 0 0 1 0
0 0 0 1 0 0 0 1 0 0
0 0 0 0 1 0 0 0 0 1
0 0 0 0 0 1 0 0 0 0
0 0 0 0 0 0 1 0 0 0
0 0 0 0 0 0 0 1 0 0
0 0 0 0 0 0 0 0 1 0
0 0 0 0 0 0 0 0 0 1
```

Number of Integral Divisors

+≠ 0 = (ιX) ∘.| ιX

1 2 2 3 2 4 2 4 3 4

Exactly Two Integral Divisors

2 = +⌿ 0 = (ιX) ∘.| ιX

0 1 1 0 1 0 1 0 0 0

Prime Numbers

```
(2 = +⌿ 0 = (ιX) ∘.| ιX) / ιX
2  3  5  7
```

More APL Examples

- Leap Year Test

```
(0 = 400 | X) v (0 ≠ 100 | X)^0 = 4 | X
```

- Test for Duplicate Elements

```
^/ (X ι X) = ι ρX
```

- Standard Deviation

```
((+/ (X - (+/ X) ÷ ρX) * 2) ÷ ρX) * .5
```

Iota


```
(define iota
  (lambda (n)
    (letrec
      ((loop
        (lambda (n acc)
          (if (= n 0)
              acc
              (loop (sub1 n)
                    (cons n acc))))))
      (loop n '()))))

> (iota 8)
(1 2 3 4 5 6 7 8)
```

Select

```
(define select
  (lambda (pred)
    (lambda (ls0 ls1)
      (map cdr
        (filter
          (lambda (x) (pred (car x)))
          (map cons ls0 ls1))))))

> ((select even?) (iota 8) '(a b c d e f g h))
(b d f h)
```

Select Explained

```
> (map cons '(1 2 3) '(a b c))
((1 . a) (2 . b) (3 . c))

> (map (lambda (x) (pred (car x))))
  '((1 . a) (2 . b) (3 . c)))
((#t . a) (#f . b) (#t . c))

> (filter (lambda (x) (pred (car x)))
 '((1 . a) (2 . b) (3 . c)))
((1 . a) (3 . c))

> (map cdr '((1 . a) (3 . c)))
(a c)
```

Tally

```
(define tally
  (lambda (pred)
    (lambda (ls)
      (apply +
       (map (lambda (x) (if (pred x) 1 0))
      ls)))))

