Von Neumann Architecture

Fetch - Decode - Execute - Cycle

Control unit – instruction register (cur. instruction)

– program counter: points to next instruction on memory

instruction: read, write, rewrite counter (change default order of execution)

(assignment) v = (x+y) (assignment statement)

read x and interpret as int

read y and interpret as int

add x and y

store result into (v)

advance to nxt instruction

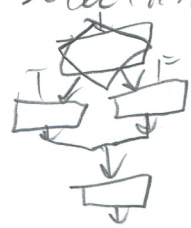
8/16/32/64-bit (bus i/o size)

(von Neumann bottleneck)

GO TO / JUMP / BRA (control modification)

Dijkstra: Go To Statements considered Harmful (1968)

1. Sequencing 2. Selection 3. Repitition



Single Entry Single Exit

Exception handling, goto better

Process-oriented → Data oriented

- Abstract Data type. (by Barbara Liskov)
  - Encapsulation
  - Information hiding No need to ~~hide~~ change

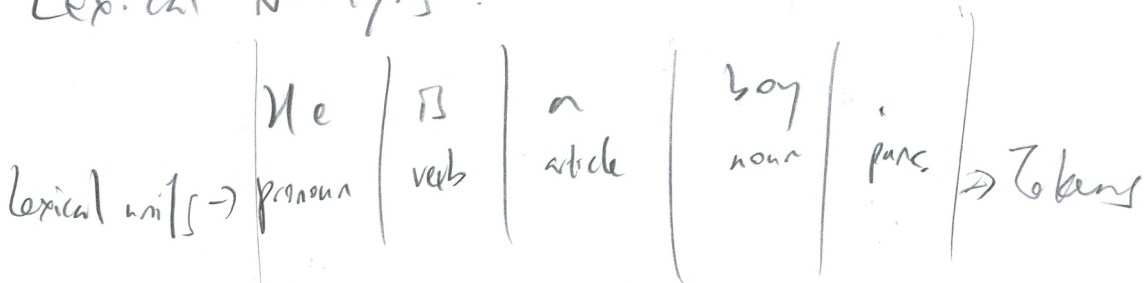
OOP

- ADT + inheritance + polymorphism (reuse)

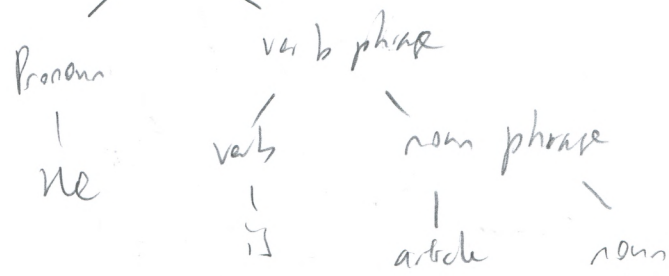
Interpreter reads & execute line by line.

Compilation: slow translation → Fast Exec.

Lexical Analysis



Syntax Analysis (Parsing)



Load module, Linking and loading

Hybrid Implementation Sys. for portability

Scientific paradigm: written rules or laws
belief of practitioners
~~tres~~sense of tr values about what is important
applicability to problems & associations

imperative: Von Neumann, cmd & var update, state oriented

OOP: ~~interactive~~ interacting objects via messages,
obj classes, inheritance

declarative: mathematical formalisms, assignment-free,
what vs. how (fp)

concurrent & : extra control mechanism
distributed

OOP: class, var, type, func (Names)

~~F~~95: max len 31
C99: no limit but only first 63 significant,
external names max 31.
C#, Ada, Java: no limit, all significant
C++: no limit, but implementers impose

Conventions: case sensitivity C-based, yes
others, not
keyword, special word, special in certain contexts
reserved, special word, cannot be user defined name.

var: name, addr, val, type, lifetime, scope

no name for dynamic memory.
pointer & references

var w/ diff. addr at diff. time due to
scope limit, func. recall.

var w/ diff. addr at diff. places in a program, diff. scope

aliases ~~var name~~ diff. var name point to same memory location
~~F~~ chr values, ~~H~~ values

