30/7/24

Programming Languages →
1) Syntax defined by a CFG
2) Semantics executed by an interpreter / compiler

Design: Semantics tells meaning of each construct enabled by grammar

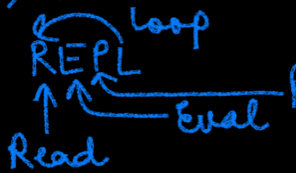
→ Implementation

FORTRAN: Math / Scientific
LISP: AI, Academic
COBOL: Business logic
PL I: Low-level systems prog.

Scheme is a version of LISP (List Processing)
Evaluator: DrRacket

Elements of PL: 1) Primitives, 2) Combinations, 3) Abstraction

How do interpreters work?




```
> 2
2
>
// waits here
```

```
> 2          > "manas"      > +
2            "manas"        <procedure: +>
> 3          Numbers        > *
3            decimals       <  "  *>
> 17.5       strings
17.5
```

Primitives

Combination?
Syntax: (<operator> <operand1> ... <operand n>)

Prefix notation

> (+ 2 3) ✓

> (2 + 3) ✗ — 2 not a procedure

Parentheses are compulsory. Every combination in parentheses;

```
> (+ (* 3 4) (- 10 5))
17
```

First operands are evaluated then procedure is applied

```
> (+ (* 3 (* 4 6)) 6)
78
```

Advantage of Prefix notation ⇒
① Multiple operands (7, 2)
② Simpler semantics

"define" keyword ⇒ Defining variables or procedures

```
> (define x 2)
> (define y (+ x 1))
> (define (add x y) (+ x y))
```

"add" is a procedure
(All procedures in scheme must have a return value)

Prefix notation

```
> (add 2 6)
8
```

```
> +                    > add
<procedure:+>          <procedure:add>
```

No distinction in primitive or user-defined procedures

```
>(define (square x) (* x x))
>(square (square 2))
16
>(define add (lambda (x y) (+ x y)))
```

Substitution model of Evaluation →

Substitution used rather than assignment

(define is not assignment in C++) rather it is substitution, like #define

```
> (define add +)
> (add 2 3 4 5 6)
20
```

```
> (define + *)
> (+ 2 3)
6
> (define define +)
> (define 2 3)
5
```

01/08/24

'foo' is just a name to a lambda

In first method, we blindly substituted first & then do any calculations

In second method of substitution, we evaluated arguments then used them

Applicative order → avoids multiple computation, takes lesser memory

To see if a PL uses applicative order / normal one,

We can use this ⇒ (define (f x y) (+ 2 y))

```
> (f (/ 10 0) 5)
```

Applicative order will give error

Normal one will not.

Applicative is called 'call by value'

Normal " "call by name"/"need"

Conditionals 'cond' a special form

```
(define (abs x)
  (cond ((> x 0) x)
        ((= x 0) 0)
        ((< x 0) (- x))))
```

Predicate Action

((> x 0) x)

Clause

OR
```
(define (abs x)
  (cond ((< x 0) (- x))
        (else x)))
```

OR
```
(define (abs x)
  (if (< x 0)
      (- x)
      x))
```

cond

Consequent

Alternative

5/8/24

```
(define (fact x)
  (if (= x 1)
      1
      (* x (fact (- x 1)))))
```

$$fact(n) = \begin{cases} 1, & n=1 \\ n*fact(n-1), & n>1 \end{cases}$$

Recursive Procedure

Recursive Process

We have to remember the future computations we have to do & then all complete quickly

Iterative factorial func^{n} →

```
(define (fact-iter prod ctr n)
  (if (> ctr n)
      prod
      (fact-iter (* ctr prod)
                 (+ ctr 1)
                 n)))

(define (fact n)
  (fact-iter 1 1 n))
```

Recursive Procedure

Iterative Process

Each step is building up our answer

```
(define (Fibonacci n)
  (cond ((= n 0) 0)
        ((= n 1) 1)
        (else (+ (Fibonacci (- n 1)) (Fibonacci (- n 2))))))
```

Tail optimization ⊢ So space is O(1) for iterative process

6/8/24

"define", "if", "cond" are not procedures but rather special forms

```
(if (< x 10)
    4
    (/ 1 0))
```
Doesn't give error despite applicative form

```
(define (foo y)
  (+ x y))
```
bound variable
free variable

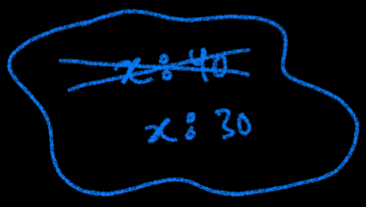
Scoping

Lexical scoping — Check where it was defined

Dynamic scoping — Check where it was called

```
> (define x 40)
```
This is not a assignment in some memory location

```
> (define x 30)
```
This rather saves the "mapping" from "x" to 30 in the global scope



If we do
```
> (define (foo)
    (define x 20)
    ....)
```
This does not destroy the "x: 30" map in global scope. It creates a "x:20" map in the scope of "foo" procedure.

```
(define (sum-ints a b)
  (if (> a b)
      0
      (+ a (sum-ints (+ a 1) b))))
```
Summation

$$\sum_{i=a/\text{by}1}^{b} i$$

```
(define (<name> a b)
  (if (> a b)
      0
      (+ <term a> (<name> <next a> b))))
```

```
(define (sum-series term next a b)
  (if (> a b)
      0
      (+ (term a)
         (sum-series term next (next a) b))))
```

