Computer Science

INDEX

NAME: Bhavan Kumar Gubrati STD.: _____ SEC.: _____ ROLL NO.: _____ SUB.: _____

S. No.	Date	Title	Page No.	Teacher's Sign / Remarks
		STRUCTURE AND INTERPRETATION OF		
		COMPUTER PROGRAMS.		
		BOOK - I		

S. No.	Date	Title	Page No.	Teacher's Sign / Remarks

It's extremely difficult to asses the correctness of large programs therefore it's essential to build large programs out of small one whose correctness we are sure of using Organizational techniques ~~of proven value~~. These techniques are treated at length in this book.

A great deal is known about algorithms particularly sort their execution time and data storage requirements. A programmer should acquire good algorithms and idioms and always attempt to improve their performance.

Another important idea in this book is that the program as a way of expressing ideas and should be written for people to read.

The goal of the book is to impart a good feel for the elements of style and aesthetics of programming & to impart a good command of the major techniques for controlling Complexity in large software systems.

In essence, the students should be able to read a 50 page long program written in exemplary style, they should know what not to read, & what they need not understand at any moment. they should feel secure about modifying a program, retaining the spirit & style of the Original author.

We control complexity by building abstractions that hide details when appropriate. we do it by building interfaces that enable us to construct systems by combining solid well understood pieces we do it by establishing new languages for design each of which emphasizes particular aspects of designs & deemphasizes others.

Programming language being used - Racket - based on Scheme dialect of LISP - LISP is the second oldest high level programming language in widespread use with different dialects.
It's called LISP derived from 'List Processor' as lists are one of the major data structures in LISP & its source code is made of lists

Scheme is a parantheses ( ) based language

(+ 2 3).

> 5.

(* 3 3)

> 9.

(+)

> 0

(*)

> 1

(/)

Argument expected 2.

Defining a constant:

(define Pi 3.14)

> Pi

> 3.14

Defining a procedure

(define (Square x)
(* x x))

Steps of evaluation for Ex:

(define (Square (* 2 3))

Square expression is evaluated to call the right function.

(* 2 3) Argument is evaluated first.

This the process of evaluation almost always, with certain Exceptions like 'Define itself'. Ex:

(define Pi 3.142)

here as per procedure if Arguments are evaluated first we would end up getting an error as 'Pi' isn't defined anywhere but directly called. this is true for many 'define' expressions.

If we define a function say.

(define (Square x) ← not define as of yet
(* x x))

it is not evaluated unless we call it.

> (Square 3) — evaluation

Suppose if I call,

> (Square (+ 2 3))

then, x in definition is called Argument Parameter
(+ 2 3) is the Argument expression
5 is the Argument value

Piglatin program.

→ Scheme is converted to emerchay
all the consonant b4 the first vowel is moved to the end and an -ay is added at the end

```
(define (Pigl wd)
  (if (Pl.done? wd)
      (word wd 'ay)
      (Pigl (word (bf wd) (first wd)))))

(define (Pl.done? wd)
  (Vowel? (first wd)))

(define (Vowel? letter)
  (member? letter '(a e i o u)))
```

→ a Program calling itself is recursion

Programming Paradigms

Programming is easy as long as the programs are small & can be structured clearly in the mind, but as they became lengthier the complexity increases & now instead of getting into the details we have to think of it in chunks to structure in our minds & this is called Programming Paradigms

Levels of abstraction

Application
High level language Ex: Scheme, Python
Low level language Ex: C.
Machine language & Architecture
Logic gates
Transistors
Quantum physics

Functions. – why functions are imp → functions are well understood & can be theorized about

* Because today we have multitasking & multicore processors if might happen that a part of the program is executed once & the sys takes up something else we might get in trouble if a part of prog depends upon what the result of a diff part is. hence it is smarter to write prog in functions to increase flexibility

this implies that $f(x) = 2x+6$ is a function since for a given input $f(x)$ gives a fixed output but $f(x) = 2x+b$ is not a function (b → var) since the output of $f(x)$ isn't fixed.

---

lets take a example. $f(x) = 2x+6$, $g(x) = 2(x+3)$ these are same functions since for a given input the results are the same but are solved in a different procedure. The systems doesn't have any functions it only follows procedures so $f(x)$ & $g(x)$ are different from the systems point of view

```
(define (buzz n)                    test          action
  (cond ((equal? (remainder n 7) 0) 'buzz)      clause 1
        ((member? 7 n) 'buzz)                   clause 2
        (else n)))
```

```
>(buzz 15)
  15
>(buzz 17)
  buzz
```

here. Cond (Probably stds for conditional) is an extension of if statement that stands for more than 2 conditional cases. format is as follows

note: parantheses generally signify Procedure. but here it's an exception

```
(cond. clause clause .....)
       (test action)
  (function arg arg..) (function arg arg...)
```

Procedure of evaluation.

local stands for definition "order"

Procedure of function (f a b) but in the same line

```
(def (f a b) (+ (g a) b))
(def (g x)   (* 3 x))
(applic (f (+ 2 3) (-15 6)))
```

Procedure followed by the computer in applicative order

```
(+ (+ 2 3) (- 15 6))
  (+ 2 3) ⇒ 5
  (- 15 6) ⇒ 9.
(+ 5 9) →
(+ (g 5) 9)
  g(5) →
  (* 3 5) ⇒ 15
(+ 15 9) ⇒ 24
```

There are 2 orders of evaluation one is applicative order as shown followed by Schemt & the other is normal order

in applicative order the argument expressions are evaluated first and then passed to the procedure

in Normal order the argument expression itself is passed to the procedure

(normal (+ (+ 2 3) (- 15 6)))

(+ (+ 2 3) (- 15 6)) →

(+ (q (+ 2 3)) (- 15 6))

(q (+ 2 3)) →

(* 3 (+ 2 3))

(+ 2 3) → 5

(* 3 5) → 15

(- 15 6) → 9

(+ 15 9) → 24

but,

(def (zero z) (- z z)) → takes a random no < 10

(appli (zero (random 10)))

> 0

(normal (zero (random 10)))

> 7

because (zero (random 10)) →

(- (random 10) (random 10)

↳ 8 ↳ 7

(- 8 1) → 7

[True as per our definition we can say 'random' is not a function]

→ If we write proper functional programs it will give the same result no matter the procedure followed so functional programming protects us from thinking what goes on when inside the computer

---

Summing the squares from a to b.

```
(define (sumsquare a b)
    (if (> a b)
        0
        (+ (* a a) (sumsquare (+ a 1) b))))
```

0 is returned only if a>b during the first input else.

(+ (* a a) (sumsquare (+ a 1) b))

check a>b

if yes,

a^2 + 0

a^2 is returned.

if no

$+ (a+1)^2$ +

$+ (a+1+1)^2$ +

$+ (a+1+1+1)^2$

+

.

.

.

+ 0

Now this can be generalized for Squares, cubes -- power n if we could. Pass a function which defines Power as an argument to Sum.

```
> (define (Sum FN a b)
      (if (> a b)
          0
          (+ (FN a) (Sum FN (+ a 1) b))))
```

FN → function that Squares, cubes ..... p^n

for this to work a function has to be defined and passed as a variable ie.,

```
(define (Square x) (* x x))
```

defined & used

```
(Sum Square 3 5)
>50
```

here no parameters is not used because the function/Procedure Square is not invoked here but is passed as data

we cannot for example do this

```
(Sum (* x x) 3 5)
```

because the system tries to evaluate the argument first ie (* x x) where x is undefined

this idea that functions can be passed as arguments, as data is important since it breaks the idea of demarkation between data & procedures (functions)

Definition - Domain & Range of a function.

Domain of a function is what kinds of things does the function take as an argument & Range is what kinds of things does it return as the result
ex: nos, sentences, ...

---

a function that takes a sentence of numbers and returns the even nos in that sentence

```
(define (evens nums)                    → base case always present in recursion to avoid ∞ loop
  (cond ((empty? nums) '())  ((= (remainder (first nums) 2) 0)
         (se (first nums) (evens (bf nums))))
        (else (evens (bf nums)))))
```

sentence combines two 2 arguments

only a function that takes a sentence and returns the words that contain the letter 'e'

```
(define (ewords sent)
  (cond ((empty? sent) '()) ((member? 'e (first sent))
         (se (first sent) (ewords (bf sent))))
        (else (ewords (bf sent)))))
```

```
> (evens '1 3 5 1 4 9 8 6 7))
> (4 8 6)
> (ewords '(get to get you into my life))
  (get life)
```

Write a function that returns all the pronouns from a sentence.

```
(define (Pronouns sent)
  (cond ((empty? sent) '())
        ((member? (first sent) '(I me you he she it ...))
         (se (first sent) (Pronouns (bf sent))))
        (else (pronouns (bf sent)))))
```

This can be generalized as follows

```
(define (kup PRED sent)
  (cond ((empty? sent) '())
        ((PRED (first sent))
         (se (first sent) (kup PRED (bf sent))))
        (else (kup PRED (bf sent)))))
```

This will work as follows.

```
> (kup even? '(5 7 3 4 6 11 2 25))
(4 6 2)
> (define (eword? wd) (member? 'e wd))
> (kup eword? '(got to get you into my life))
```

Predicate is a function that gives (whose range is) true/false mathematically.

if there is an expression say $ax+b$ where a & b are constants, then $ax+b$ is a function of 'x' i.e.

$$x \longmapsto ax+b$$

(x maps to $ax+b$)

can also be written as

$$\lambda x.\ ax+b$$

This idea can also be used while defining functions i.e.

(define (Square x) (* x x))

is actually a short form for

(define Square (lambda (x) (* x x)))

So instead of defining the predicate function initially by passing it on to keep we can also do this

(keep (lambda (wd) (number? 'e wd)) '(living for the benefit))

---

* Higher Order function/Procedure is a function that takes another function as an argument or returns one Ex: keep above or we can think of integrals & differentials that take functions as an argument & return one too.

This is imp since long Prog are made intelligible by making them smaller through generalizing (thereby avoiding repetition)

and further we can imagine these generalized 'Keep kind of functions as independent modules that do a certain Procedure, and therefore can be used indiscriminately so: we have a dataset of 1000 datapoints that have to be checked to know if it satisfies a certain condition Now 'Keep' function can be used by a say a '1000 coprocessor parallely to handle all datapoints at once. This is an important advantage of functional programming

---

* (define Square (lambda (x) (* x x)))
  here we are assigning a procedure as the value of variable Square.

