CS339: Abstractions and Paradigms for Programming

Sequences as Conventional Interfaces

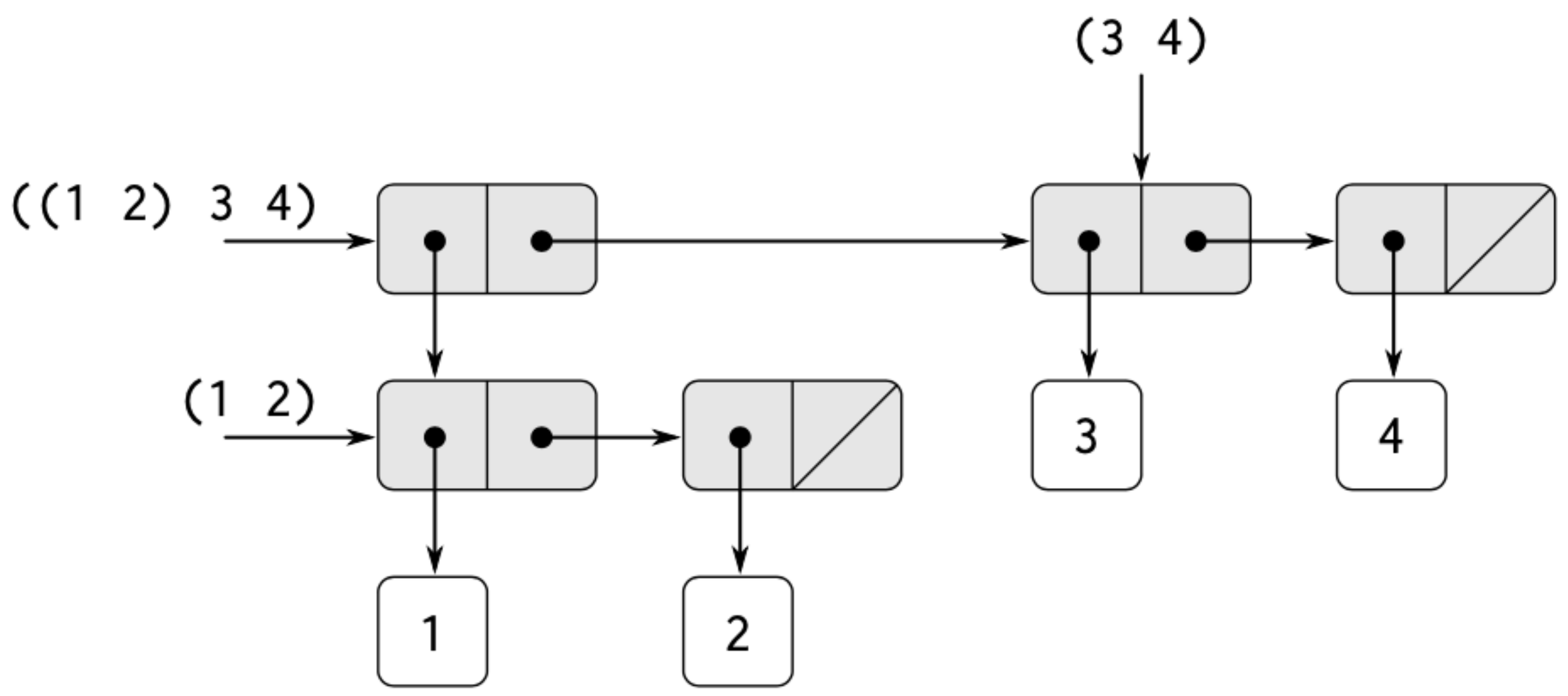
Manas Thakur
CSE, IIT Bombay

Autumn 2024

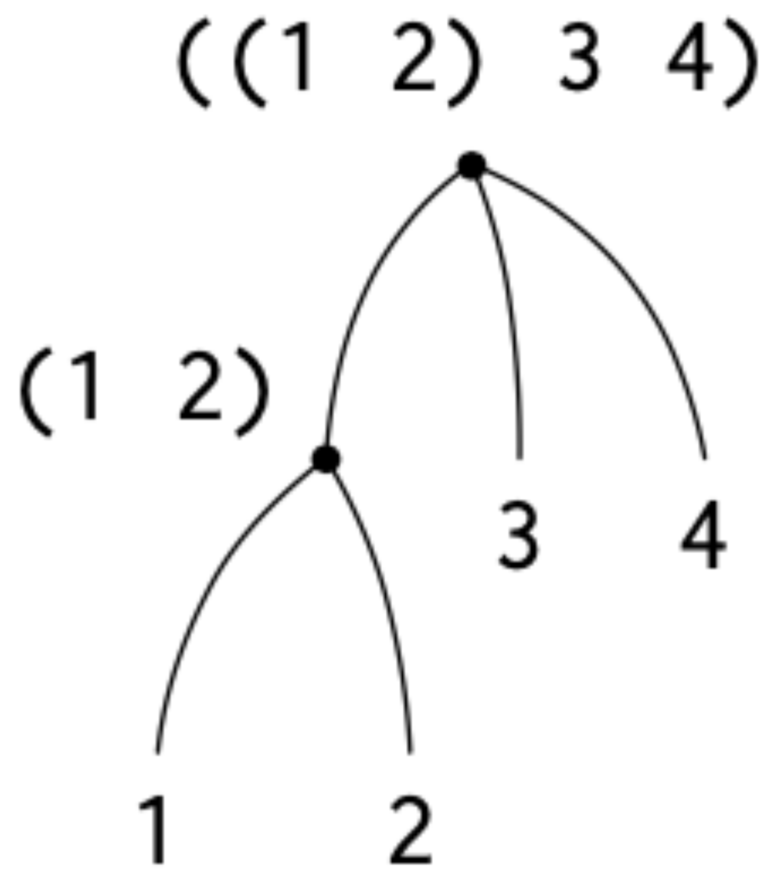
Let's get back to lists

- Our lists look very similar to trees:

```
(list (list 1 2) 3 4)
```



- What's the corresponding tree?



Sum the squares of the odd leaves of a tree

```scheme
(define (sum-odd-squares tree)
  (cond ((null? tree) 0)
        ((not (pair? tree))
         (if (odd? tree) (square tree) 0))
        (else (+ (sum-odd-squares (car tree))
                 (sum-odd-squares (cdr tree))))))
```

List the even fibonacci numbers till *fib(n)*

```scheme
(define (even-fibs n)