12/8/2024

① Core of all functional langs, ② Useful for specifying semantics → type system

Program is a CFG. E.g. $M \rightarrow x$ ① (term) (variable)

$\mid \lambda x.\ M$ (abstraction) ②

$\mid M_1\ M_2$ (application) ③

$\mid (M)$ {4th form for simplification}

$\mid$ constants

$\lambda x.\ M$ — x is a parameter to the function M

$M_1\ M_2$ — M_1 must be a function so we can apply it

Anything possible using Turing machine is possible with λ-calculus and vice versa

$\lambda x.\ x \equiv$ (lambda (x) x) {Identity function}

$\lambda x.\ x+1$ We didn't define '+' or '1'

M_1 M_2

$(\lambda x.\ x+1)\ 2$ (Applying to 2)

How to read multiple arguments? $\lambda x, y.\ x+y$

(define (add x y) (+ x y))

$\equiv$ (define add (lambda (x y) (+ x y))

How to show it as a single argument?

$\equiv$ (define add (lambda (x)
(lambda (y) (+ x y))))

$\equiv \lambda x.\lambda y.\ x+y$

$(\lambda x.\lambda y.\ x+y)\ 2\ 3$

$\Rightarrow (\lambda y.\ 2+y)\ 3$ Outer lambda "consumes" 2

$\Rightarrow 2+3$

Some rules ⇒ ① Application binds tighter than abstraction

$\lambda x.\ xy = \lambda x.\ (x\ y)$ ✓

$\neq (\lambda x.\ x)\ y$ ✗

② Application associates to left

$xyz = (x\ y)\ z \neq x\ (y\ z)$ ✗

Substitution model follows β-reduction ⇒

$(\lambda x.\ M)\ N \Rightarrow_\beta [N/x]\ M$

$(\lambda x.\ x+1)\ 2 \Rightarrow_\beta [2/x]\ x+1$

$= 2+1$

E.g. $(\lambda x.x)\ \underbrace{((\lambda x.x)\ (\lambda z.(\lambda x.x)\ z))}_{M_2}$

$\underbrace{(\lambda x.x)}_{M_1}$

$=_\beta\ (\lambda x.x)\ \underbrace{(\lambda z.(\lambda x.x))\ z)}_{M_2}$ $\quad =_\beta\ (\lambda z.\ \underbrace{(\lambda x.x)\ z})$ Application before abstraction

$=_\beta\ \underbrace{\lambda z.z}$ Normal form (those which can't be further simplified by β-red^n)

E.g. $((\lambda xyz.\ x\ z\ (y\ z))\ (\lambda x.x))(\lambda x.x)$

Application associates left

$=_\beta\ \lambda yz.\ (\lambda x.x)\ z\ (y\ z)\ (\lambda x.x)$ → leftmost outermost redex (Call by name)

$=_\beta\ \lambda z.\ z\ ((\lambda x.x)\ z)\ =_\beta\ \lambda z.zz$

Another way ⇒

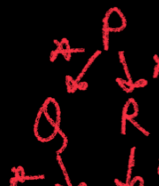
$\lambda yz.\ (\lambda x.x)\ z\ (y\ z)\ (\lambda x.x)$ → leftmost innermost redex (call by value)

$=_\beta\ \lambda yz.\ z\ (y\ z)\ (\lambda x.x)\ =_\beta\ \lambda z.\ z\ ((\lambda x.x)\ z)\ =_\beta\ \lambda z.zz$

↳ Outer lambda y returns a lambda z to "z applied (y z)"

Why? as application comes before abstraction & the leftmost application after · is "z (y z)".

Church-Rosser Theorem ⇒



(different valid ways of reduction)

(Same normal form)

Normal forms are UNIQUE IF THEY EXIST

Example where Normal Form didn't exist ⇒

$(\lambda x.xx)(\lambda x.xx)$

$=_\beta\ (\lambda x.xx)(\lambda x.xx)$

(Infinite recursion)

13/8/24 **Lambda calculus (Contd.)**

zero: (lambda () '()) (succ x): (lambda () x) one: (succ zero)
(pred x): (x)

Defining such rules is "Programming based on axioms"
Only way of reducing lambda expressions is β-red^n.

$(\lambda y.\ x+y)\ x \ =_\beta$ ⊗ $= 2*x$ ✗

x: free (Check ENV for its mapping)
y: bound
⊗: bound this free variable erroneously!

Hence we need to perform α-conversion before blindly using β-red^n

$\lambda x.x \equiv \lambda z.z \equiv \lambda u.u$ (we can rename the local variable freely)

So, $(\lambda y.\ x+y)\ x \equiv (\lambda y.\ z+y)\ x \ =_\beta\ z+x$

M → x
| λx.M
| M₁ M₂
| (M)
| C
} syntatic sugar

C = Constants: Those whose definitions don't change
0 → zero (alias)
+ : add-church
} ∈ 'C'

Let's define boolean (for if cond^n) →

if true M N ⇒ M
if false M N ⇒ N

We can define them like :-

true: λxy.x	≡ λx.λy.x
false: λxy.y	≡ λx.λy.y

if ((λxy.x) M) N
App binds left
$=_\beta$ ((λy.M) N) $=_\beta$ M
apply

Similarly $(\lambda xy.y)\ M\ N =_\beta N$

They can be used for AND, OR operations ⇒

OR x y ≡ λxy if x true y

{ If x is true, return true else return y }

E.g. → (λxy if x true y) true false
$=_\beta$ (λy if true true y) false
$=_\beta$ (λy true) false $=_\beta$ true

