CS 655

①

Mr	Raphael	Finkel
Dr	Rafi	Goldstein
Prof.	רפאל	
–		

---

client

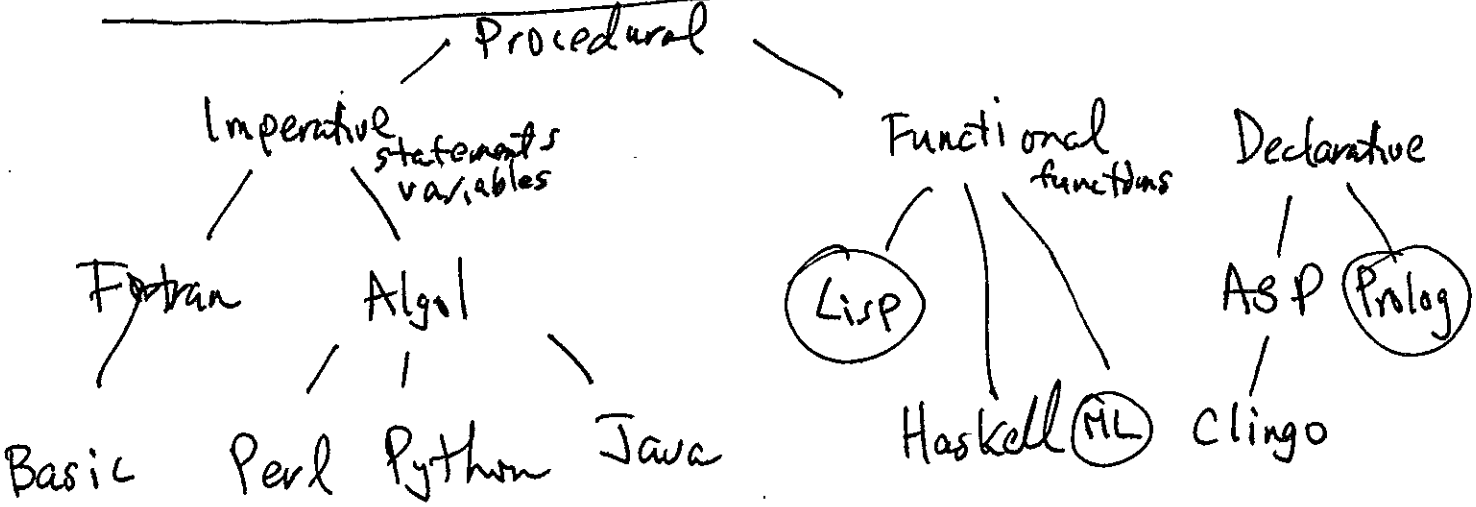
specification.

implementation

programmer

compiler

[syntax
semantics



Control

- sequence (;)
- goto
- functions - subroutines - procedures.
    - parameter passing modes.
        - Fortran: reference.
        - C: value.

actual / formal
↓ ↘
at point of call. inside procedure.

- Algol: name.
- ~~Pascal~~
- Ada: in / out

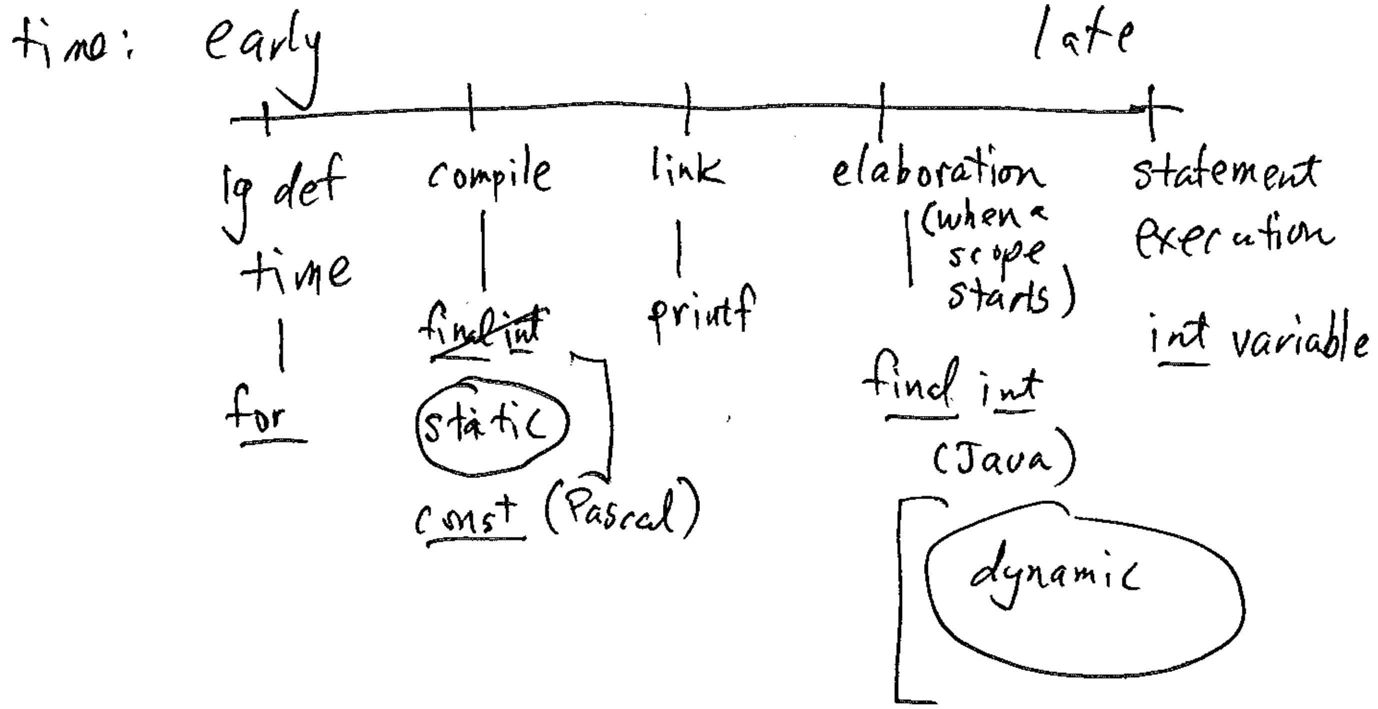
nestable
- if
- while / do-until
- for
- case

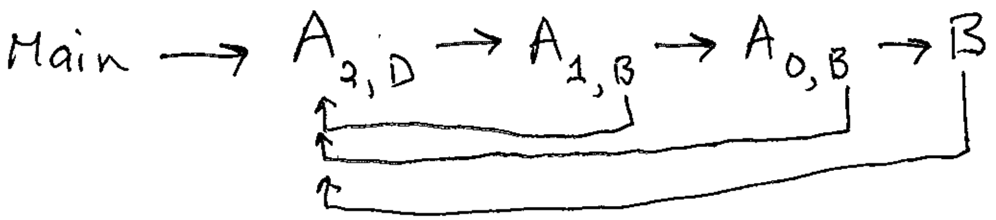
iterators/generators.

lambda expressions.

Theme: Binding time

binding: connection between a name and its value.