* Procedure without a name.
  ((lambda (x) (word x x)) 'foo)
  >foofoo

* Any first class data type can be passed as an argument to a procedure
* character, strings are not first class in older languages like C, aggregate (like Sentence) are hardly first class but they can be given printer in some languages which in turn are first class
* Lambda idea is one of the key ideas in the 'Lisp' family of languages and is adopted in modern languages like python.

First class Data types

can be
- value of a variable
- an argument to a procedure,
- the value returned by a procedure
- a member of an aggregate (Something like an array or structure where diff elements are combined)
- anonymous.

* One of the chief design principle of scheme is to have everything definable within scheme to be first class. or manipulatable

* Other ideas that some other languages are based upon include * it should be easily compilable
* it should look natural
etc

Procedures that return procedures

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

> (define plus3 (make-adder 3))
> (plus3 8)
  11.
> (define plus5 (make-adder 5))
> (plus5 (plus3 2))
  > 10.
```

plus3.
↓
(make-adder 3)
↓
lambda (x) (+ x 3)
↓ returned
(lambda x (+x3) 8)

8+3 = 11

```
> (define (compose f g)
    (lambda (x) (f (g x))))
> ((compose first bf) '(she loves you))
> loves
```

here both the arguments & Return value of compose is a function/Procedure

---

when the return value of function is a function it is useful to imagine the returned function in the place of the function call for clarity

---

Since it is easier to code in LISP these languages are used as rapid prototyping languages and once all the design iterations are done it is then coded in faster compiled languages like C

```
(define (roots a b c)
  (define (roots1 d)
    (se (/ (+ (-b) d) (* 2 a))
        (/ (- (-b) d) (* 2 a))))
  (roots1 (sqrt (- (* b b) (* 4 a c)))))
```

- here instead of saying

```
d = (sqrt (- (* b b) (* 4 a c)))
```

and then using se, the function is defined first and ~~evaluated~~ the expression is passed directly. here since scheme uses applicative order (sqrt (exp)) is calculated only once if it used normal order it would calculate it twice.

Still better we can write.

```
(define (roots a b c)
  (lambda (d)
    (se (/ (+ (-b) d) (* 2 a))
        (/ (- (-b) d) (* 2 a)) ))
  (sqrt (- (* b b) (* 4 a c))) ))
```

This can be further abbreviated as follows

```
(define (roots a, b, c)
  (let ((d (sqrt (- (* b b) (* 4 a c)))))
    (se (/ (+ (-b) d) (* 2 a))
        (/ (- (-b) d) (* 2 a)) )))
```

binding name → binding value

body

This are not two different ways of doing the same thing but the latter is an abbreviation for the former.

Let Syntax

```
(let bindings body)
  ( binding1, binding2, ..... )
  (name value/exp)
```

Interesting question.

Suppose we have

```
(define (roots a b c)
  (let ((d (sqrt (- (* b b) (* 4 a c)))))
    (-b (-b))                    ; here instead of -b can we use d.?
    (2a (* 2a)))
  (se (/ (+ -b d) 2a)
      (/ (- -b d) 2a))))
```

here, a ~~tempting~~ way to think is ~~that~~ ~~it~~ evaluation starts from left d is evaluated first so can be used in the next binding. This is "Wrong" to understand this we need to expand this as follows

```
((lambda (a b c)
   (se (/ (+ -b d) 2a) (/ (- (-b d) 2a))
   ((sqrt (- b b) - arg2 . arg3)
```

here. args are evaluated first ~~without~~ assigning ~~there~~ any names like (a b c) so when arg2 is evaluated arg1 is not assigned any name. for the same reason recursion isn't possible

Instead we can use

```
(let * ((a 3) (b (+ a 5))) (* a b))
```

which will work because

```
(let ((a 3))
  (let ((b (+ a 5)))
    (* a b)))
```

One let is nested inside another so a is assigned by the second let

Let variables are only valid inside its body

In the previous programs we could have easily said, $d = \$sqrt(b^2 - 4ac)$ which is internal definition. Internal def^n work as a Let rack and boil down to the same mechanism. Global def^n on the other hand are a diffent story and are generally frowned upon in the programming world because they give risk to case where a change in one gives risk to an unexpected change else where

---

Analysing the units of time taken by an algorithm is called time Complexity. It's done by first counting the no. of primitive operations executed.

1. Assign a value to a Variable (independent of the size of the value; but the variable must be a scalar)
2. Method Invocation ie calling a function or Subroutine
3. Performing a Simple arithmetic Operation (divide OK, log not)
4. Indexing into an array (1D)
5. following an object reference.
6. Returning from a method.

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

(define (square sent)
  (if (empty? sent)
      '()
      (se (square (first sent))
          (square (bf sent)))))
```

Primitive ops
> Empty.
> if
> first
> bf.
> Square isn't prim but its def^n is just 1 const time.
> Se is compl but here its 1 const time.

~~(define (sort~~

(squares '(6 23 5 10 7))
> (36 529 25 10000 49)

total no of times the proc is called = 6 (including empty)
Empty sentence num has only 2 const ops.

∴ run time ~~2~~ 6N + 2
where N is the length of the input

Sorting Programme.

```
(define (Sort sent)
  (if (empty? sent)
      '()
      (insert (first sent)
              (Sort (bf sent)))))

(define (insert num sent)
  (cond ((empty? sent) (se num))
        ((< num (first sent)) (se num sent))
        (else (se (first sent) (insert num (bf sent))))))
```

→ Here range of sort is a sentence that has to be kept in mind for all cases

> Sort (6 23 5 100 7)

> (5 6 7 23 100)

tracing it.

→ Sort with sort = (6 23 5 100 7)
→ Sort with sort = (23 5 100 7)
→ Sort with sort = (5 100 7)
:
→ Sort with sort = (7)
→ Sort with sort = ()
→ Sort returns ()
→ insert with num = 7 sort = ()
→ insert returns (7)
→ insert with num = 100, sort (7)
→ insert with num = 100, sort ()
→ insert returns (100)
→ insert returns (7, 100)
→ Sort returns (7, 100)
→ insert with num = 5, sort = (7, 100)
→ ~~Sort with num = 5, sort (7~~
insert returns (5, 7, 100)
→ Sort returns (5, 7, 100)
→ insert with num = 23 sort = (5, 7, 100)
→ insert with num = 5 sort = (7, 100)
→ insert with num = 23 sort = 100
→ insert returns (23 100)
→ insert returns (7, 23, 100)
→ insert returns (5, 7, 23, 100)
→ Sort returns (5, 7, 23, 100)
→ insert with num = 6 sort = (5, 7, 23, 100)
→ insert with num = 6 sort = (7, 23, 100)
→ insert returns = (6, 7, 23, 100)
→ insert returns = (5, 6, 7, 23, 100) → Sort returns (5, 6, 7, 23, 100)

here we are not building a data structure like an array, storing it & changing its value. In functional programming we donot change the value of anything. We instead write a program that returns the required value. "Think like, what value does this function have to ~~write~~ return & express it as an expression."

Now, Time complexity analysis of Sort,

SORT

Insert. run time estimation.

1) cond.
2) empty?
3) <
4) first
5) se.

} case I

2)
4)
5) se
6) first.
7) bf

} case II

3) Empty
1) Cond
2) empty
3) Se.

∴ Run time ∝ 7N + (5) [3 empty, 5 run is small]
(worst case)
N is no of item in subarr

or on an Avg.

Run time ∝ 7 $\frac{N}{2}$ + 5

Sort time esti
1) if.
2) empty? } Constant time.
3) first
4) bf.
+ insert call
+ recursion to sort

RT $\propto N\left(4+\frac{7N}{2}+5\right)+2$

Runtime $\propto \frac{7N^2}{2}+9N+2$

Here,

* if the sys takes 't' See to sort 5000 digits it will take almost 4 times as much to sort 10000 nos (because of N^2 at large N values other items in the exp are neglected)

because of this behaviour the 2 in the denominator won't very significant because the sys speed doubles quite quickly these days which will make the 2 in the denominator quite insignificant. Thus the difference b/w ~~the~~ the run times of linear type algos ($9N+2$) and Quadratic types ($\frac{7N^2}{2}+9N+2$) is essential & needs to be captured in writing. Therefore we define,

$f \in O(g) \longleftrightarrow \exists N, K \mid \forall x > N \;\; |f(x)| \leq K|g(x)|$. (N corresponds to Point P in the graph in the next page.)

where $N, K \neq 0$

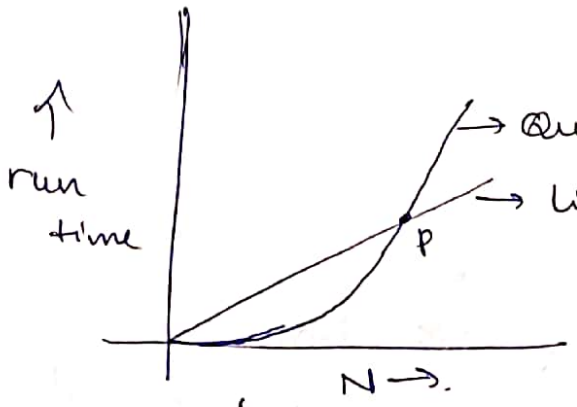
$f \in \Theta(g) \longleftrightarrow f \in O(g) \wedge g \in O(f)$

i.e.

$f \in O(g) \longleftrightarrow \exists N, K \mid \forall x > N \;\; |f(x)| \leq K|g(x)|$

i.e

There exists N, K such that for all x greater than N. Absolute value of $f(x) \leq K$ times abs of $g(x)$



(no of elements in the input)

Now suppose we have a linear function

$f(N) = N + a$

Quad.

$f(N) = N^2$.

we know that Point P is $N = 1$

i.e Quad is < linear below 1 & N can't be 0.1 0.2 → is discrete etc

But we have to take into account the effect of constant.

i.e.

$f(N) = 1000N$ so.

$f(N) = 10N^2$

P is an arbitrary point.

An example showing the importance of this diff b/n, Linear & Non linear Run time functions.

