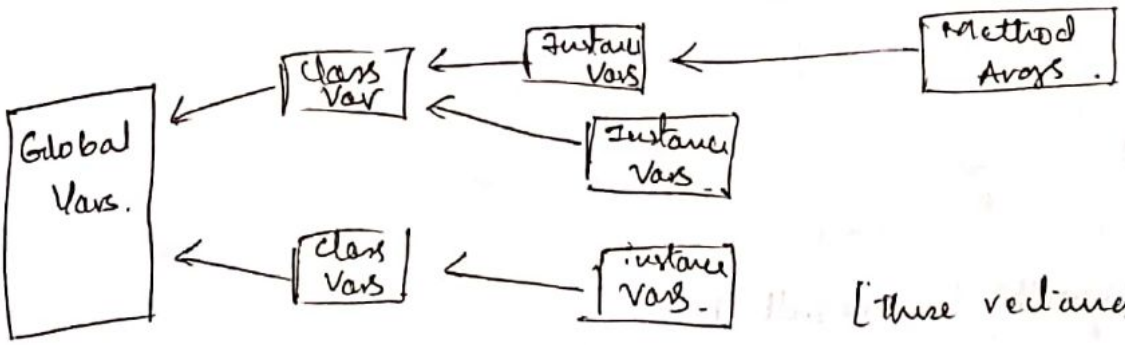
I N D E X

Name SHRAVAN KUMAR GUVVADI. Std_____ Sec_____

Roll No._____ Subject_____ School/College _____

Sl. No.	Date	Title	Page No.	Teacher Sign/ Remarks
		STRUCTURE AND INTERPRETATION		
		OF COMPUTER PROGRAMS		
		BOOK II		

Environment Model of evaluation



[these rectangular blocks are called frames]

this diagram will be the essence of todays lecture.

- Environment model isn't limited to Obj Oriented programming as we will see OOPs comes for free once we have the Schemes idea of Scope of Variables & Lambda
- we will demonstrate the intricacies of the environment model by writing a program for a counter

1) (define counter 0)

```
(define (count)
  (set! counter (+ counter 1))
  counter)
```

this program works but the problem is that we need to define a global variable which we wish to avoid because we may wish to have more than one count so we prefer local variables.

2)
```
(define (count)
  (let ((counter 0))
    (set! counter (+ 1 counter))
    counter))
```

this Program doesn't work. As each time we call 'count' the counter is reset to '0' and set! raises it to '1'

(We have written this prog. because let is the way we create local variables)

but, ... No parenthesis

```
(define count
  (let ((result 0))
    (lambda ()
      (set! result (+ result 1))
      result)))
```

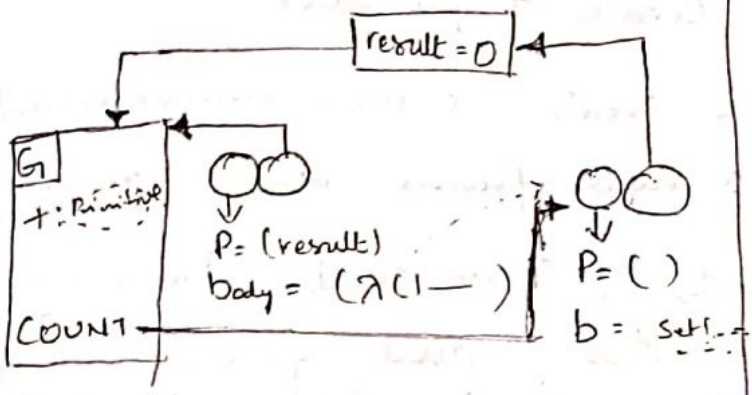
this program works, i.e > (count) ← Parenthesis
1
> (count)
2
> (count)
3

Thus we have managed to make a local state variable

* When we write (define (count) we are actually abbreviating the lambda i.e
(define (count) is equivalent to (define count (λ () ...

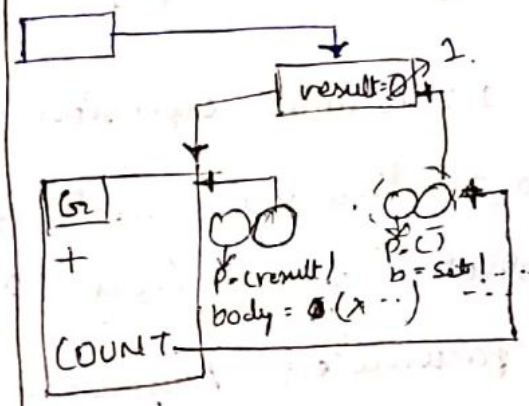
but here, a lot in between the define & λ() which translates to, define count to be the result of making a binding from result to '0' & with that binding active create a procedure of No arguments.

To understand this we need to understand the working of the environment model of evaluation



(Global frame)

Execution when the procedure is called.



> (count)

* since count is called without any argument it creates an empty frame which points to the existing latest environment

```
(define count
   (let ((result 0))
      (lambda ()
         (set! result (+ result 1))
         result)))
```

```
(define count
   ((λ (result)
      (λ () ____ ))
    0))
```

* when the computer sees (define count : it understands that we intend to add a binding to the global frame, whose name is count & its value is everything inside (( )) to get its value we need to evaluate everything inside it ie '0' viz zero self evaluating & the λ expression which gives rise to a procedure represented by two bubbles ○○ the left bubble contains the info found in the λ exp namely, Parameters & the Body of λ exp.

the right bubble in the current environment i.e the environ in which we solve the λ.

Rule NO. 1 : Lambda expression creates a procedure

Rule NO. 2: Procedure invocation creates a new environment it does that by creating a new frame with values of its parameters / Arguments. , So we create [result=0] and we have to extend this new frame to the existing environment to which the right-bubble is pointing to.

Next we do the λ() inside it & point it to the current environment i.e [result=0]. this λ expression returns the resulting procedure as its return value, so the λ(result) expression also returns it as its return value because of this the resulting procedure becomes the return value of (count)

In essence : the name 'COUNT' gives us the λ() procedure which remembers its environment. Now we can see what happens when we call the procedure.

λ(count)

even though while defining count we didn't use paranthesis around it to signify that it's a procedure. Since it's return value is a procedure we have to call it as one

* Since count has no argument but it's an invocation it creates an empty frame
* Now we try to evaluate set! procedure which has '+' first we look for '+' in the current envi [ ] then in [result=0] & from there to [ ] (Global envi)

why we lookup result [ ] → [result] & thus 1 is self evaluating
now we do 0+1 = 1 & thus set! looks for result to modify
it [ ] → [result=0] 1.

• So what made this work is the Right bubble of procedure which remembers the environment & they modify this environment result = 1 & retain it since the procedure count calls Or points only to λ() procedure & therefore (if result=0)) is not executed again.

---

the 3 technical pieces of Object Oriented programming.
are 1) Message Passing.
2) Local state
3) Inheritence

Message passing is simple we write a prog λ(message) which has a 'cond' clause which says if message = this do this & this's, if message = that do that & that & so on

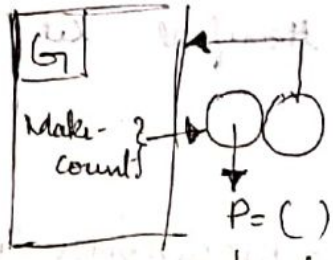
Local state, we have achieved through environment model

: Move than One count , the only diff from count is parenthesis

```
(define (make-count)
  (let ((result 0))
    (lambda ()
      (set! result (+ result 1))
      result)))
```

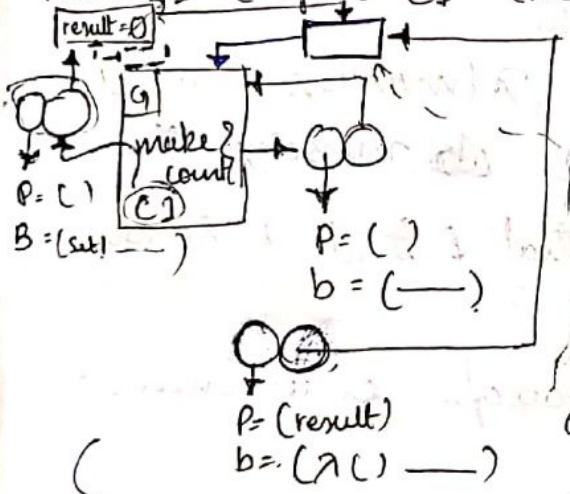
=>

```
(define make-count
  (lambda ()
    ((lambda (result)
       (lambda ()
         (set! result (+ result 1))
         result))
     0)))
```



that's it we don't evaluate the body just the lambda expression ie when we write (define square x) we don't evaluate this. → (* x x)

Now (define C1 (make-count))



Now to return this value to C1 we need to evaluate its sub expressions.

thus the key here is that unless we call the procedure & thereby create a frame the body isn't evaluated

so when we say (define C2 (make-count)) we create another empty frame which creates another separate body evaluation thus making a fresh counter. all this is done just by binding in one change from the counter to make-count ie defining make-count as a procedure/function (by using ( ) in define).

when we call C1 again it forms another frame which is empty & extend the environment that C1's right bubble is pointing at. & then evaluate the body thereby changing

Result = ~~0~~ 1.

* the frame created by a procedure is used to extend an environment & this environment which is to be extended is given by the right- bubble. (This is why we have right bubble)

* when we call a procedure & make a new frame we can use this to extend environment in 2 ways 1) extend the environment the Right bubble points to or the environment in which the procedure was created. → Lexical Scoping

2) extend the current environment or the environment in which we call the procedure
→ Dynamic Scoping.

e.g. eg: (f 1 type) > (define (1. make-count) of the scheme prompt then the Emoi Extension would happen with global environment as scheme prompt's emoi is global

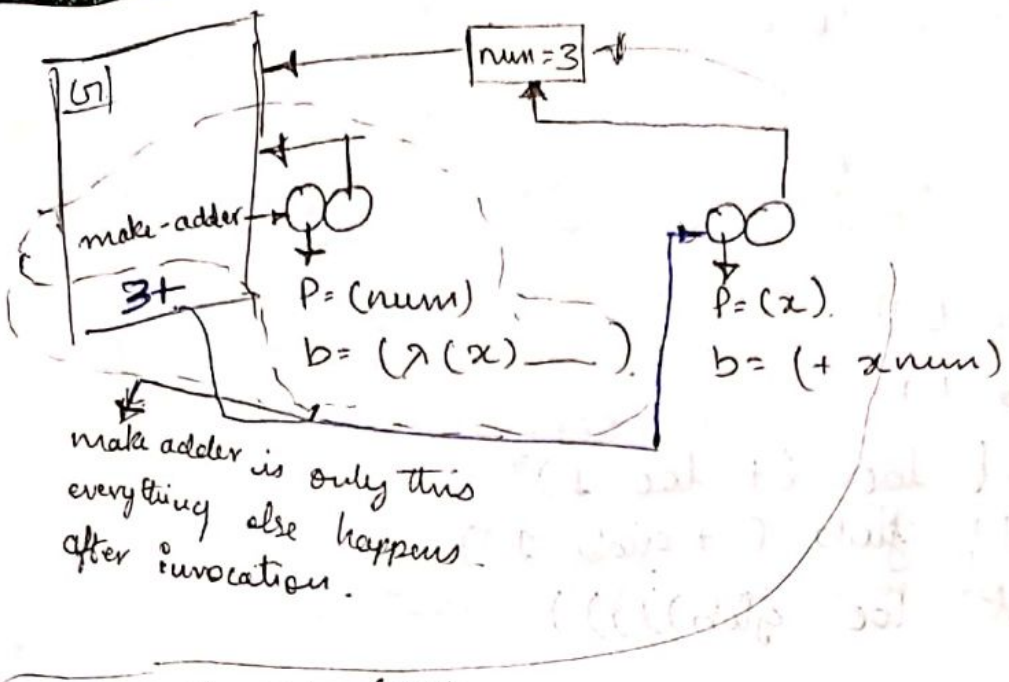
Substitution Model	Environment Model
1) Recursively evaluate all Subexpressions	1) Recursively evaluate all Subexpressions in current environment
2) Substitute actual arg values for formal params in body	2a) Make a frame, Binding formal Params to actual arg Values 2b) Extend procedure's env. with new frame
3) Eval Modified Body.	3) Eval body in new Env.

> (define (make-adder num)
    (lambda (x) (+ x num)))

which when expanded.

> (define make-adder.
    (lambda (num)
      (lambda (x) (+ x num))))

> (define 3+ (make-adder 3)).

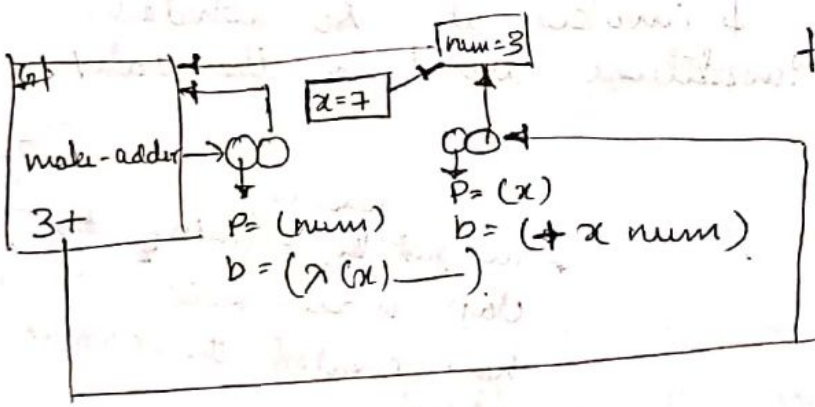


* we have written 3+ in the global environment because that's where we typed this

Now, if we say.

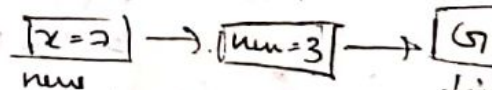
> (3+ 7)

> 10 this is done as follows



* here we create a new frame with x=7 Now we cannot extend it to global env. even though the current env is global (it is typed in global) because '3+' is binded to a procedure. which points at our environment with num=3.