> ((tally even?) (iota 8))
4
```

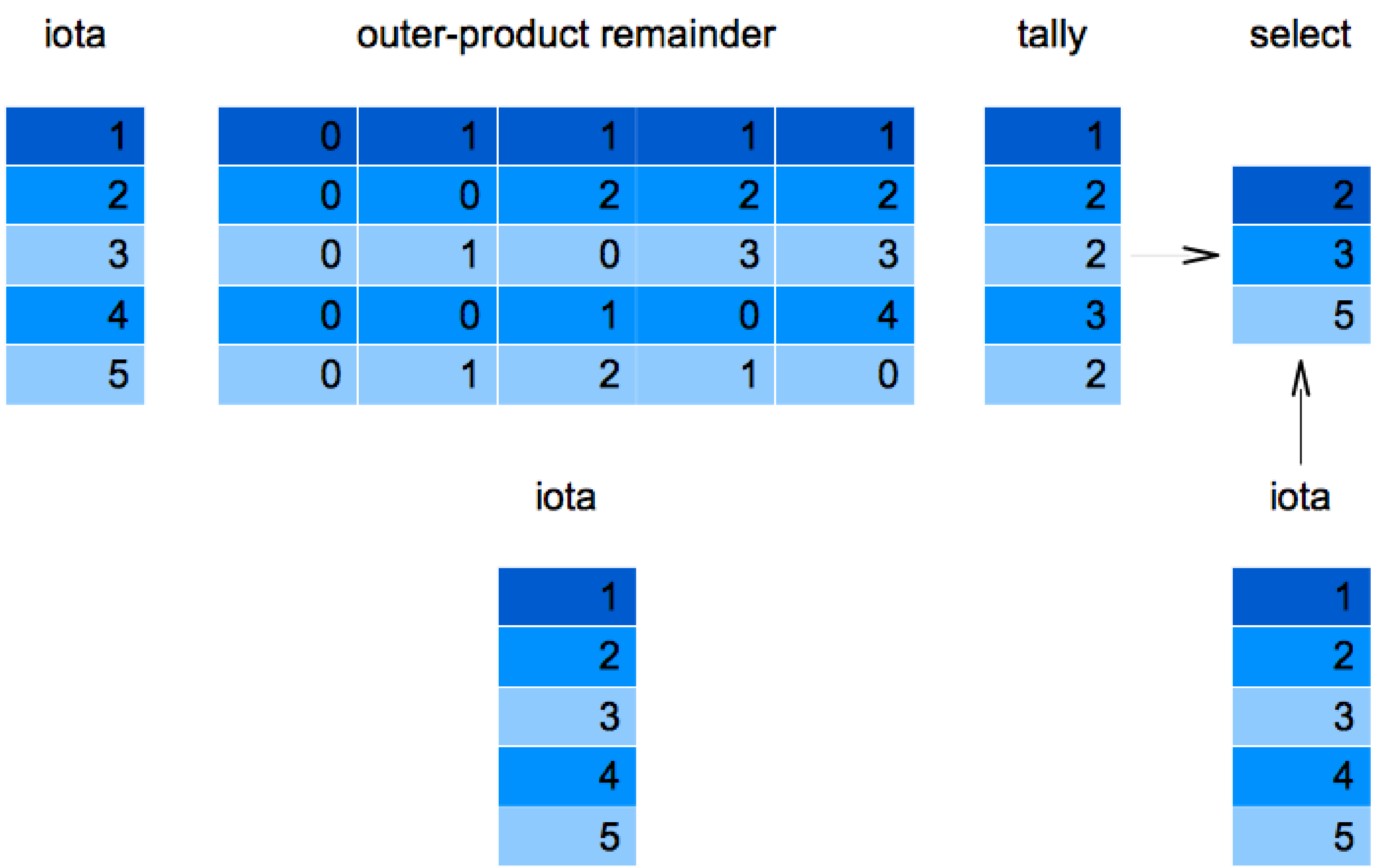
Outer Product

```
(define outer-product
  (lambda (proc)
    (lambda (us vs)
      (map (lambda (u)
        (map (lambda (v) (proc u v)) vs)) us))))