  (define (next k)
    (if (> k n)
        nil
        (let ((f (fib k)))
          (if (even? f)
              (cons f (next (+ k 1)))
              (next (+ k 1))))))
  (next 0))
```

Is there any similarity between these two programs?

Recall our higher order list procedures

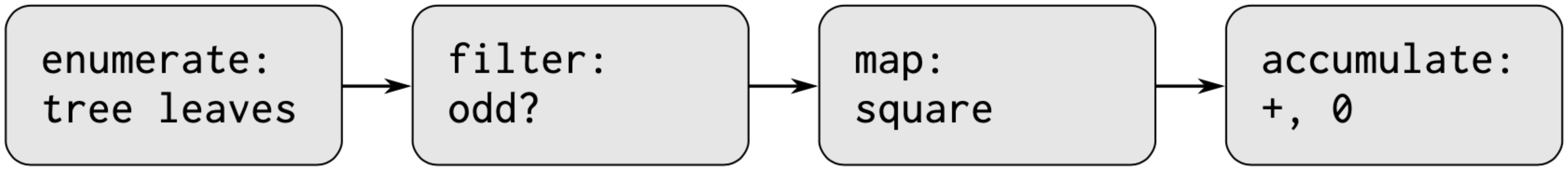
```
(define (map f l)
  (if (null? l)
    '()
    (cons (f (car l))
          (map f (cdr l)))))

> (define l '(1 2 3 4))
> (map (lambda (x) (* x x)) l)
> (1 4 9 16)
```

```
(define (filter pred l)
  (cond ((null? l) '())
        ((pred (car l))
         (cons (car l)
               (filter pred (cdr l))))
        (else (filter pred (cdr l)))))

> (filter odd? '(1 2 3 4 5))
> (1 3 5)
```