(λxy if x true y) f t
$=_\beta (\lambda y.\ y)\ t$
$=_\beta$ t

Define AND x y :
λxy. if x y false

Getting recursion in λ-calc ⇒ Y-combinator ← closed combinator (without free variables)

$$Y = \lambda f.(\lambda x.\, f(x\ x))\ (\lambda x.\, f(x\ x))$$

Applying the combinator ⇒

$$YF =_\beta (\lambda x.\, F(x\ x))\ (\lambda x.\, F(x\ x)) =_\beta F((\lambda x.\, F(x\ x))\ (\lambda x.\, F(x\ x)))$$
$$=_\beta F(YF)$$

F: some function

Put some base case ⇒ E.g. Factorial program

$$F = \lambda f.\lambda x.\, \text{if } (x=0) \text{ then } 1 \text{ else } x \times f(x-1)$$

→ F takes 2 arguments one f & other x

$$YF\,3 =_\beta ((\lambda x.\, \text{if } (x=0) \text{ then } 1 \text{ else } x * f(x-1))\ 3$$
$$= F(YF)3$$
$$=_\beta 3 * (YF\,2) = 3 * (F(YF)2)$$
$$=_\beta 3 * 2 * (YF\,1) \ldots = 3 * 2 * 1 = 6$$

$YF\,3$
$\equiv (YF)\,3$
(App binds left)
$=_\beta F(YF)\,3$

YF supplied to F

Gives $3 * (YF\,2)$

& continues

19/8/24

Rational numbers comprises of 2 things "n/d". Define +, -, *, / procedures

$\frac{2}{3}$ & $\frac{4}{6}$ are different as they have different numer, denom.

We make a separate abstraction "rat" for rational numbers.

If we don't have this abstraction then we will have to remember somewhere separately

'make-rat' is like a constructor and 'numer' and 'denom' are like selectors.

Procedures are closures hence can be used for abstraction in 'make-rat'

```
(define (make-rat n d)
  (lambda (choose)
    (if (= choose 0)
        n
        d)))
```

```
(define (numer x)
  (x 0))
(define (denom x)
  (x 1))
```

Data abstraction helps in definitions of '*_rat' functions being same irrespective of implementation of rational numbers we do.

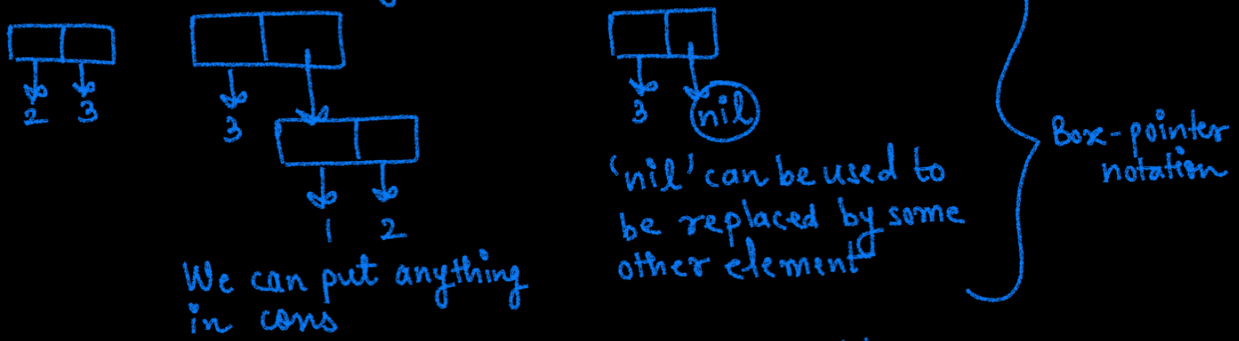
(cons x y) — in-built pair making func^n

(car x) ≡ x.first

(cdr x) ≡ x.second

```
(define (make-rat n d)
  (cons n d))
```

'nil' represents 'nothing' in scheme (like NULL in C++)



We can put anything in cons

'nil' can be used to be replaced by some other element

} Box-pointer notation

> (define n (cons 1 (cons 2 (cons 3 4)))) → n = (1 2 3 . 4)

> (car n) → 1, (cdr n) → (2 3 . 4)

> (car (cdr n)) → 2, > (cdr (cdr n)) → (3 . 4)

≡ (cadr n) (abbreviation) ≡ (cddr n)

> (caddr n) → 3 (cdddr n) → 4

← right to left evaluation)

(cddddr n) → Data type error (4 is not a pair)

(cdddddr n) → Undefined (max 4 d/a wale c****r defined)

'scheme' came from LISP hence it must've lists

> (list 2 3 4) → (2 3 4)

> (cons 2 (cons 3 4)) → (2 3 . 4) ← This dot says that 3 & 4 were together in a cons

> (cons 2 (cons 3 (cons 4 nil))) → (2 3 4) nil is not printed & we got rid of the dot

```
> (define l (list 1 2 3 4))
> (cdddr l) → (4)
> (cddddr l) → () ← nil
> (caddr l) → 4
> (car (cddddr l)) → contract violation
```