> ((outer-product cons) '(1 2) '(a b))
(((1 . a) (2 . a)) ((1 . b) (2 . b)))
```

Prime Numbers

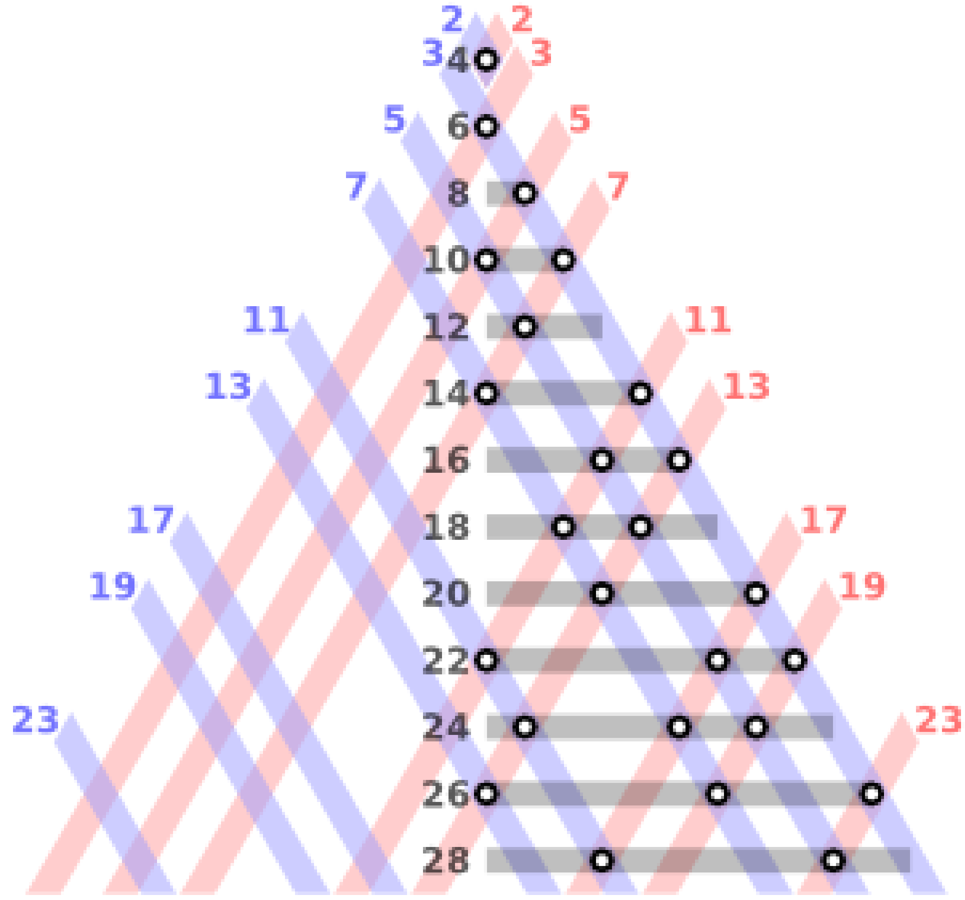
```
(define primes
  (lambda (n)
    (let ((ls (iota n)))
      ((select (lambda (x) (= x 2))
       (map (tally zero?)