Hybrid implementation ~~for i in {'stdio', 'stdlib', 'string', 'math'}~~

Cause: 1. had to ~~fa~~ translate OOP into machine code
2. portability } multi-language support

.NET of MS is family of lang. Visual Basic, C#...
diff. lang for diff purpose
multi language support

↳ compile into intermediate language → compilation
Save Applied code in logs for optimization interpretation

Compiler in C (bootstrapping) maybe init compiler (BCPL)? (Fortran)?
minimal subset

→ Von Neumann → ~~Algol~~ Programming Methodology
floating pt. & decimal Precision

Abstraction: General ideas, not specific manifestation
- variables & assignment abstract away from storage fetch & move
- control structures abstract away from jump.
- generic abstraction parts of program away from type of values which they operate in instead of reusability.
  - data abstraction consists objects & operation characters/behaviour
  - control abstraction → sequencing with. actions
  - procedural & modular abstraction specifies actions of a computation on a set of input & ~~output~~ output objects.

Var : Type : a set of of values that the var can hold

e.g. Boolean = {TRUE, FALSE}

int = {[-32768 : +32767]}

ascii = set of std: ascii table

float = set of ieee floating-pt. std.

value : l-value is addr, write to

r-val is val. read & act.

Abstract mem cell : phy cell or collen cell of the var.

Binding : bind attributes ~~to~~ to it.

Binding time : Lang. design time -- bind operators to operations. * - multiply
- dereferencing

Lang. implementation time -- bind type ~~to name~~ ~~bind~~ to representation
e.g. int = 4 byte

compile time -- bind var to type

load time -- bind C/C++ static var

((load)) after linker, allocate mem to static

static : program lifetime var

runtime -- bind nonstatic local var to mem cell

static binding : before runtime, & remains unchanged throughout execution

dynamic binding : occurs during execution or change during execution.

explicit declaration: int x; → statically typed (helps catch bugs)

implicit declaration: i, j, k, l, m, n in Fortran

type inferencing: C#, ML var x = 1; ← still should declare type

type error ⇒ helps debugging; help catch inference (int)

In statically typed language, is the var possess the type

In dynamically typing, is the content of the var possess the type

Dynamic typing: high cost, difficult type err. detection by compiler

can do polymorphism w/o inheritance

Storage Bindings & lifetime

allocation & deallocation

Static: alive when ~~the~~ till program terminates

in C: bounds to mem ~~before~~ execution

load time

static

in ~~compile time~~ before run-time or at start of run-time

adv: efficiency: dir. addressing, history-sensitive

disadvantage: no recursion (lack of flexibility)

list-sensitive; foo(){ static int x=0; printf(x); x+=1; }

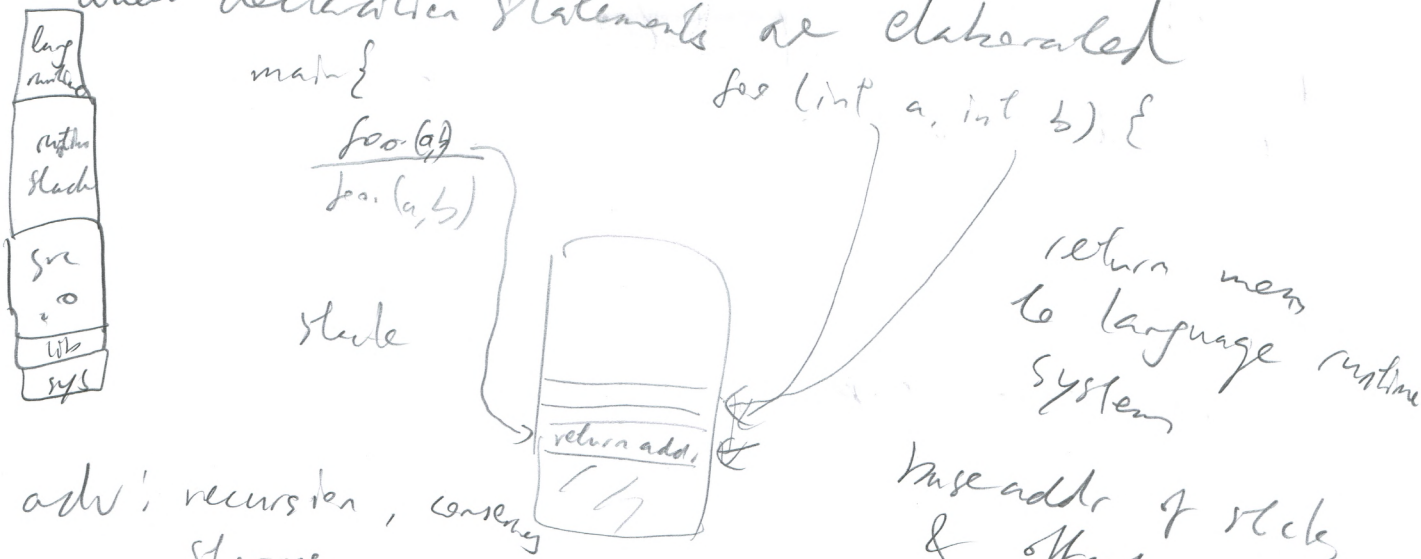
foo() = 0
foo() = 1
foo() = 2
...