→ then it looks at '+' and searches in

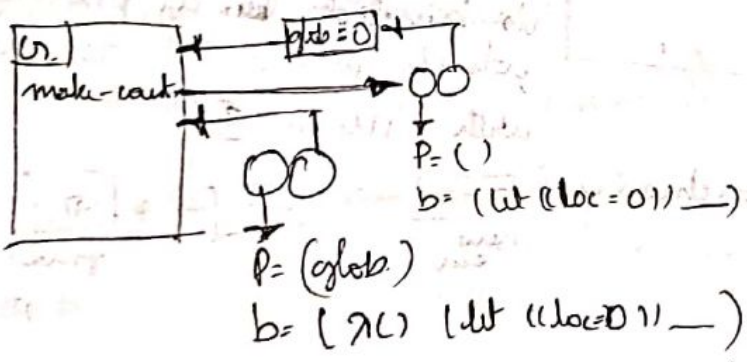


* Now we will make a counter with class variable.

```
(define make-count
  (let ((glob 0))
    (lambda ()
      (let ((loc 0))
        (lambda ()
          (set! loc (+ loc 1))
          (set! glob (+ glob 1))
          (list loc glob))))))
```

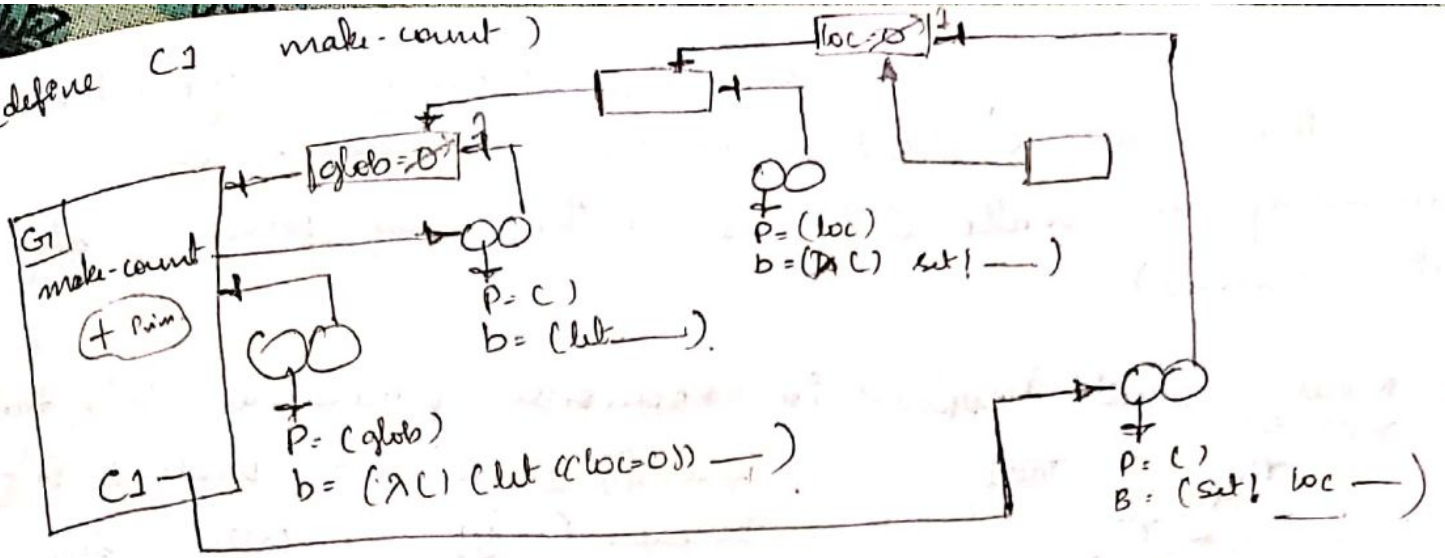
* here the make-count will have a binding with whatever is returned by the `Let' [outermost].
* `Let' creates a procedure & invokes it so whatever is returned from this invocation will be the value of make-count.

* here just by defining the class 'make-count' we have created the class var 'glob' when we



* That's it we need to stop here for the binding of make-count because this is the value returned by the outermost Let.

> (define C1 make-count)



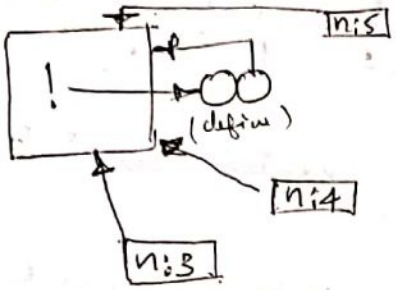
* By creating an instance C1. we have also created the inst var loc.
* If I define C2 as well. then it would create a new empty frame + everything will have to be duplicated.
  ↳ Printing to glob=0.

so they will share same class variable glob but diff instance var loc.

* Now if I call >(C1) Now it creates another empty box that would point to loc=0 because the right bubble of C1's binding points at it + the evaluation happens as follows.
  * the body of C1 is evaluated in the new environment ie the empty box.
  * looks for '+' in → loc → → glob → G1
  * looks for 'loc' in → loc
  * 0+1 = 1
  * looks for loc in → loc
  * looks for '+' in loc → → glob → G1
  * looks for glob in → loc → → glob
  * 0+1 = 1
  * looks for glob in → loc → → glob

* if we keep calling '(1' we keep creating empty boxes but that's not a problem in scheme because the memory is reallocated if nothing is printing at it. (it seems)

* Now what happens in recursive procedures like Fact
> (n=5)!



here n:4 doesn't point to n:5 because (n=4)! is called separately in the global env. therefore we don't end up getting a long chain of frames when we call say 1million Factorial. but just all frames printing at G this is an advantage of Lexical scoping.

*** While coding if we want to create a local state variable the method is to create a 'x' inside a let. ie., if you want to create a class var the x of the class should come inside a let of the intended class var unlike for instance vars

---

Now, we will try to integrate the message passing into 'make-count' procedure.

```
(define make-count
  (let ((glob 0))
    (lambda (f)                    ; this is value we put
      (let ((loc 0))               ; instantiation variable
        (lambda (msg)
          (cond ((eq? msg 'local)
                 (lambda (value)
                   (set! loc (+ loc value))
                   loc))
                ((eq? msg 'global)
                 (lambda (value)
                   (set! glob (+ glob value))
                   glob))
                (else (error "No such method" msg))))))))
```

constructor for class. This is what make count actually is

→ reads the message and returns a procedure. — Dispatch procedure

object →

* We have written a OOP program without explicitly using method, ~~global~~ class vars or instance vars or we needn't use ask to call these methods ie.,

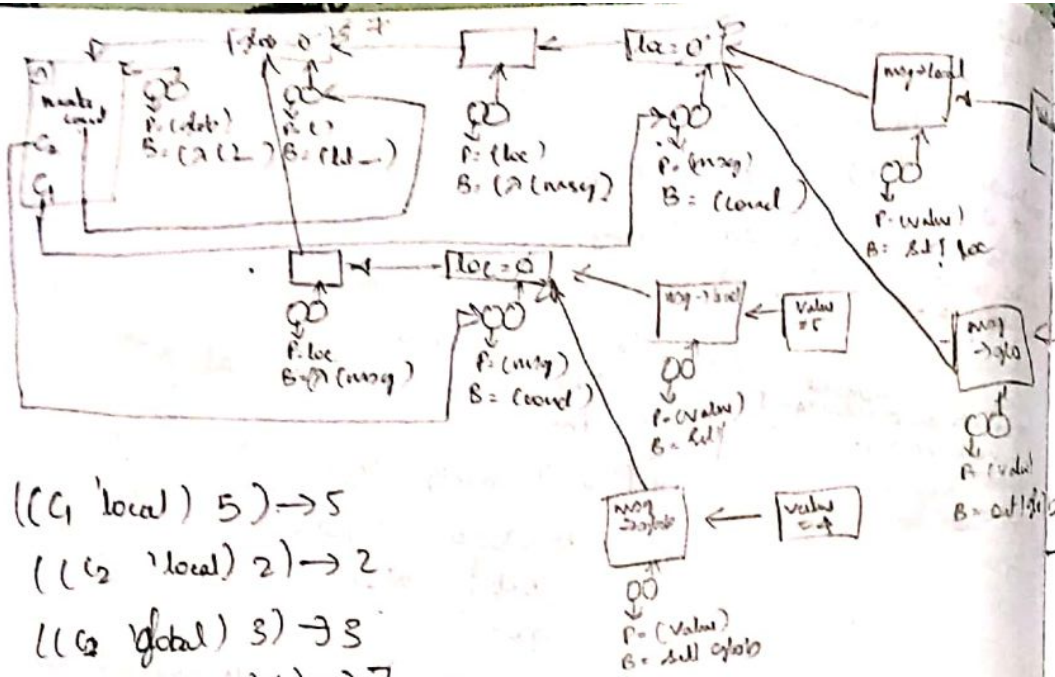
```
> (define c1 (make-count))
> ((c1 'local) 5)
5
```

Since the message passing returns a procedure we call that procedure with an arg, here '5'

ie. in OOP sys.

(ask obj msg arg) ⟹ ((obj msg) arg)

```
> (define c2 (make-count))
```



((C1 'local) 5) → 5

((C2 'local) 2) → 2

((C2 'global) 5) → 5

((C2 'global)) → 7.

Now we take the first command ((C1 'local) 5), C1 'local is a procedure call so we create a new frame. & C1 is pointing to the procedure we go there & find a cond clause & here we match the msg. cond clause returns a procedure which is then invoked with an argument 5. This invocation makes another frame & the set! is executed. Same for ((C2 'local) 2) → 2

Now if we ask the Sys the value of local or global variable the Sys wouldn't know how to do it as we have not made a method for this. So we can write in the cond clause

((eq? msg 'loc) loc)

this will work if we write

(C1 'loc)

but in an actual OOP sys we write

(ask C1 'loc)

which translates to ((C1 'loc)) ... if we had args it would come here

So this doesn't work. So we have to write the method such that it returns 'loc by taking an empty arg ie.

((eq? msg 'loc) (λ() loc))

Inheritence: can be done in 2 ways

1) delegation: after writing all the methods in the cond clause instead of writing else - error we write (ask parent obj)

```
ie (define ____)
     (lambda (self)
       (let ((loc 0)
             (Parent-obj ... make parent obj ...))
         (lambda (msg)
           (cond
             ((eq? msg ...
             ((eq msg ...
             :
             (else (... ask Parent Obj ...)
```

Mutable Data

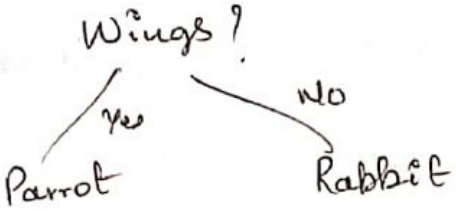
Consider a "game-like code

> (animal-game)
Does it have wings? Yes
Is it a parrot? NO.
I give up, what is it? eagle

Please tell me a question whose answer is YES for an eagle & NO for a parrot.
Enclose the answer in quotation marks.
"Is it the National Bird of USA?"
"Thanks. Now I know Better".

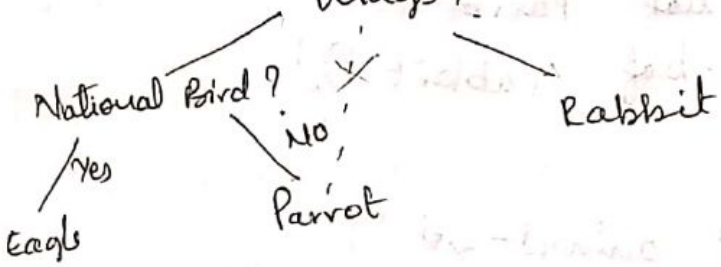
> (animal game)
Does it have wings? Yes
Is it the national bird of USA? Yes
Is it a Eagle? Yes.
"I Win!"

What's interesting about this program is that it learns when we play the game. it starts out with a Data structure that looks like this.



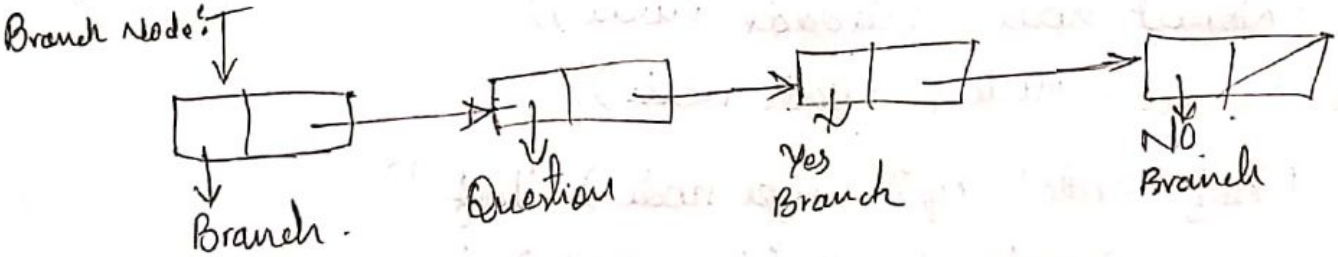
and when it doesn't get it right it takes the input & modifies its data structure as follows.

(this is Binary tree. Since it is a Yes/No tree)

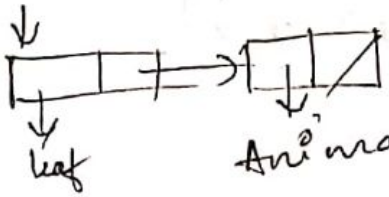


** the most important point here is that we are modifying the existing structure by breaking the existing link between the Wings? & Parrot & making a new link between the wings? & the National Bird

further Breaking down the Tree structure into Branch nodes & Leaf Nodes.



Leaf Node



[we have typed tagged the nodes as Branch & leaf]

The corresponding code is as follows:

```
(define (make-branch q y n)
  (list 'branch q y n))
(define (make-leaf a)
  (list 'leaf a))
```

} constructors

```
(define animal-list
  (make-branch "Does it have wings?"
    (make-leaf 'Parrot)
    (make-leaf 'rabbit)))
```

(define (animal-game) (animal animal-list))

here animal is a helper function/Procedure & for no good reason the selectors are defined within it

```
; (define (animal node)
  (define (type node) (car node))
  (define (question node (cdr node))
  (define (yespart node) (caddr node))
  (define (Nopart node) (cadddr node))
  (define (answer node) (cadr node))
  (define (leaf? node) (eq? (type node) 'leaf))
  (define (branch? node) (eq? (type node) 'branch))
  (define (set-yes! node x)
    (set-car! (cddr node) x))
  (define (set-no! node x)
    (set-car! ((cdddr node) x))
```

Yes or NO

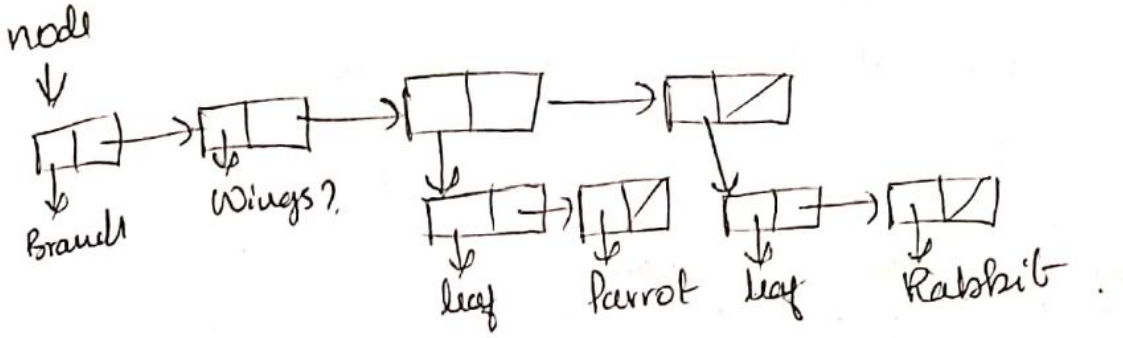
```
(define (yorn)
  (let ((yn (read)))
    (cond ((eq? yn 'yes) #t)
          ((eq? yn 'no) #f)
          (else (display "Please type Yes Or No")
                (yorn)))))

(display (question node))
(display " ")
(let ((yn (yorn)))
  (let ((next (if yn (yespart node) (nopart node))))
    (cond ((branch? next) (animal next))
          (else (display "Is it a")
                (display (answer next))
                (display "? ")
                (let ((correctflag (yorn)))
                  (cond (correctflag "I win!")
                        (else (newline)
                              (display "I give up, what is it?")
                              (let ((correct (read)))
                                (newline)
                                (display "Please tell me a question to know ")
                                (display "answer is YES for a")
                                (display correct)
                                (newline)
                                (display "and NO for a")
                                (display (answer next))
                                (display ".")
                                (newline)
```