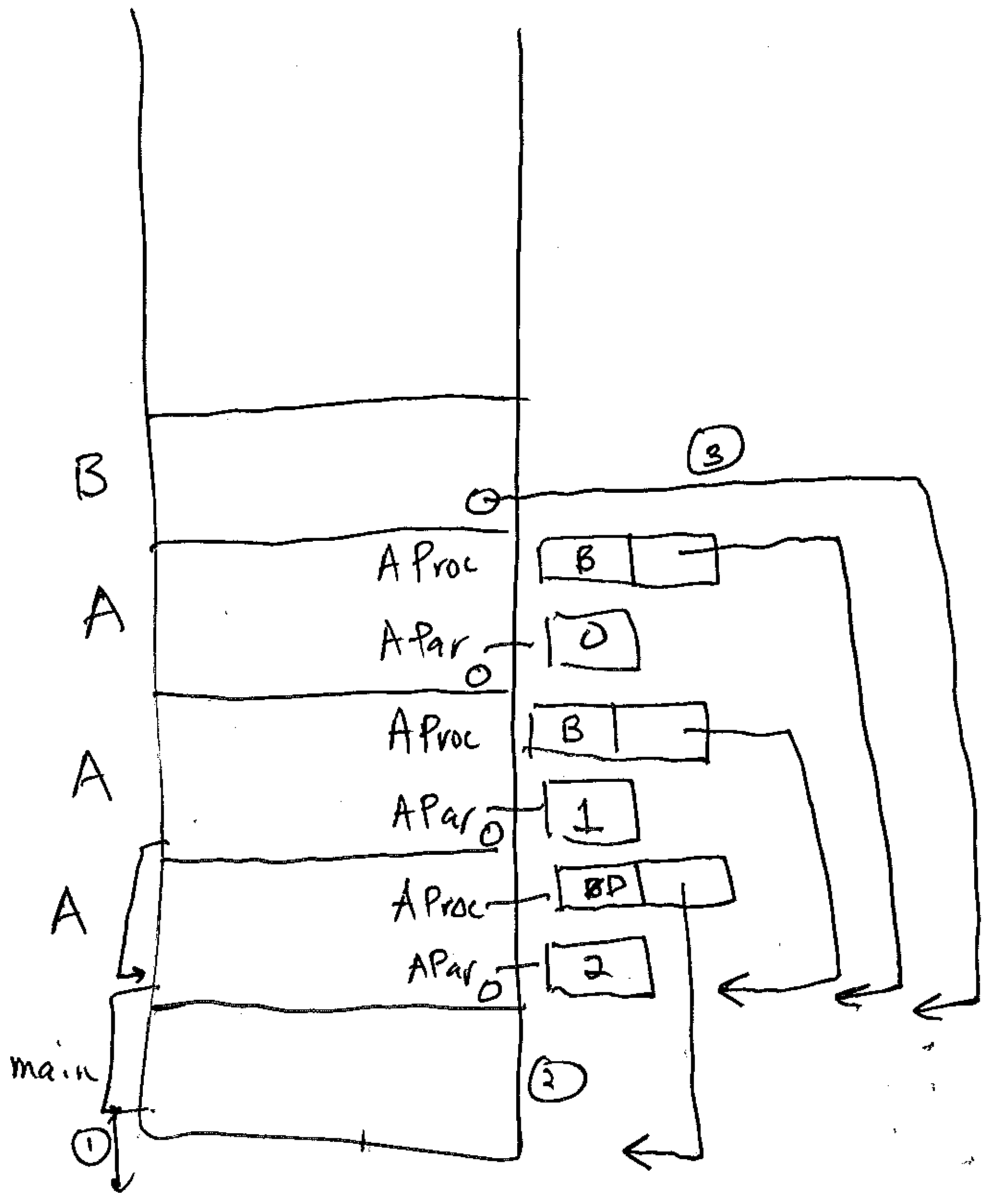




pass a procedure by closure

(code address,
non-local referencing environment)

NLRE = pointer to stack



① dynamic chain (to caller's frame)
② frame = activation record
③ static pointer

Block structure

Introduced in Algol

Block: nestable name scope

nestable → one inside another

name → identifier introduction. (declaration)

scope → region of code where identifie exists

local
non-local: inherit
hide
explicit import/export
global

Elaboration time

constants get values.

dynamic-sized types are bound to their size

space allocated for variables.

NLRE: the identity of non-local identifiers.

Deep binding: NLRE of P is determined when P elabora

Shallow binding: " P is invoked.

Iterators (generators)

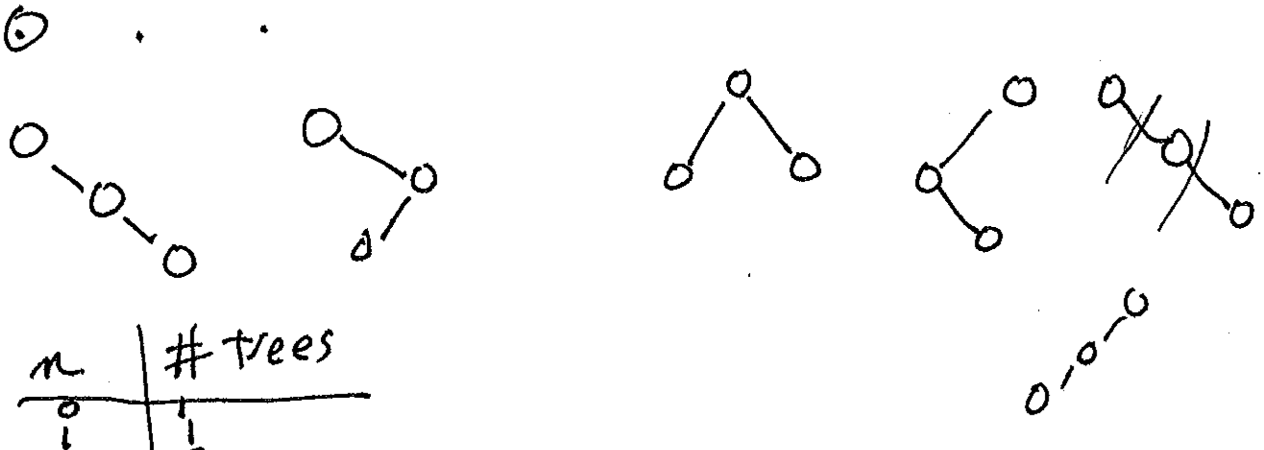
Python, JavaScript, C
↑
my macros.

Generalization of <u>for</u>:
iterates over a set of values
generated on demand by iterator.

Like a procedure, but it presents values
by <u>yield</u>, not <u>return</u>.
↓
maintains its referencing environment
and program location.

Useful for generation combinatorial structures

Example: All trees with n nodes.



n	# trees
0	1
1	1
2	2
3	5

Imperative languages

Statements: modify the current <u>state</u>

↓

variables

variable: identifier

bound to a type (typically staticly)

bound to value (dynamicly, repeatedly)

L-value: use on LHS of assignment ("address")

R-value: use on RHS ("content of address")

type: set of R-values

set of operations

built-in (+)

programmer-defined (procedures)

Conversion of types

coercion: f = i (implicit conversion)

cast: i = (int) f (explicit conversion)

non-converting cast

i = f <u>qua</u> int

reinterpret_cast<int> f

operations: functions or operator symbols (+, *)

infix *

postfix -> .

prefix (C: *)

semantics: lazy, eager

↓ unary: !

binary +, *

ternary : ?