20/8/24

nth element of list — (1st element) == (car lst)

sum of ALL element of list → sum += ith element + sum of ALL in rem list

First n elements → ith element + (n-i) elements from rem list

Insertion sort →

```
(define (insert num lst)
    (cond ((null? lst) (cons num nil))
          ((<= num (car lst)) (cons num lst))
          (else (cons (car lst) (insert num (cdr lst))))))

(define (isort lst)
    (if (null? lst)
        nil
        (insert (car lst) (isort (cdr lst)))))
```

≡ (list num)

Trees → BST

```
(define (make-tree datum left right)
    (list datum left right))
```

(datum t) → (car t)

(left t) → (cadr t)

(right t) → (caddr t) ~~(cddr t)~~

as (cddr t) is (cons _ nil)

22/8/24


```
(define (foldr f v l)
  (if (null? l)
      v
      (f (car l) (foldr f v (cdr l)))))
```

```
(define (sum l)
  (foldr + 0 l))
```

E.g. l ← (list 2 3 4 5)
↳ (+ 2 (foldr + 0 '(3 4 5)))

(list a b c d) ≡ '(a b c d) is another representation.

```
(define (length l)
  (foldr (lambda (x y) (+ 1 y))
         0 l))
```

foldr = fold 'right', we start folding the list from right.
foldl = fold 'left', " left

foldr is a recursive process, foldl is an iterative process!

```
(define (sum-of-sq-of-even ans l)
  (if (null? l)
      ans
      (if (is-even (car l))
          (sum-of-sq-of-even (+ ans (sq (car l))) (cdr l))
          (sum-of-sq-of-even ans (cdr l)))))
```

Sum of squares of even numbers of a list

map/filter/reduce


```
(define (>10 l)
  (if (null? l)
      nil
      (if (> 10 (car l))
          (cons (car l) (>10 (cdr l)))
          (>10 (cdr l)))))
```

Only > 10 elements → predicate

```
(define (sqr-even l)
  (map square (filter even? l)))
```

```
(define (sum-sqr-even l)
  (foldr + 0 (map ----- )))
```

We are making new lists, but in old one, we did not make new list, just iterated.

27/8/24 OOP

Object? group multiple items into a single abstraction
multiple of them of a certain kind (E.g. dogs & not 4-legged mammal)
operations are abstracted
Distinguishing too

In scheme, use lambdas to make structures (make-rat) { (make-rat 2 3) (make-rat 4 5) Parametrized constructor
(add-rat), (multi-rat) are operations to the class

Difference b/w the implementations:-

(multi-rat x y)

Passing 2 arg & function is doing the work

x. multi-rat (y)

↓ acts as a receiver for y & is implicit to x, x is reacting to this input.

Subtle point?

Complex numbers → Two representations: Rectangular, Polar, Euler etc

Rectangular → Good for add/sub
Polar → Good for multi
} Have both representations!

Operations shouldn't depend on representations
Make the selectors see what representation is used & then do the suitable method.

Different selectors with same name!

29/8/24

Add a suffix?

Attach a tag ('polar' 'rectangular')

eq? → Checks equality

'rectangular 'polar

~~strings~~ quoted text

Bad about tags?

- Can't add more representations
- Any implementation of operations need to know about these tags

Message-passing?

Objects tell what needs to be done

3/9/24

```
(lambda (x) (lambda (y) (* x y)))
```

This outer lambda binds x

Scope of x

Scope of y

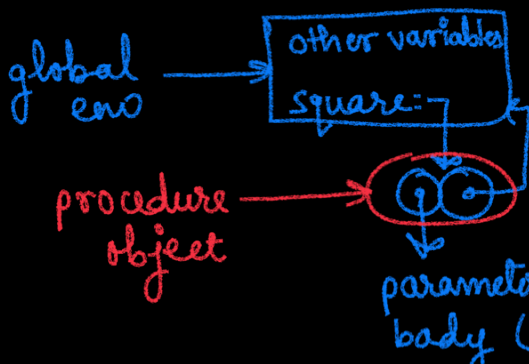
Each frame stores bindings & environment is a chain of those.

To find the binding for a name, check each frame in the chain of environment

Undefined, unbounded are same

If found, we stop & don't proceed further search.

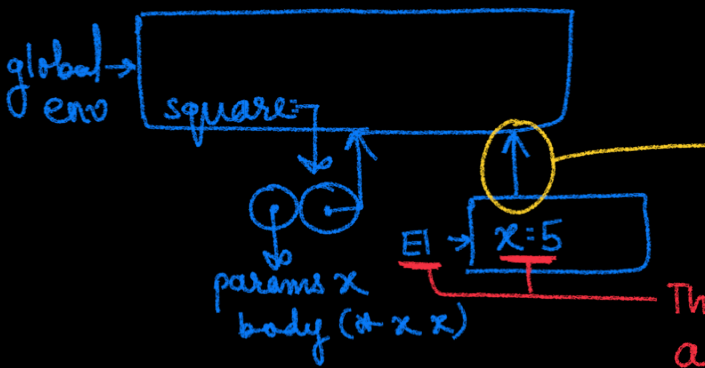
```
(define (square x) (* x x))
```




```
class ProcObj{
    List params
    List body
    Env ptr
}
```