like instance var in objects

in Java, C++ class var use static

instance var has their own instances

stack-dynamic: storage bindings are created for var when declaration statements are elaborated



adv: recursion, conserves storage,

cons: overhead of allocation and deallocation, not trivial. sensitive

explicit heap dynamic: allocated & deallocated by explicit dir. specified by programmer, takes effect during exe. referenced ~~thru~~ thru ptrs / ref. e.g. new, malloc...

pros: dynamic storage management

cons: inefficient & unreliable

implicit heap dynamic

: allocation & deallocation by assignment statements (experimental)

flexibility (generic code) inefficient

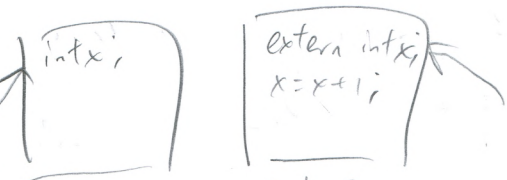
cons: all dynamic

applied occurrence: in function, operating binding occurrence: int x;

Scope: local, declared in {unit} non local, visible in unit but not declared there

static Scoping

C/C++



definition
✓ malloc

file1.c file2.c

declaration . X malloc

FP:

let

var name = exp

...

in

exp

end;

Dynamic Scoping

Pros: convenience (big structure) (useless) (~ pass ref)

Cons: subprogram exe, var are visible to all subprog.

impossible to stat. type check

poor readability

Python: can read parent declared var

~~Ao~~ Data types

Primitive Data Types (Not made from other data type)

Int (signed/unsigned)

lots byte, short, int, big, big big

8?

float 1 + 8 + 23 bits double 1 + 11 + 52 bits

sign exp frac sign exp frac

4B 8B

complex decimal (0-9) bool string

substring ref

~~Limited~~ String

Static Length

Limited Dynamic Length C/C++ string, arr of char end '\0'

Dynamic length

Linked List Array of pointers Adjacent memory

Ordinal type can easily associated w/ pos int.

Enumerated type statically typed language

statically typedef lang

enum days x; x = Sun

enum solar y; y = Sun

Enum, not allow allow coercion, or assigned a value
may
outside its defined range

C/C++ enum coerced into int.