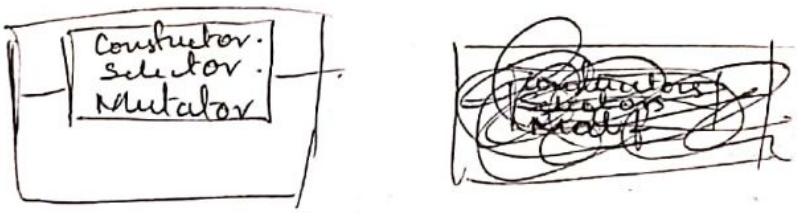
cond used instead of if construct because we have multiple clauses in else. This is not possible in if

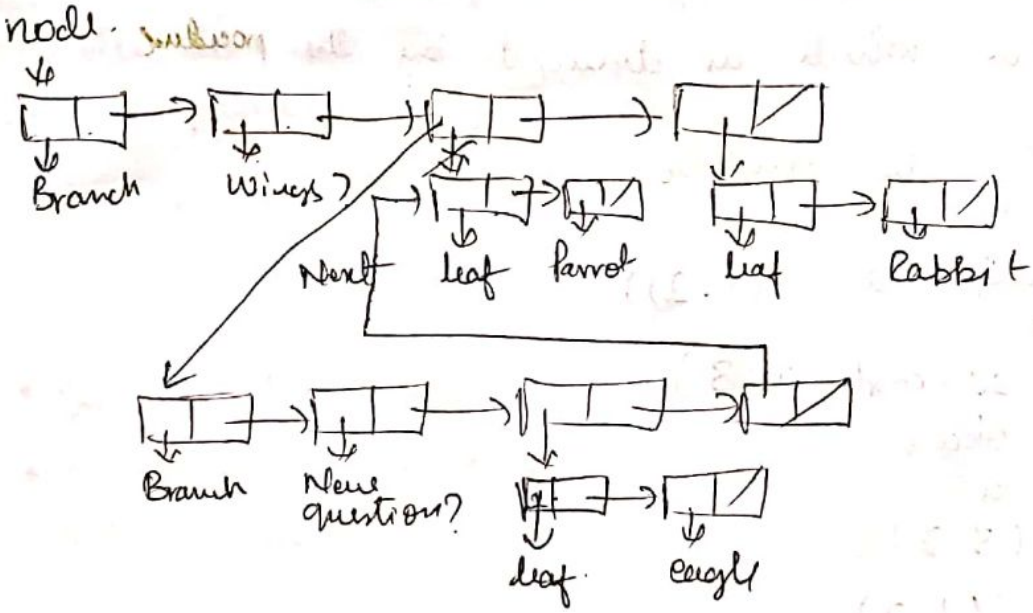
```
(display "Enclose the question in")
(display "quotation marks")
(newline)
(let ((newquest (read)))
  (if yn
      (set-yes! node
                (make-branch
                  newquest
                  (make-leaf correct)
                  next))
      (set-no! node
               (make-branch
                 newquest
                 (make-leaf correct)
                 next)))
  "Thanks. Now I know better.")))))
```

Our Data structure



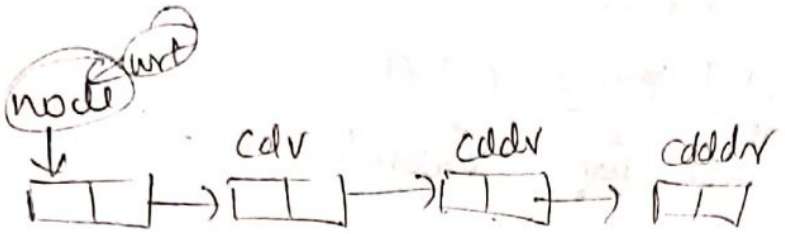
- correctflag is a variable whose value is true or false
- Predicate is a procedure whose value is true or false.
- Now we update our datastructure dia.





* Set-car! & Set-cdr! are 2 Scheme primitives

(set-car! pair new-car)
(set-cdr! pair new-cdr)



* Set-car changes a element → second arg should be a element for lists
* Set-cdr changes the structure of data → second arg is a sublist

↓ for just pair both are elements

* how this implemented in LISP
LISP has only one datastructure & its pointers.
So what set-car! & set-cdr! get are pointers to a memory location which is denoted by the pointer

* quoted lists cannot be mutated.
may not
Ex: (set-car! a) > (define a '(1.2))
a
> (set-car! a 3)
okay
> a
(3.2)
> '(1.2)
> (1.2)

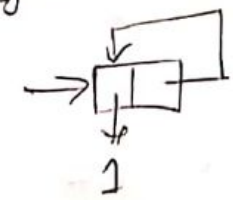
It works but its illegal because as a measure for space efficiency the equal quoted data may be combined & depends on implementation. But its okay to test it on (define a (cons 1 2))

* Set!& set-car!/set-cdr! are similar in terms of deviation from functional prog but are very diff in terms of implementation ie set! is a special form. whereas the latter are primitives

set! changes the binding by looking through the frames as discussed, first arg are symbols and are not evaluated

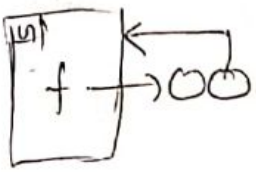
* set car! & set cdr! were not absolutely essential to this prog instead of changing the structure we could have just made a new one by copying the existing except for the req change. So here, this is just a efficiency hack

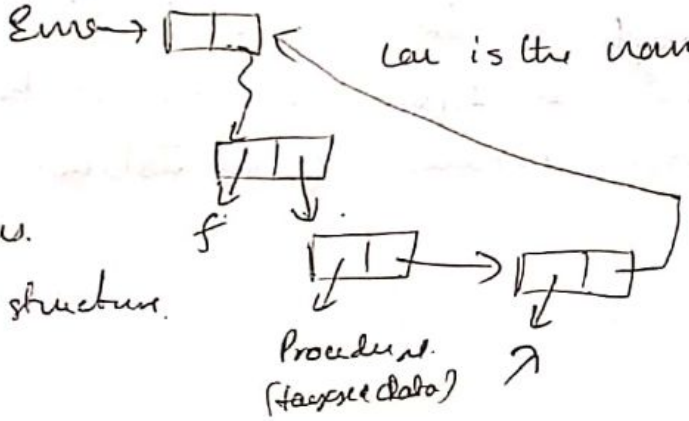
* But when are set-car & set-cdr essential Ans:- in making circular list ie.,



ie.,
```
(define ones (cons 1 '()))
(set-cdr! ones ones)
```

where else do we use circular list? Let's think about actual implementation of procedures like this,



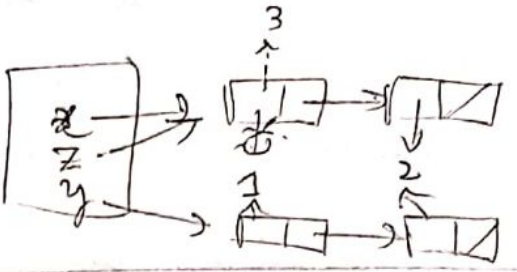


* Env is a list of frames.
* a frame is a list structure full of bindings.

* We can do everything that mutation does by functional prog. But we might use it because its efficient or because it better reflects our thinking

* Equality.

```
('define x (cons 1 2)).
x
(define y (cons 1 2))
y
(define z x).
```



x = (1, 2)
y = (1, 2)
z = 1, 2)

```
>Set-car! x 3
>x
 3.2.
> y.
 1, 2
> z.
 3.2
```

```
(eq? x y)
#t.
(eq? x z.)
#t.
(eq? y z)
#f.
```

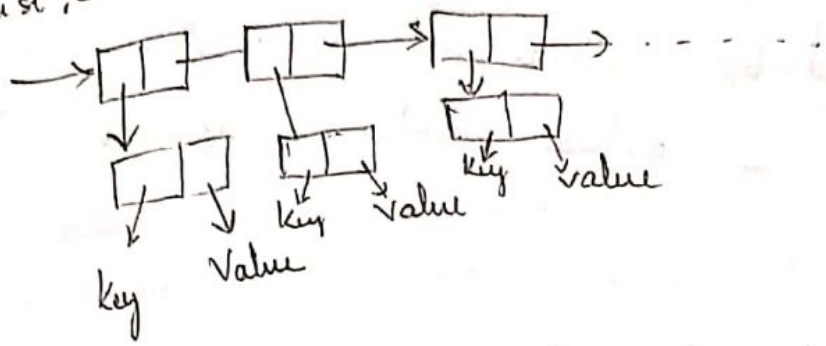
```
(equal? x y)
#t
(equal? x z)
#t
(equal? y z)
#t.
```

* eq asks identity questions that is are they the same entity or do they represent the same memory location.
* equal just compares the values
* Scheme goes through a lot of trouble to make sure that everything that is equal is eq. but this is not true for lists because of mutation

Tables

We have used tables in Data directed programming with 'get' & 'put'. Here we see how they work. First we have to learn about a new data structure called 'Association list' or A list

A-list :-



A-list is a list each of whose elements is a pair whose CAR is a key and 'cdr' a value.

- We have a primitive called 'assoc' that takes a 'key' & a A-list as arguments & returns the key-value pair corresponding to its 'key' argument. It's different from get such that get returns only the value but assoc returns both key-value pair

```
> (assoc 'foo '((a.b) (c.d) (foo.baz) (e.garply)))
(foo.baz)
```

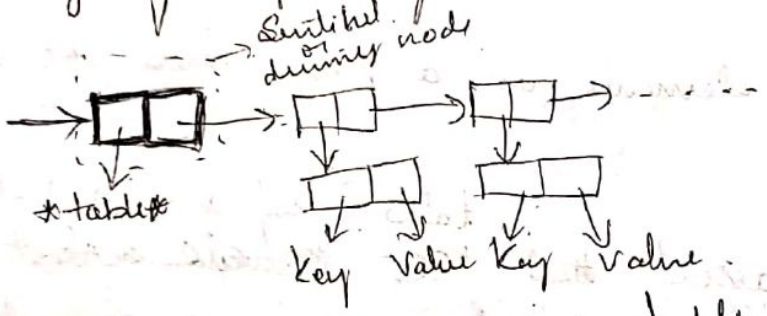
if you give a key that doesn't exist it returns false ie #f

- another difference is it lets us know whether the value associated with the key is false #f or the key doesn't exist. Since in the former case it returns the key value pair. But in get it's not possible to know the difference b/n the two.

* Another reason Assoc returns the key value pair is Mutation to change the value of a key pair we need to know the 'key'.

* We have another primitive called 'assoq' the difference is the way it tests whether 'keys' are equal is 'eq' & not 'equal'
↓ identity ↓ value

* Basically this is a table since it associates keys with values. But the actual table is represented slightly differently. i.e.



This is a One dimensional table here the only diff from a Association list is the first pair with *table* which might seem like a 'type tag' but its actual function is pretty different.

(Pointer Pair)
If we check the diagram of the A list we see that all the pairs are similar except the first one since no pair is pointing to it [it might point to it Name & variable binding]. The problem arises when for example we might want to delete a pair we can just set-cdr! to the new value Ex:-

But if we wanted to do the same to the first Pair we wont be able to do that.

* Another reason for *table* pair is that it allows us to create a empty table while we can create it use in our program along the way. we might be tempted to think that we can create a empty list and use it as a table for this purpose but in that case all the empty table would be 'eq'. & would create problems So if we say make-table cons (*table*) (l) to make multiple empty list
list they will all be 'equal' but not 'eq'

Now, that we have understood the basic structure of table we will look into the code for get & put

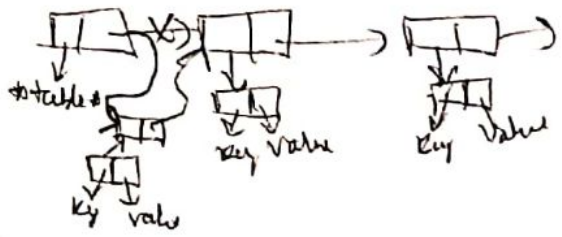
```
(define (get key)
  (let ((record (assoc key (cdr the-table))))
    (if (not record)          ----> we can use just 'record'
        #f
        (cdr record))))

(define (put key value)
  (let ((record (assoc key (cdr the-table))))
    (if (not record)
        (set-cdr! the-table
                  (cons (cons key value)
                        (cdr the-table)))
        (set-cdr! record value))
    'ok))
```

'Put'
first searches the key in the existing list using assoc if not found creates a new list with this pair at the front & set cdr of the table to the new list.

---

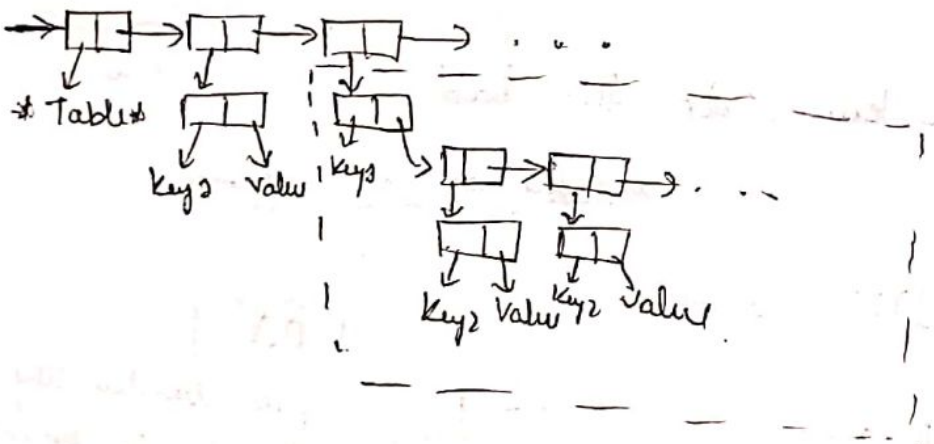
ie



* this is not the best way to implement tables as the Runtime $\Theta(n)$ we can have better efficiency such as $\Theta(\log n)$ & $\Theta(1)$ → # table.

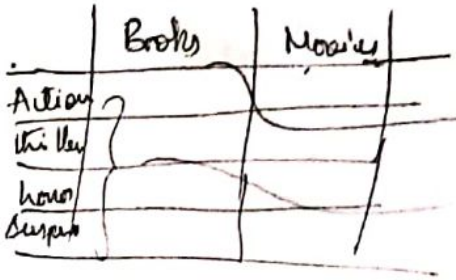
* Another example is that of representing frames in the environment model. Generally the global frame is represented with a # table & the Other frames with the A-list (Association list) since the former is bigger & will be time expensive with A-list

2-D table



This is the Representation of a 2-D table, Note that the dia within the rectangle represents a 1-D table by itself except that it doesn't have a *table* sentinel but it's part is played successfully by the Keys Pair.