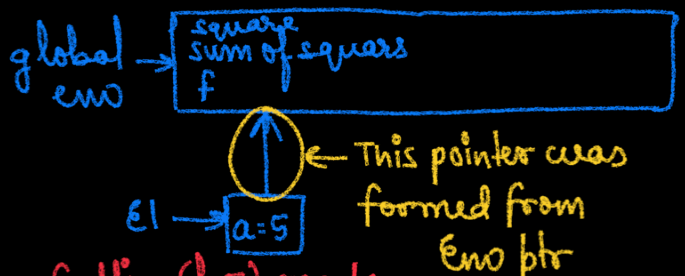
One ptr

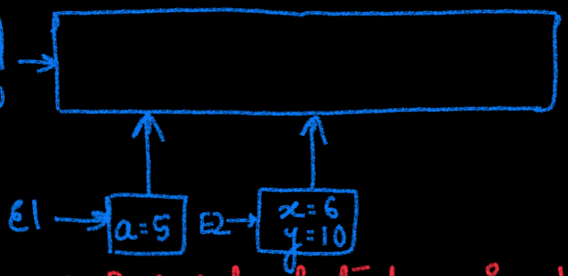
What happens during procedure applications →

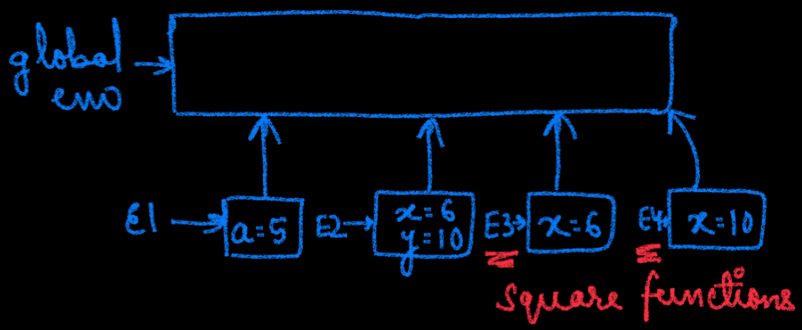


Calling a function creates a new frame in a new environment & binds arguments with parameters & evaluation is done inside the new env.

What if nested func^{n} is applied?

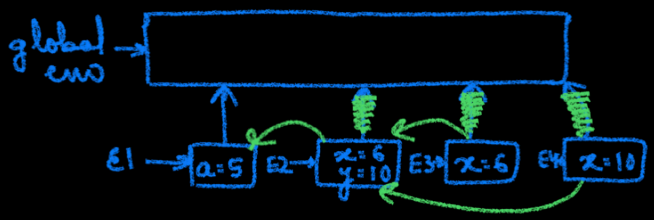






Square functions

To make Scheme Dynamic Scope, just make the parent ptr point to the fn call.



5/9/24 Java class has 2 parts ⇒
① HAS-A : composition
② IS-A : Defines inheritance

In instantiation of a class we create an object of the class.

Initialization happens in constructor of a class

```
Person P = new Person("A", "X");
```

Methods perform actions and take/give messages.

In Java, class names have to start with a capital letter.

```
System.out.println()
```

System → class
out → Some field in class System that points to smth containing the 'println'
println() → member method

Child class inherits features from parent class / superclass and they can extend some more ".

Polymorphism means 'multiple forms'.

The 'process' generated by the program is the one which takes multiple forms.

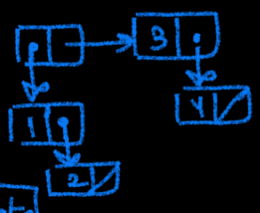
```
Person p;
if (*)          // * : Some input taken at runtime or a random num generated at runtime
    p = new Prof();
else
    p = new Student();
p.foo();
```

→ Simple Runtime Polymorphism

10/9/24

(list (list 1 2) 3 4)

Box pointer



interchangable

Sequences become primitives. Use map, filter, accumulate

Didn't write much 😐

12/9/24

(car (cdr (stream-filter prime? (enumerate-interval 10K 1M))))

So inefficient. Are objects truly time dependent?

STREAMS

Ⓜ A data structure

(cons-stream a b) ⇒ stream (say s)
(stream-car s) ⇒ a
(stream-cdr s) ⇒ b

the-empty-stream ⇒ '(), so (stream-null? s) ⇒ (eq? s the-empty-stream)

```
(define (stream-ei low high)
  (if (> low high)
      the-empty-stream
      (cons-stream low (stream-ei (+ low 1) high))))
(define (stream-filter pred s)
  (cond ((stream-null? s) the-empty-stream)
        ((pred (stream-car s)) (cons-stream (stream-car s)
                                            (stream-filter pred (stream-cdr s))))
        (else (stream-filter (stream-cdr s)))))
```

Looks very same as lists (even the hofs)

But now, this is much faster ⇒

(stream-car (stream-cdr (stream-filter prime? (stream-ei 10K 1M))))

Let's see the data struct in detail ⇒

(cons-stream a b) ⇒ (cons a (delay b)) — (delay b): promise

(stream-car s) ⇒ (car s)

(stream-cdr s) ⇒ (force (cdr s)) — force: force the promise

How to delay the evaluation of an expression? ⇒ Store in a lambda

(delay <expr>) ⇒ (lambda () <expr>), (force promise) ⇒ (promise) — (promise): Call the λ

(stream-car (stream-cdr (stream-filter prime? (stream-ei 10K 1M))))