N.	$t_1(N) = 3N^3$	$t_2(N) = 19500000N$
10.	3 microsec	200 milli
100.	3 sec.	20 sec
100000	35 days	32 min.

This is the importance of designing a proper algo & keeping in mind its running function.

This also demonstrates the unimportance of constant factors even when they are as big as 19500000 as shown above. But with the exception of Videogames industry because they push the hardware

its absolute limit.

$f \in O(g) \longleftrightarrow \exists N, K > 0 \mid \forall x > N \quad |f(x)| \leq K \cdot g(x)$

$f \in \Theta(g) \longleftrightarrow f \in O(g) \wedge g \in O(f)$

To be studied

Runtime $\propto$

$\Theta(1)$
$\Theta(\log N)$
$\Theta(N)$
} Searching

$\Theta(N \log N)$
$\Theta(N^2)$
} Sorting

Searching generally takes linear time, because it generally functions as check if 'a' is what we req NO then check 'b' etc. this checking is a constant time activity due to which the Run time increases linearly with the lenght of the input. If the Runtime becomes proportional to $\log N$ then its a serious advantage since $\log 1000 = 3$. i.e instead of Run time rising ppl to 1000 it is ppl. to 3. this becomes achievable when the data isn't randomly ordered & follows some order [Ex: like looking up in a phonebook]. we can further optimise search to constant time ie $\Theta(1)$ using hash tables (#)

$\Theta(N^2)$ in sorting is already discussed in insertion Sort. In $\Theta(N \log N)$ we take the data cut it in half so further cut sort rearrange etc because of which we have $\log N$ & $N \log N$ is the limit for these cases of sorting

$\Theta(N^3)$ - Matrix Multiplication

$\Theta(2^N)$
$\Theta(N!)$
$\Theta(N^N)$
} intractable: meaning we can write programs to do these things but for any meaningfull sized dataset the program will never finish running

imagining 2^N, if we need to double the running time in a (linear) $\Theta(N)$ then we need to double the input size. in case of $\Theta(N^2)$ we need to increase the data size by 40%. but in $\Theta(2^N)$ we only need to increase the input data size by 1 new data point

---

Space complexity analysis

Lisp languages were criticised for having to use procedure Calls for everything & for not having iterative mechanisms like for, while etc. because procedure calling was deemed expensive in terms of memory consumption.

But, even in functional programming we can achieve the same efficiency provided we write the program iteratively.

```
(define (count sent)
  (if (empty? sent)
      0
      (+ 1 (count (bf sent)))))
```

— A

```
(define (count sent)
  (define (iter wds result)
    (if (empty? wds)
        result
        (iter (bf wds) (+ 1 result))))
  (iter sent 0))
```

— B

(count '(i want to hold your hand))

> 6.

Now Analysing 'A'

M1 (count '(she loves you))

(+ 1 (count '(loves you)))

M2

(+ 1 (count '(you)))

M3

(+ 1 (count '()))

M4.

Now, M4 returns '0' to M3 returns 1 to M2 returns 2 to M1 & M1 returns 3 thus each of these Memory chunks has to wait until the last one evaluates & returns. ie they have to remember to whom they have to return this result

M_1 (count '(she loves you))

M_2 (iter '(she loves you) 0)

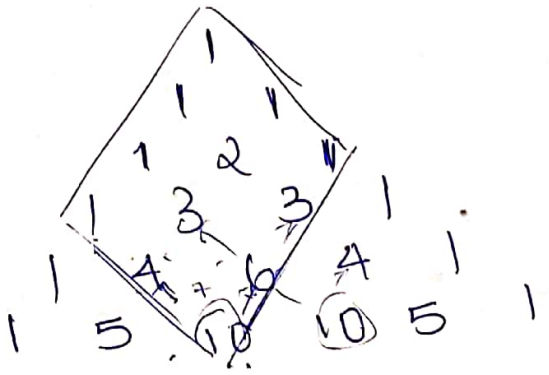
M_3 (iter '(loves you) 1)

M_4 (iter '(you) 2)

M_5 (iter '() 3)

hw. the work is done the way down instead of doing it the way up thusfor. no memory chunk is kept waiting. hence in Memory is used efficiently.

Pascal Triangle



Computing Pascal's Δ^{le}

$\Theta(n)$

```
(define (linear x)
  (if (empty? x)
      se/word (constant (first x))
              (linear (bf x))))
```

$\Theta(n)$

```
(define (exp x)
  (if (empty? x)
      combine (exp (first x))
              (exp (bf x))))
```

$\Theta(n^2)$

```
(define (quad x)
  (if (empty? x)
      (combine (linear (first x))   Θ(n)
               (quad (bf x)))))
```

$n\Theta(n) = \Theta(n^2)$

Computing a digit (say row 20 column 3) in a pascal Δ^{le} is $a\Theta(2^N)$ problem as is evident and hence consumes extraordinary amounts of time when coded.

Unless we code it such that all the nos in each row is calculated & not just the ones we need

ie to calculate 10 in the fig only the nos in the diamond shaped fig are req. But the Program that's coded to calculate all the nos in the Δ^{le} works far better because the usual code computes lot of nos repeatedly drastically slowing the run time.

Chapter 2: Data abstraction

Q. An abstract data-type is a datatype that is in the mind of the programmer & not internal to the programming language. Ex: Say 'Cards' → houses.
Spade Diam etc — Skyline etc Graffiti

* Constructors & → Selectors in datatypes.

We can make code more readable by giving synonyms like (define remaining Cards butlast) These are called Selectors. It's called so because it takes an instance of a datatype & pulls out a piece of it

---

* The Chapter-1 was about flow of control ie how we organize sequence of events in a program using recursion & higher order functions as our chief tools

---

* Chapter-2 is all about how we organize, structure & represent data

→ Data abstraction -
→ Pairs
→ Sequences

Now Consider a function that gives the total hand in a card game ie the sum total of the ranks of all the cards a person has Ex: if we have 3 of spades, 10 of J and 4 of diamonds represented by 3S, 10C & 4d then the program will return 3+10+4 = 17 as the total ha (here the king, Ace, Queen & Jack are ignored for simplicity

```
(define (total-hand hand)
  (if (empty? hand)
      0
      (+ (butlast (last hand))
         (total-hand (butlast hand)))))
```

this program works fine but when this program is a part of a much larger program & when certain modif are needed to be made it can cause a lot of confusion because of butlasts which mean diffent in each usage in the above program. therefore

```
(define (total-hand hand)
  (if (empty? hand)
      0
      (+ (card-rank (one-card hand))
         (total-hand (remaining-cards hand)))))
```

```
(define card-rank butlast)
(define card-suit last)

(define one-card last)
(define Remaining-cards butlast)
```

Card suit is
Spades
clubs
Diamonds &
hearts

here, we have added only synonyms which make the program much more understandable and also easily maintainable, ie because suppose the representation of cards need to be changed from 10S (for 10 of spades) to S10 since the former isn't legal in most languages we can just go to definition of card-rank and change it to butfirst instead of butlast otherwise we would have to scan the entire program for relevant changes

The idea of 'card' here is an 'abstract Data type' that exists only in the mind of a programmer for the computer its just a word. [since input will be '10S']

To have an abstract Data type the first thing we need is 'Selectors', in this example

Selectors are. Card-rank &
Card-suit

next we need 'constructors' we may have more than one constructor, here we have only one

Constructor - make-card

i.e, (define (make-card rank suit)
(word rank (first suit)))

Now, we can change the implementation to have just the nos from 0-51

```
(define (make-card rank suit)
  (cond ((equal? suit 'heart) rank)
        ((equal? suit 'spade) (+ rank 13))
        ((equal? suit 'diamond) (+ rank 26))
        ((equal? suit 'club) (+ rank 39))
        (else (error 'say what'))))
```

this is the new 'make-card' of the constructor for our Abstract Data type

Now for selectors,

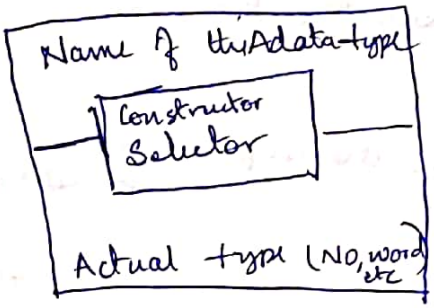
```
(define (card-rank card)
  (remainder card 13))

(define (card-suit card)
  (nth (quotient card 13) '(heart spade
                            diamond club)))
```

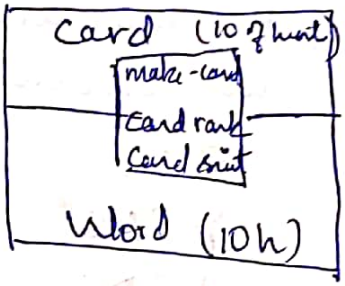
Selectors are card rank & card suit

After this we can use the program total based as it is without making any changes because as long as the changes we make to the 'Constructors' matches the changes we make to the 'Selectors' all other parts of the program function seamlessly. This is the chief reason why data abstractions are important they make the programs readable and maintainable.

Diagramatic representation of Abstract Data type



Ex:

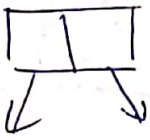


even if a program has two procedures that manipulate the data. The procedures in the internal block in the above diagram are the only procedures that actually know how the data type looks (two or word etc).

Some languages have a notation for representing abstract data types built into the language. like in Java an Abs data type is represented as an object when we define an obj class and have names for constructors & selectors defined there. But we don't really need that as long as we define the selectors and constructors & stick to using them, the language need not know about our Abs data type

Pairs

every programming language has different ways to aggregate data (ie put things together). the main data aggregation in Scheme is ~~pointers~~ pairs that looks like this

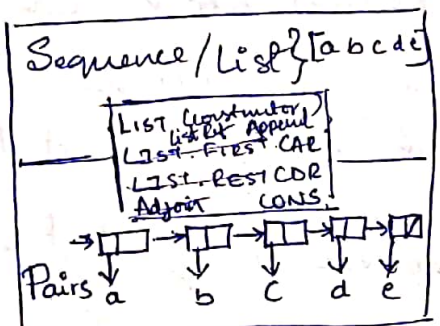


for Ex: we can represent 3/4 in a pair with left half as 3 & right half as 4 [3|4]

we would expect there would be constructors & selectors for pairs namely,

make-pair	cons (construct indicates the importance of Pairs as a datatype)
pair-left	CAR.
pair-right	CDR.

they are instead called



in each pair the CAR ~~(circle)~~ represents an element of the pair where as the CDR represents the pair in the list. the CDR of the last pair is an empty list.

here list is a abstract data type made out of pointers. but the constructors & selectors for lists are same as that for pointers.

```
> (cons 'a (cons 'b (cons 'c '())))
  (a b c)
> (list 'a 'b 'c 'd)
  (a b c d)
```

here, cons and List are procedure so as per applicative order the arguments get evaluated first & then the rest of the procedure.