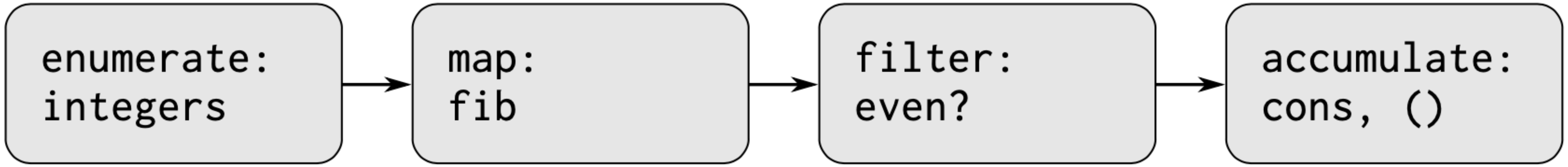
```
(define (accumulate op v l)
  (if (null? l)
    v
    (op (car l)
        (accumulate op v (cdr l)))))

> (accumulate + 0 '(1 2 3 4))
> 10
```

Same as *foldr*

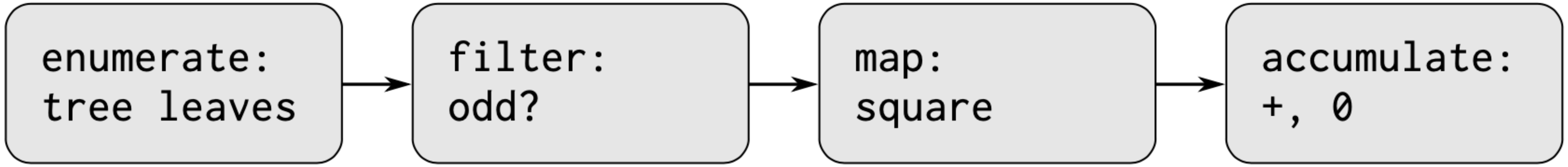
Sum the squares of the odd leaves of a tree