⇓

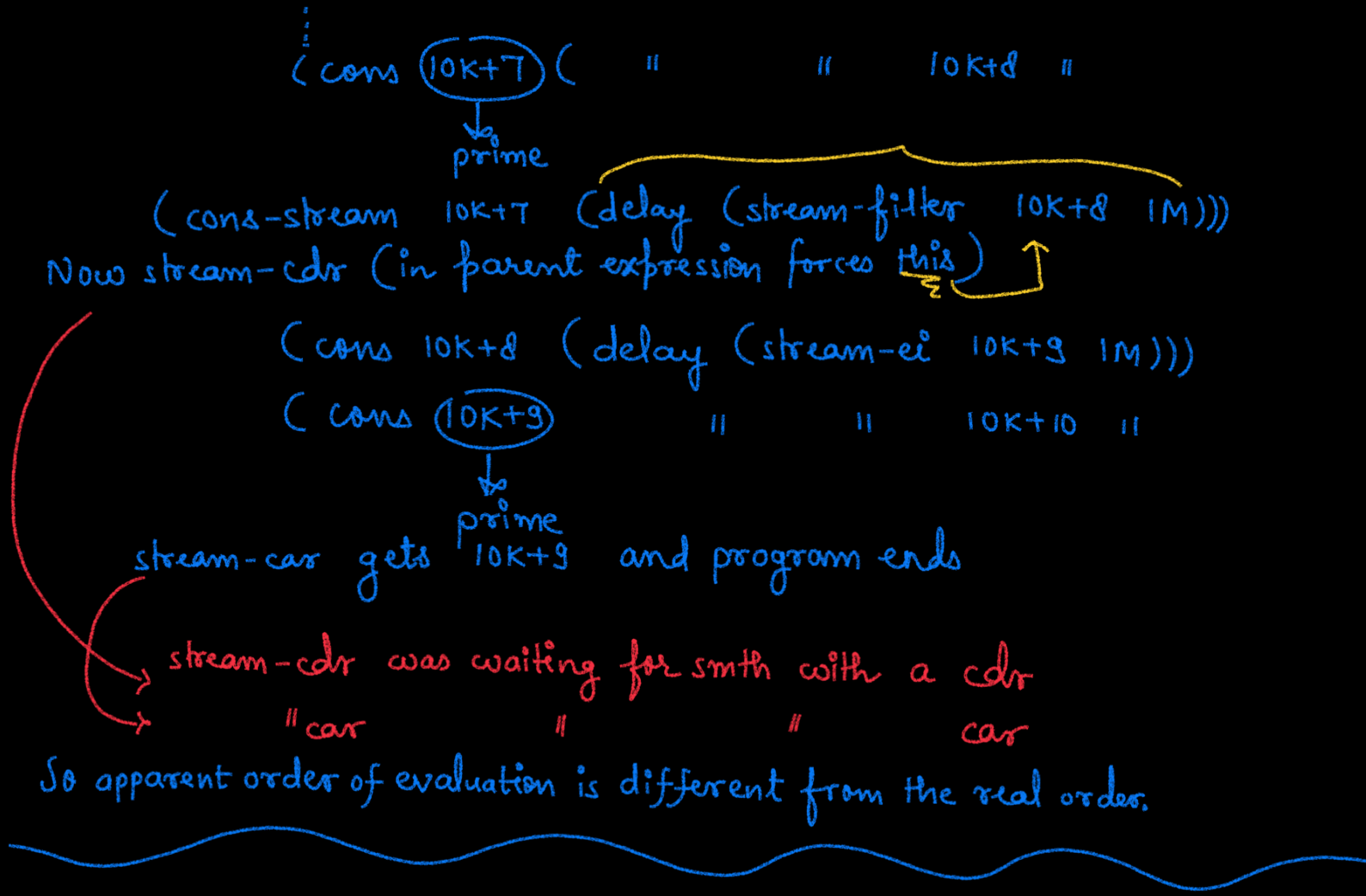
(cons 10K (delay (stream-ei 10K+1 1M)))

⇒ 10K not prime so stream-cdr is called cdr {aka promise}

⇒ promise is called →

(cons 10K+1 (delay (stream-ei 10K+2 1M)))

⋮


(cons 10K+7 (" " 10K+8 "
prime
(cons-stream 10K+7 (delay (stream-filter 10K+8 1M)))
Now stream-cdr (in parent expression forces this)
(cons 10K+8 (delay (stream-ei 10K+9 1M)))
(cons 10K+9 " " 10K+10 "
prime
stream-car gets 10K+9 and program ends
stream-cdr was waiting for smth with a cdr
" car " " car
So apparent order of evaluation is different from the real order.

23/9/24 Midsem answer discussion :||

```
(define-syntax
  (syntax-rules ()
    (delay p) (lambda () p)))
```
→ Works like #define in C++
Direct text replacement in code

||ly for (cons-stream)...

Infinite list of integers? Use stream!

```
(define (integers-from n)
  (cons-stream n (integers-from (+ n 1))))
(define integers (integers-from 1)) ⟶ (1 procedure)
(define (stream-ref n s)
  (if (= n 0)
      (stream-car s)
      (stream-ref (- n 1) (stream-cdr s))))
```
nth index element from s

(stream-ref 100 integers) ⟶ 101

```
(define (divisible? a b)
  (= (remainder a b) 0))
(define not-div-5 ⟶ (all numbers not div by 5)
  (stream-filter (lambda (x) (not (divisible? x 5))) integers))
```

(stream-ref 10000 not-div-5) ⟶ 12501

Same stream for fibonaccis

Parallelization possible in functional programming.