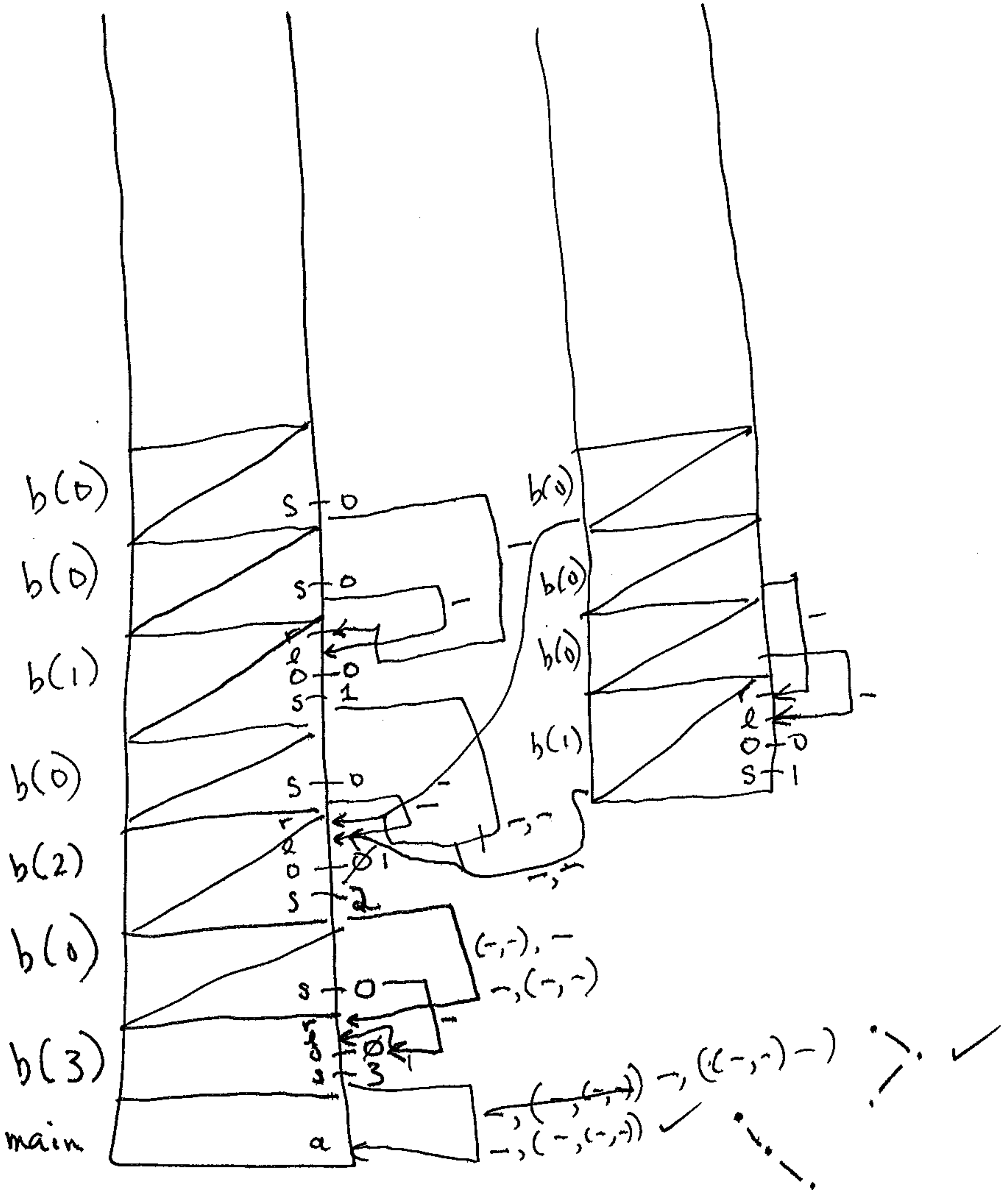
"arity"

operations
- overloaded: multiple definitions,
  distinguished by type of operands.
  number result
  - resolution: determining which definition is meant
    - typically: staticly.

types
- primitive (basic): no accessible components.
  - int, float, long, char
- structured: separately accessible components
  - array: associative or indexed.
    string number
  - record (struct)
  - disjoint unions
  - class

constant:
- like variable, but has no L-value.
- Java: final.

Trace of binGen(3)



$i \Leftarrow$ setjmp(buffer) : puts pc, sp into buffer,

longjmp (buffer, i) : restores situation that was stored,
causes setjmp to return, returning i.

use:

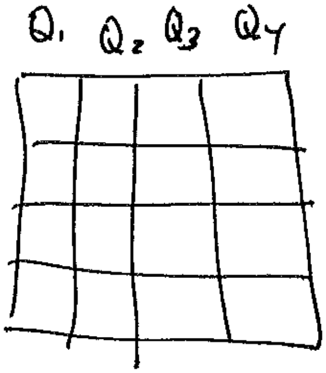
```
switch (setjmp(buf)) {

    case 0:   . . .

    case 1:   . . .
      .
      .
      .
}
```

Macros:

IterSTART, IterFOR, IterDONE,
IterYIELD, IterSUB

n - Queens

n = 4

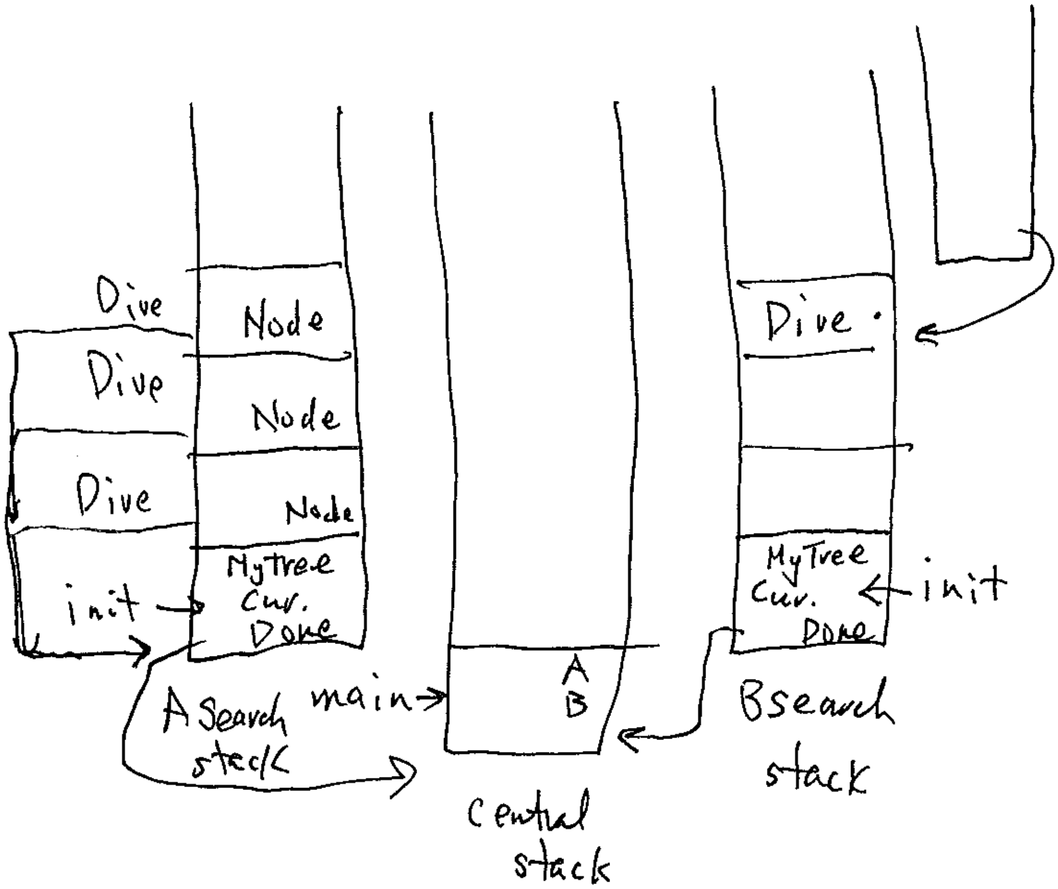


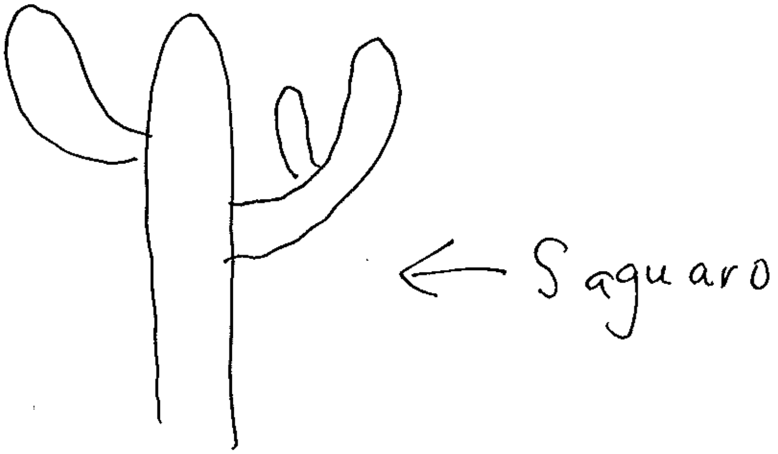
```
for (q1 = 0 .. 3) {
  if ok(q1)
  for (q2 = 0 .. 3) {
    if ok(q1 .. q2)
    for (q3 = 0 .. 3) {
      if ok(q1 .. q3)
      for (q4 = 0 .. 3) {
        if ok (q1 .. q4) then
          print(Q1 ... Q4)
      }
    }
  }
}
```

```
nest depth = 0 .. ~~3~~ n-1 {
    for q[depth] = 0 .. n-1 {
        if ok(q, depth) then
            deeper;
    }
}
do print(Q)
```

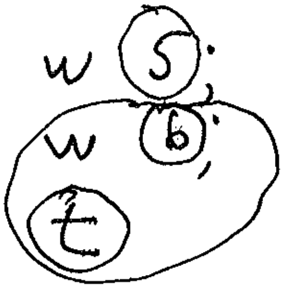
General coroutines (as in Simula67) require

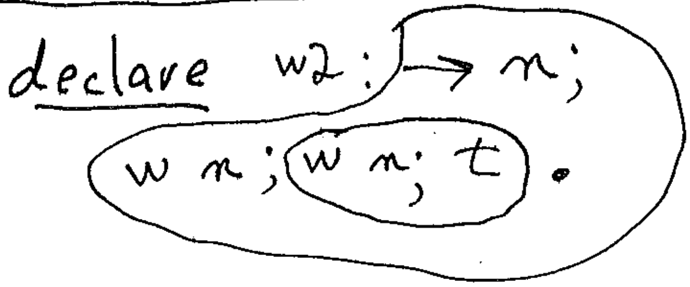
1) each coroutine has its own stack.
2) The NLRE of a coroutine can be shared among coroutines.





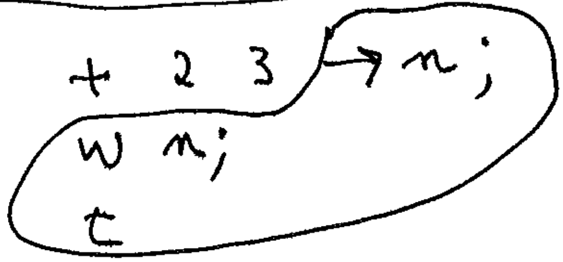
cactus stacks.

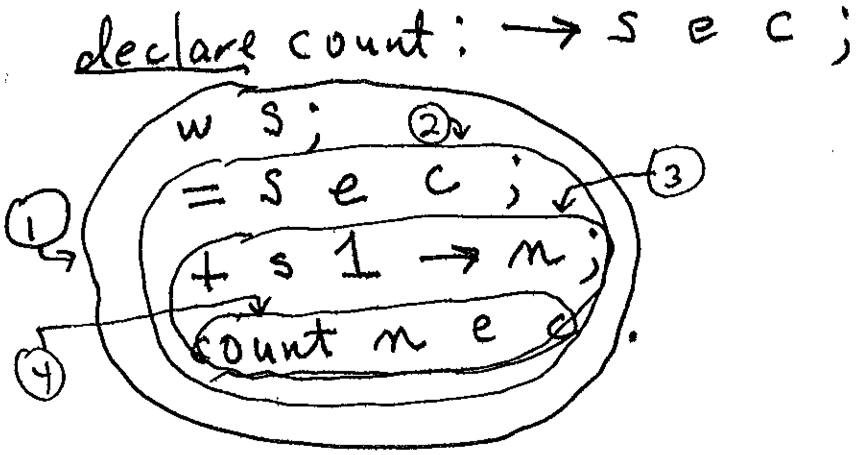




declare w2: → n c;
w n; w n; c .

w2 7;
w 9;
t





count 1 3 t .