List the even fibonacci numbers till *fib(n)*

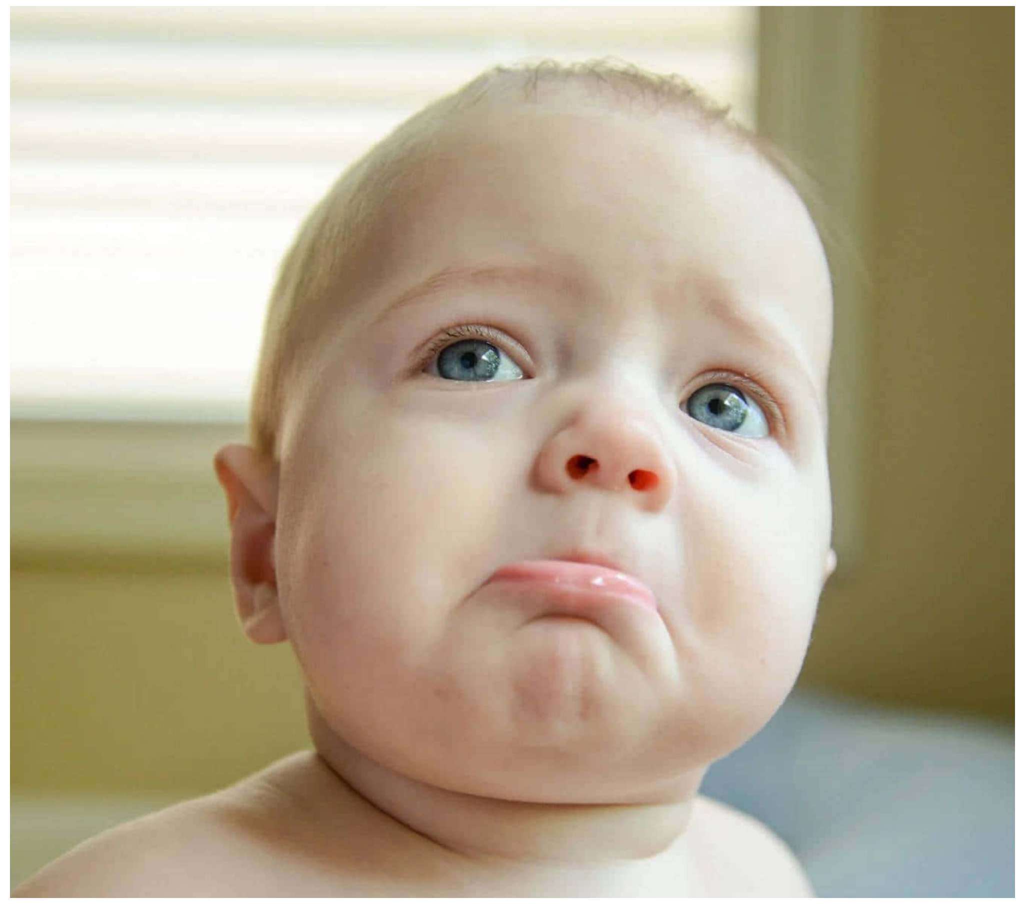


Sum the squares of the odd leaves of a tree




```scheme
(define (sum-odd-squares tree)
  (cond ((null? tree) 0)
        ((not (pair? tree))
         (if (odd? tree) (square tree) 0))
        (else (+ (sum-odd-squares (car tree))
                 (sum-odd-squares (cdr tree))))))
```

Don't match!