Efficiency vs Reliability
C/C++ Rust

Array: static array
efficiency
static int x[100];

fixed stack-dynamic
space optimality
int x[100];

stack dynamic
flexibility
int x[i];
run-time

(fixed)heap-dynamic
flexibility

Heterogeneous Arrays, elements need not be same type

Row major: <type> var [n_N][n_{N-1}]...[n_1] (C/C++)

declared

- ~~offset~~
- offset(i) = n_i offset(i-1) i ∈ N find var [x_N]...[x_1]
- offset(1) = n_1 offset(1) = x_1 sizeof(<type>)

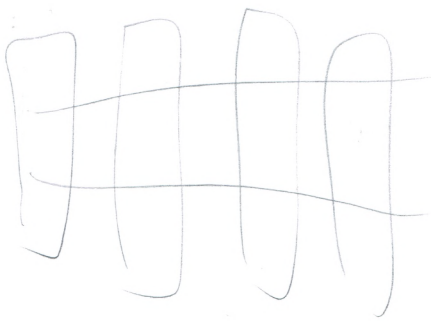
Addr of var[x]...[] = addr + $\sum_{i}^{N}\left(x_i \prod_{j}^{N}\right)$

Addr result: addr + $\left(\sum_{i=2}^{N}\left(x_i \prod_{j}^{i-1} n_j\right) + x_1\right) \cdot sizeof(<type>)$

Column major: (x,y)

Same formula in row major

~~result: addr~~ but var [n_1][n_2]...[n_N]
var [x_1][x_2]...[x_N]



Records are ~~fast~~ faster than array as pre-computed offsets

Assumed init at 0

~~Shape~~ Union type
e.g.

```
type Shape is (Circle, Triangle, Rectangle);
type Colors is (Red, Green, Blue);
type Figure (Form: Shape) is record
    Filled: Boolean;
    Clr : Colors;
    case Form is          # Discriminant for tag run-time type checking
        when Circle => Diameter: Float;
        when Triangle =>
            LeftSide, RightSide: Integer;
            Angle: Float;
        when Rectangle => Side1, Side2: Integer;
```

```
x.Form = Triangle; x.LeftSide = 5.7; x.RightSide = 2.3;
x.Form = Circle; x.Diameter = 5.8;
print x.LeftSide; ← error Ada
```

Free unions unsafe, discriminated unions safe, Java/C# X unions

A reference var is always automatically and implicitly dereferenced, and final/const pass by value (pass address also)

pass by reference
ref. var cannot be updated, once inited.

Java obj var behave sometimes like pointers, some times like ref. var.

Heap dynamic lifetime, malloc → dealloc / malloc → dealloc / gb
ptr. lifetime (Ada)

leak can be caused by lifetime of ptrs,
ptrs changed, exception occured, not reach free

int a[10];
a is an int array of length 10

int *p = malloc(sizeof(int)*10)
p is a pointer pointing to a chunk of memory of 40 byte

a is const addr of a[0]
&a == &(a[0])

pointer var = var store addr of var
value = addr of var

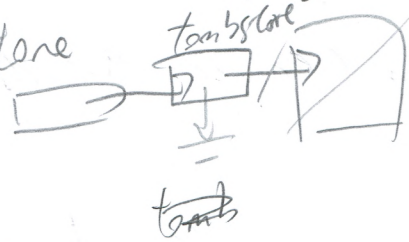
dangling ptr.
- out of bound by ptr. arithmetic
int a[5]; *(a+7)=2;
- scope & lifetime
int *foo(){ int i; return &i; }
- deallocated

memory leakage
- out of name scope
- not deallocated

{ int *a = malloc(); }

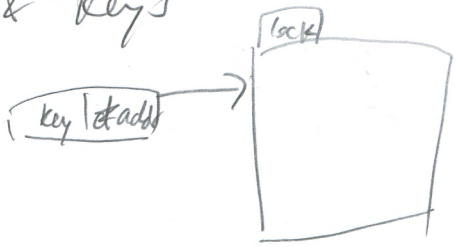
~~Tombstone~~ Dnlg Dangling Ptr.