            ((outer-product remainder) ls ls))
       ls))))

> (primes 10)
(2 3 5 7)
```

Prime Numbers Explained



Goldbach's Conjecture

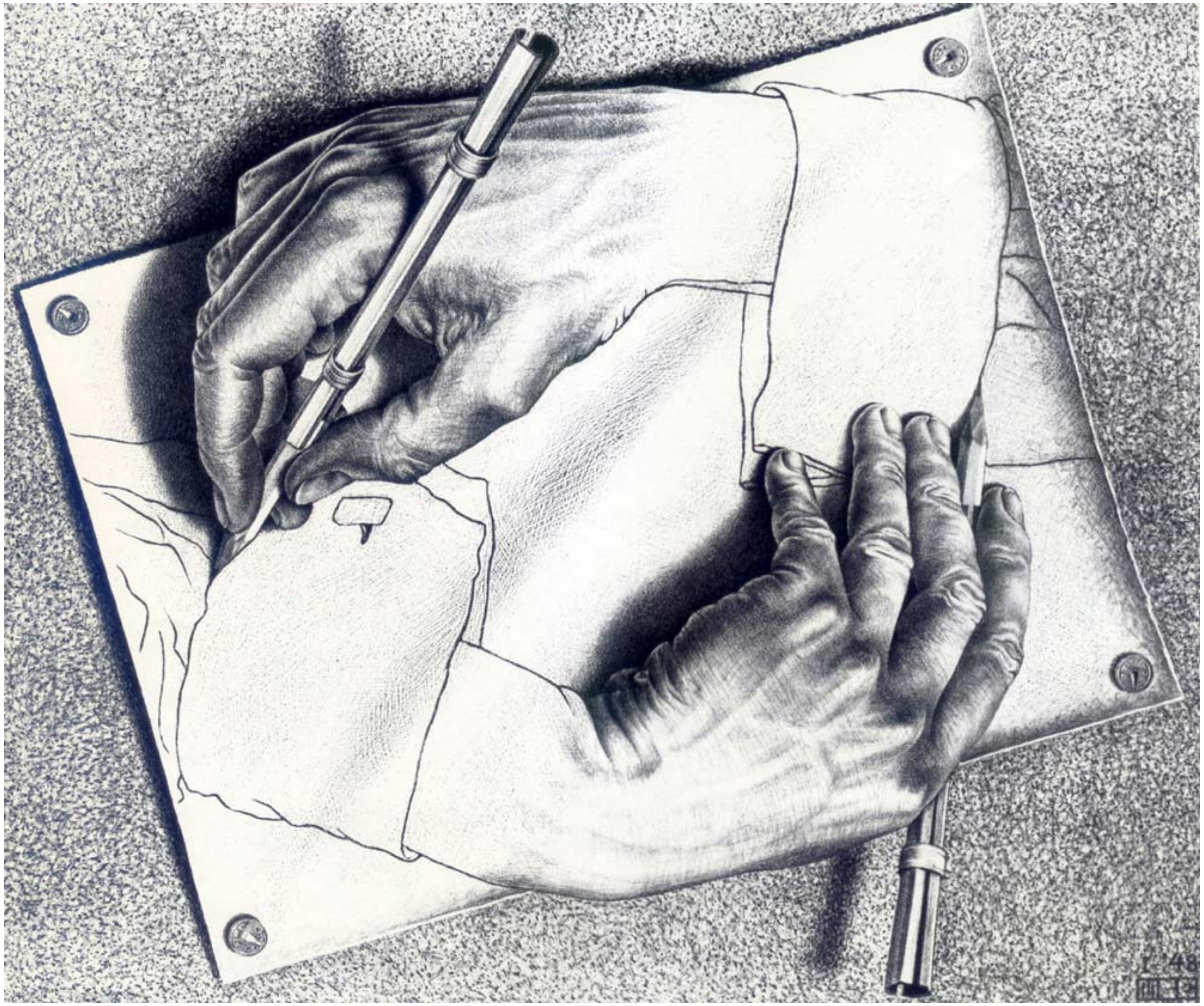
Every even integer greater than 2 can be expressed as the sum of two primes.



Goldbach's Conjecture