My way of imagining this is that key 1 represents the title of a row & then is a 1-D table below this title



* the code for 2D table is in the book. straight forward.
* 'Get' part is
* 'Put' part has 3 cases
  → we say (Put key1 key2 Value)
  3 args

case 1: we find key1. but not key2. create key2 & save value in it

case 2: we find both key1 & key2. change its value.

case 3: we don't find key1. create key1 within it key2 & save value.

we need to write.

(define (make-table) (list '*table*))

and not

(define make-table) '(*table*))

because in the latter case all the empty tables would end up being same ie eq so any change we make in one would end up reflecting in all others and another reason is as stated before we cannot mutate quoted data

using tables.

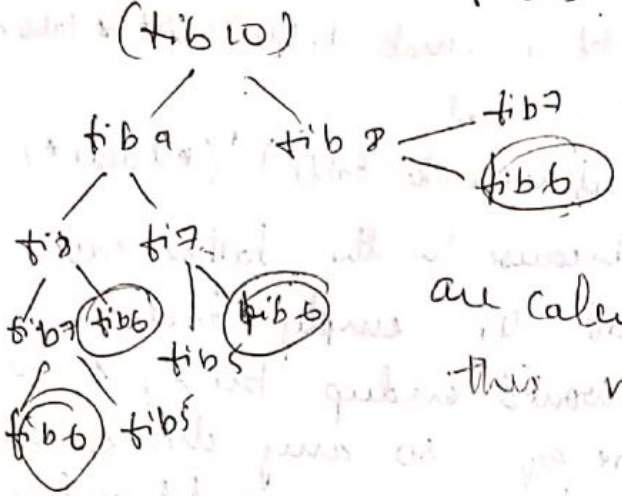
for ex: take the following code that lets you find the fibonacci no

```
(define (fib n)
  (if (< n 2)
      n.
      (+ (fib (- n 1))
         (fib (- n 2)))))
```

here as we can see calculation of each fib no leads to the calculation of 2 more and so on so. its $O(2^n)$ program. A better way to do it is to use the same idea and storing all the calculated fibonacci nos in a table for later use. the imp point here is that if we want to calculate the fib of say 50 we will end up calculating fib of all nos with 50 & 2 and all these nos will be saved in the table for later use.

```
(define (fast-fib n)
  (if (< n 2)
      n.
      (let ((old (get 'fib n)))
        (if (number? old)
            old.
            (begin
              (put 'fib n (+ (fast-fib
                              (- n 1))
                             (fast-fib
                              (- n 2))))
              (get 'fib n))))))
```

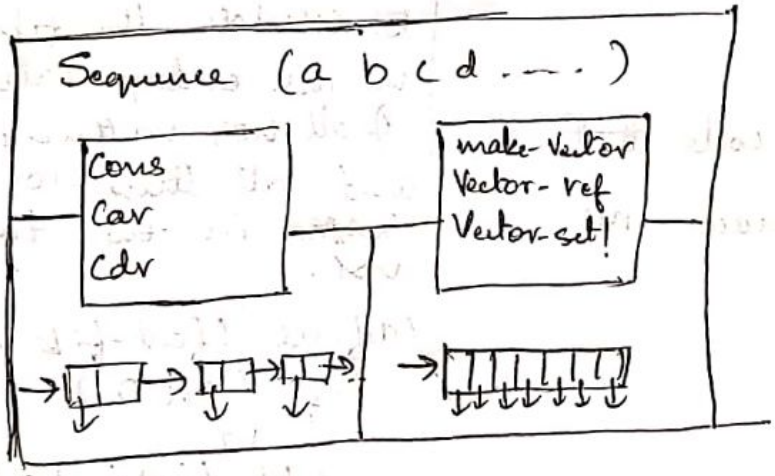
This works as follows



how the same values are recalculated multiple times our fast fib code just make sure that this values are calculated once stored & reused this method is called.

Memoization

Vectors: This is a different way of implementing the abstract of a sequence, i.e. we want to abstractly say 'here's the first element' 'give me the 3rd element' and so on & until now we have been doing it with lists which are built using pairs.



- Vectors are also called 'arrays' in some languages (matlab?)
→ similar to lists, Vectors are made of pointers but unlike lists they can have any no of these pointers

> (define v (make-vector 6))
v.
> v.
#(#[unbound] #[unbound] #[unbound] #[unbound] #[unbound] #[unbound])
> (define v (make-vector 2 'foo))
(foo foo)

---

* we cannot add elements to vectors like we used to with lists by using 'cons'. here to do that we have to recopy the entire vector and add the element we wish to stick in.
* Eventhough list and vectors are abstractly the same there benefits of having multiple representation

car (car (car (....

	List		Vector
n^{th} element	list-ref	$\Theta(N)$	Vector-ref $\Theta(1)$
add new element	cons. $\Theta(1)$		Vector-cons. $\Theta(N)$ → make a new vector of $N+1$ and copy N elements → takes linear time

Map in Vectors.

```
(define (vector-map fn vec)
  (define (loop newvec n)
    (if (< n 0)
        newvec
        (begin (vector-set! newvec n (vector-ref vec n)))
               (loop newvec (- n 1)))))
  (loop (make-vector (vector-length vec)) (- (vector-length vec) 1)))
```

Consider if we were writing a code for playing solitaire. We have two jobs at hand - shuffling and distributing the cards. for shuffling the cards maintaining the cards as Vectors makes the process $\Theta(N)$ where as if it were in lists it would have been $\Theta(N^2)$. But while distributing having it in lists is convenient [the card we have currently is car, the rest of the deck is cdr once it is distributed we cons it]

* As shown above, converting datastructure to a more convenient one, is a general methodology for better efficiency

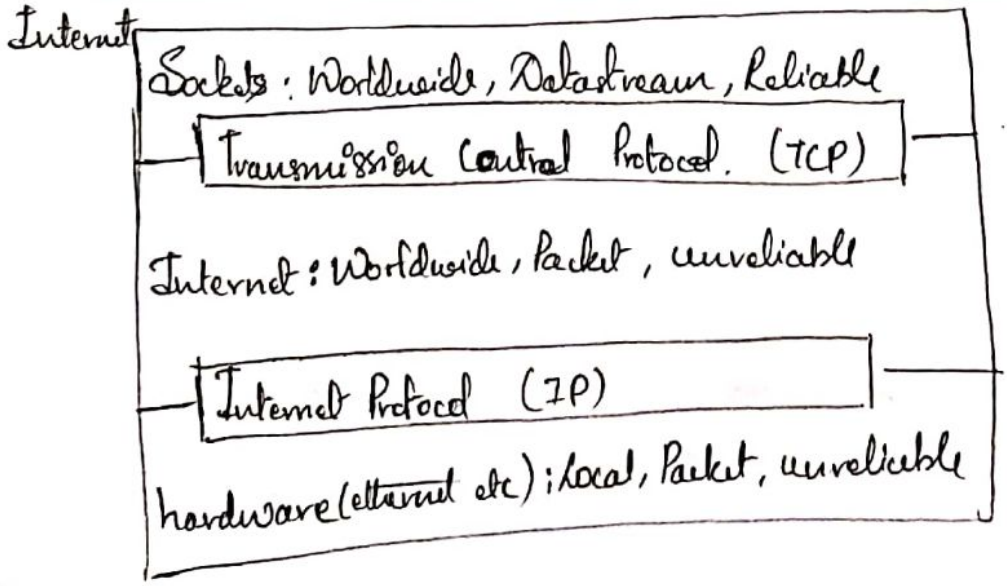
CLIENT / SERVER. PROGRAMMING.

Key Idea: the program organization upto now was done such that there is a starting point or a highest level procedure call that we do (or something else) that runs for a while & produces an output and then it stops.

But in a client server model we see something very different, ie we have a server process that sits there until somebody makes a request for service some-where in the world & then sort of wakes up long enough to handle the request then it goes back to sleep & waits for another one.

Sometimes it'll have to handle more than one request (for Ex google servers) it accomplishes this using some sort of parallelism called 'threads' which isn't hardware parallelism but software parallelism & it carries out all these requests nicely in a shared data space.

The concept of waiting around until somebody requests service is called 'Callbacks'.



The previous diagram shows the organization of Networks (to internet)

* The bottom most Part in the hierarchy is the hardware level network like a local area network established in an office using ethernet etc. It works in sending bursts of information called packets & is unreliable.

→ Generally when we send a req for say reading an article in New York times. Our msg goes over several hops from our Internet service provider to our ISP's ISP to somebody important like Comcast who are very big ISPs.

& then service providers send msg cross country & from then several hops to our destination. So making this work is the job of the 'Internet protocol' which is a software specification which is handled by all PC's these days. & Nos like "128.32.253.149" are addresses and every comp on the net has a table that is a function from address ranges to out of all the PC's I can talk to whom should I send the msg.

→ IP is unreliable in the sense the packets might not be received /received in the right order etc

→ Above that we have another software layer called TCP viz complicated. It's job is to take a bunch of packets with unreliable delivery & give an illusion of a reliable stream of data. It's done as follows.

* It puts Sequence no for packets we are sending out (So that the reciever can rearrange just in case)
* Reciever has to send acknowledgement packet for every packet (in case we don't recieve we resend it)
* If the reciever recieves the same packet twice we ignore

3

→ Packets are generally small (1kb etc) since the Process is unreliable.

(define (im-server-start)
Start the server

set! server-socket (make-server) makes server socket it creates an abstract datatype that listens on a particular Port. it takes an optional argument i.e a Portnumber

The way the internet works is that there are assigned Port numbers for every service ex: http is Socket 80, ssh is 21

these nos are allocated by ICANN a consortium to do this stuff

(when - Socket - ready server-socket (λ ()
(begin.
(Print sensing new client connection)
(Procedure for handshake
↓
argument is a Socket

This means that we dont use the ← with a client but create Server socket for conversation Server socket for each client. Thus we achieved a new server socket for each client. "One Server many clients"

- Then when a client msgs a multi-thread starts

3 way handshake.

client "hello" → Server

Server "welcome" → client

client "thanks" → server.

Concurrency

→ what happens when multiple processes are sharing Data.

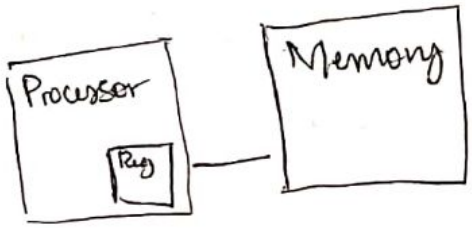
Ex; Multiple people doing online reservation on a flight

→ In such cases using mutation can be problematic. One way to avoid these problems arising from Mutation and parallelism is by using functional programming.

→ To understand the problems arising from mutation & Parallelism. we need to first understand the way the Machine runs a code. for this we take a simple example of mutation.

(Set! x (* x 1)) + we use symbolic simplified version of machine language

```
LOAD  r8, x
ADD   r8, 1
STORE r8, x
```



* There is a Processor + Memory + they had a connection in between.
* Memory is RAM + it is generally in GBs

* Inside the processor there is a much smaller memory called 'Registers'. Machines have 32/64 Registers so 32 diff nos the processor can be thinking about at the same time
* The instructions are to Load Data from memory to Register & store Data from Register back to memory. (R8 here is Register no 8)
* The processor cannot do arithmatic for the stuff in memory. It can do it only on the stuff in the Register
* The thing can be a bit more tricky ~~bcoz~~ with the 'cache' but these things are ignored here.
* So lets take an example of 2 people adding 1$ to the same account at the same time here instead of 102$ we end up with 101$

X = 100

Process A	Process B
Load (100)	Load (100)
Add (101)	Add (101)
Store (101)	Store (101)

↓ Time

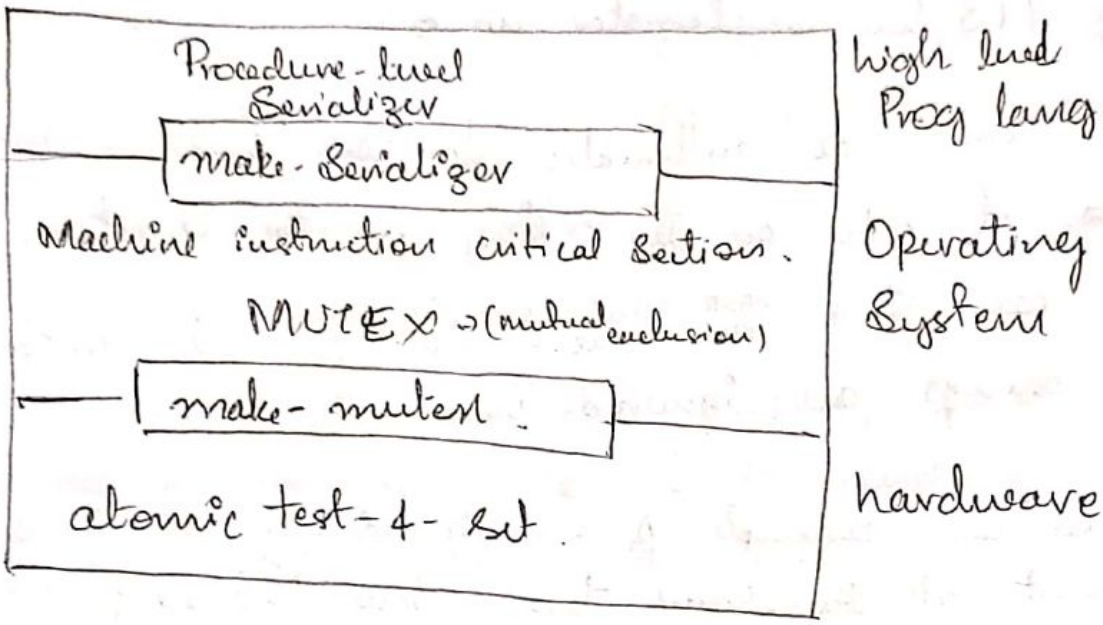
But instead if they were executed one after the other we would have had 102 $

X = 100

Process A	Process B
Load (100)	
Add (101)	
Store (100)	
	Load (101)
	Add (102)
	Store (102)

* What we have to do therefore is to make sure that nothing interrupts (load, ADD & store) instructions when multiple 'loads' are accessing the same variable (data).

*



→ A programmer in a high level language would want to say take this procedure and protect it & nothing more

→ the mechanism we have for that is called 'Serializer' & we have a constructor for that

→ the mechanism works as follows:- when the instruction is translated to machine level language it says here is the beginning & here is the end of a critical section i.e a set of instructions that can't be interfered with. The mechanism is called Mutex & it has a constructor. The mutex capability is provided by the Operating System windows, Linux, Mac OS etc

→ At the lowest level what we need is to be able the read & write a location in memory in one single uninterruptible operation & it is called atomic Test & set

+ this capability is provided by hardware so all architectures these days provide some feature to support this.

for ex in Java we can define obj classes such that every method in that class is protected against other methods of the same instance or if we have class variables we would need another layer of protection to ensure two different instances of the same class can't both be running in 2 threads at the same time.