* as we can expect the list constructor (ie >(list 'a 'b 'c 'd)) is made from the recursion of cons

>(List (a b) (c d))
> ((ab) (cd))
> (cons (ab) (cd))
> ((ab) c d)
> (se '(ab) '(c d))
> (a b c d)

because cons is basically a dyin ie. it takes the second argument as a list and joins it to the first arg which it considers as a single element

>(Append '(a b) '(cd))
>(a b cd)

se & append seem similar but, if we do this,

>(Append 'a '(b c))
we get a error because Append expects only lists

>(append '(a b) 'c)
> (a b . c) not an error but a improper list

Improper lists

→ instead of an empty list we have a number e which makes this an improper list

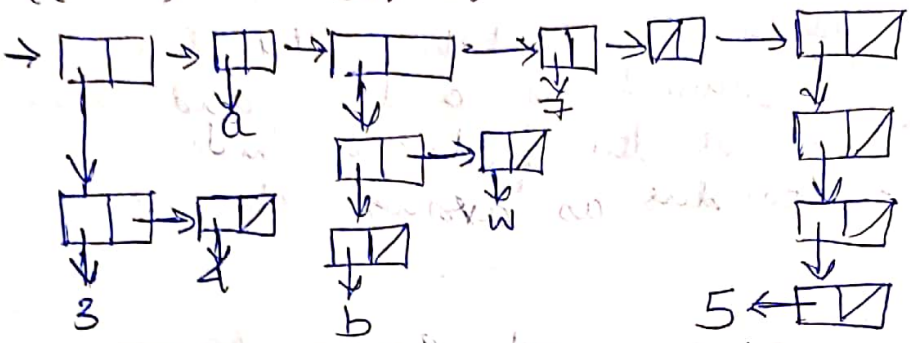
(a . (b . (c . (d . e)))) → Pair notation:

Scheme never prints a dot followed by open parenthesis

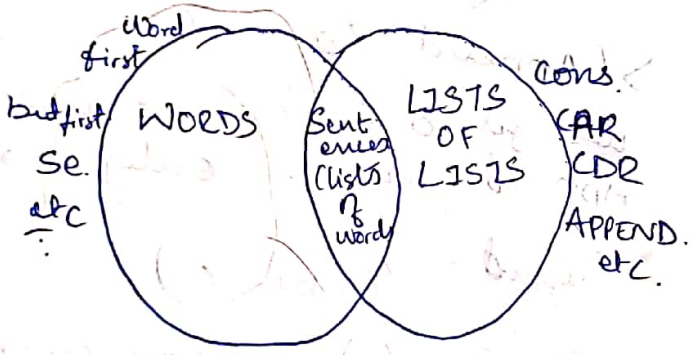
* We use 'lists' as constructors when we already know the exact length of the list we are making
* We use 'cons' as constructors in recursive form when we donot know the exact length
* append is generally a joiner of lists

Let's try a diagram for this

((3 4) a ((b) w) 7 () (((5))))



Relationship b/w words, Sentences & Lists



* Neither is the subset of other
* commands like first bf. donot work on lists except for lists of Words ie sentences
* ~~cons CAR etc do not work on words except for sent~~
* if we have list of sent we would use CAR & CDR to take apart sentences but first & bf to analyze words.

Tail call?

Every

(define (first-letters sent)
  (every first sent))

>(first-letters '(how comes the sun))
>(HCTS)

Keep : takes a predicate & a sentence & returns only those args. for which Predic is true

>(keep even? (1 2 3 4 5))
(2 4)

Higher-order functions for lists

Every	MAP
Keep.	FILTER
Accumulate	

Accumulate

Takes a Procedure & a sentence. applies the procedure sequentially to all the items in the sentence.

```
> (accumulate + '(6 3 4 -5))
> 8.
```

MAP

* map applies procedure element-wise to the elements of the lists and returns a list of results

```
(map (lambda (n) (expt n n)) '(1 2 3 4))
```

Procedure applied to each element ⇒ (1, 4, 27, 256)

Basic Structure of Any Interpreter

→ Read.
Eval.
Print loop } Repl.

⇒

```
(define (calc)
  (display "calc: ")
  (flush)
  (print (calc-eval (read)))
  (calc))
```

Read → Eval → Print → loop

Read takes the input and turns it into a Box & Pointer diagram & its input is given to calc-eval it takes in an Expression & returns the value of that expression. & print prints the evaluated value

```
> (load "calc")
Okay.
```

Read → calculated → Printed

```
(+ (* 3 4) (* 5 (+ 6 7)))
77
```

```
(define (calc-eval exp)
  (cond ((number? exp) exp)
        ((list? exp) (calc-apply (car exp) (map calc-eval (cdr exp))))
        (else (error "calc: bad exp:" exp))))
```

[flush: the os doesn't print unless it sees the end of line character. flush allows it to do that]

; Evaluate an expression

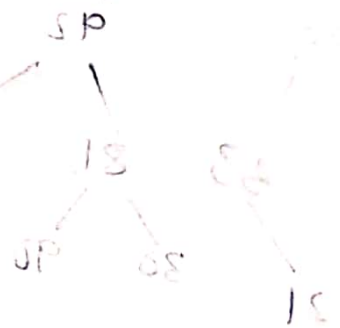
```
(define (calc-eval exp)
  (cond ((number? exp) exp)
        ((list? exp) (calc-apply (car exp) (map calc-eval
                                                (cdr exp))))
        (else (error "calc: bad expression:" exp))))
```

* here what we need to realize is the expression can either be a number or a list. if its a number. then its self evaluating (taken care in the first exp of cond) if its a list then the car of list is always a with operator. (+ - * /). (taken care by calc-apply procedure). Now we are only left with cdr of lists which can have nos. (taken care in first line of recursion) or further list (the recursion will go deeper) if the list in the cdr has multiple entries the map will take it wider

* All the properties of a programming language manifest themselves in their interpreter. In the above program. the applicative order is manifested as (map calc-eval (cdr-exp)). ie if we remove map calc-eval & instead pass the cdr-exp as a second argument the calc-apply it would be normal order ie [we are passing argument expression not any value therefore Normal order]

```
(define (calc-apply fn args)
  (cond ((eq? fn '+) (accumulate + 0 args))
        ((eq? fn '-) (cond ((null? args) (error))
                           ((= (length args) 1) (- (car args)))
                           (else (- (car args) (accumulate + 0 (cdr args))))))
        ((eq? fn '*) (accumulate * 1 args))
        ((eq? fn '/) (cond ((null? args) (error))
                           ((= (length args) 1) (/ (car args)))
                           (else (/ (car args) (accumulate * 1 (cdr args))))))
        (else (error))))
```

→ * (1) args.
  ↳ to ensure that 1 is returned when args is empty

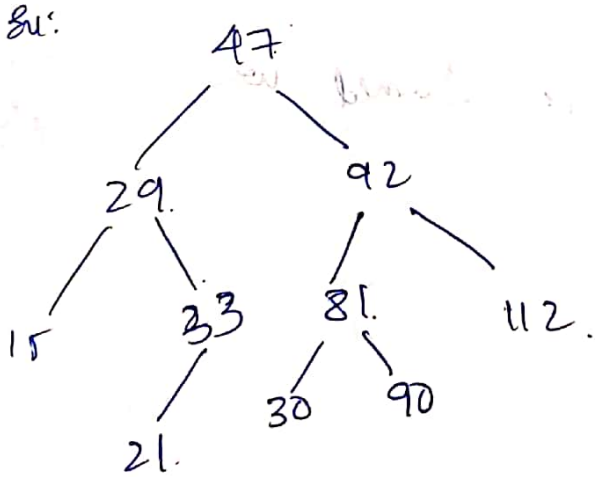


Trees: two dimensional hierarchical datastructure its an entirely abstract data type & has nothing already built into scheme like list (list is both primitive & abstract or in between them).

Ex:

- One use of a hierarchical data structure is to introduce hierarchy
- Another can be demonstrated in Binary search tree
  - each node has a left child & a right child only either or none.
  - every node has a property that the node to its left are smaller in value & the node to its right are higher.

Ex:



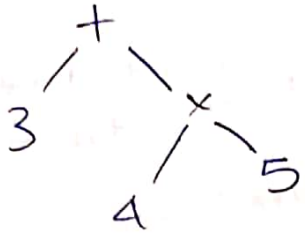
Lets search 70
→ 47 is smaller than 70 so everything to its left can be ignored
→ 92 > 70 so everything to its right ignored
→ 81 > 70 — " —
→ 30

only 4 nodes sum to arrive at the conclusion that 70 doesn't exist

Run time $\Theta = \Theta(\log_2 N)$.

Parse Tree

$3 + (4 \times 5)$



Read 3 → + → × → 4 → 5

Left → middle → right.

all languages no matter their syntax build a parse tree (built by a Compiler n interpreter).

---

* Node - is a piece of a tree. (can a datum)

USA is 'not' a node but the entire family below USA is the node. which implies nodes are also trees. ie. both are the same. This is an important idea.