Tombstone



Cons: Waste Space, not efficient
extra check, extra redirection
will raise exception if freed

Locks & Keys



Cons: extra space, inefficient

Heap Management

Reference Counter

easy method

inefficient
(space + time)

Cons: complication to handle loops

self-reference / circular linked list

Mark sweep method

Lazy approach

1. mark all heap mem cells as garbage (marking phase)
for loop

2. DFS search tree and mark not garbage

(clearing phase) 3. for loop free garbage

strong typing: ~~detection~~ always detect type errors

C, C++: parameter type checking can be avoided / Union Type

Name type ≡ use same declaration or same type name
(幾乎) Pascal, same type but not in same place
easy to implement but restrictive

~~Structure type~~ ≡ struct, union, class

Structure type ≡ def a{int b; int c} def ~~a~~ f int b; int c}
typ a = type b

operator: '+' operand: 'a', 'b' ⇐ a + b

precedence: NOT AVAILABLE IN SMALLTALK

associativity: L to R / R to L (**) (some unary)
left associative right associative

a := if then else ; (expr)

precedence z = x + foo(&x); ... unstable side effect
foo(y)
static x ...
return x+y
functional side effects

Sol. 1. No 2-way No non-local ref parameters in func, 2. No non-local ref in func.

cons: ~~limits~~ less power for lang.

2. Fix operand evaluation order
cons: ~~less~~ less optimization

3. Referential Transparency
Any 2 expr have same value can be substituted for one another anywhere in prog.

narrowing conversion: float → int not include all origin type

widening conversion: int → float at least approx. all origin type
(promotion)

Type coercion: ↓ type error detectability

Ada X coercion

Selections

dangling else: if ...
if ...
else ? which if belongs to

2-way selection (if ... then ... else ... , ? :)

multiple-way selectors

Ruby → case
when and then action
else
end

(PHP) →
expr

for loop in C

counter-controlled loop

for (expr.1 ; expr.2 ; expr.3 ;)

(goto)
legal to branch into
body of a for loop in C

```
expr.1;
while (expr.2)
{
    ...
    expr.3;
}
```

Logically-controlled loop

pretest while post test do...while / repeat...until

Java X branch into loops

User-controlled loop single

still single-entry ~~multiple~~ exit (break, continue)

Iteration on Data Structure

foreach loop over elements

PHP: current: 1 element in array next: move cur to next ele. reset: moves cur to first ele.

Java 5.0: 1 table interface e.g. Array List

C: for(p=root; p!=NULL; p=traverse(p)){ }

C#: for(auto obj: objs){}

Blocks do iteration (OOP e.g. Ruby)

3.times{ code block } list.each{|value| puts value}

1.upto(5){|x| print x, " "}

GOTO, IF...GOTO... (program selection)

↑ readability: Exception handling, FSM

try-catch

Subprogram

~ Objects called and act upon message
sending message to obj ~ parameter passing
(method invocation/calling)

method is operating on the object
func is operating according using parameters.

subprogram definition, ~~body~~ body

parameter profile (number, order, type of parameters)

protocol (parameter profile + ret type)

subprogram declaration provides protocol int foo(int a, int b);

definition int foo(int a, int b) { }

foo : ~~(~~int × int → int

signature

protocol

~~parameter type~~

formal parameters : declared in ~~function~~ parameter profile

int foo(float y, int z)

actual param : params passed into subprogram

arguments : expr of parameters to be passed.

positional parameters vs. keyword parameters (keyword correspondence)

Ruby blocks (anonymous) passing of code in function

```
def fib_up_to(max)
  i1, i2 = 1, 1
  while i1 <= max
    yield i1
    i1, i2 = i2, i1+i2
  end
end

fibonacci(100) { |num| print num }
```

Sub programs → procedure (subroutine) (collection of codes)
→ function (mathematically based, ∄ with return)

procedure (inserting codes into designated place)
- modifys parameters directly

function (a subprogram on parameters that gives output)
- returns, theoretically no side effects
- may have side effect if have state / pass by reference/addr
- modern lang use func to replace subroutine which is special case of functions.