[* Until windows NT. the system used to crash quite often and this was taken for granted by the users. But after NT there was significant improvement because the programmers paid attention to concurrency issues]

```
> (define x 10)
  (parallel-execute (lambda () (set! x (+ x 1)))
                    (lambda () (set! x (+ x 1))))
```

this will fail as expected.

```
> x
101
```

So we make a serializer.

```
> (define x 10)
  (define s (make-serializer))
  (parallel-execute (s (lambda () (set! x (+ x 1))))
                    (s (lambda () (set! x (+ x 1)))))
> x
102
```

> Serializer thunfor is a procedure that takes a procedure & returns a protected procedure.
(has argument the same)

> how No two procedures protected by the 'same' Serializer can intervene. that means if we had 's' protecting one code & 't' serializer the other the execution will fail.

> Imagining a realistic Scenario with thousands of threads running in pd some are allowed the intervene & some aren't

> the way a Serializer works is by using mutex. So it does, acquire. Mutex S. instead of (load → add → store)

(load + add → store)

Release. Mutexs.

& the (acquire Mutex) waits if someone is using the same Mutex

> To make sure that each imp shared variable (each Bank account for ex) has its own serializer. typically a Bank account will be a abstract data type & will include the Balance & a Serializer & we will have to write methods that use these serializers to operate on the account

* What can go wrong because of parallelizations.
  → Incorrect result (two pass in same seat)
  → deadlock.
  → inefficiency
  → unfairness.

* Deadlock
Suppose we were reserving two variable at once but in opposite orders. ie

Parallel-execute (SAt (λ ——)))
(t (S (λ ——)))

this is a deadlock. ie.

A	B
Acquire S	Acquire t
Acquire t	Acquire S.
Can't do it	can't do it
B finishes	A finishes.

To take care to ensure that deadlock doesn't happen the program can be fastidious in resource allotment ie in money exchange b/n two accounts ie A to B + B to A simultaneously leading to deadlock in such cases we serialize based on Bank account number value. always serialize the higher value bank account first by doing so one of them fails to acquire & continues once the other is done

* Another common thing done these days is wait for the deadlock to happen have an entity that regularly checks for such cases. & have it kill one of the procedures stuck in deadlock

* inefficiency: if we use a single Serializer & do everything in Serial. things always give the correct result but the process is inefficient
* unfairness: what if a proj never gets to run because another Prog sneaks in & takes the Serializer first

---

* The right answer in a parallel processing scenario is the answer we get without parallelism. this doesn't mean we need to serialize (or Parallelism without parallelism) but we allow processes to overlapp except in scenarios which are critical like reading & writing in memory. (if one is using x memory location & the other y it can run in vel if both are using x it is a problem & we serialize)

A Simple make-serializer code.

```
(define (make-serializer)
  (let ((in-use? #f))
    (lambda (Proc)
      (define (Protected-Proc . args)        ; → takes any no of arguments ie the Proc can have any arg.
        (if in-use?                          ;   ↳ list of args
            (begin
              (wait-a-while)                 ; Never mind
              (apply Protected-Proc args))   ; Try again
            (begin
              (set! in-use? #t)              ; Dont let anyone else in
              (let ((result (apply Proc args)))
                ; call the original Procedure
```

The Operating system does this → 'I can run right now run somebody else'

(set! in-use? #f)
result))); Finished but others in queue
Protected-proc)))

* This code is almost right but has a problem that the problem of parallelization that we are trying to solve can occur here. ie more than one proc can do the (if in-use? together & subsequently get to the critical memory read - write section. Therefore we have to check if the serializer is in-use & then check if anyone else is checking & the critical section. which has the test of in-use & running the Proc. Therefore we have to make sure that nobody else sneaks in while we check in-use? & set it to #t this is nothing but Atomic test & set - which is implemented at the hardware level.

* why can't we implement the entire program at Atomic level? → it leads to inefficiency. ie it cannot diff that if one Proc is acting on memory 'x' & the other on 'y' I can allow etc that has to be done by the program.

---

* Parallelism can occur even if the Processor isn't multiprocessor because of something called interrupts. ie if the System is running a Program & we type or press a key the hardware reads this & raises an interrupt asking the System to stop what its doing & take care of the new request. Because of such interrupts there is competition & threading. Because of this the Systems used to crash. Now that we have understood all this - these things are taken care of

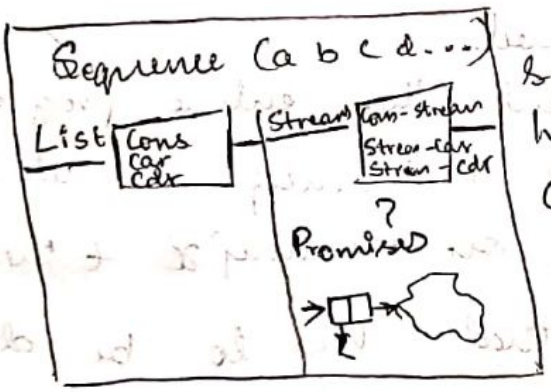
Ex: (set! x (* x x))

```
Load    r8, x   <------ Interrupt which changes value of x
Load    r9, x           now we have different values in
Multiply r8, r9         r8 & r9 & we dont get a square
Store   r8, x           causing the system to crash
```

* Functional programming is insulated from the problems of Parallelism as its free from local states ie No 'set!'

STREAMS.

Streams are an alternate implementations of an abstract idea viz. Sequence.



One way to implement a sequence is by lists which has constructor – cons & selectors car & cdr

the behaviour of stream is diffent from lists, we demonstrate it as follows.

* An integer N is prime iff its only factors are 1 & N.
  ..... iff it has no factors in [2 .. N-1]

```
(define (Prime? n)
  (null? (filter (lambda (x) (= (remainder n x) 0))
                 (range 2 (- n 1)))))
```

Program 1

```
(define (range from to)
  (if (> from to)
      '()
      (cons from (range (+ from 1) to))))
```

>(Prime? 25) #f

>(Prime? 127) #t

>(Prime? 1000000) (takes time) segmentation fault error.

* the program first creates a list of almost a million (2 lacs) numbers & then checks the remainder for each. & after all this it checks if the list is empty?

Now consider,

Program 2

```
(define (Prime? n)
  (define (iter factor)
    (cond ((= factor n) #t)
          ((= (remainder n factor) 0) #f)
          (else (iter (+ factor 1)))))
  (iter 2))
```

this Prog works much faster in cases like >(Prime? 1000000) but for a true prime (Prime? 1000003) it still has to go through all the possibilities there is no way around this

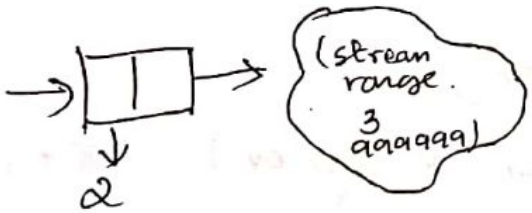
Now, we write another program,

```
(define (stream-prime? n)
  (stream-null? (stream-filter (lambda (x) (= (remainder n x) 0))
                               (stream-range 2 (- n 1)))))
```

Program 3

```
(define (stream-range from to)
  (if (> from to)
      the-empty-stream
      (cons-stream from (stream-range (+ from 1) to))))
```

here > (stream-prime? 1000000) will work as fast as the previous prog. (1000003 is same as before). Now we will understand why when we say, (stream range 2 999999) it actually starts out similar to lists, ie with a pair with car =2 but its cdr points at a 'Promise' to Compute (stream range 3 999999) ie,



thus Range is $\Theta(n)$ fuction but stream-range is a $\Theta(1)$ fuction

So, now we need to understand what a promise is, it is just a function ie,

Promise: (λ() (stream-range 3 9999999))

* Consider the last line of stream range, which says

(cons-stream from (stream range (+ from 1) to))))

(cons-stream a b)

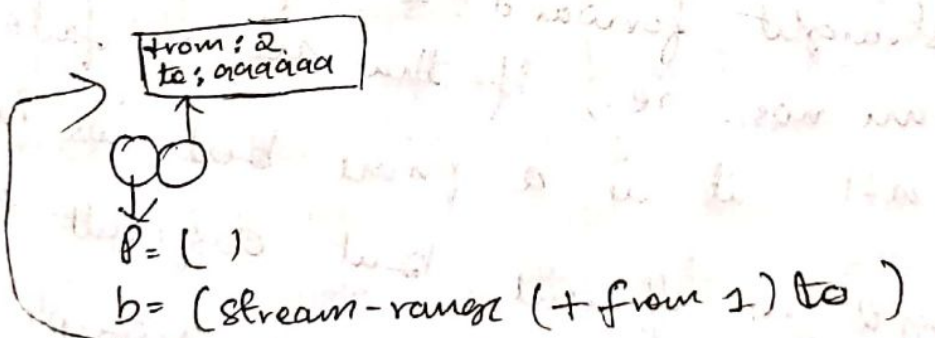
which is actually simplified to

(cons a (delay b)) where delay is a special form.

(delay Exp) ⟹ (λ() Exp)

hence in our case.

(λ() (stream range 3 9999999))



It is called promise since it remembers what to do & in what context I should do it. But it keeps it as a promise & doesn't execute it right away

Now that we know how a constructor works lets see the selectors,

```
(define stream-car car)
(define (stream-cdr s)
    (force (cdr s)))
```

force is a part of scheme standard like delay but is opposite of it.

```
(define (force P)
  (P))      just invoke it to execute it
```

here, what we have done is decouple the form of program from the sequence of events when it Runs. In the previous 'Prime?' example Program 1. is straight forward & represents how we think about prime nos. ie., if there are no factors of 'n' b/n 2 & n-1 it is a prime but its inefficient when as Prog 2 is efficient but difficult to understand & isn't straight forward. Thus we decouple the two by defining 'stream'.

```
(define (stream-filter predicate data)
  (cond ((stream-null? data) the-empty-stream)
        ((predicate (stream-car data))
         (cons-stream (stream-car data)
                      (stream-filter predicate (stream-cdr data))))
        (else (stream-filter predicate (stream-cdr data)))))
```

Hence for every no the program checks & promises are made one by the stream filter other by the stream range.

---

* Making streams as the main reason for creating streams is as such efficiency.

---

* The question is if streams works so well then why don't we make it default instead of lists, why / do we write it as a special form? Ans: There are many languages written that way which have streams as the default sequence type, these languages are functional languages ie, they don't allow progs that are not functional in style. The reason this is not the default method in non-functional progs is that in such languages there is mutation & such mutation combined with concept of promises can cause issues.

* Streams are also called lazy lists

* Now consider the prev proc

(define (range from to)
~~(if (> from to)~~ lets eliminate this 2 lines
~~()~~
(cons-stream (range (+ from 1) to))))

so we have eliminated the base case from this proc so we can expect it to get into an infinite loop but it doesn't because it will just evaluate the car & promise to do the rest.

* Fibonacci

(define f (cons-stream 1 (stream-map + f (cons-stream 1 f))))

this works because the stream f has (1 ...) so that we add (1 f ⇒ (1 1 ...) which means

f { 1 1 2 3 5 ...
(stream-cons 1 f) { 1 1 2 3 ... & so on.

2 3 5 8 ...

this is like just in time stream generation.

* Generating Primes

→ Idea: * write down all the integers starting from 2.
* the first integer on the list is prime
* Cross out all nos that are divisible by this no
* repeat second & third step

```
(define (sieve streams)
  (cons-stream
    (stream-car streams)
    (sieve (stream-filter
             (lambda (x) (not (divisible? x (stream-car streams))))
             (stream-cdr streams)))))

(define primes (sieve (stream-cdr integers)))
```

> (Primes 20), helper to print
> (SS Primes 20)

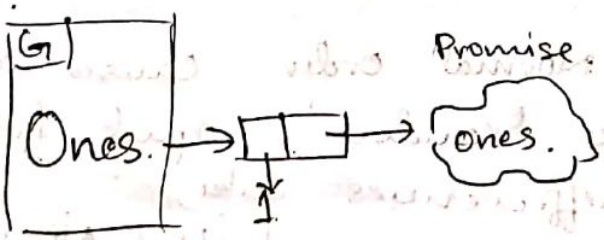
(2 3 5 7 . . . . . 71 . . .)

Thus we can create an infinite data set within the PC's memory and get the dataset of any length out of this infinite set as & when we want.

Implicit Stream definition

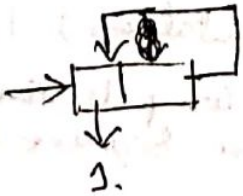
> (define ones (cons-stream 1 ones))

looks like it doesn't work if it wasn't for the 'streams' concept.



1 1 1

this is near to the car list- ie.,



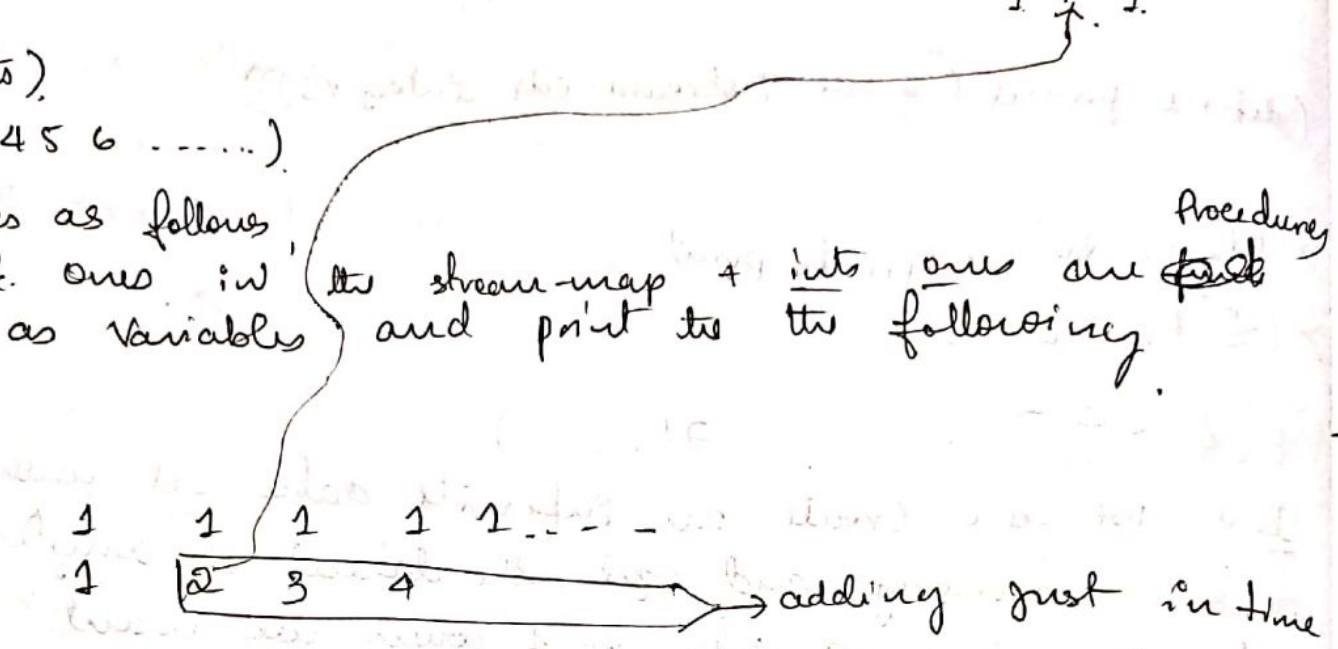
when we make the promise there isn't a 'Ones' but when we invoke it & the promise is executed Ones is created in the Global environment

∴ Ones is just a stream of infinitely many ones.

ie, (define ones (cons 1 2))
(set-cdr! ones ones)
Non-functional

but the other list was constructed in a Non-functional way. using set-cdr! but using stream we can construct it in a functional way.

```
> (define ints (cons-stream 1 (stream-map + ints ones)))
> ints
> (ss ints)
> (1 2 3 4 5 6 .....)
```



Similarities between Streams and Normal Order of evaluation

Since in normal order instead of argument values argument expressions are sent to the procedure (in contrast to applicitive order) which itself causes 'delay' which is the characteristic property of streams.