① s = 1
e = 3
c = t

②

③

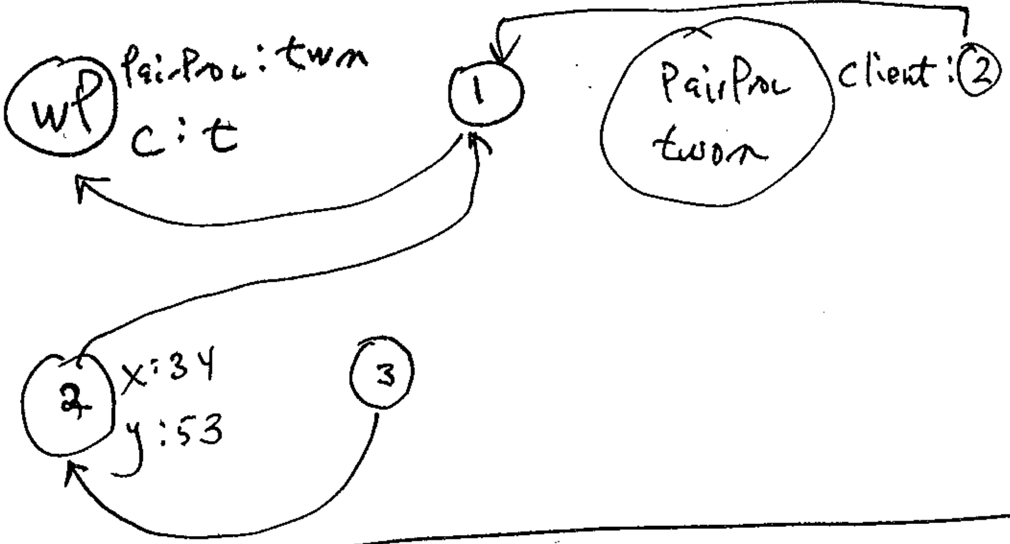
④ n = 2

① s = 2
e = 3
c = t

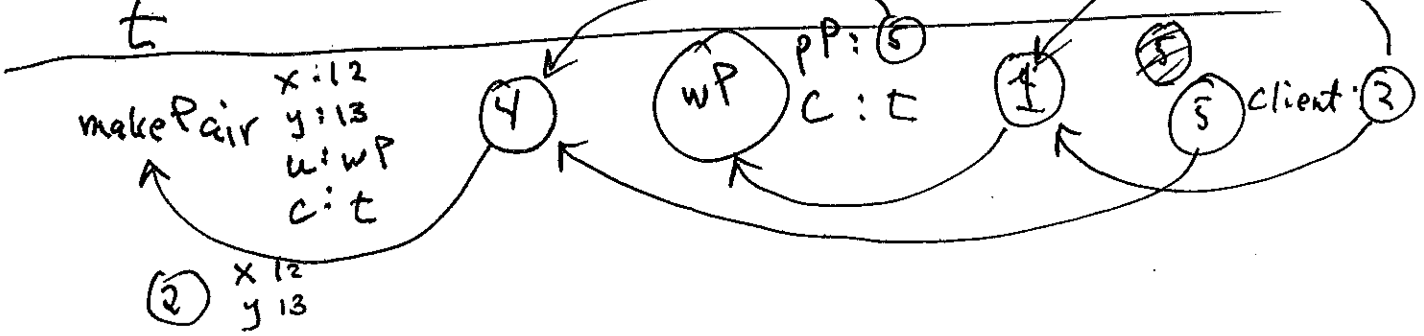
②

③

```
declare twon : → Client;
    Client 34 53 .
declare wP : → PairProc C;
    PairProc → x y;
①→ ~~write~~
    w x;      ③
    w y;          ②
    C.
wP twon;
t.
```



```
declare makePair : → x y User C ;
④ User (→ Client ; Client x y); C .
             ⑤
makePair 12 13 wP;
t
```

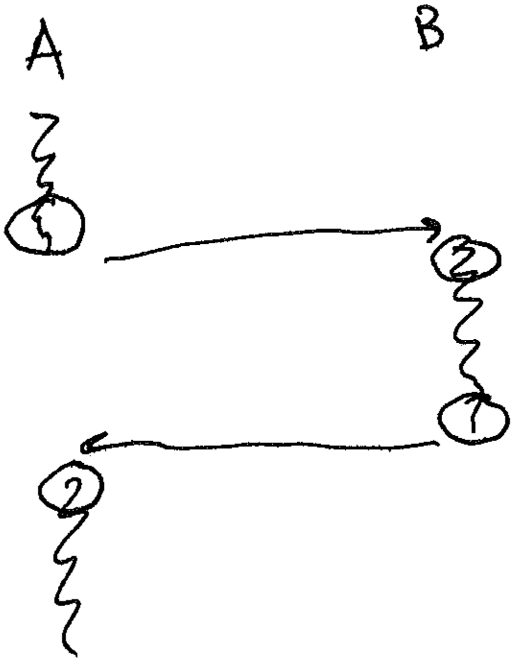


Parameters : passed to a function

formal : name inside function

actual : expression at calling point.

linkage: machine-oriented mechanism by which a call is accomplished, and the return is accomplished.



cause control to jump

pass parameters

set up B's frame

pass result back to A

reclaim B's frame

Parameter-passing modes

value

result

value result

reference

formal shares L-value of the actual.

name

all accesses to the formal directly access the actual. ~~te~~ ↓ may be array ref, or exp.

Jensen's device:

```
function accum (A, j)
    result = 0
    for j = 1, to 10 do
        result += A;
    return result;
```

~~accum(j^2, j)~~

```
accum(X*X, X)
accum (X+7, X)
```

Implementing pass by name:

each parameter is represented by 2 procedures: L-value, R-value

↓ called "thunks"

---

macro: formal is expanded as needed to the text of the actual.

$\text{foo}(\underbrace{3+}_{\text{actual}})$

```
function foo(macro t)
    return t 7;
```

Types: set of values, with operations.

property of R-values.
property of identifiers (in some lgs).

strong typing means the compiler
1) knows the type of every R-value and identifier
2) enforces type compatability
on assignment
on actual-formal binding.

compatable means
type equivalent
or convertable (coercible)
or a subtype.

type equivalent
structural: looks the same in memory.
a: int[3] b: int[3] c: float[3]
d: {z:int, w:int} e: {j:int, k:int}
~~name~~ laxity: field names
f: {a:{b:int, c:int}, d:int}
g: {a:int, b:int, c:{d:int,}}
laxity: flatten

^a ^b ^c

h: int[1..3] laxity: array bounds
j : int['a'..'c'] laxity: subscript type

To enforce:

represent each type as a string.

int	i
int[3]	[3 i
{a:int}	r a,i}
^int	p i
l:{a:int, n:^l}	r a,i, n p-5

check string equality

hash the string

Name equivalence: type constructor equivalence.

base types: int

array: each instance is new.

a,b: array [1..3]

pointerTo

enum

struct /record

derived

strict: usually allow declaring new types

type a3 = array of int[3]

var a, b : a3;

laxity (declaration equivalence):

var a,b: array of int[3]

First-class value:

can be returned from a function

can be stored in a variable

plus:

Second-class

can be passed ~~as~~ as an actual parameter.

plus:

Third-class

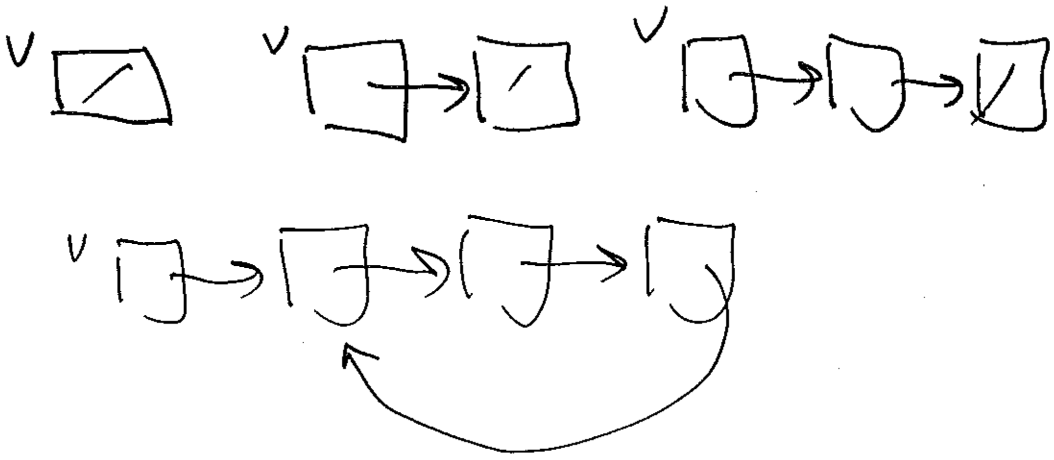
used in ordinary way.