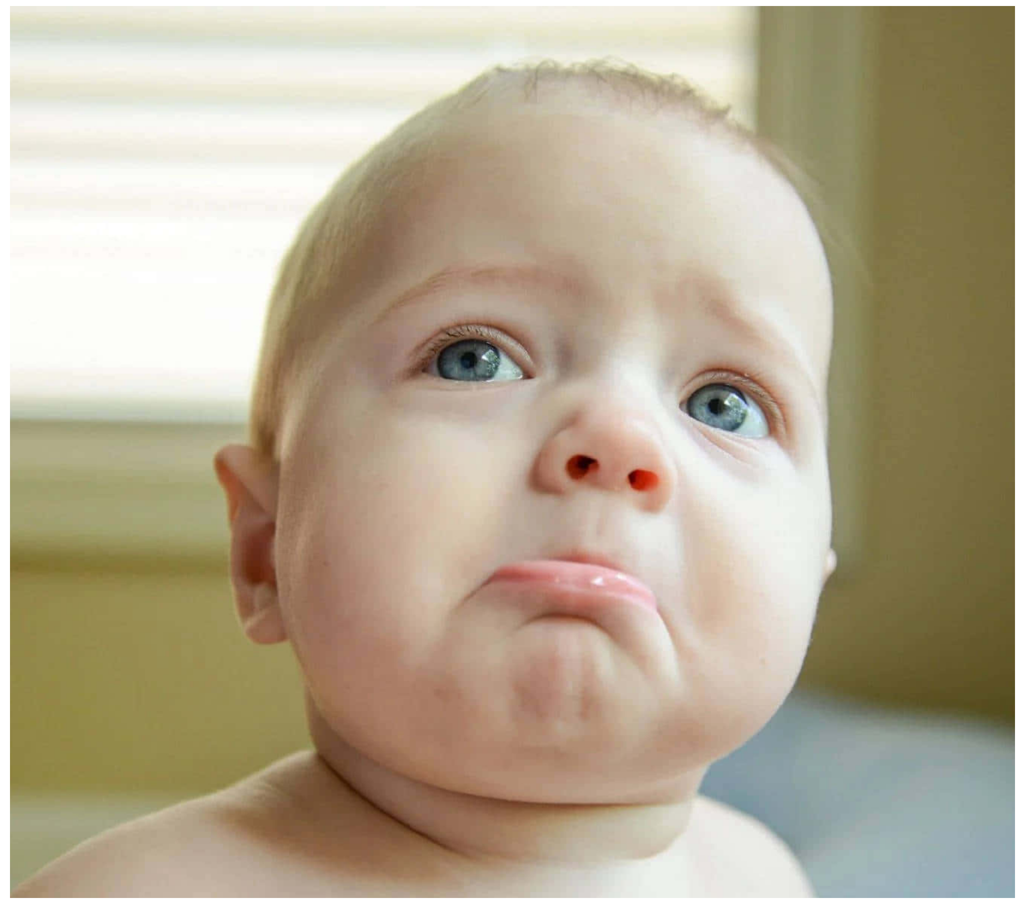


List the even fibonacci numbers till *fib(n)*

enumerate: integers → map: fib → filter: even? → accumulate: cons, ()

```
(define (even-fibs n)
  (define (next k)
    (if (> k n)
        nil
        (let ((f (fib k)))
          (if (even? f)
              (cons f (next (+ k 1)))
              (next (+ k 1))))))
  (next 0))
```

Don’t match again!



Enumerators

```
(define (enumerate-interval low high)
  (if (> low high)
    '()
    (cons low (enumerate-interval (+ low 1) high))))

> (enumerate-interval 1 5)
> (1 2 3 4 5)
```

```
(define (enumerate-tree tree)
  (cond ((null? tree) '())
        ((not (pair? tree)) (list tree))
        (else (append (enumerate-tree (car tree))
                      (enumerate-tree (cdr tree))))))

> (enumerate-tree (list 1 (list 2 (list 3 4)) 5))
> (1 2 3 4 5)
```

Using Sequences as Interfaces

Sum the squares of the odd leaves of a tree

enumerate: tree leaves → filter: odd? → map: square → accumulate: +, 0

```
(define (sum-odd-squares tree)
  (accumulate
   + 0 (map square (filter odd? (enumerate-tree tree)))))
```

List the even fibonacci numbers till *fib(n)*

enumerate: integers → map: fib → filter: even? → accumulate: cons, ()

```
(define (even-fibs n)
  (accumulate
   cons
   nil
   (filter even? (map fib (enumerate-interval 0 n)))))
```

A PL for specifying computations using sequences

- What constitutes a programming language?
  - Primitive expressions; means of combination; means of abstraction
- We have created a new **programming language** that defines *sequences as conventional interfaces*:
  - Sequences (implemented using lists) are like signals flowing from one stage to another
  - `map`, `filter` and `accumulate` (the new vocabulary of our language) represent general patterns of processing sequences
  - **Advantage:** Largely decoupled modules defined in terms of operations on sequences
    - Key to reducing complexity of large software

Modularity in full glow

- List the squares of the first n+1 fibonacci numbers:

```
(define (list-fib-squares n)
  (accumulate
   cons
   nil
   (map square (map fib (enumerate-interval 0 n)))))
```

- Multiply the squares of the odd integers in a sequence:

```
(define (product-of-squares-of-odd-elements sequence)
  (accumulate * 1 (map square (filter odd? sequence))))
```

- Can you see a BIG problem with our elegant modular programs?

Checking if a number is prime

- Find the smallest divisor (>1) of a given number n
- If the divisor is the same as n, then n is prime

```
(define (smallest-divisor n) (find-divisor n 2))

(define (find-divisor n test-divisor)
  (cond ((> (square test-divisor) n) n)
        ((divides? test-divisor n) test-divisor)
        (else (find-divisor n (+ test-divisor 1)))))

(define (square x) (* x x))

(define (divides? a b) (= (remainder b a) 0))

(define (prime? n)
  (= n (smallest-divisor n)))
```

Add all prime numbers in an interval

- Mixed up non-modular program:

```
(define (sum-primes a b)
  (define (iter count accum)
    (cond ((> count b) accum)
          ((prime? count)
             (iter (+ count 1) (+ count accum)))
          (else (iter (+ count 1) accum))))
  (iter a 0))
```

- Nicely separable modular program:

```
(define (sum-primes a b)
  (accumulate +
              0
              (filter prime?
                      (enumerate-interval a b))))
```

Second prime between 10000 and 1000000

Very very inefficient!

```
(car (cdr (filter prime?
                  (enumerate-interval 10000 1000000))))
```

- **Where are we headed next?**
  - A way to make our nice and elegant modular programs as efficient as their non-modular counterparts!
  - A way to model the real world without the cons of assignments!
  - A world in which substitution model still works!
  - A new way of viewing life!

- **Stream processing.**