As already discussed Normal order causes problem when we use commands like 'random', apart from this it can give rise to inefficiencies like,

```
> (define (square x)
    (* x x))
> (square (some-long-computation))
```

in applicitive order we first compute this say it takes 1hr & then we square it

in Normal order we say

* long comp ↓ long comp ↓ 1hr + 1hr = 2hrs

* Streams - have a similar possibility for inefficiency, ie everytime ie when we say stream-cdr (some stream) we aren't just selecting a memory location but we are making a computation & when this computation is complicated (say Prime of 3000) it can cause a lot of delay when compared to a list (a list which contains 3000 primes)

This inefficiency is solved as follows.

(delay exp) ⟹ (memoization (λ() exp))

This way every new calculation the procedure does is automatically saved & the procedure does a look up before calculating anything new.

Understanding Memoization property in streams

```
>(define rands (cons-stream (random 10) rands))
rands
>(ss rands)
> 2 2 2 2 2 2 ....
```

here we get the same no repeatedly because rands is a variable & not a func which means random 10 is calculated only ones & the 1st template of '2 rands' is repeated again.

↓ 2 rands

But,

```
> (define (rand-stream) ← Procedure
    (cons-stream (random 10) (rand-stream)))
rand-stream
> (define rands (rand-stream)) ← variable
> (ss rands)
(7 8 1 2 0 3 3 0 5 5 . . .)
```

here the 'random 10' is called ie invoked again & again

only if we ask the sys to recalculate rands, ie

```
>(ss rands)
(7 8 1 2 0 3 3 0 5 5 . . .)
```

we get the same values ie because the System doesn't recalculate instead gets values from memoized data.

* There are 3 big ideas about why we have streams. the first ex. of calculus
* Making more efficient a finite computation Ex: Primes
* We can use the stream to represent infinite data streams
* It allows the representation of time varying information. Or. [Ex: representing random nos] representing info coming in overtime

[Stream has the same sense as 'Streaming' video]

* Since stream allows the representation of time varying info it helps represent the properties of Object Oriented programming with in functional programming domain & its a very powerful idea. when we are dealing with time varying problems for ex: I'm doing withdrawals & deposits continuously how much money do I have. We use oop concept of mutation & as we have seen this causes concurrency issues for which we have to build tedious mechanisms to overcome. A natural thing to have is a structure with sequence of events in the time in which you are running Chanoz is running.

* Ex: >(define (user-stream) → scheme primitive
(cons-stream (read) (user-stream)))

Userstream

* the behaviour of this Program is interactive

>(ss (user-stream) 5)

1000
100
-300
50
5000
} imagine deposits

>(1000 100 -300 50 5000 . . .)

* it is as though I have all the details of my future deposits / withdrawals available with me right now.

> (define my-account (cons-stream 1000 (stream-map + my-account (user-stream))))

account starts with 1000

(ss my-account 5)

> 1000 > -300 > 4000 > -60 > 1000

> (1000, 1100, 800, 1200, 1140 . . . .)

We addressing stream of 'myaccount' viz stream of all the balances my account is ever going to have with stream of all the deposits & withdrawals I am ever going to make.

* Thereby we just modeled Time Varying local state without using mutations, Entirely functionally

* But Streams are not used for bank accounts because just like concurrency issue with Helism here eventhough there can't be concurrency issue since we are using functional programming but we do have to handle parallelism & it gets complex when we have streams

---

* Everything in a stream has to be reachable in finite no of steps ie,

Consider. >(SS ints)
(1 2 3 4 5 6 ---)
> (define neg-ints (stream-map - ints))
neg-ints
> (SS neg-ints)
(-1 -2 -3 ---)
> (define all-ints (cons-stream 0 (stream-append ints neg-ints))
> (SS all-ints)
(0 1 2 3 4 ---)

we have to 'stream-cdr' infinite no of times to get to -1. Therefore it is as good as not having them. (but in certain sense we do have them)

* we can make a stream of all the rational nos but not real nos because we wont be able to reach in finite no of steps

Shell programming

Concept:

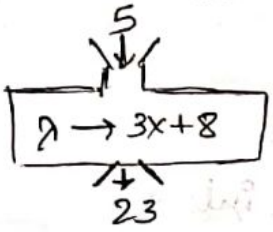
The kernel is a low level abstraction that is in charge of the computer and responsible for making the applications run. Shell is a high level abstraction around the kernel for humans to interact.

People keep writing new shells.

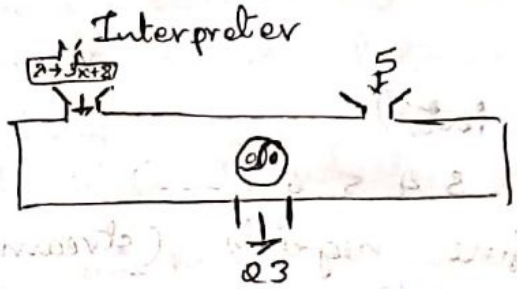
Metacircular Evaluation

The idea of interpreter:

Usual Procedure:



Interpreter



The interpreter is a universal program, that can take a procedure in some sort of encoding (like text) & args as inputs & give output.

We wrote an early version of scheme interpreter in scheme ~~the exact~~ the reason & justification for this was discussed along with it, but it should also be noted that very standard interpreters like Stk are also written for the most part in scheme itself. Except for modules on storage allocation system etc which need to be written in something else

Scheme Interpreter in Metacircular Evaluation

in Scheme 1. we say (eval exp).

in Metacircular Evaluator we say (eval exp environment)

Environment is list of frames which has bindings from names to values.

```
(define (scheme)
  (display '>')
  (Print (eval (read) the-global-environment))   REPLoop
  (scheme))
```

↳ env
it's actual a global variable whose value is a list of just 1 frame. ie the global frame.

```
(define (eval exp env)
  (cond ((self-evaluating? exp) exp)        → No env here this is a special case
        ((symbol? exp) (lookup-in-env exp env))
        ((specialform? exp) (do-special-form exp env))
        (else (apply (eval (car exp) env)
                     (map (lambda (e) (eval e env)) (cdr exp))))))
```

will contain each special form & how to evaluate it

Procedure we are calling

Procedure application.

Same as before λ is introduced just to make sure we indicate the env for evaluation

```
(define (apply Proc args)
  (if (Primitive? Proc)
      (do-magic Proc args)
      (eval (body Proc)
            (extend-environment (formals Proc)
                                args
                                (Proc-env Proc)))))
```

this is just a selector for abstract Procedure data type

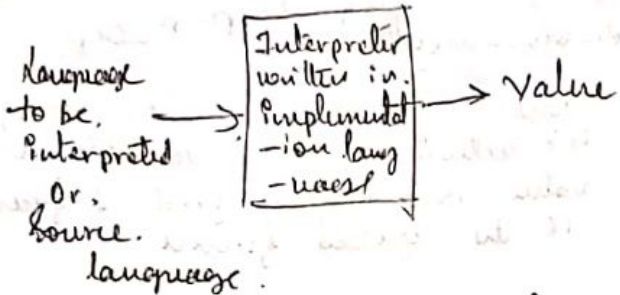
shows to which environment to extend

↳ selector for Procedure's right bubble

Here is the Big change from Scheme-1. In Scheme-1 we used substitute to replace args with arg values. in here we donot use substitution instead we pass an an environment & in this environment we evaluate the Procedure Body directly. The environment is got by extending the exist env with a new frame consisting of the formal Parameters we find in the left bubble of Procedure

* We don't extend to current environment that would be dynamic scope. Scheme uses lexical scope therefore we see what the right bubble of the procedure points to and extend accordingly

* Suppose we are writing a interpreter for some language.



So, what are the properties we would want in a source language we need it to be 'very simple' no complicated notation should it do different things in different ways etc. the implementation language on the other hand should be very powerful, should have capabilities for manipulating structured data, text, OOPs etc

* A possible reason as to why ( ) is such an important part of scheme can be understood by looking at Logo another Lisp language create for children it doesn't use ( ) which means we cannot distinguish when we are invoking a procedure and when we are passing it as data which in turn means we cannot pass it as data. which stunts Logo's usability & capability.

Lexical Scope

Local state variable (OOP)

fewer bugs.

Faster compiled code

Dynamic Scope

First class expressions (instead of first class procedure) make it simpler

Easier debugging

Allows "semi global" variables [i.e I define a global variable and in each helper procedure I assign a value to it accordingly

* Lexical scope + λ → extends to environment in which the procedure was created (not called) give us the ability to create local state variables but are persistent even though they are local they don't disappear between procedure calls because the dispatch procedure for the object remembers the environment that contains the binding for local variables. These features give us Object Oriented programming for free. [Even though C has lexical scope it doesn't support OOP because it doesn't have λ i.e we cannot define a procedure within a procedure].

Map Reduce

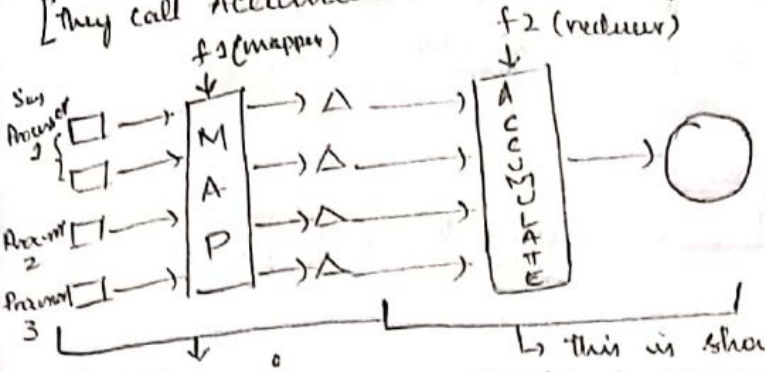
Parallelism in the form of map reduce - An algorithm for handling parallelism that uses functional ideas & makes a lot of things easy (which otherwise would be hard)

The proprietary implementation of this idea is from Google. The open source implementation is called 'hadoop.' [Btw strictly speaking the actual implementation is not completely functional]

Google has to process huge amounts of data everyday & do things like make indexes for web pages so that when we search for something they needn't go through everything & they already know which pages are going to be relevant to your topic. That kind of processing can't be done on a single processor computer instead what they do is they buy racks & racks of these computers & put them in a rack. They take the data & distribute them among these processors. ie. One file can a live in pieces in all these different computers & when we want to process it we ask all these comp to do their bit of processing. The problem is computers fail once in a while (if we are using 1000 comp the prob of failure is 1000 times the failure of 1 comp) so we have to have the same data stored in multiple systems + A software system that is robust against system failure i'e identify when comp dies and reassign the task.

Before introduction of this Mapreduce kind Algorithm ppl at google had to worry about all these + concurrency issues. In 2003 they realised that most of what they are doing can be simply be represented by Map and Accumulate

[They call Accumulate as Reduce].



Parallelism as apparent

↳ this is shown in a simplified version. even this is a bit activity mostly see the Detailed dia.

* this is the basic version of the idea. they built this in a way that was flexible enough to solve all Googles particular problems & also the robustness + Parallelism (or concurrency) so that the application developer needn't have to think about it.

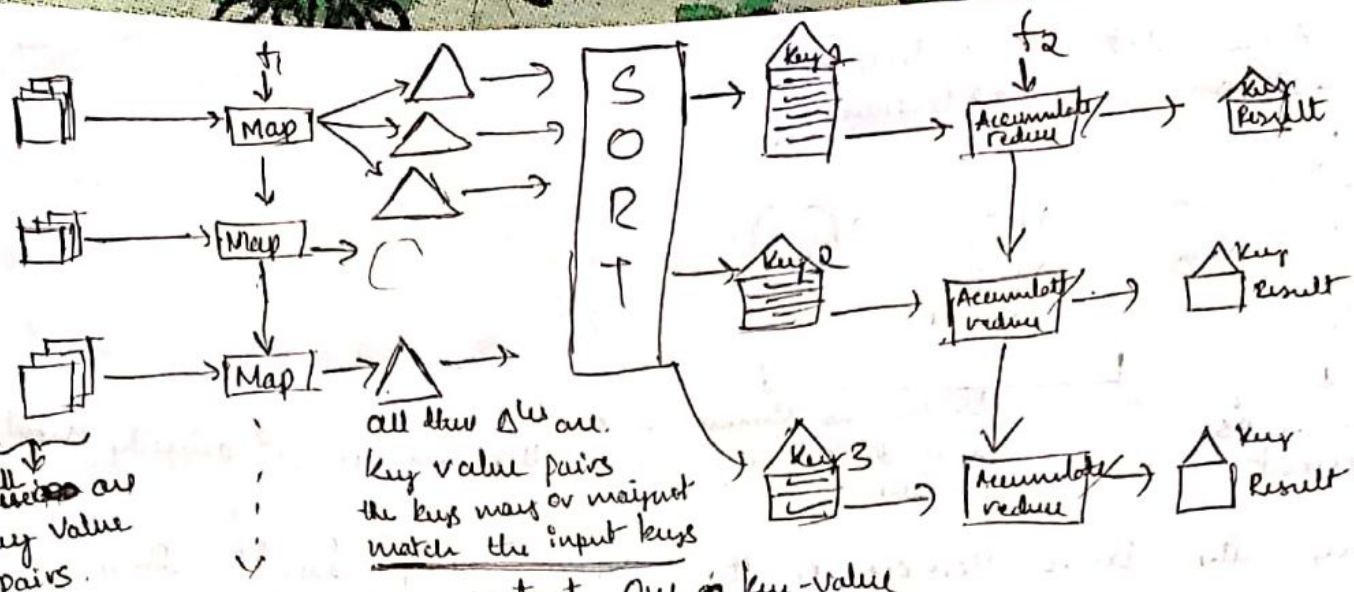
Conceptual representation

(define (mapreduce mapper reducer base-case data)
(accumulate reducer base-case (map mapper data))

..... > function used for mapping
——— > function for accumulate

* Mapreduce is designed to work only on Key-value pairs, as input
↓
like Assoc list or Dictionary in python

* the key-value idea is introduced to establish the similarity between values so that they can be grouped and then reduced. why do we need to do this? Suppose instead of questions like 'I want one no which is equal to the sum of all the words shakespeare ever wrote' we are facing 'for every word in shakespeare tell me how often that word is used'. in such cases we group the same word appearing multiple times based on its key & then apply accumulate on each (in red) for the former the top diagram holds good.



mappers can output one key-value pair or multiple key-value pair or no key value pair. (mapper in input is a sentence & we want output on each word.)

the input key to map & output key may or may not be same

* Sort: the system sorts based on 'key' & all the files with same 'key' are grouped into one group & sent to one computer

* the reduction happens per key basis & thus is val & the accumulate doesn't get any keys i.e once they are sorted the keys are discarded.