List of trees → forest

Tree code.

```
(define make-tree cons)  → constructor
(define datum car)
(define children cdr)    } Selector

(define (leaf? node)
    (null? children node)))
```

leaf is a node without children.

here children is a list. by itself its not a tree but its elements are nodes in other words trees so children is a forest

```
(define (treemap fn tree)
  (make-tree (fn (datum tree))
             (map (lambda (child) (treemap fn child))
                  (children tree))))
```

(2)

Sample

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

```
(define (leaves seq)
  (map (lambda (x) (make-tree x '())) seq))

(define t1
  (make-tree 1
             (list (make-tree 2 (leaves 3 4))
                   (make-tree 5 (leaves 6 7 8)))))
```

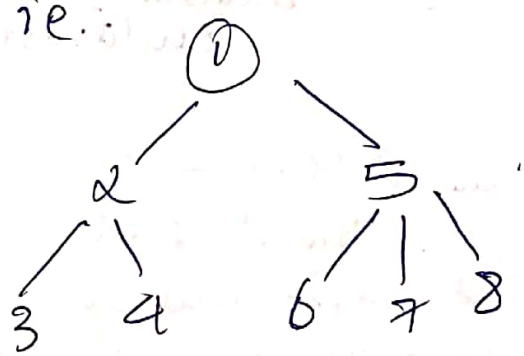
(1) See in order

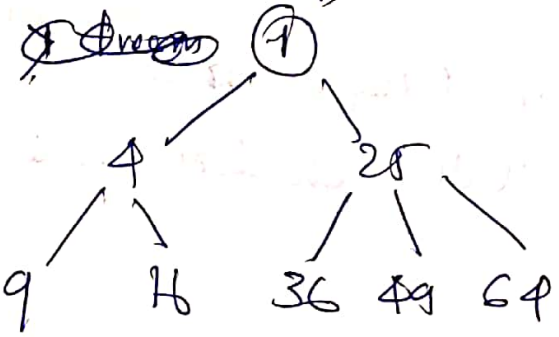
> t1

(1 (2 (3) (4)) (5 (6) (7) (8)))

> (treemap square t1)

i.e.:





The beauty of this procedure is that it applies a function on each number of the tree & the solution is also a tree of the same structure & this is achieved from a 3 line of code. Base case is buried inside tree map. we have two Base cases { Vertically → - leaf nodes
horizontally → - after the last child

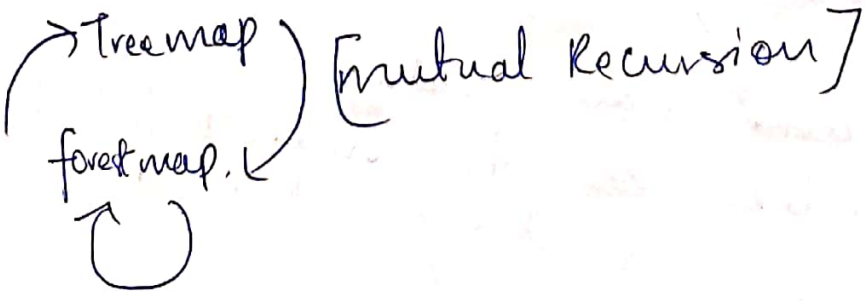
here, map calls treemap & } this is mutual recursion
treemap calls map. } which gives rise to 2-D control structure

---

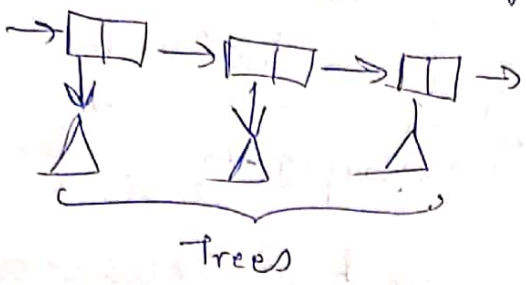
Same code, written without map

```
(define (treemap fn tree)
  (cons-tree (fn (datum tree))
             (forest-map fn (children tree))))

(define (forest-map fn forest)
  (if (null? forest)
      '()
      (cons (treemap fn (car forest))
            (forest-map fn (cdr forest)))))
```



A forest is actually a list; each of whose elements are themselves trees, they look like this

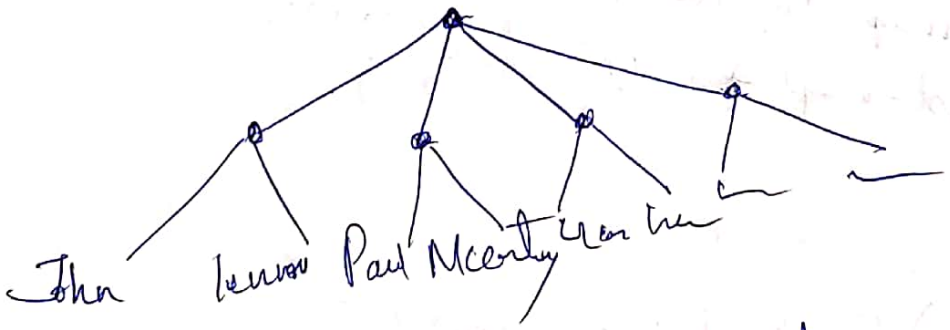


Deep Lists: Just like trees Deep lists are another form of 2D datastructure.

Ex: ((John Lennon) (Paul mccartney) (George Harrison) (Ringo st...))

This is a list of list of words

Ordinarily we may use CAR & CDR to manipulate the outer list & first, but first etc to manipulate the elements of inner list (since they are sentences) but we can also imagine this as a tree, shown below



here Datum doesn't have values but it's tree like, so to manipulate this we can imagine something like tree map.

ie.,
```
(define (deep-map fn lol)                    ; list of lists
  (if (list? lol)
      (map (lambda (element) (deep-map fn element))
           lol)
      (fn lol)))
```

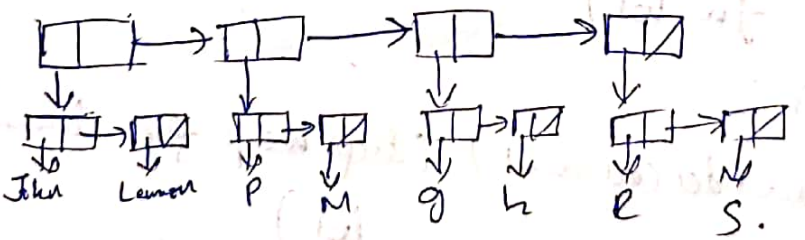
This would work for a deep list of any complexity

---

	2 children	n children
abstract data type	binary tree	tree
Just Pairs	car/cdr recursion	deep list

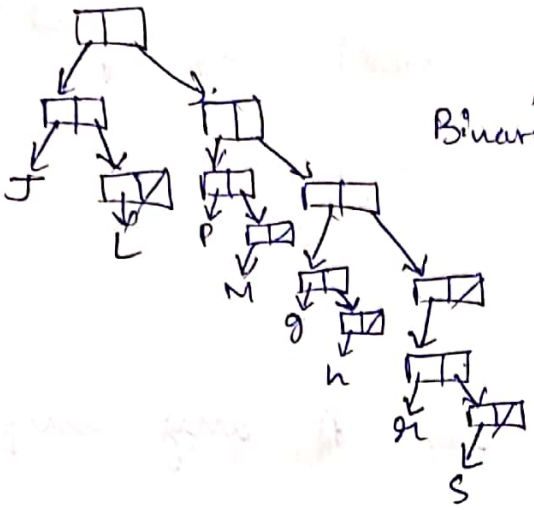
Car/cdr recursion.

Consider a list of lists.



list of 4 elements each of the 4 elements in turn are a list of 2 elements which are words.

Now we can imagine this list of list like this



this reminds of a Binary tree.

```
(define (deep-map fn xmas)
  (cond ((null? xmas) '())
        ((pair? xmas)
         (cons (deep-map fn (car xmas))
               (deep-map fn (cdr xmas))))
        (else (fn xmas))))
```

does the work & map

Now compare this car/cdr recursive version with the Deep list version of the code viz.

```
(define (deep-map fn lol)
  (if (list? lol)
      (map (lambda (element) (deep-map fn element))
           lol)
      (fn lol)))
```

This version understands that the list is made up of pairs with elements in the cars but when as the top one doesn't know much about the lists even though it uses CAR & CDR (~~accessors~~ selectors) because it would work even if CDR didn't contain either a null

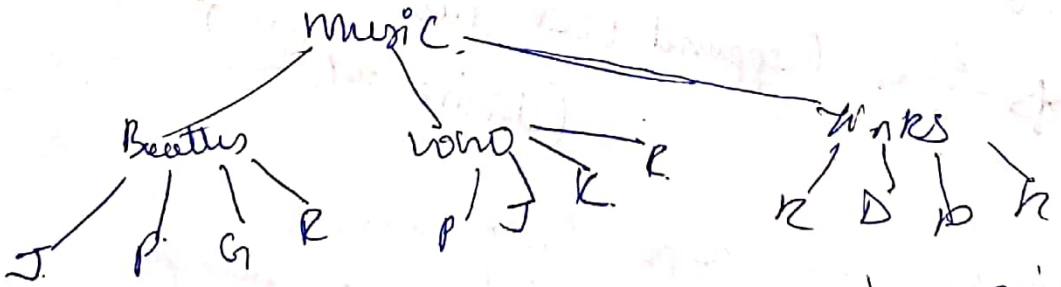
& another pair like in lists, instead if it were something like (2.3) it [2|3] if would still work. in the ~~top version~~ bottom version we say it understands list because it uses maps.

Filter → Tree version of Keep

```
(define (filter Pred Seq)
(cond ((null? seq) '())
      ((Pred (car seq))
       (cons (car seq) (filter pred (cdr seq))))
      (else (filter pred (cdr seq)))))
```

Here even though there are 2 consecutive recursions. but this is not a tree recursion like the previous example because here either of the 2 recursion is executed where as in the previous one both are executed. So when we see 2 recursions which are consecutive -ly executed its possible that it is a tree recursion so there is explicitly or implicitly a tree like datastructure involved.

Suppose we have an abstract tree



We want to go through this tree printing out the Datum at each step. There are 2 ways of doing this Depth first & Breadth first.