Ref Env.
Stack-Dyn. Local Var
Pros: recursion, storage of locals shared among some subprograms
Cons: Allocation / de-allocation, init time
Indirect addressing
Cannot be history sensitive

Static Local Var
Dic. opp. Pros & Cons of stack dynamic

Param. Passing Model: in, out, inout modes

Pass by Value formal param = local var
{param} = {in value}
~~Pros~~: Normally implement by copying
Can by address (access path)
But space & ~~to~~ time costly ∴ duplication
But ~~write access~~ write-protect in subprograms
access cost more ∴ indirect addr.

pass by result: foo(int out x, int out y, int z)
like pass by ref. ... return x, y, z
problematic like foo(a, a, c) → x=x, a=y
~ side effect

pass by value result:
or pass by copy cons: these of pass by result
local duplications. values.

pass by reference (pass by sharing
Cons: slower access to formal param.
unwanted side effects
unwanted aliases
pros: efficient access

pass by name:
outdated, textual replace, problematic, C preprocesser
e.g. swap(name int x, name int y) { int tmp;
tmp = x; x = y; y = tmp; }
...
a = 3; b = 4; swap(a, b);
= tmp = a; a = b; b = tmp;
but in swap(i, list[i])

All passes are on the runtime stack

C supports pass by value only, but emulates pass by references by passing pointers as parameters.

Java supports all parameters are passed by value, object are passed by reference in behind.

Type checking parameters: Fortran 77 & original C: none
FORTRAN 90+, Pascal, Java, Ada, always required
ANSI C &C++, choice is made by user.
JS, PHP, no type checking, Python, Ruby, impossible

Subprogram passing

1. Ref. environment — by func. pointers / delegate (ref)

Shallow binding:

```
fun sub1() {
    var x;
    fun sub2() {
        print x;
    }
    fun sub3() {
        var x;
        x = 3;
        sub4(sub2);
    }
    fun sub4() {
        var x;
        x = 4;
        subx();
    }
    x = 1;
    sub3();
}
```

← deep binding (static scoping)

← ad hoc binding

← shallow binding (dynamic scoping)

Generic ≠ Sub programs

generic or polymorphic functions

- 1 func. def., diff. argument types e.g. printf

Ad hoc polymorphism ≡ overloading

Subtype polymorphism = var of type T can use any type derived from T

Parametric polymorphism ≡ C++ template creates func at compile time

```
template <class Type> {
    Type max(Type a, Type b) { return (a > b ? a : b) }
```

Closures

```
function makeAdder(x) { return function(y) { return (x+y) } }
var add10 = makeAdder(10)
add10(20) == 30
```

closure

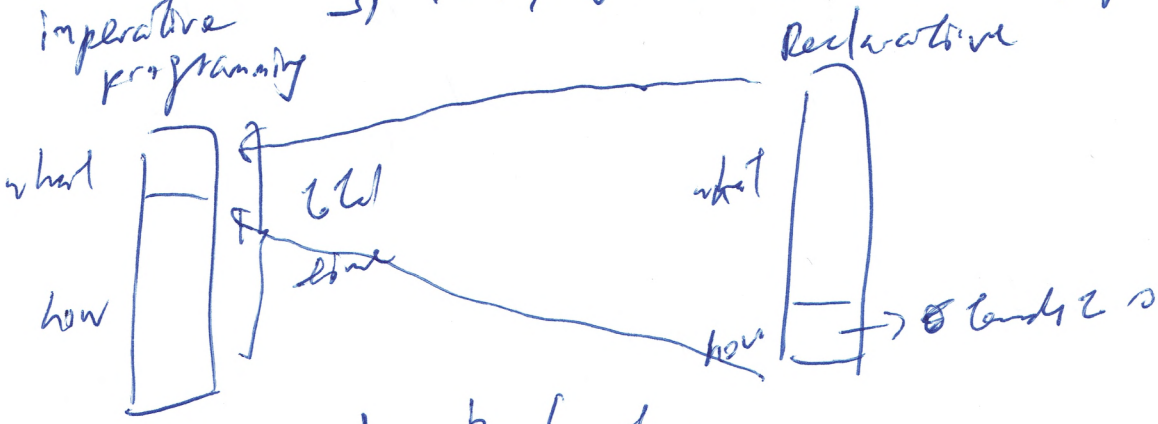
multi-dim array passing

What vs **How**

the problem is to solve the problem

$n! = n \cdot (n-1) \cdots 1$

→ `r=1; for i=1..n r=r*i; print(r);`



Properties of declarative

- Simple program semantics "What You See Is What I Mean" (WYSIWIM)
- Higher program understandability & verifiability
- Referential Transparency
  - Close to Math
  - Compute by value, not effect
  - Meaning of program is independent of execution.