(3, (4, 5))   ((3, 4), 5)

int * (int*int)   (int*int) * int

---

type TA = ^TA;

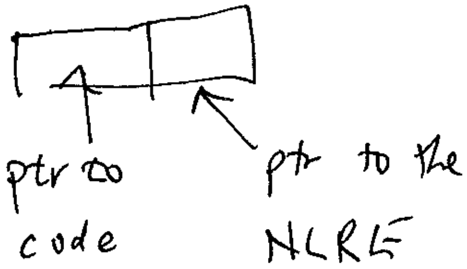


Procedures: f

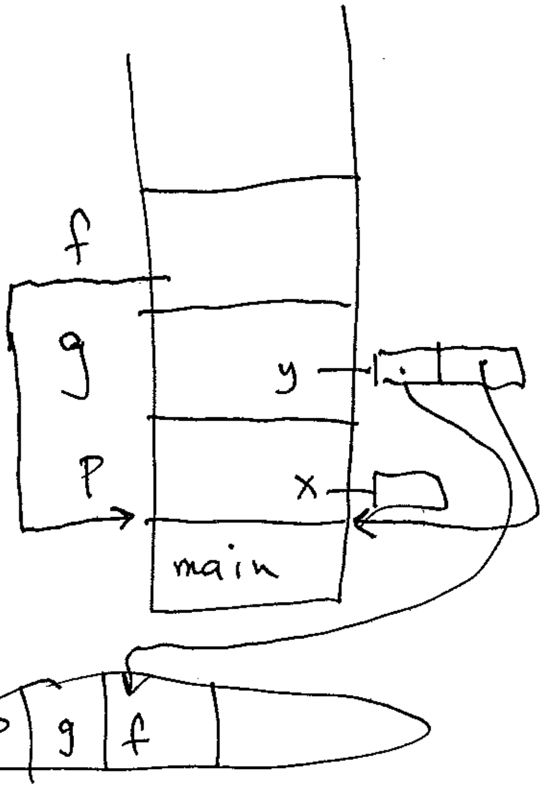
at least 3rd class

to make 2nd class

need to make its NLRE available when it is called. Therefore pass f as a <u>closure</u>.

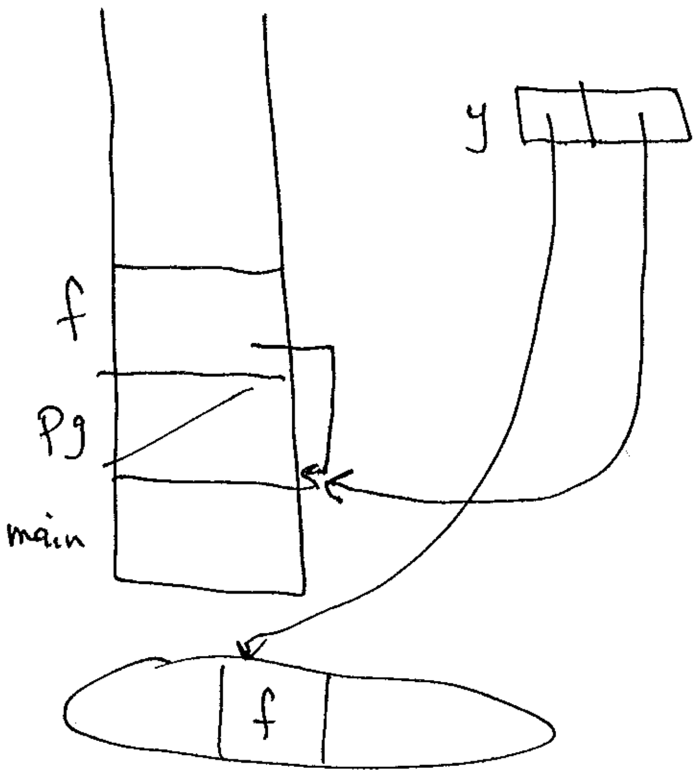


```
P [ x
    f [
      [
    g(f)
  [

g [ (y)
    y()
  [
```



to make 1st class

```
y
p [ x
    f [
      [
    y := f
    ~~g()~~
  [
g [ y()
  [
main { p()
     { g()
```



problem: dangling NLRE problem

solutions:

Pascal: procedures are 2nd class

C: no nested scopes, so no need for closures

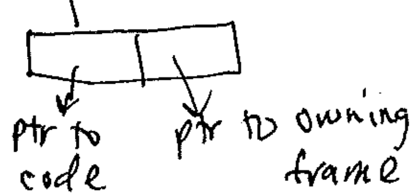
Modula-2: only top-level procedures are first-class.

Carve all frames from heap, reclaim by reference count.

What about labels?

usually 3rd class

to make 2nd class: pass as a closure, goto unwinds the stack



first-class labels?

use a heap

or disallow!

2nd class types.

Java: introspection.

every class can be converted to an instance of Class.

that instance has methods for inspecting details.

it is possible to instantiate it again.

. . .

What is polymorphism?

1) Static procedure overloading. (Java, Ada, C++ . . .)

Compiler resolves for any call.

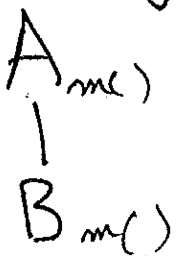
By its signature. (number, types of parameters)

↓

call's and overloaded possibilities'

2) Dynamic method binding. (overriding)

Resolution: dynamic dispatch.



var b : ~~B~~ A

b = new B()

b.m()

3) Types can include type identifiers (like 'a in ML)

4) ~~B~~ Passing a type as a parameter. (Russell, Java)

5) Generic packages (Ada), templates (C++), (Java)

the generic parameter: can be constrained

is a compile-time binding.

Alonzo Church, Burklay Rosser, Paul Kleene ㉓

Lambda calculus: origin of Lisp

Term: identifier, like x

abstraction, like $$\overbrace{(\lambda\ \underset{\uparrow \atop \text{parameter}}{x}\ .\ \underbrace{(* \ x\ 2)}_{\text{body}})}^{\text{syntax}}$$

application, like $(f\ \ x)$

Syntax: most parentheses are optional

abstraction, ~~is~~ application are left-associative

application has higher precedence.

curried functions have syntactic sugar

$$(\lambda x.\ (\lambda y.\ (\lambda z.\ T))) = (\lambda\ x\ y\ z\ .\ T)$$

In a term T, the identifier x is either <u>free</u> or <u>bound</u>

x is <u>bound</u> in $\lambda x.\ T$

x is <u>free</u> in T if:

T is x

T is $(F\ \ P)$ and x is free in F or in P

T is $(\lambda y.\ B)$ and $x \neq y$ and x is free in B

$(\lambda x. y)(\lambda x. z)$: free $\{z, y\}$ bound $\{x\}$

$(\lambda x. y)(y\ z)$: free $\{y, z\}$ bound $\{x\}$

$(\lambda y. (y\ z))$: free $\{z\}$ bound $\{y\}$

$(\lambda x. (\lambda y. z))$: free $\{z\}$ bound $\{x, y\}$

$(\lambda x. (\lambda x. z))$: free $\{z\}$ bound $\{x\}$

β reduction

$$(\lambda x. T)\ P \xRightarrow{\beta} \cancel{Z}\ \underbrace{\{\overset{\text{actual}}{P} / \overset{\text{formal}}{x}\}\ T}$$

replace all free instances of x in T by P. Read: "P for all free x in T"

$\{a/b\}\ b = a$

$\{a/b\}\ a = a$

$\{a/b\}(\lambda c. b) = (\lambda c. a)$

$\{a/b\}(\lambda b. b) = (\lambda b. b)$

$\{a/b\}(\overset{(\lambda z. z)}{\underline{(\lambda\ \underline{b}.\underline{b})}}\underline{(b\ c)}) = ((\lambda b. b)(a\ c))$

$\{(\lambda x. y)/x\}(x\ y) = (\lambda x. y)\ y$

α renaming

$$(\lambda x. T) \xRightarrow{\alpha} (\lambda y. \{y/x\} T)$$

1 $(\lambda a\, b\, c . (a\ c)(b\ c))(\lambda a . a)(\lambda a . a) \Rightarrow \alpha$

2 $(\lambda a\, b\, c . (a\ c)(b\ c))(\lambda z . z)(\lambda y . y) \Rightarrow \beta$