24/9/24

Sieve of Eratosthenes:-

~~1~~ ~~2~~ ~~3~~ ~~4~~ ~~5~~ ~~6~~ 7 ~~8~~ ~~9~~ ~~10~~ 11 ~~12~~ 13 ~~14~~ ~~15~~
~~16~~ 17 ~~18~~ 19 ~~20~~ ~~21~~ ~~22~~ 23 ~~24~~ ~~25~~ ~~26~~ ~~27~~ ~~28~~ 29
~~30~~ ...

2 3 5

```
(define (sieve s)
  (cons-stream (stream-car s)
               (sieve
                 (stream-filter
                   (lambda (x) (not (divisible? x
                                      (stream-car s))
                   (stream-cdr s)))))
(define primes (sieve (integers-starting-from 2)))
(stream-ref 5 primes) ⇒ 13
        "  100  " ) ⇒ smth-smth
```

Similarly,

```
(define primes
  (cons-stream 2 (stream-filter prime? (integers-from 3))))
```

Another way to check whether a num is prime or not?

```
(define (prime2? x)
  (define (iter ps)
    (cond ((> (square (stream-car ps)) x) #t)
          ((divisible? x      "     ) #f)
          (else (iter (stream-cdr ps)))))
  (iter primes))
```

Instead of looking for divisors, we look for the prime divisors

```
(define primes
  (cons-stream 2 (stream-filter prime2? (integers-from 3))))
```

↓

we need some initial prime no. so that there is a (stream-car ps)

Memoization would make this very more efficient if we have a ton of queries for this. (As for every prime2? call, makes primes' everytime!

square-iter — how iter? Coz of state variables 'guess'
Each guess is a different state!

```
(define (sqrt-stream x)
  (define guesses
    (cons-stream 1.0
                 (stream-map (lambda (guess)
                               (sqrt-improve guess x))
                             guesses)))
  guesses)
```

1.0 map (1.0 (map ...))

1.0 1.5 (map 1.5 (map ...))

smith .....

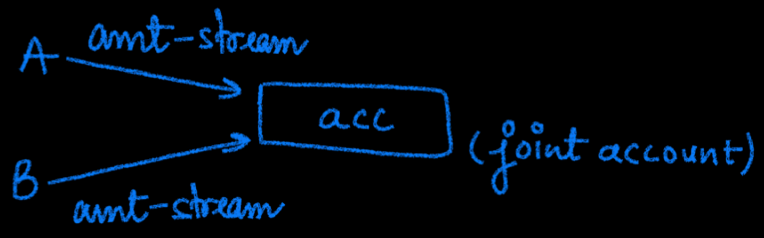
```
(define sqrt2s (sqrt-stream 2))
(stream-ref 0 sqrt2s) → 1.0
        ||  1    ||   → 1.5
        ||  5    ||   → 1.41421...
```

streams hide states of iterative process!

```
(define (make-withdraw balance)
  (lambda (amount)
    (set! balance (- balance amount))
    balance))
```

} preserves state of balance!

```
(define (stream-withdraw balance amt-stream)
  (cons-stream
    balance
    (stream-withdraw (- balance (stream-car amt-stream))
                     (stream-cdr amt-stream))))
```



(Not good as time as a factor is needed for this merge to be fair!)

26/9/24 Haskell

- Lazy, typed

Function: map from domain to a range

f :: D → R
sin :: Float → Float
age :: Person → Int
add :: (Int, Int) → Int

Applying func^n?
No parentheses, only space separations, E.g. f x

Composition?
f :: Y → Z, g: X → Y ⇒ f.g :: X → Z

(f.g) x :: f (g x)

f says if you give me a value in Y then I will give a Z value otherwise won't work
So this type signature is sort of some contract to be followed by user

commonWords :: Int → ([Char] → [Char])

'→' means some f^n

What does this mean?
This is smth like it takes an integer, returns a lambda that takes [Char] and returns [Char]

```
(define add
  (lambda (x y)
    (+ x y)))
```

⇒

```
(define add
  (lambda (x)
    (lambda (y)
      (+ x y))))
```

CURRYING → Any Function only takes 1 argument in Haskell!

Type Synonymns same as typedef

Currying:-

map :: (a → b) → [a] → [b]

(a → b): 1st input; [a]: 2nd input; [b]: output

Any thing on left to an arrow is the 1st argument to that f^n.

Haskell typical functions are mostly composition of fn^s.

30/9/24

`[]` empty list — `[1,2,3]` list (comma sep)

`x:xs` (cons) → `x` car, `xs` cdr

cons is non-strict in both arguments

(strict smth means eagerly eval it & non-" " " lazily ")

```
null :: [a] → Bool   {Checks if list is empty}
null [] = True
null (x:xs) = False

map :: (a → b) → [a] → [b]
map f [] = []
map f (x:xs) = f x : map f xs

filter :: (a → Bool) → [a] → [a]      (a → Bool) → predicate
filter f [] = []
filter f (x:xs) = if f x
                  then x : filter f xs
                  else filter f xs
```

indentation

Since we have the types in Haskell, equality f^n can check the types & know which code to run { Eg. In scheme, integers ⇒ = proced, strings ⇒ eq? "

Enumerations!