```
(define goldbach
  (lambda (n)
    (let ((primes (primes n)))
      (apply append
        (apply append
         (outer-product
           (lambda (x y)
             (if (= n (+ x y))
                 (list (list x y))
                 '()))
          primes
          primes))))))

> (goldbach 98)
((19 79) (31 67) (37 61) (61 37) (67 31) (79 19))
```

Quines



C Quine

```
char data[] = {35,105,110,99,108,117,100,101,32, ... };

#include <stdio.h>

main() {

  int i;

  printf("char data[] = {");

  for (i = 0; i < sizeof(data); i++) printf("%d,",data[i]);

  printf("};\n\n");

  for (i = 0; i < sizeof(data); i++) printf("%c",data[i]);

}
```

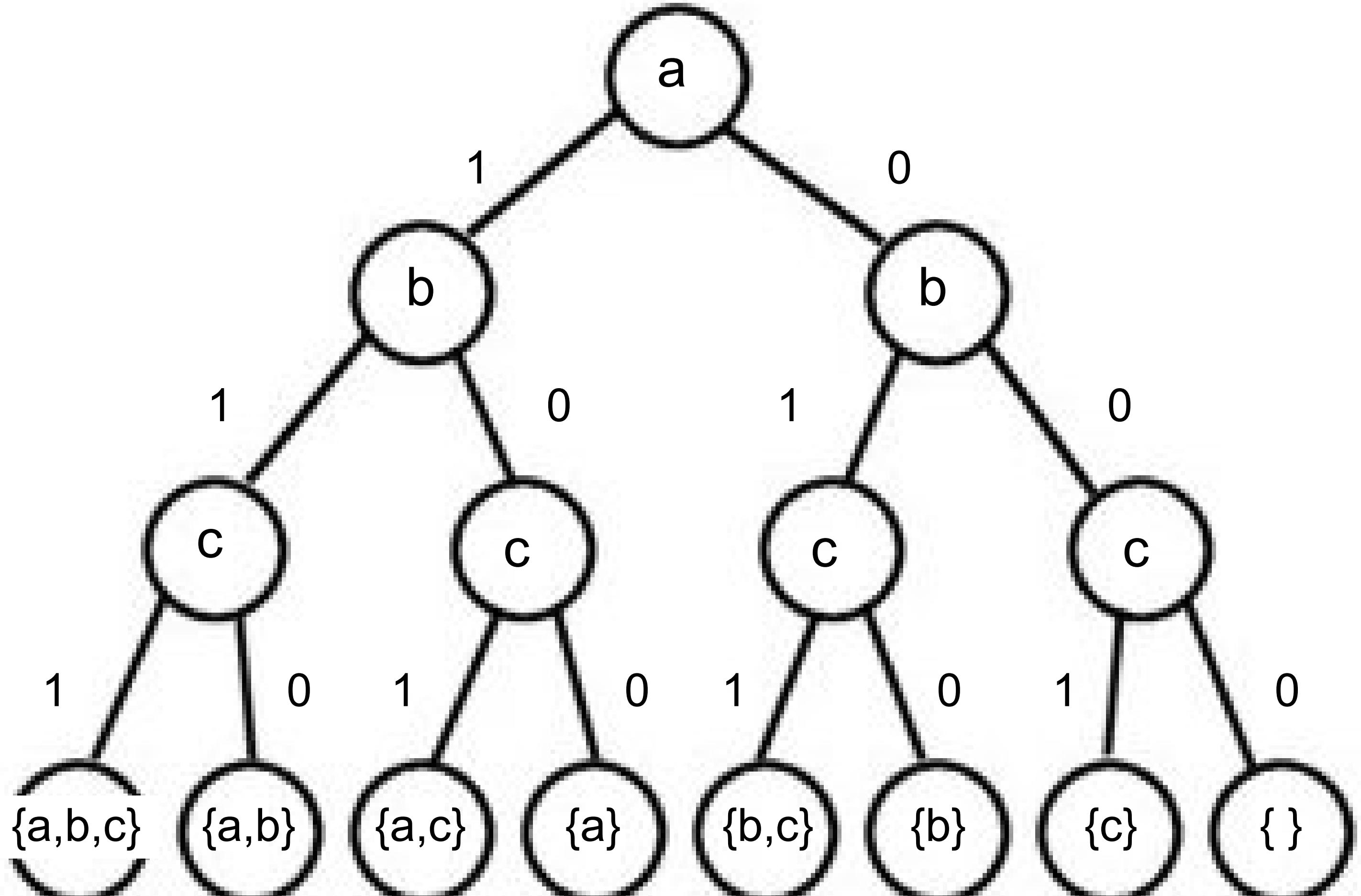
Scheme Quine

```
((lambda (x) (list x (list 'quote x)))
'(lambda (x) (list x (list 'quote x))))
```

APL Quine