3 $\underbrace{(\lambda b\, c . ((\lambda z . z)\ c)(b\ c))}\ \underbrace{(\lambda y . y)} \Rightarrow \beta$

4 $(\lambda c . (\underbrace{(\lambda z . z)\ c})\underbrace{((\lambda y . y)\ c)}) \Rightarrow \beta\beta$

5 $(\lambda c .\ c\ c)$

Church-Rosser Theorem

$$\begin{tikzcd}
 & S \arrow[r] & U \\
T \arrow[ur] \arrow[dr] & & \\
 & R \arrow[uur] &
\end{tikzcd}$$

given then ∃ ↑

applicative order: run innermost β first

normal order: run outermost β first

Normal form: no β reduction is possible

$\underbrace{(\lambda x . (x\ x))}_{F}\ \underbrace{(\lambda y . (y\ y))}_{P} \overset{\beta}{\Rightarrow}$

$(\lambda y . (y\ y))(\lambda y . (y\ y))$

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

$Y g$

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

$$Y g = \boxed{(\lambda \textcircled{f}.\ (\lambda x. f(x\ x))(\lambda y. f(y\ y)))}\ \textcircled{g}$$

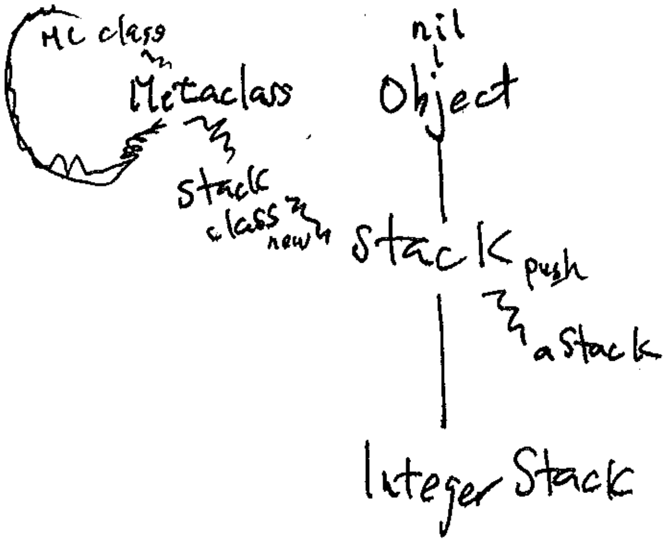
$$\xrightarrow{\beta} \underbrace{\boxed{(\lambda x.\ g(x\ x))(\lambda y. g(y\ y))}}_{①}$$

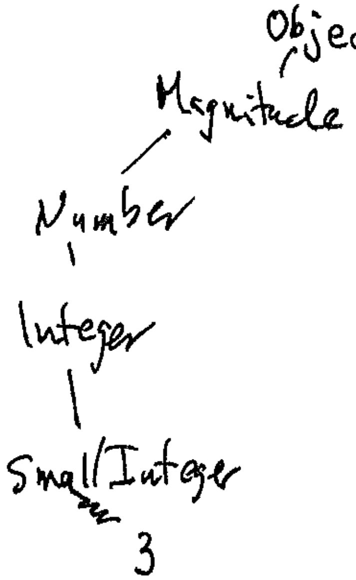
$$\xrightarrow{\beta} g\underbrace{(\ (\lambda y. g(y\ y))(\lambda y. g(y\ y)))}_{②}$$

$$= g(Y g)$$

$$(\lambda x.\ F\ x) \overset{\eta}{=} F$$

Lambda	ML
F P	F P
λx. T	fn x => T
if B T F	if B then T else F
{A/x} T	let val x = A in T end





hierachies:

— subclass-of hierarchy (superclass)

~ is-a hierarchy (class)

Intellectual history of object-oriented lgs

1) Records in Cobol (1960)
   fields are visible, variable.
   and then in C, Pascal

2) Classes in Simula (1967)
   fields can be procedures.
   NLRE is the record in which they sit.
   Subclasses inherit fields, possibly extending and overriding.
   Some control over visibility

3) Abstract data types (ADT) 1972
   CLU (clusters) Modula (Modules) Ada (Packages)
   Export a type and operations
   Clients create instances on stack or heap

4) Monitors : ADTs with concurrency control.
Guards (mutually exclusive procedures) protect program, not data.

5) Classes (Smalltalk, C++, Java)
No "type" export, rather export of variables and procedures.
Client builds instances of the class and then has access to the exported fields.

O-O : what is it?

Nomenclature : *objects* (instances of *classes*)
↘ types

communicate by sending *messages* (procedure calls) to invoke *methods*.

current state of an object is defined by the values of its *instance variables*

set of callable methods is the *protocol* of the class.

members = instance variables ∪ methods

Data Encapsulation

of an object / of its class

Can only affect state by invoking methods.

↑ inspect

Inheritance

Subclasses inherit the members of their superclass.

purpose of subclass:

1) specialize
2) reuse code

Overriding – deferred binding – dynamic binding

an ~~instance~~ subclass method overrides all accessible methods in superclasses having same signature; resolved dynamically based on instance's class.

A foo = new B()

foo.m() //uses B's version

foo.f() //uses A's version

foo.g() //compile-time error

Overloading ÷ static binding

methods withing a class overload each other if thay have the same name, different signatures

$A_{m(),\ m(\cdot)}$

Method resolution in Java

1) widening >Boxing>Varargs
Varargs
2) W, B disallowed
B, V OK
3) B, W OK (B, S = superclassing)
4) W, V vs B, V mutually exclusive
5) no widening of wrapper classes

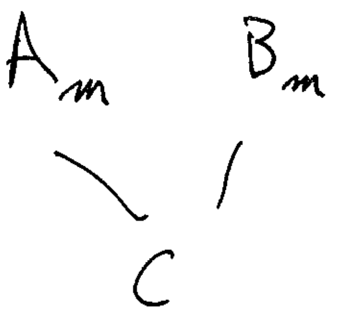
Methods	Invocation	Resolution Called	Rule
f(Integer i), f(long i)	f(5)	long	1
f(int... i), f(Integer i)	f(5)	Integer	1
f(Long l), f(int... i)	~~int i~~ f(5)	int...	1, 2
f(Long l), f(Integer ...i)	f(5)	Integer... i	1, 2
f(Object o), f(Long l)	f(5)	Object o	2, 3
f(Object o), f(int ... i)	f(5)	Object o	1, 3
f(Object o), f(long l)	f(5)	long l	1
f(long... l), f(Integer...i)	f(5)	error.	4
f(long...l), f(Integer i)	f(5)	Integer	1
f(Long l)	Integer i; f(i)	error	5
f(Long l), f(long... l)	Integer i; f(i)	long...	1, 5

lg	mode	other instance same class	related	inherited?	instance of other class
Smalltalk	variable	n	—	y	n
	method	y	—	y	y
C++/Java	public	y	y	y	y
	protected	y	y	y	n
	private	y	y/n	n	n
Java	package-private	y	y	y*	n
Eiffel	(default)	y	y	y	y
	specified	y	y	y	n
	none	n	n	y	n

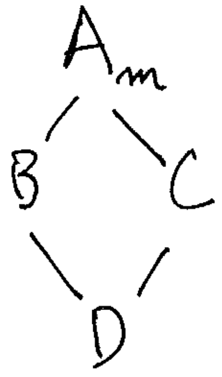
Unusual features of Java

interfaces
  provides a contract
  specifying methods,
    variable declarations
  class may <u>implement</u> a list
    of interfaces.
effectively homoiconic
  serialization
    instance ⟷ string
~~intro~~
introspection
  class ⟷ instance of Class

Multiple inheritance



C foo = <u>new</u> C();
X foo.m()
B.foo.m()



d.m()

A class is very like a type.