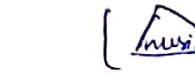
Depth first

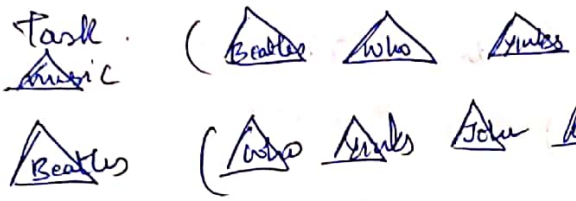
```
(define (depth-first-search tree)
  (print (datum tree))
  (for-each depth-first-search (children tree)))
```

Depth first search in just 3 lines (for-each = map)

Breadth first

We use Queue to do breadth first search





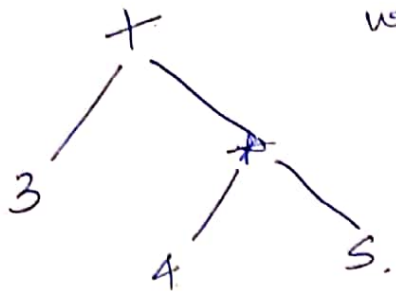
Breadth first search

```
(define (bfs-iter queue)
  (if (null? queue)
      'done
      (let ((task (car queue)))
        (print (datum task))
        (bfs-iter (append (cdr queue)
                          (children task))))))
```

---

for most cases we use depth first search but sometimes Breadth first is useful for example when searching the min-max tree in chess we can't use depth first as it can go to almost infinite depth & not return a answer so we do Breadth first search here.

Parse Trees



we can read this in 3 ways

INORDER

LEFT
DATUM
RIGHT

(3 + (4 * 5))

PREORDER

DATUM
LEFT
RIGHT

(+ 3 (* 4 5))

POSTORDER

LEFT
RIGHT
DATUM

3 4 5 * +

Post Order: Hp calculator & some comp languages work this way, here the logic is Datafirst & then operator

---

Scheme Interpreter written in Scheme

* The point of this self contradictory topic is to educate about interpreters. Secondly it can be used to upgrade scheme by adding different features to it.

* The Book makes a point about the universality of programmable computer i.e., we can do one computer program viz Interpreter that can do any computation

Basic Version of the Scheme interpreter

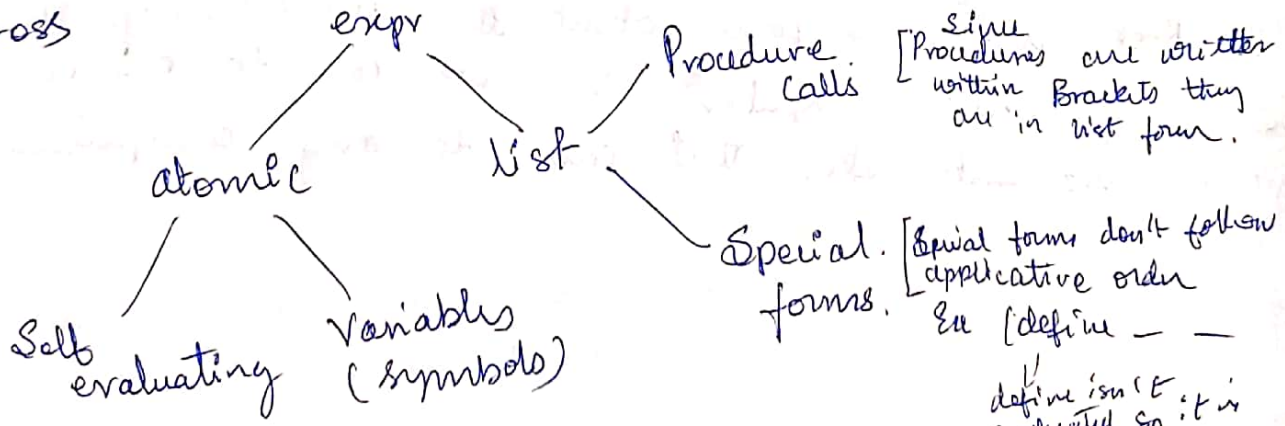
```
(define (scheme)
   (Print (eval (read)))
   (Scheme)
```

} Repl

```
(define (eval exp)
   (cond ((Self evaluating? exp) exp)
         ((Symbol? exp) (look-up-global-value exp))
         ((Special-form? exp) (do-Special-form exp))
         (else (apply (eval (car exp))
                      (map eval (cdr exp))))))
```

```
(define (apply proc args)
  (if (primitive? Proc)
      (do-magic Proc args)
      (eval (Substitute (body proc) (formals proc) args))))
```

Possible types of expressions that we can come across



* here in cond exp if eval error in ignored case
* here the symbols that eval comes across are global variables only because of the substitution model followed. Because of which all local variables are substituted with value before evaluation.
* So the type of symbols we come across will be global Var and primitives.
* we'll have to write different cases for each of the special forms. & is one of the reasons why the interpreter code will be lengthy
* here the car will be the subexpression whose value is the procedure that we want like '+' '>' etc or it might be the name of a procedure that we wrote whose value will be the procedure.
* The cdr exp is the list of actual arg expressions
* It must be observed that this is a tree recursion here
* The apply here either gets a primitive procedure or a > procedure. If it's primitive then we needn't currently the > its evaluation (hence do magic) if its a lambda exp the following happens.
  (it will be written in machine level language)
* we create an abstraction for the Procedure written in lambda ie

(λ [(x y z)] → formals.
   [(+ x (* y z))])
        ↓ Body

* this is how substitute works Eg: ((λ(x) 10 (+ x ((λ(x)(* x 3)) (+ x 5)))
  So our substitution function should be able to carry out this procedure
  Actual Scheme doesn't run on substitution model but on Environment model as subs model is only adequate for functional programming

Value = 55

* here we see that 'eval' calls 'apply' and 'apply' calls 'eval' which is mutual recursion and forms the heart of the interpreter & also the tree recursion.

* So how can we justify the 'tree' recursion in an interpreter we can justify it as there is hierarchy in scheme expressions (viz the essence of tree data structure) ie scheme expressions can have sub expression which in turn can have further sub exp and so on another way is expressions can call procedures which have bodies with expressions & hence hierarchy

* things we have ignored for now: those primitives work ie how does it do arithmatic etc & Memory allocation to understand which we have to study Operating Systems

---

Points on Moderately detailed version

* here we are taking the procedures of stk as primitives for our scheme interpreter which means that if I define something in stk it'll be a primitive for my interpreter which in turn means that to evaluate it I need the call the eval of stk itself.

* when I write a quote expression say 'FOO it is read by read procedure converted into a list as follows (quote FOO) so to evaluate if we take the cadr.

* We are using if expression of the scheme to evaluate if expressions in our interpreter & it isn't cheating.

* In apply-1. we check if the 'Proc' is a primitive/stk defined procedure by using stk's Procedure? & if it is use 'apply' viz also stks apply to invoke the procedure involved.

* the empty case in Substitute is for λ exp within λ exp case as explained by

* If have ignored 'Define' in our interpreter hence cannot be used

Moderately detailed Version of Scheme Interpreter:

```
(define (scheme-1)
  (display "Scheme-1 : ")
  (flush)
  (print (eval-1 (read)))
  (scheme-1))

(define (eval-1 exp)
  (cond ((constant? exp) exp)
        ((symbol? exp) (eval exp))   ; this eval is underlying scheme's eval not ours
        ((quote-exp? exp) (cadr exp))
        ((if-exp? exp)
         (if (eval-1 (cadr exp))
             (eval-1 (caddr exp))
             (eval-1 (cadddr exp))))
        ((lambda-exp? exp) exp)
        ((pair? exp) (apply-1 (eval-1 (car exp))
                              (map eval-1 (cdr exp))))
        (else (error "bad expr:" exp))))

(define (apply-1 proc args)
  (cond ((procedure? proc)
         (apply proc args))   ; using underlying scheme's apply.
                              ; also procedure? is stk.
        ((lambda-exp? proc)
         (eval-1 (substitute (caddr proc)   ; the Body
                             (cadr proc)    ; the formal Param
                             args           ; the actual args
                             '())))         ; bound Vars.
        (else (error "bad proc:" Proc))))
```

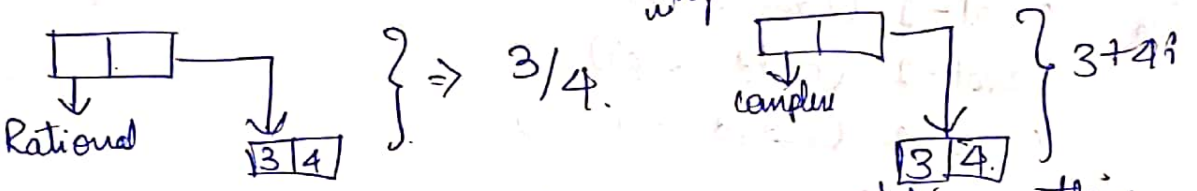
Some trivial helper procedures / data abstractions / substitute etc.

* but we have returned the expression itself for the value of Lambda expression because unless the procedure is invoked there's no work. But this won't be true in actual case & will be discussed later

---

Generic Operators

The problem we are trying to solve is how do we handle different data differently, for ex: we represent 3/4 as [3|4] why if we were to represent 3+4i we may write [3|4] but if we are writing a arithmatic operator program such representation is confusing. So to solve this we can represent it as.



So we call this tagged data type, & we define this abstract data type as follows.

```
;; tagged data.
(define attach-tag cons)
(define type-tag car)
(define contents cdr)
```

A similar system is implimented in most languages

The reason this is important is because as we know the nos we type in are stored in the system as a bunch of 1s and 0s. ~~But the same~~ & we have different programs to interpret this as Integers or floating points (Real) etc i.e. the same set of 1s & 0s will be represented differently by integer prog & differently by floating program. So it is essential that the system knows how to interpret it.

As an extension of this it is impossible to tell the actual value of a string of ones & zeros just by looking at it as it depends on how it is encoded as a integer or as a complex no or a valued etc.

This can be solved in two ways One is to add a type tag on every no (whether as a pair or in other ways) another way is that the programmer tells the computer what to expect at different instants of time.

The following are the trade off (b/n computer time & human time)

1) Efficiency: In the former case (viz Schemes case scenario) the Computer takes longer time as it has to figure out what are the nos we are dealing with (this is the current done topic of discussion)

in the latter case when the programmer has told the system what to expect the corresponding machine instructions are already deployed

2) Debugging: if we explicitely declare the type. The compiler might catch a few bugs

3) Ease of programming

Ex: (define square x)
(* x x)

(define (int-(int-square (int-x)))
(* x x)

(define (float-(float-square (float x)))
(*. x x).