```
1φ22ρ11ρ'''1φ22ρ11ρ'''
```

Powerset

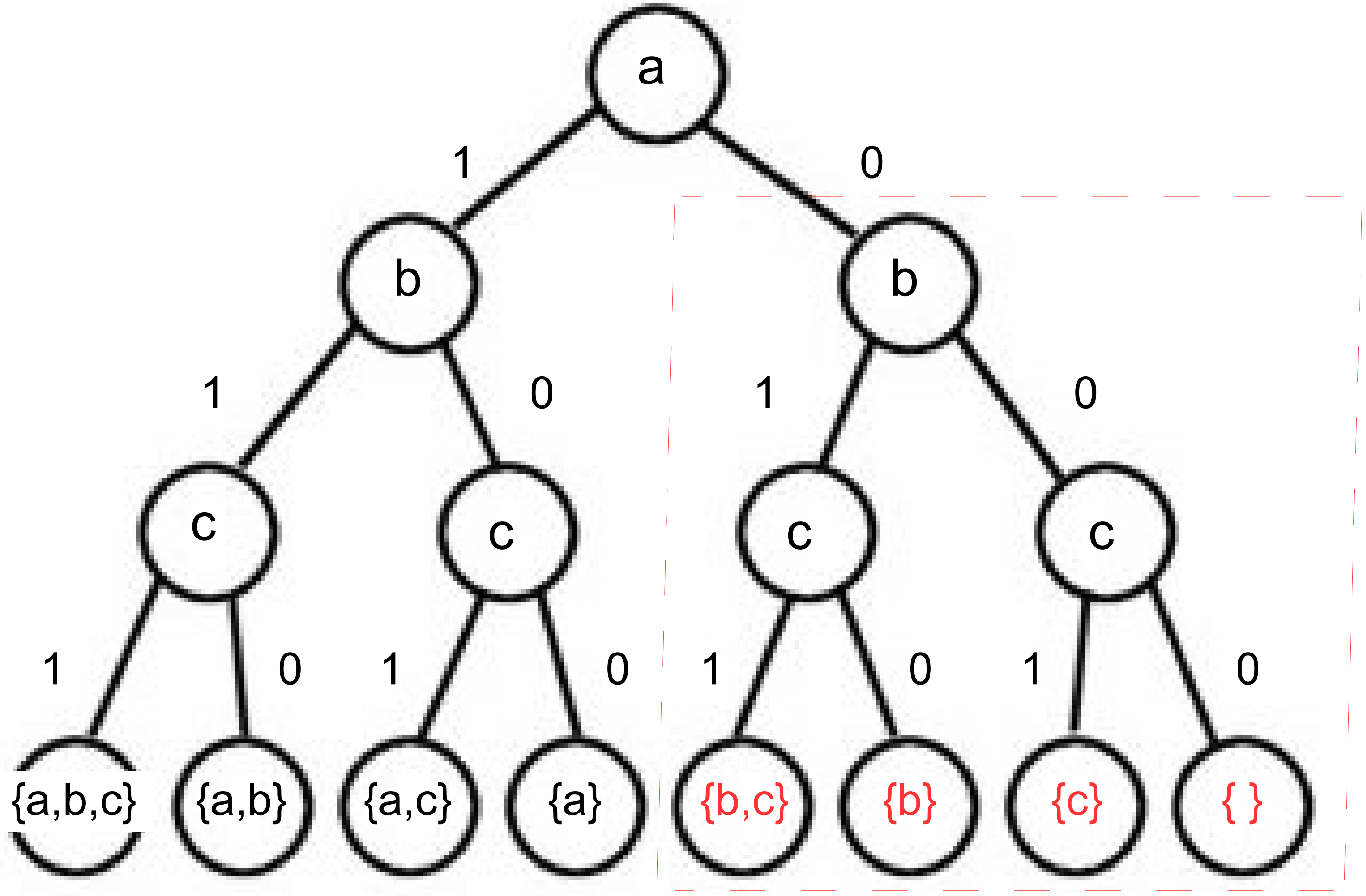


Results which Grow Fast

- Space complexity gives a lower bound on time complexity.
- A result of size $O(2^n)$ cannot be computed in less than $O(2^n)$ time!
- To grow this fast, a recursive function must call itself twice in every step.

Opening an Oyster

Powerset



Powerset

```
> (powerset (cdr '(a b c)))
((b c) (b) (c) ())

>(map (lambda (x) (cons 'a x))
      (powerset (cdr '(a b c))))
((a b c) (a b) (a c) (a))
```

```
(define powerset
  (lambda (xs)
    (if (null? xs)
        '(())
        (let ((half (powerset (cdr xs))))
          (append (map (lambda (x) (cons (car xs) x))
                       half)
                  half))))
```

Make Change