* If we were to put a scheme wrapper around this viz written in Java / hadoop it will have to Run scheme interpreters on the worker machines in the cluster & ship out your local environment to all the machines to carry out f_1 and f_2 which are scheme functions. this will obviously slow things down

```
(define (mapreduce mapper reducer base-case data)
  (Groupreduce reducer basecase
               (sort into buckets (map mapper data))))
```

```
(define (groupreduce reducer base-case buckets)
  (map (lambda (subset)
         (make-kv-pair
          (kv-key (car subset))
          (accumulate reducer base-case (map kv-value subset))))
       buckets))
```

substitute it just car

```
(define (my-mapper kv-pair)
  (map (λ (wd)
         (make-kv-pair wd
                       (kv-key kv-pair)))
       (kv-value kv pair)))
```

First Variant Metacircular Evaluator : Analyzing Evaluator

because it is not translated to machine code instead it is translated to stk scheme from metacircular procedure [this kind of same language to same language compilers are present in Java, Pascal etc]

* It's a sort of Compiler.
* Consider, factorial function.

```
(define (fact n)
    (if (= n 0)
        1
        (* n (fact (- n 1)))))
```

Java → Java Virtual machine → JVM interpreter (JVM)

* Let's see what happens when we ask for the factorial of 5

```
> (fact 5)
```

this is a procedure call & how do we know that → see the compiler code → the system keeps on checking all the conditions from self evaluating? to cond? & it doesn't match & then it comes to application? (Viz basically a fancy word for procedure call) i.e a list but not any of the above (i.e self evaluating? to cond?).

then we evaluate 'fact' check - self evaluating - false & then variable? Yes → we lookup its value in the current environment viz the global environment because we are typing at the scheme prompt.

then we evaluate 5 viz self-evaluating. & then we are ready for 'apply'.

Apply sets up a new environment where 'n' from the code is binded with 5. & with this we evaluate the body using the same evaluate as before, we find: if ✓

application ✓
= → primitive ✓
n → var ✓
0 → prim ✓
1 → prim ✓
application ✓
+ → prim
n → var

→ its a variable we find in the global env (Primitive)

```
(define (me-eval exp env)
  (cond ((self-evaluating? exp) exp)
        ((variable? exp) (lookup-variable-value exp env))
        ((quoted? exp) (text-of-quotation exp))
        ((assignment? exp) (eval-assignment exp env))
        ((definition? exp) (eval-difinition exp env))
        ((if? exp) (eval-if exp env))
        ((lambda? exp)
         (make-procedure (lambda-parameters exp)
                         (lambda-body exp)
                         env))
        ((begin? exp)
         (eval-sequence (begin-actions exp) env))
        ((cond? exp) (me-eval (cond->if exp) env))
        ((application? exp)
         (me-apply (me-eval (operator exp) env)
                   (list-of-values (operands exp) env)))
        (else
         (error 'unknown expression type - EVAL 'exp))))
```

→ maps evaluate on operands

then we come to (* n (fact (- n 1)))) — this part.

application ✓
fact → variable → value → same procedure
application ✓
- → Prim
n → var.
1 → Prim.

(- n 1) = 5 - 1 = 4. ∴ we have fact(4)

Now we call factorial procedure with n = 4 & we again endup at apply. What apply does is that it creates a new environment with n bound to 4 & in that environment it evaluates the body starting from 'if' & everything runs same as before except for the value of n.

But for every ~~Dec~~ sub-exp in the procedure the system has to look at all the conditions in 'eval' again which it has already done before (for n = 5 case). So it makes sense if the system can remember what each expression/sub expression is. [the idea is like memoization but not memorization]

Even after implementing this factorial is going to remain a $\Theta(n)$ procedure. but it is going to make a significant difference to the constant factor

So how do we implement this idea? :- We split up the 'eval' (i.e me-eval) into 2 parts one that only looks at the form of the procedure & once it is seen we pass the environment in which to calculate it. ie, (define (me-eval exp env)
((analyze exp) env))

```
(define (analyze exp)
  (cond ((self-evaluating? exp)
         (analyze-self-evaluating exp))
        ((variable? exp) (analyze-variable exp))
        ((quoted? exp) (analyze-quoted exp))
        (assignment?
         definition?
         if?
         lambda?
         begin?
         cond?
         application?
        else
          error)

(define (analyze-self-evaluating exp)
  (lambda (env) exp))

(define (analyze-quoted exp)
  (let ((qval (text-of-quotation exp)))
    (lambda (env) qval)))

(define (analyze-variable exp)
    (lambda (env) (lookup-variable-value exp env)))
      .
      .

(define (analyze-if exp)
  (let ((pproc (analyze (if-predicate exp)))
        (cproc (analyze (if-consequent exp)))
        (aproc (analyze (if-alternative exp))))
    (lambda (env)
      (if (true? (pproc env))
          (cproc env)
          (aproc env)))))
```

```
(define (analyze-lambda exp)
  (let ((vars (lambda-parameters exp))
        (bproc (analyze-sequence (lambda-body exp))))
    (lambda (env) (make-procedure vars bproc env))))

  .
  .
  .

(define (execute-application proc args)
  (cond ((primitive-procedure? proc)
         (apply-primitive-procedure proc args))
        ((compound-procedure? proc)
         ((procedure-body proc)
          (extend-environment (procedure-parameters proc)
                              args
                              (procedure-environment proc))))
        (else
         (error
          "unknown procedure type -- execute-application"
          proc))))
```

- thus it is pritty straight forward that since analyze returns a 'λ' expression (with 'env' as an argument) it is analyzed only once no matter how many recursions happen, only the environment changes. thus we have made the compiler / interpreter much more efficient.

* Most work in designing compilers is finding special cases and making it Robust. like getting non-scheme expressions etc

---

Lazy Evaluation

Now, consider a prog written with a goal to generate the following sequence.

(1 1) (1 2) (1 3) (1 4) (1 5) (1 6) . . .
(2 2) (2 3) (2 4) (2 5) (2 6) . . .
(3 3) (3 4) (3 5) (3 6) . . .
(4 4) (4 5) (4 6) . . .
(5 5) (5 6) . . .
(6 6) . . .
.
.
.

Produces the first element seen as a special case

the Prog will be as follows

```
(define (pairs s t)
  (cons-stream
    (list (stream-car s) (stream-car t))
    (interleave
      (stream-map (lambda (x) (list (stream-car s) x))
                  (stream-cdr t))
      (pairs (stream-cdr s) (stream-cdr t)))))
```

delayed by cons-stream

here the first-element (1, 1) is seen as a special case in the prog which looks un-aesthetic, here is another attempt

```
(define (louis-pairs s t)
  (interleave
    (stream-map (lambda (x) (list (stream-car s) x))
                t)
    (louis-pairs (stream-cdr s) (stream-cdr t))))
```

Now, this will not work, the reason being the absence of cons-stream; As discussed cons-stream is a special form described such that it delays the evaluation of its cdr. [the essence being the sequence we are trying to generate should behave as a stream within itself & the prog above doesn't allow that to happen]

Another way to look at it is that the reason we can have recursions in stream without a base case is because eventhough in essence we are generating a infinite list the delay prevents the procedure from going into an infinite loop [viz absent in the above code]

One way to fix the above code is by introducing the delay & force artificially as follows,

Fixing Louis' code

```
(define (fixed-pairs s t)
  (interleave-delayed
    (stream-map (lambda (x) (list (stream-car s) x))
                t)
    (delay (fixed-pairs (stream-cdr s) (stream-cdr t)))))
```

```
(define (interleave-delayed s1 delayed-s2)
  (if (stream-null? s1)
      (force delayed-s2)
      (cons-stream (stream-car s1)
                   (interleave-delayed (force delayed-s2)
                                       (delay (stream-cdr
                                               s1)))))))
```

Another way we can make it work more elegantly is by having Normal order instead of applicative order. [i.e we wont evaluate argument expressions until we really need them]. Now lets see how do we achieve this by modifying our Metacircular evaluator.

(define (mc-eval exp env)
  (cond ((self-evaluating? ...
         variable?
         ⋮
         lambda?
         ⋮
        ((application? exp)
         (mc-apply (actual-value (operator exp) env)
                   (operands exp) env))
        (else (error ...

Same as before.

** Instead of mc-eval we don't evaluate Operands (viz what defines Normal order)

Instead of this we can map delay over argument expressions & later when required we force it with environment either way we cannot stick to our usual understanding of Normal Order as just sending argument expression alone we are sending argument expression & the environment for its evaluation as promises

(define (actual-value exp env)
  (force-it (mc-eval exp env)))

↳ checks if we have a promise
if not return it
if so force it

as the result of eval. can be a promise we force it

Reminder: what is a promise
((λ (x) x)
 (+ 2 3)) → this procedure returns 'x' as a promise to evaluate '2+3'

```
(define (me-apply procedure arguments env)
  (cond ((primitive-procedure? procedure)
         (apply-primitive-procedure
          procedure
          (list-of-arg-values arguments env)))   ; either we have received args as text will env or just promises
        ((compound-procedure? procedure)
         (eval-sequence
          (procedure-body procedure)
          (extend-environment
           (procedure-parameters procedure)
           (list-of-delayed-args arguments env)
           (procedure-environment procedure))))
        (else
         (error
          'unknown ))))
```

here we do the evaluation

returns list of Promises instead of values which will again start from eval & get evaluated one by one.

```
(define (list-of-arg values exps env)          ; → it's basically Map
  (if (no-operands? exps)
      '()
      (cons (actual-value (first-operand exps) env)
            (list-of arg values (rest-operands exps)
                  env))))
(define (list of delayed args exps env)
  (if (no-operands? exps)
      '()
      (cons (delay-it (first-operand exps) env)
            (list of delayed args (rest-operands exps)
                  env))))
```

```
(define (eval-if exp env)
  (if (true? (actual-value (if-predicate exp) env))
      (me-eval (if-consequent exp) env)
      (me-eval (if-alternative exp) env)))

(define (driver-loop)
  (prompt-for-input input-prompt)
  (let ((input (read)))
    (let ((output
           (actual-value input the-global-environment)))
      (announce-output output-prompt)
      (user-print output)))
  (driver-loop))
```

This is Basically the usual Read - eval - print Loop

but instead of eval → we have actual-value to retain the ~~property of Normal order~~ printout ↓ even when we have promises

value

* An interesting point here ~~in that~~ in the previous case of Analyzing evaluator we had to change almost ~~all~~ the entire compiler code (all the cond clauses) but the resultant nature of the language is almost the same. But here we have introduced very little change but the resultant impact on the language is significant (change from applicative order to Normal order)

* The achilles heel of Normal order is mutation

Force it :

```
(define (force-it? obj)
  (if (thunk? obj)
      (actual-value (thunk-exp obj) (thunk-env obj))
      obj)).   ↳ calls force it again on new-val
```

* thunk is a procedure with no arguments ie its identifying if it is a promise.

* here we have mutual recursion (Not the branching type of mutual recursion we had in trees) we have to handle cases like (I (I (I (x))))

```
(define (delay-it exp env)
  (list 'thunk exp env))
```

---

* We had discussed a draw back of Normal order ie

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

(square (fibonacci 100))
```

↳ calculated twice in Normal order. but once in applicative order.
here insufficient

we can overcome this by using Memoization

Memoization Version of force it

```
(define (force-it obj)
  (cond ((thunk? obj)
         (let ((result (actual-value
                        (thunk-exp obj)
                        (thunk-env obj))))
           (set-car! obj 'evaluated-thunk)
           (set-car! (cdr obj) result)
           (set-cdr! (cdr obj) '())
           result))
        ((evaluated-thunk? obj)
         (thunk-value obj))
        (else obj)))
```

↳ just data abstraction

the idea is to replace the obj expression with the calculated value once the first evaluation of the expression is done

---

List of all the changes we made to change the applicative order to Normal order

me-eval : Application? clause → just 1 clause.

me-apply : changes in both primitive & non primitive procedure.

REPL, IF : me-eval → actual-value.

we invented : delay-it, force-it

* the Result of all this is that instead of having a applicative natured language and having streams as a special case, we have made everything work the way streams work

---

The concept of continuation

Consider the following procedure,

(+ (* 2 3) (- 10 7))

Now we can ask what happens to the result of the subexpression * 2 3 the answer is

(λ (result) (+ result (- 10 7)))

* Continuation is generally 'what to do next'.

* Now suppose we define all the primitives as continuations ie cp+, cp-, cp*, cp/ as follows

(define (cp* a b cont)
  (cont (* a b)))

and so on for other primitives

we can re-write (+ (* 2 3) (- 10 7))

as (cp* 2 3 (λ (Prod)
  (cp- 10 7 (λ (diff)
    (cp+ Prod diff
      (λ (x) x)))))))

* This idea comes in handy when the programmer doesn't want to use recursion + it's like ↳ Procedure call & return & wants to clearly express the sequence of actions in the explicitly program. If we see the prog again it tells the System to do x^n first calls the result prod then differe Calls the result diff then addition & returns it

Logic Programming [Ex: Prolog]

Till now we have seen the following Programming paradigms

1) Functional.
2) Object Oriented.
3) Client / Server
4) Data Parallel (when data is distributed among computers and all of them are doing the same stuff ie Mapreduce)

the 5th one is

5) Logic Programming / Declarative Programming.

Since this is a completely different programming paradigm these differences are reflected in its interpreter as we will see, ~~we in a Normal Prog~~ Till now our idea of programming was to create Procedures or Recipes for executing something but in logic programming we give the System some Data & then ask it Questions.

Here is a Demo,

;;; Query input:

(assert! (Brian likes Potstickers))

Assertion added to database

;;; Query input:

(assert! (Ramesh likes Potstickers))

Assertion added to database

;;; Query input:

(?who likes Potstickers)

? indicates a variable

;;; Query results:

(Ramesh likes Potstickers)
(Brian likes Potstickers)

} Did have the Procedures that we wrote returned one value like the result of the Query. You'll be more than one this is a very important property of logic programming

;;; Query input:

(assert! (Brian likes ice cream))

;;;

~~(?who likes)~~

(Brian likes ?what)

;;;

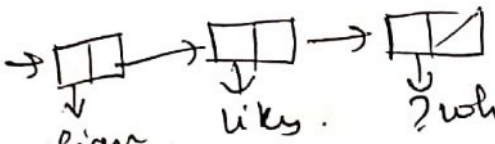
(Brian likes Broccoli)
(Brian likes Potstickers)

we don't get the ice cream Result

;;;
(brian likes ?what)

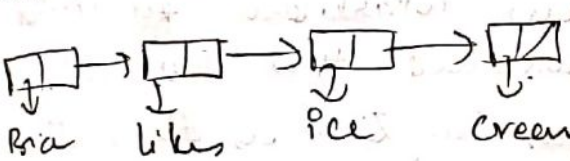
;;;
(Brian likes ice cream)
(Brian likes Broccoli)
(Brian likes Potstickers)

here the pattern matching works by matching the cars and cdrs since this is list.



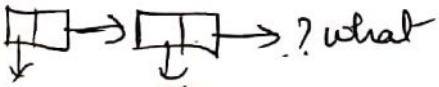
matches with

Doesn't match with.



if I had said.
(Brian likes (ice cream)).
it would have worked.

But, in the second case we have,



Since ?what becomes the spine of this pattern & therefore matches all the 3.

This is the concept of pattern matching.

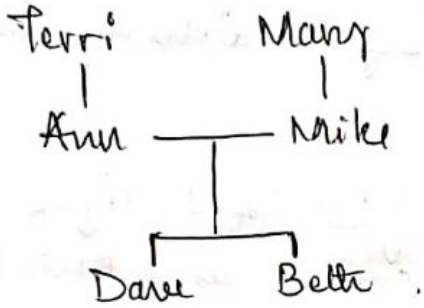
But, by convention we use the relation / verb first i.e

assert! (likes Brian. Potstickers)
(likes Brian ?what).

* But the point here is that it is working by Pattern matching & there is no Procedure called likes. the reason this convention is developed in efficiency because the way that assertions are stored in our systems is by indexing them by their first words, i.e, all the sentences with first word 'like' are grouped together & so our

---

Consider the following Example.



if we want to build this relationship in our database then we need to make numerous assertions for each node (because the sys won't understand that Ann & Mike will be the parents of Dave if I just say Dave is the brother of beth) This can be overcome by making the system understand the rules