$[x \mid 2x > 12 \text{ and } 3x < 80 \text{ and } x \in [1,\ldots,10]]$ (Math)

x: Unbound; x: bound → This bounds x

```
[x | x ← [1..10], x*2 > 20, x*3 < 80]   (Haskell)
```

comma works as "and"

These should be defined to the right of x ← [ ] otherwise Haskell identifies it unbounded

```
[x | x ← [1..100], isPrime x]
take 100 [x | x ← [1..], isPrime x]
map f xs = [f x | x ← xs]
filter p xs = [x | x ← xs, p x]
```

```
concat xss = [x | xs ← xss, x ← xs]
```

01/10/24

Strongly typed, weakly typed are just relative terms. E.g. Java is stronger typed than C. (C can type-cast pointers).

Untyped lang. ⇒ Assembly?

'data' keyword is used to define new data types

```
data Shape = Circle Float | Rectangle Float Float
```

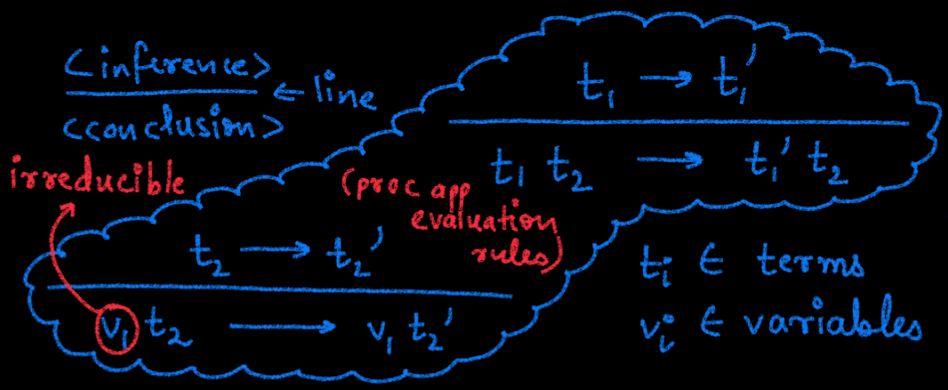
```
:type Circle
⇒ Float → Shape
|| Rec
⇒ Float → Float → Shape
```

These are not new data types
These are just names given a constructor of type Shape
Hence f :: Circle → Int gives error

3/10/24 (Simply) Typed Lambda Calculus ⇒

Grammar :- $M \to x$
$\mid \lambda x{:}T.\ M$ ← Type (for unbounded variables)
$\mid M_1\ M_2$

Γ → type environment (binding b/w var & types)



(t_1 simplies to t_1')
$(t_1\ t_2)$ // $(t_1'\ t_2)$

$t_i \in$ terms
$v_i \in$ variables

? What are these 2 trying to enforce?

These 2 rules enforce that in application, operator should be simplified first then operands. (acc to above inference rules)
In untyped λ-calc, we simplify M_1 first then M_2 in $(M_1\ M_2)$ application
Now // (or rules) can't check whether M_1 is a procedure or not. If M_1 is a variable then application fails. Hence types can help here.

"$x:T \in \Gamma$" ≡ "$\Gamma \vdash x:T$" (If we lookup Γ, we'll find a "$x:T$" binding

"$\Gamma \vdash \lambda x:T_1.\ t_2$" What will be the type of the lambda?
↓
term (type unknown?)

We need to know types of parameters & the return value (in that environment)

$$\frac{\Gamma \vdash x:T_1 \qquad \Gamma \vdash t_2 : T_2}{\Gamma \vdash \lambda x:T_1.\ t_2 : ?}$$

? → $T_1 \to T_2$

$$\frac{\Gamma \vdash x:T_1 \qquad \Gamma \vdash t_2 : T_2}{\Gamma \vdash \lambda x:T_1.\ t_2 : ?}$$

2 times?

? → $T_1 \to T_2$

What if?

$$\frac{\Gamma \vdash t_2 : T_2}{\Gamma \vdash \lambda x:T_1.\ t_2 : T_1 \to T_2}$$

shouldn't this work?
No, as x is most likely present in t_2 & to determine type of t_2 (in Γ) we need type of x (in Γ) as well!
(Notice the "in Γ" highlights)

Application:

$$\frac{\Gamma \vdash t_1 : T_1 \to T_2 \qquad \Gamma \vdash t_2 : T_1}{\Gamma \vdash t_1\ t_2 : T_2}$$

If types don't match, we'll get stuck.

Type-soundness Theorem ⇒ "Well-typed programs cannot go wrong!"
~ Robert Milner (1978)

↓ (Types for everything)

$$\frac{\Gamma \vdash t_1 : \text{Bool} \quad \Gamma \vdash t_2 : T \quad \Gamma \vdash t_3 : T}{\Gamma \vdash \text{if } t_1 \text{ then } t_2 \text{ else } t_3 : T}$$

① t_1 should be Bool
② t_2 & t_3 have same types

If t_2, t_3 had different types, then they should be assigned some 'union' of types?
But is union defined?
Subtyping helps!

if true then 0 else "Hello"
(T_1) (T_2)

Subtyping ⇒

$S \leq T$
(Monkey) ↑ subtype (Animal)

$\forall S, S \leq U$ (Universal type needed so that union of types is always possible)
↓
java.lang.object

① $S \leq S$ ② $\frac{S \leq T \quad T \leq U}{S \leq U}$ ③ $\frac{T_1 \leq S_1 \quad S_2 \leq T_2}{S_1 \to S_2 \leq T_1 \to T_2}$

contravariance ↙ ↘ covariance