This is how programmer explicitely tells the sys the type. This makes the computation a bit faster by a constant factor only

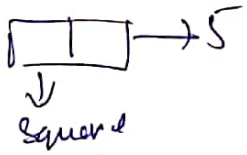
* Lets take an example involving shapes

```
(define attach-tag cons)
(define type-tag car)
(define contents cdr)

(define (make-square side)
    (attach-tag 'square side))

(define (make-circle radius)
    (attach-tag 'circle radius))

>(define s5 (make-square 5))
>s5
(square . 5)
```

ie 


```
> (area s5)
25
> (perimeter s5)
20
> (perimeter c5)
31.415
```

to imagine, we make the conventional style following table

	Area	Perimeter
Square	$s \mapsto s^2$	$s \mapsto 4s$
Circle	$r \mapsto \pi r^2$	$r \mapsto 2\pi r$

we will look at 3 different ways of representing this in the computer

1) Conventional style : This is the method used by everyone b4 anybody ~~thought~~ seriously about this

;; Conventional style

```
(define (area shape)
  (cond ((eq? (type-tag shape) 'square)
         (* (contents shape) (contents shape)))
        ((eq? (type-tag shape) 'circle)
         (* pi (contents shape) (contents shape)))
        (else (error "unknown shape -- Area" shape))))
```

The idea is to write a procedure for shape which calculates it for diffrent shape. Var for perimeter etc

[Error here is a stk primitive it prints the error message stops what the prog is doing and goes back to the prompt $ or Read-eval-print loop]

The major disadvantage of conventional style is ~~that~~ its very hard to manage. i.e if we wanted to add a new shape then the pre existing code has to be carefully modified [it's easier to add new opp like ~~Area,Peri~~]

So people have ~~thought~~ about organizing info about diffrent Operators & types. The general topic here is 'Generic Operators'. Generic Operator is a Operator that works independent of the type of data you have Example: Schemes '+'

2) Data directed Programming:

Data Directed	Area	Perimeter
Square	$s \mapsto s^2$	$s \mapsto 4s$
Circle	$r \mapsto \pi r^2$	$r \mapsto 2\pi r$

We create this entire table as a data structure
So we create a 2D data structure with columns as Operator names & Rows as type names

;; Data-directed Programming.
[Two programs 'Put' and 'Get' are provided as a Primitive]

```
(Put 'square 'area (lambda (s) (* s s)))
(Put 'circle 'area (lambda (r) (* pi r r)))
(Put 'square 'perimeter (lambda (s) (* 4 s)))
(Put 'circle 'perimeter (lambda (r) (* 2 pi r)))
```

```
> (Get 'square 'area) 4)
> 16
> (Get 'square 'area)
  # [closure args t = (s) 8134 tes8]
```
↳ a procedure is returned

this is how put & get work
(put 'lastname 'yakko 'warner)
(get 'lastname 'yakko)
> Warner.
get matches the first 2 args & prints the 3rd one.

* Data directed programming doesn't mean that we build a 2-D datastructure. It means we build into a datastructure the info about what the progr should do [somehow encoding]
Thus it's a very general idea.

```
(define (operate op obj)
  (let ((proc (get (type-tag obj) op)))
    (if proc
        (proc (contents obj))
        (error "unknown operator for type")))))
```

we have used let here so proc is defined as the function returned by get

```
(define (area shape)
  (operate 'area shape))

(define (perimeter shape)
  (operate 'perimeter shape))
```

→ This is how we benefit:- Let's say we want to invent a new data type say Δles we needn't change anycode that we have written, all we have to do are 2 puts for area & perimeter only for a new operation we do a bunch of puts.

* here strictly speaking get isn't a function because if we use get with a bunch of args we get one answer we put a diff value for same bunch of args & then use put again we get diff values. So for same arg we got diff answers so strictly speaking get isn't a function. however if we do a bunch of puts & treat the table as a constant then it behaves like a function [we'll learn how to make tables ie write put & get shortly]

It's interesting to note that define has mar property & its implemented with something mar to a table.

* Data-Directed Prog doesn't mean making a 2D data structure it is a very general idea * It also doesn't mean that we are doing Data directed Prog if we have put & get here it is DD prog only because we put & get procedures ie it is DDP only if we put & get "verbs" not "nouns" i.e data

* Functional programming is an abstraction over the abstract data types. because even if we write a functional code what goes on within the system isn't functional with changing states etc. It's just a discipline we choose to follow i.e., It's easier to understand if I write my code using make-trees than to think about what car & cdr of my tree are.

3) Message Passing.

Message Passing	Area	Perimeter.
Square	$S \mapsto S^2$	$S \mapsto 4S$.
Circle	$r \mapsto \pi r^2$	$r \mapsto 2\pi r$

here the data is intelligent if it takes the operation as input & does what's required

;; Message passing.

```
(define (make-square side)
  (lambda (message)
    (cond ((eq? message 'area)
           (* side side))
          ((eq? message 'perimeter)
           (* 4 side))
          (else (error "unknown message")))))
```

(define (operate obj op)
(obj op))

Instead of telling Area with S4 Call S4 with Area as arg.

my

```
(define (make-circle radius)
  (lambda (message)
    .
    .
    .
    .
    .
```

> 54
[closure arglist = (message) 8168754] → Procedure.

> (area 55)
25

* In some sense this is a step backward as Data Driven code was far more manageable, but Message Passing is important as it's one of the key ingredients in Object Oriented Programming

~~The topic for today is data types, different kinds of data~~

For the same (3) cases as above what are we going to do when we have more than one arguments each of which might be a different data type.

* Datadriven programming is a very powerful area that it threatened the jobs of many programmers. years ago the Big Corp would buy their computers to do their payrolls & along with it they would hire programmers to write their payroll codes as this code had to be custom made for each company as each company would have diff format of employeenos diff structure of pay etc

But nowadays we have ready programs that can be bought which come with the flexibility to modify as req. these codes will have Datadriven structure in them. Another Ex: to data directed Prog ie Parser Generator Lect 16: Generic Operations Spring 2008

The point involved in Generic Operators or Generic Procedures

Procedures that can operate on data that may be represented in more than one way. Ex: 3, 3/1, 1/3, 0.3333...

the main technique involved is to work in terms of data objects that have type-tag, data directed prog

* In the previous section we saw how to design sys in which data objects can be represented in more than one way. The key idea is to link the code that specifies the data operations to several representations by means of Generic Interface Procedures.

Generic Interface Procedures: For ex: the code that would identify the real part separately; imaginary part for diffrent representations of complex nos

ie GIPs are the code that would filter all the relevant stuff out of all the diffrent representations the data can possibly have.

Now we will see how to use this same idea not only to define operations that are generic over diffrent representations but also to define operations that are generic over different kinds of arguments.

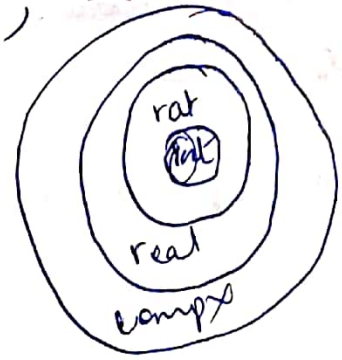
Prob involved:

$(3 + 2/3)$ } Are diffrent cases which means
$(2/3 + 3)$ } for addition we would have to write $4 \times 4 = 16$ diffrent Prog

(1, int) : integer
(int, Rat) : Rational
(int, Re) or (Rat, Re) : Real
(Rat, Rat) or (Re, Re) : Complex

and if we add a new type we will have to write 9 diff cases.

But, we have the following Relationship



So what we can do is when we have 2 diff types to be operated on we will raise the lower one to the higher's order & operate. By doing so we would have to write only 4 procedures for 4 diff data types + 3 procedure for raising. + Give levels (or ranks) to know which one to raise

	Adding +	raise	level
Int	+	(raise x) (make rat x 1)	1
rat	+ rat	(raise x) (make real (/ numx denomx))	2
real	+	(raise x) (make-comp x 0)	3
complex	+ C		4

Doubt: why can't we raise everything to complex and have only one way of doing things.

we can't do that because raising nos results in data loss. especially while raising from Rational to Real $\frac{1}{3}$ when raised to real becomes 0.33333 (viz not exactly $\frac{1}{3}$)

As a compromise we can have 2 representations of complex one with ration & the other with Real, this is possible and is one of the basis of language design. ie how imp is it for me to exactly represent big nos.

Possible breakage cases :-

So how Square is both a Rect & a Rhombus so how do we Raise it?

Parallelogram

Rectangle Rhombus

Square

the Arg is dispensing on the other Arg. is it's a Rhombus or a Rectangle.

how to Raise?

* how do we do diadic Operators (e.g. +) in a message passing System.

here, if we say 3 + 4

message passing works as follows,

it asks '3' to do (+ 4) → viz the message

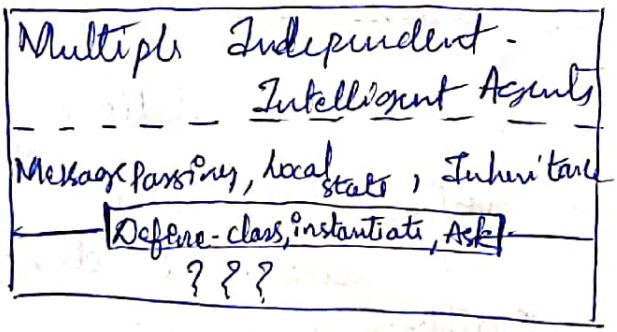
So it places extra emphasis on 3 making the process Asymmetric. In doing so '3' or say the integer should know how to add itself to another integer, Rational, Real, CS etc if its an integer then its straight forward otherwise it would have to Raise itself (or in case Anti-Raise) & pass the same message to the new Raised Obj a

---

Object Oriented Programming

(Functional Programming is done)

We'll look under the hood of OOP



The main reason Behind Obj Oriented Programming is to have Multiple Independent Intelligent Agents.

Here instead of Running the Program we think of Running them independent Agents.

here the idea isn't efficiency [Parallel use of cores like in functional] instead its a metaphor [Like mathematical functions for functional Prog]

Functional prog is suitable when prob are very well defined & has a well defined soln. But when there is a need for flexibility we go for OOP ex for this is simulations Prog :- Exploring diff possibility for traffic routing to avoid congestion. we model behaviours of diff cars, trucks into diff objects etc. Ex 2: Button, Sliders etc in a interactive GUI

To realise this we need 3 things:
1) Message Passing - Already discussed
2) Local State
3) Inheritance.