```
(define (add query)
  (add-rule-or-assertion!
    (add-assertion-body
      (query-syntax-process (list 'assert! query)))))
```

```
(add '(mother-of Dave is Ann))
(add '(father-of Dave is Mike))
(add '(mother-of Beth is Ann))
F of Beth is Mike
M of Mike is Mary
M of Ann is Terri
```

conclusion

```
(aa '(rule (grandmother-of ?gc is ?gm) ;; gc = grandchild
                                        gm = grandmother
           (and (mother-of ?parent is ?gm)
                (or (mother-of ?gc is ?parent)
                    (father-of ?gc is ?parent)))))
```

Body

Now we can ask questions about which we haven't made any explicit assertions like,

(grand-mother-of ?who is Terri)

> (gra— of bette is Terri)
(g— of dave is Terri)

* while writing rule we write the conclusion first. followed by the body.
* there are 3 key words 'and', 'or' & 'not' with their usual meanings (not has a special extra meaning discussed later)
* here we should note that rule is not exactly saying 'for each element in — — check — —' it is just defining what being grandmother means.

Working Mechanisms

Step.	Scheme	Query
0.	Evaluate subexpressions	Nothing to evaluate
1.	Bind parameters to arguments	1. Try to match conclusion (Bind Vars to Term)

Explanation of Rule 1. in Query.

when we ask (grandfather ?who is Terri)

we match with the conclusion (grandfather ?gf is ?gm)

& then make the binding

?gm → Terri
?who → ?gf

2. Evaluate body.	2. Evaluate body as a query (just as we had typed it + constrained by the above bindings)

what the Read-Eval-Print-loop does is it gets back a result & prints it, what the query-driver loop does is it doesn't get back a single result instead it gets back a stream of frames. Each frame is a binding of variables to values, we have a bunch of them possibly infinitely many. the driver takes each frame & takes it back to query & based on the frame plugs into the query based on the frame & then prints it out.

- (append '(a b) (c d e) ?what)

;;; Query results:

(append (a b) (c d e) (a b c d e))

for comparison scheme version of append

```
(define (append a b)
  (if (null? a)
      b
      (cons (car a) (append (cdr a) b))))
```

in query.

```
(a b)  (c d e)  d
```

name given for this recursive result

```
(aa '(rule (append (?u . ?v) ?y (?u . ?z))
           (append ?v ?y ?z)))
```

here we are saying ?z is the result of appending ?v & ?y

```
(aa '(rule (append () ?y ?y)))
```

here the system doesn't know that there is a special relation called append instead it knows that there is a list of length 4 whose first word is append. & there can be more than 1 such rule & that is how we achieve if, else, cond. without actually using them.

* Another important thing to remember is that we cannot use composition of function approach on these lists because these are procedures & these are no return values. ie,

(mother ?gm (parent ?gc))) is wrong
doesn't return anything

Since this is not an algorithm but just a pattern matching it can go backwards & give the possible bindings
;;;

(append ?this ?that (a b c d e))

will give all the possible append of a variable that can give (a b c d e) total 6

() (a b c d e)

(a) (b c d e)

.
.
.

(a b c d e) ()

Logic Programming Under the hood

Query system works by pattern matching

↓

It's done in list structure

↳ lists & sublists

it may or may not include variables

Pattern (query) vs Just text (assertion in the form of data) : Match → matches word to word whatever is in the place of variable is matched

Pattern (query) vs Pattern (conclusion of rule) : Unify.

In the case of matching/Unifying with rules it can be a bit complicated because in the place of the variable there can be another variable Ex:

Query : (foo ?x) corresponding rule : (foo (the ?y))

?x = (the ?y)

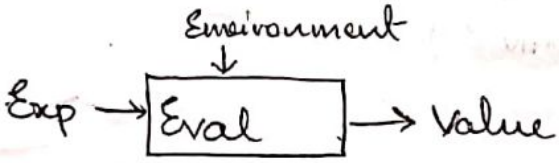
if we set say ?y = elephant then we dont have a problem

but instead if we set ?y = ?x then we are in trouble.

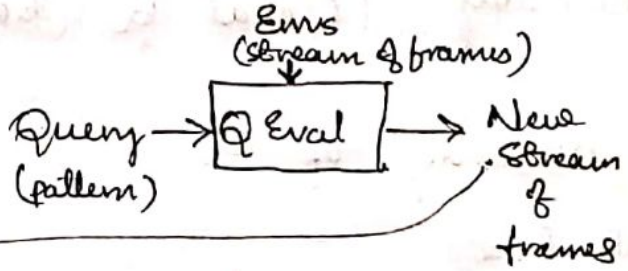
Getting the logic right for when 2 different things can be equal was the hard part of logic programming & this Algo is called Unification

---

Scheme



Query language



* in Query we get more than one value as output because we do pattern matching in multiple environments & print the result

1. New bindings added - we match the conclusion of the rule & then we evaluate the body & find new bindings

2. Frame cloned - Qeval also has database of all assertions as input along with Query & env & we match the query with one of the assertions & put it in the output

3. Frame rejected : when matching an assertion/query give rise to bindings which conflict with the bindings we already have.

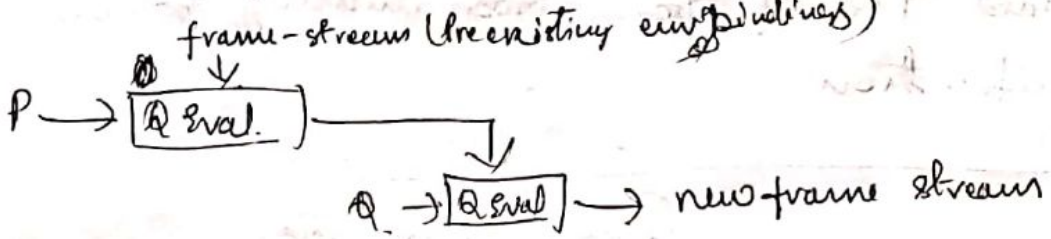
* Global environment is maintained as a stream of frames ∴ the initial stream is {()} stream with a null frame

* the function of Qeval is to simply match the query with each environment (frame) in the stream of environments

* It does for each ∂ (stream) take the query we typed & for every variable replace it with the binding we find in this frame & print that

* Working of AND.

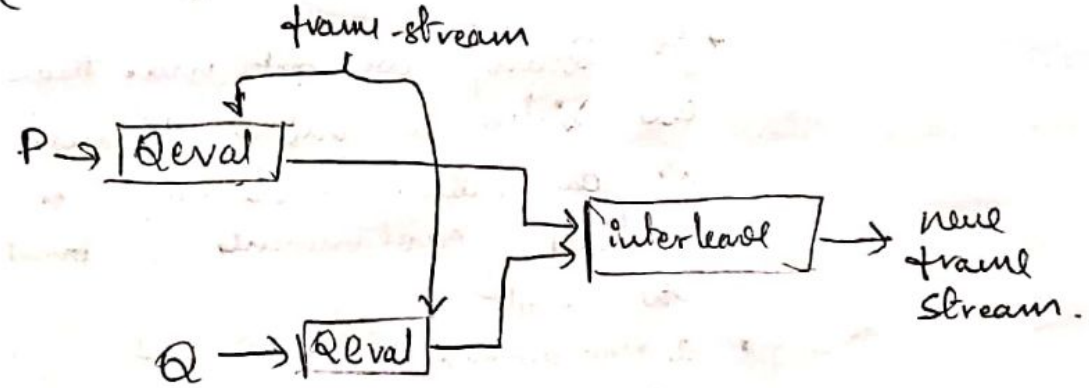
(AND P Q)



Basically (Qeval Q (Qeval P environment))

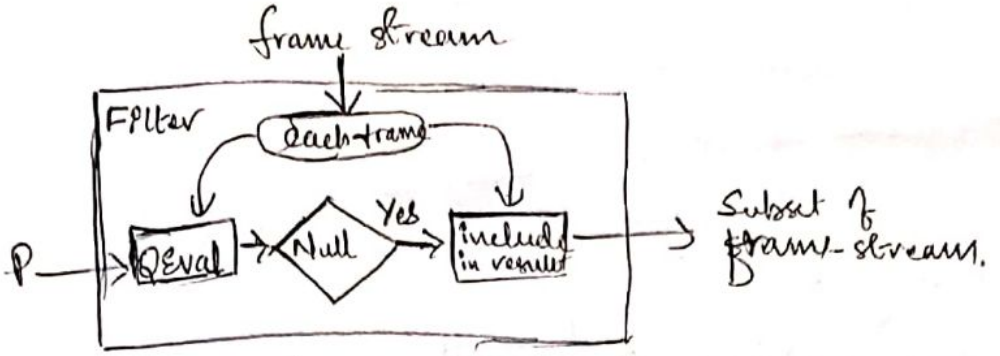
* Working of OR.

(OR P Q)



[Interleave - we cannot append as this could be infinite streams]

* Working of qeval not



we take P qeval it in a particular frame/env if we get a null then we include that frame in result → repeat

---

* Query system is Dynamically scoped. therefore the bindings in the rule we apply at a particular instance has to be different from any previous bindings of the same name. so each variable is prefixed/suffixed with a no from a counter.

Consider the following code.

(qa '(rule (append (?u . ?v) ?y (?u . ?z)) → conclusion
(append ?v ?y ?z))) → body

(qa '(rule (append () ?y ?y))) → rule won't only conclusion no body

Now if we were to say.

(append ?a ?b (aa bb))

we have 3 possibilities which together constitute our result

() (aa bb)
(aa) (bb)
(aa bb) ()

(append ?a ?b (aa bb))
(append () ?y1 ?y1)

append = append.

?a = ()
?b = ?y1.
?y1 = (aa bb)

frame.

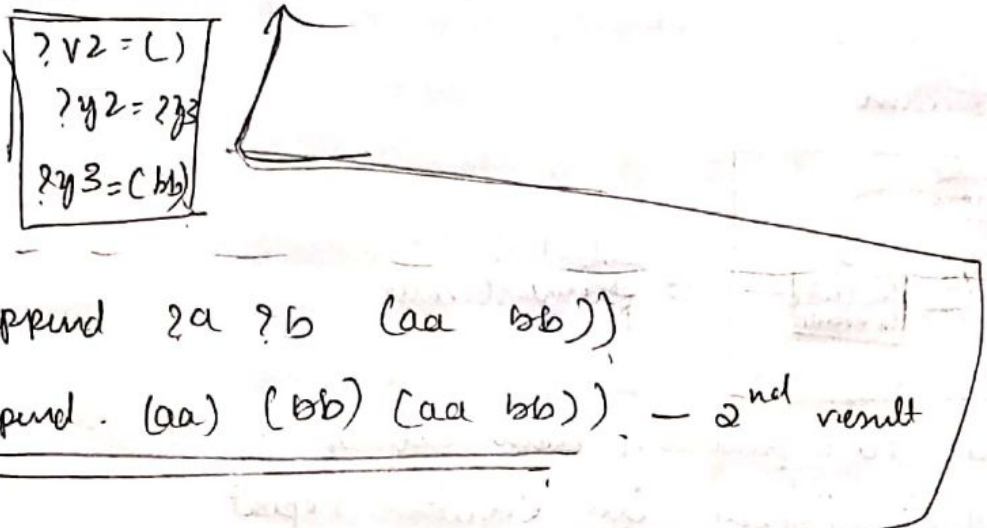
→ 1 result

(append ?a ?b (aa bb))
(append (?u2 ?v2) ?y2 (?u2 . ?z2))

?a = (?u2 . ?v2)
?b = ?y2
?u2 = aa
?z2 = (bb)

Now we go to the body
(append ?v2 ?y2 ?z2)
(append ?v2 ?y2 (bb))
Now we look for other rules (continued in next pg)

(append () ?y3 ?y3)



Now instead of this if we take the recursive Rule we get the 3rd Result.

This whole process explained above is called 'Unification'. ~~The Unifier Algorithm~~ The most significant thing to note here is that this 'Unifier' Algorithm is a Universal algorithm i.e., it can solve any solvable problem give we give all the data. i.e in all the previous paradigms we had to write the algo to do a particular task but in logic programming we just give the Data & define things & the System uses its Universal 'Unifier' Algorithm to find the solution. Thats how powerful this Algorithm is

i.e. in the append example it might appear as if we have written an algo but we haven't we have just defined what append means & the System has used the 'Unifier' to do the rest. in another example when we say $n!$ $\begin{cases} \text{if } n = 1 \Rightarrow n! = 1 \\ n \cdot (n-1)! \quad \text{if } n > 1. \end{cases}$

we are not writing the Algorithm but we are just defining what factorial means.

Review.

Abstraction → Abstraction hides

Abstraction.

Voluntary submission to a discipline in order to gain expressive power.

1. Functional Programming.

$f(g(x))$

Focus: Repeatable input-output behaviour, composition of functions to layer complexity

hidden: Side effect mechanisms (assignment) internal control structures of procedures

even if the program we write is functional what happens inside the system is full of states & isn't functional, this is hidden.

2. Data Abstraction.

Focus: Semantic view of data aggregates → Importance is given to the meaning of the data rather than its form or how and the bits stored in the memory

hidden: Actual representation in memory

3. Object Oriented Programming

Focus: time varying local state metaphor of many autonomous actors

hidden: Scheduling of interactions within the one computer procedural methods within an object

=> OOPs is good for programming things that change overtime like 7! doesn't change overtime but how fast is my car going does.

4. Streams

focus: Metaphor of parallel (not necessarily hardware parallelism) operations on data aggregates
Signal processing model of computation
hidden: actual sequence of events in the computation

5. Programming languages

focus: Provide a metaphor for computation embody common elements of large groups of problems

hidden: technology-specific implementation medium - storage allocation, etc.

Ex: functional languages like scheme/logo vs logical prog like query. Why diff languages? their design principles etc

6. Logic Programming

focus: declarative representation of knowledge inference rules

hidden: Inference algorithms

Note: Each of these abstractions can be approached 'from above', focusing on the view of computing that the abstraction provides, or 'from below', focusing on the techniques by which the abstraction is implemented. In the metacircular evaluator we emphasize the view from below, since we have been working all along with the view from above. In the query evaluator we emphasize the view from above barely mentioning the implementation techniques. In our discussion of object programming both views are used.