```
(define make-change
  (lambda (amount coins)
    (let ((ls (powerset coins)))
      (car (select
            (lambda (x) (= x amount))
            (map (lambda (ls) (apply + ls)) ls)
            ls))))))
```

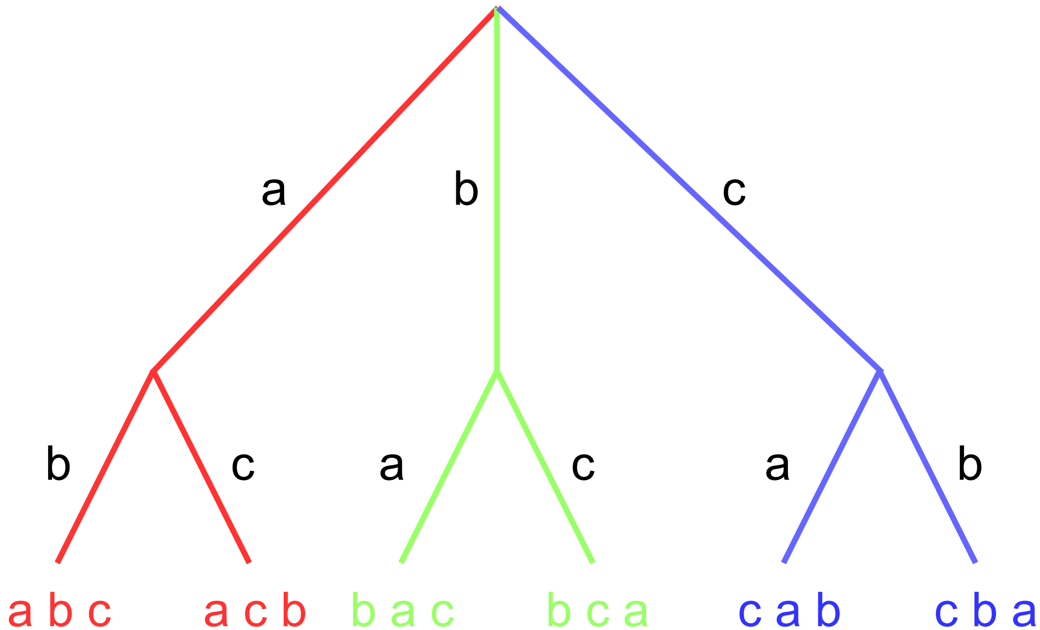
Make Change Explained

```
>(define half (powerset '(25 10 10 5 5 5 1 1 1)))
> half
((25 10 10 5 5 5 1 1 1)
 (25 10 10 5 5 5 1 1)
 (25 10 10 5 5 5 1 1)
    .
    .
    .
 (1 1)
 (1)
 (1 1)
 (1)
 (1)
 ())
```

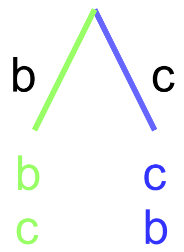
Make Change Explained

```
>(map (lambda (ls) (apply + ls)) half)
(63 62 62 61 62 61 61 60 58 57 ... 1 2 1 1 0)

>((select (lambda (x) (= x 57)))
  (map (lambda (ls) (apply + ls)) half)
  half)
((25 10 10 5 5 1 1)
 (25 10 10 5 5 1 1)
          .
          .
          .
 (25 10 10 5 5 1 1))
```

Permutations of {a,b,c}



Permutations

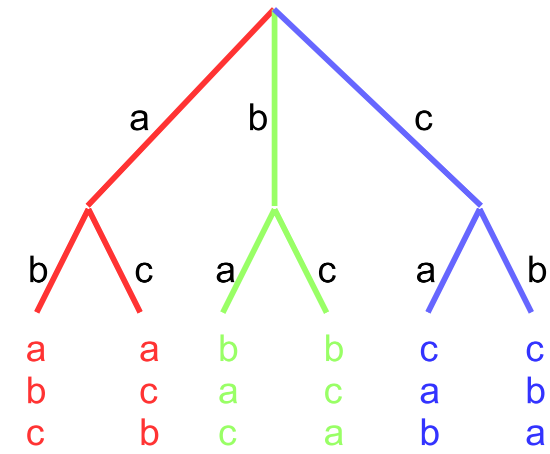



```
> (permutations '(b c))
((b c) (c b))

> (permutations (delete 'a '(a b c)))
((b c) (c b))