1) Message Passing: is through which one object communicates with each other like pls do such and such, pls give such & such etc

2) Local state: here is where we are breaking with functional programming. Because in Func Prog when you give the same Input to func it gives you the same Output it isn't allowed to remember about its history. where as Objects do remember

3) Inheritance: "This kind of obj is a bit like that kind of obj except for ... ."

* the most important point here is that Object Oriented Prog acts as a Veneer & we can do the same stuff that we do in Oops ~~without using~~ just by using the regular stuff

Vocabulary:

* Class is an object say 'CAR'
* Instance is a particular object 'My CAR'

So 'define class' is a constructor for classes

* Instantiate is a selector for classes but a Constructor for Instances

* Ask is the way we do message passing. *
→ 'Ask this obj such & such'

---

under the hood obviously we use 'λ'

```
(define-class (complex real-part imag-part)
  (method (magnitude)
    (sqrt (+ (* real-part real-part)
             (* imag-part imag-part))))

  (method (angle)
    (atan (/ imag-part real-part))))
```

→ name of the class
real-part imag-part → instance variables - something for which I have to provide a value when I create a instance
(magnitude) → messages
(angle) → messages.

```
(define-class (counter)
  (instance-vars (count 0))
  (method (next)
    (set! count (+ count 1))
    count))
```

> (define 3+4i (instantiate complex 3 4))

3+4i

> (ask 3+4i 'magnitude)
↳ observe Quote the message is just a word

> (ask 3+4i 'real-part)

3 [* here we have not written a method for the identif of real part but we get the right result, this is because we get a ready made procedure with the definitions of all types of Object Variables (There are 5 types of instantiation Var is one of them)] → [not all Obj Sys do this its an emphasis on hiding info. So in such cases we would write (method (real-part) realpart)

The counter class discussion

(define C1 (instantiate counter))
(define C2 (instantiate counter))

> (ask C1 'next)
1
> (ask C1 'next)
2.
> (ask C1 'new)
3.

} reason OOPS is not functional

> (ask C2 'next)
1.

here C2 is ind of C1's memory

This works because of 2 things

1) Instance-vars

This is different from Instantiation Variable so that we needn't provide a Value for it during instantiation

In third prog, (instance-vars (count 0)) is very much to 'let' in its syntax but unlike let it remembers its state

2) Set! (set sayy)

We use to define instead of saying define a new value it says change the ~~Value~~ Value of an existing ~~Variable~~ Binding.

* Set! is the reason we are not doing functional prog.

```
(define-class (doubler)
    (method (say stuff) (se stuff stuff)))

> (define dd (instantiate doubler))
dd. to let the sys know.
```

→ message passing → name of the instance to which message is passed

→ method name passed as message

→ arg for the method passed as message.

```
> (ask dd 'say 'hello)
(hello hello)
```

```
(define-class (counter)
  (instance-vars (count 0))
  (class-vars (total 0))
  (method (count)
    (set! total (+ total 1))
    (set! count (+ count 1))
    (list count total)))
```

> C3 + C4 defined.
> (ask C3 'next)
(1 1)
> (ask C3 'next)
(2 2)
> (ask C3 'next)
(3 3)
> (ask C4 'next)
(1 5) → because its class defined & not instance.

> (ask C3 'count)
1
> (ask C3 'total)
5
> (ask counter 'total)
5
If we wanted to reset the variable we would have to write another method for that since this is class defined

---

Simulating behaviour of people in the world.

```
(define-class (person name)
  (method (say stuff) stuff)
  (method (ask stuff) (ask self 'say (se '(would you please) stuff)))
  (method (greet) (ask self 'say (se '(hello my name is) name))))
```

> (define bh (instantiate person 'brian))
bh
> (ask bh 'ask '(study hard))
(would you pls study hard)

→ message that calls the ask method that we define.

→ Self is a instance variable that every object gets automatically & value is the obj (instance) itself

Clauses within define-class are :
1) Method
2) Instance-Vars
3) Class-Vars
4) Parent
5) default-method
6) ~~Class vars~~
6) Initialize

```
(define-class (pigger name)
  (parent (person name))
  (method (pigl wd)
    (if (member? (first wd) '(a e i o u))
        (word wd 'ay)
        (ask self 'pigl (word (bf wd) (first wd)))))

  (method (say stuff)
    (if (word? stuff)
        (ask self 'pigl stuff)
        (map (lambda (w) (ask self 'pigl w)) stuff))))
```

```
> (define pp (instantiate pigger 'porky))
  pp
> (ask pp 'say '(for no one))
  (orfay onay oneay)
> (ask pp 'ask '(go to the window))
  (ouldway ouyay easeplay Ogay otay ethay indowway)
```

this is happening because we have defined (parent (person name)) which means pigger inherits from person all the methods including ask but the 'say' within the ask is taken from pigger & not Person. we take method from Person only if we haven't defined it in pigger

- There are different methods in which inheritance can be done here its done through 'delegation' here we create a person object within a pigeon object. so when we pass a message to pigeon it says is this a msg that I understand, if so I deal with it if not I send it to my internal person to handle

> (ask pp 'greet)
——— (no value returned)
this is because the program is stuck trying to convert 'my' to pig as it doesn't have any defined vowels [infinite loop] [greet within person → say → pigeon]

- (define-class (square)
  (default method (* message args)
  (method (7) 'buzz))

> (ask s 8)
64.
> (ask s 7)
buzz.

Because of the default method it can take all msgs
so > (ask s 'gaply) → even if there is no method called gaply it takes it as a msg because we have a default method & it thinks gaply is an argument for that

* Initialize - think of it as a method i.e (define initialize) but its thru internally within every class. This is the method through which it assigns a value to itself ie 'self'. So if we explicitly initialize something in the code it is just included with the implicit one & nothing else.

---

* Suppose we have written the following code.

```
(define-class (counter)
  (instance-vars (count 0))
  (class-vars (total 0) (counters '()))
  (initialize (set! counters (cons self counters)))
  (method (next)
    (set! total (+ total 1))
    (set! count (+ count 1))
    (list count total)))
```

* Here once we write this code & lets say we have not created any instances [instantiation not done] in such cases. Class vars would still exist but instance vars will not & we wont have access to any explicitly written methods unless we create an instance. But the instantiate method which is invoked when we instantiate is one method the class knows to use.

* Some object oriented programming languages doesnt have this distinction b/w class & instance. So when we create a class actually it creates an instance & when we create an instance, it just makes a copy of it

pg:-

* I know not completely clear as to exactly how this is working, but my thoughts are, 'cons self counters'. everytime a new instance is created it adds it 'self' to the counters list & hence the mapp & so on.

* Procedure to overcome the 'great' infinite loop problem because of 'my'

  pigl method as it is. ✓ ∴ (method (say stuff)
  (if (word? stuff)
  (if (equal? stuff 'my)
  (usual 'say stuff) (ask self pigl-stuff))
  (map (lambda (w) (ask self 'say w)) stuff)))

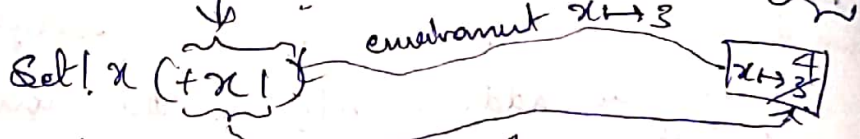
* usual works as following ←
  it works way to 'ask' that
  Ask wi: it follows this flow chart:-
  1. method in this instance? (class)
  2. method in the Parent? (class) Usual.
  3. method a Default?
  4. Error.

  but usual doesn't do the step ① instead starts from step ②.

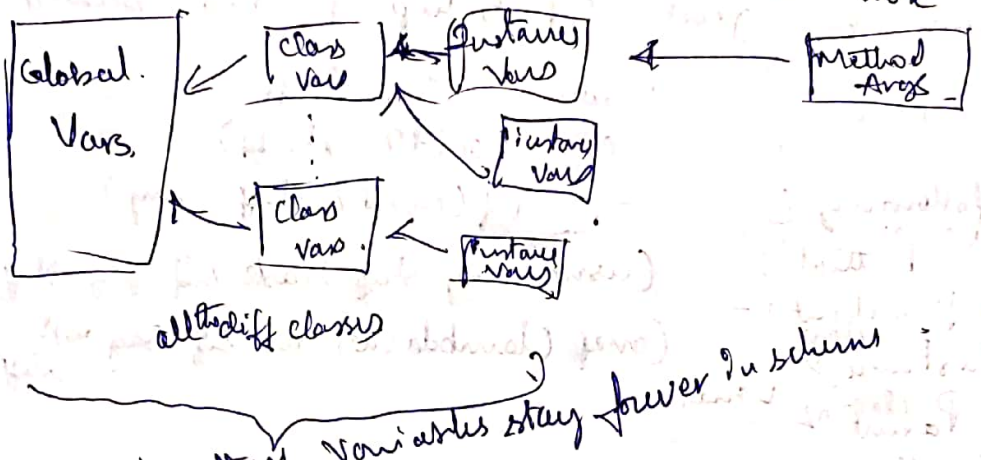
* The independent windows in Windows Op systems are programmed in a OOp fashion because they are independent entities by nature. Each window usually doesn't speak to another window & so it is natural to program them that way. Eventhough we can use 'Not OOPS' way to code them it will be extremely difficult & complicated. So OOPs is just the natural way to do certain things

* when we move away from functional programming especially by using Set! we can no longer use the substitution model. ie (Set! x (+ x 1)) x; if we call Set! on 3 then it would work 3 + 3 1 3 ↙ which makes no sense. even if we retain x ↓ even then since we have already made x=5 it would print '5' & not 4. So we would have to use the idea about variables used in 'C' where we treat variables like shoe boxes where we can put & remove things as we wish & it is called the environment Model

which basically says evaluate this expression in the this environment



So a General Environment will look something like this



We generally hear that global vars are bad but all the primitives are defined as global vars ex: '+' is binded to a procedure that does addition. & it is a global var.