Styles

- Logic / relational — deduction or proof oriented
- Functional — function application oriented
- Equational — rewriting (1980s)
- Algebraic — constraint-solving

SW Eng.

- Correctness
- dynamic & interactive env
- Rapid prototyping & exploratory programming for complex problem w/ no clear sol. available at start
- Knowledge is declarative & requires fusion of program & data

Logic Programming (Declarative Programming)

facts functor(x, y). rules functor(X, Y) :- fact(X, Y).

Derivation by reduction & substitution

composite objects. Functors (will not calculate)

latitude(p), longitude(q) → loc(p, q) in deg.

e.g. loc(north(45), east(72))

records = trees

sum(0, X, X).

sum(s(X), Y, s(Z)) :- sum(X, Y, Z).

sum(X, Y, s(s(0))).

~~sum~~(0, X, X) → sum(0, s(s(s(0))), s(s(s(0)))).

sum(s(X), Y, s(Z)) :- sum(X, Y, Z).

→ sum(~~s(0)~~X, Y, s(s(0))).

~~→ sum(s(0), Y, s(s(0)))~~.

→ sum(0, s(s(0)), s(s(0))).

→ sum(s(0), s(s(0)), s(s(s(0)))).

Conceptual Model of ~~FP~~ FP is of an oracle
expr has value, and gives the expr a name
1. create a val, which is a function
2. to name the value

= val : fn : real * real -> real

function
- fn((x:real),(y:real)) => x*x + y*y;
-> fn((...)) y*y; (5.0, 2.0);
val it = 29.0 : real

val sumsq = fn((...)) ...; ← bind value

-> sumsq(5,c); int

val name = fn => fun (in ML)

~~val sumsq =~~ fun sumsq ((x:real), (y:real)) = x*x + y*y

function in ML is 1st class citizen: act like ordinary value -> can ~~assi~~ be assigned, passed as argument, returned as values from fns

<u>No coercion in ML</u> but can overload

(real 1);
real :
val it = fn : int -> real

<u>ML</u> is strongly typed -> no coercion

For recurse:

: input : int

val rec fact = fn n => <u>if n<=0</u> then 1 else <u>n*fact(n-1);</u>

output

ML takes only 1 element in argument, but can be in tuple w/ more than 1 element

$f(x) = x+1$ $f(x) \to 3$ $f: 2 \to 3$ $f = 2 \to 3$

higher algebra

pattern matching _ underscore: universal selector

local declaration let val p = (area(2,0)) in
sqrt(p*(p-a)*(p-b)*(p-c)); Heron's
↑ efficiency and ↑ readability

datatype CAPNAMES = ... | ... | ... ;

case in ML

```
(fn dir => case dir of North => 0
                     | East => 90
                     | South => 180
                     | West => 270);
```

or (fn cases => eval) input;

pattern matching match case from top to bottom

like prolog

st_card in eg. of lec means greater than card in poker

```
(v1:int) > v2
: v1 is int : v2
  is int
```

Lists

[1,2,3,4,5,6] = 1::(2::3(::...))

hd([]) = head of list

tl([]) = tail of list

```
fun sumList list = if null list then 0
                   else (hd list) + sumList(tl list);
```

∴ if null returns bool,

homogeneous → int list

∴ int list → int

or

```
fun sumList [] = 0
  | sumList(a::l:ll) = a + sumList(tl(a))
```

a great list = a + sum List list;

Union type

```
datatype num = i of int | r of real
```

Records

all tuple

'a binTree = empty | node of 'a bTree * 'a * 'a binTree;

function appl. highest precedence

double in 3

fn(fn x => 2 * ln(~x))

len: length of any list

list pre written

double length.

```
fun map(f, []) = []
  | map(f, x :: xs) = f(x) :: map(f, xs);
```