>(map (lambda (p) (cons 'a p))
      (permutations (delete 'a '(a b c))))
((a b c) (a c b))
```

Results which Grow Even Faster

- Space complexity gives a lower bound on time complexity.
- A result of size $O(n!)$ cannot be computed in less than $O(n!)$ time!
- To grow this fast, a recursive function must call itself n times in step n.
- It can only do this by mapping itself across a list of size n.

Permutations

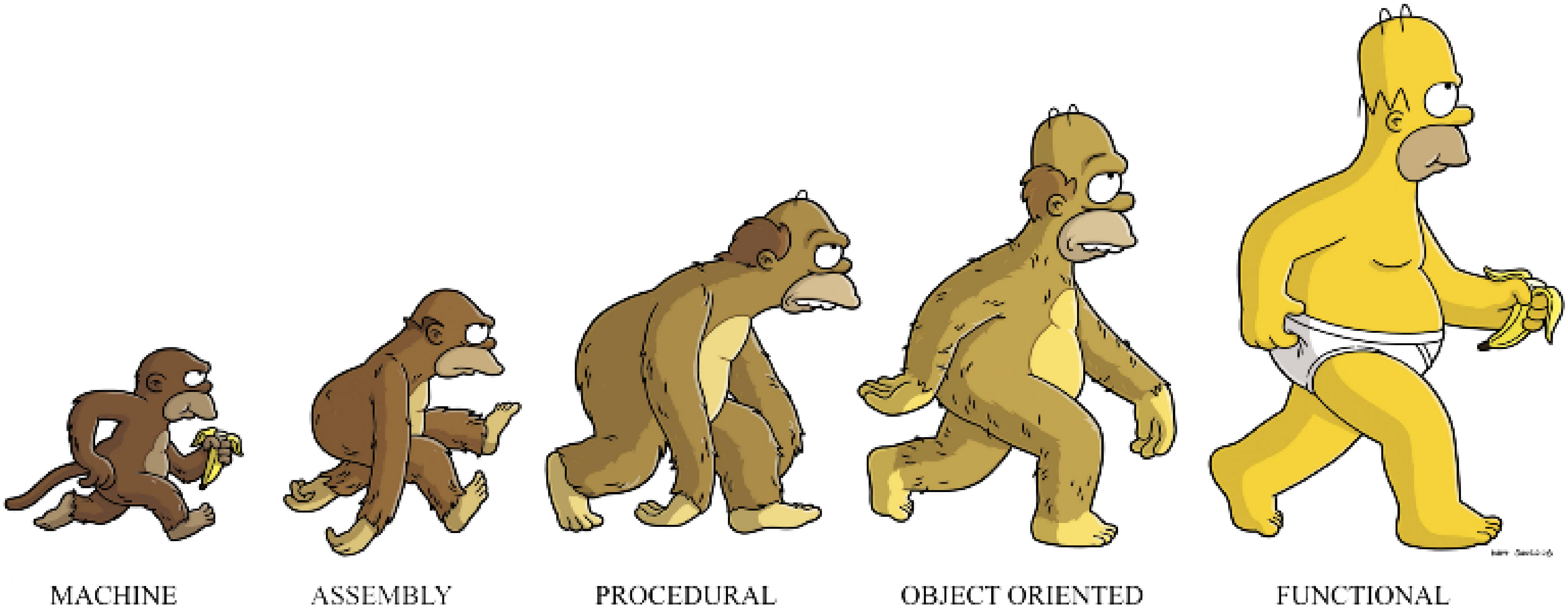



```
>(map (lambda (x)
       (map (lambda (p) (cons x p))
            (permutations (delete x '(a b c)))))
     '(a b c))
(((a b c) (a c b)) ((b a c) (b c a)) ((c a b) (c b a)))
```

```
> (apply append
   (map (lambda (x)
         (map (lambda (p) (cons x p))
              (permutations (delete x '(a b c)))))
       '(a b c)))
((a b c) (a c b) (b a c) (b c a) (c a b) (c b a))
```

Permutations

```
(define permutations
  (lambda (xs)
    (if (null? xs)
        '(())
        (apply append
          (map (lambda (x)
                 (map (lambda (p) (cons x p))
                      (permutations (delete x xs))))
               xs)
```


MACHINE
ASSEMBLY
PROCEDURAL
OBJECT ORIENTED
FUNCTIONAL