Smalltalk: a class is a value

Values are created on heap (typically)

Methods can be dynamically inserted into classes (Smalltalk) with immediate effect on all objects in the class (and ~~subcla~~ subclasses)

Duck Typing (Python, Smalltalk)(JavaScript)

If an object satisfies a protocol, you can treat it as a member of the class that has that protocol. A little like structural equivalence.

C# formal parameter can have type "dynamic"

go disallows overloading.

Concurrent Programming

Basic idea:

multiple simultaneous threads of execution.

- how to specify (start, stop)

how they can communicate with each other

how to implement threads (storage, schedulin

exclusion of critical regions. (synchronization)

To specify

fork (~~proc.~~ procedure name) → cookie

↖ opaque reference

~~f~~ join(cookie)

↖ waits for the thread to terminate.

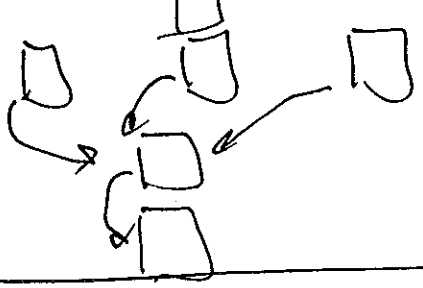
---

cobegin

⋮

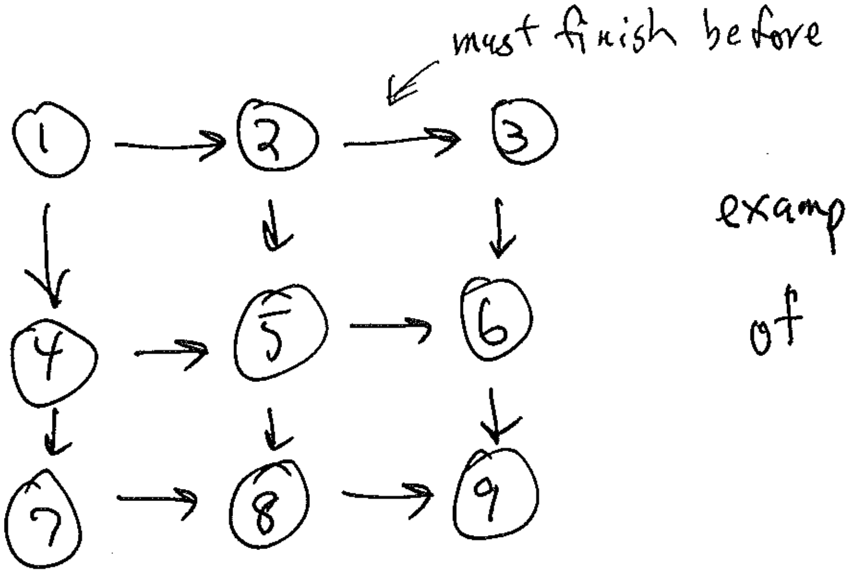
coend

leads to cactus stacks



Modula: call a process, same syntax as a procedure.

Go: call any procedure with keyword go.



example of inadequacy of cobegin

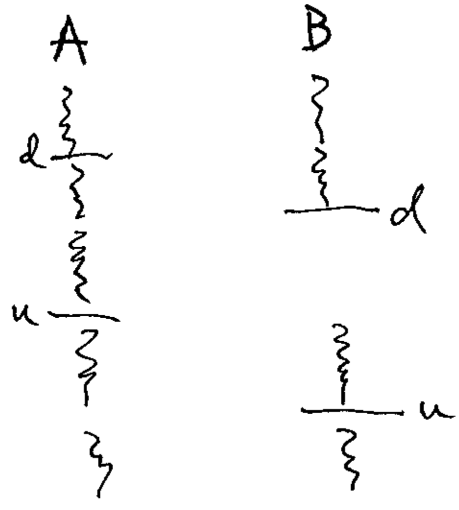
Basic synchronization method: <u>semaphores</u>.
Counter, list of waiting threads.

up ( ): increments counter, c
unblocks first waiting thread if $c \geq 0$.

down( )
decrements c
blocks caller if $c < 0$.

standard use:
initialize c to 1.
critical region: section of code that must run in exclusion.

down( )
R
up( )



Java:

```
    synchronize(){


    }
```

Conditional critical regions

Monitors (C.A.R. Hoare) (Tony)

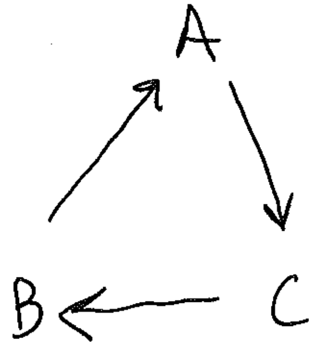
abstract date type accessed by mutually exclusive guard ~~proc~~ procedures that can block on condition variables.

standard examples

- Bounded buffer
- Readers + writers : broadcast (signalAll) introduced
- Dining philosophers

Standard difficulties

- Deadlock (circular wait)
- Starvation (some threads make progress, at least one never does)
- Livelock (no thread makes progress, but they all continually do work)



Livelock example
(CSP)

Formal semantics: what does a program <u>mean</u>?

Idea comes from formal syntax

BNF (1960)

Attributed grammars to even cover type correctness.

Three approches

Axiomatic (Hoare 1967)

Operational

Denotational

Axiomatic

place <u>assertions</u> around code fragments.

provide <u>axioms</u> allowing you to prove correctness of assertions.

notation: ~~{P} Q {R}~~ "if P holds before executing S, then condition Q holds after, if S terminates"

precondition → $\{P\}\ \underset{\substack{\uparrow \\ \text{code}}}{S}\ \{Q\}$ ← post-condition

assertions

Weak and strong predicates

if $P \Rightarrow Q$ we say P is <u>stronger</u> than Q.
Q is <u>weaker</u> than P.

weakest predicate : true
strongest predicate : false

A: $\{\underset{\substack{\uparrow\\ \text{pre}}}{P}\}\ S\ \{\underset{\substack{\uparrow\\ \text{post}}}{Q}\}$

<u>strengthen</u> the precondition has the effect of <u>weakening</u> ~~the~~ A

say $R \Rightarrow P$

$\{P\}\ S\ \{Q\} \Rightarrow \{R\}\ S\ \{Q\}$

Axioms try to be stated in strongest way:

weakest precondition
strongest postcondition.

$\{y = 12\}\ x := \underbrace{y+2}_{E}\ \{\underbrace{x = 14}_{Q}\}$

Axiom of assignment

$\{Q_{x \to E}\}\ x := E\ \{Q\}$

$\{y+2 = 14\}$
$\{y = 12\}$

Axiom for conditionals

$$\frac{\{B \wedge P\}\, S_1\, \{Q\}, \quad \{\neg B \wedge P\}\, S_2\, \{Q\}}{\{P\}\ \underline{\text{if}}\ B\ \underline{\text{then}}\ S_1\ \underline{\text{else}}\ S_2\ \{Q\}}$$

Axiom of iteration

$$\frac{\{B \wedge I\}\, S\, \{I\}}{\{I\}\ \underline{\text{while}}\ B\ \underline{\text{do}}\ S\ \{\neg B \wedge I\}}$$

↑ loop invariant

```
{true}
{1 = 1!}
count := 1;
{1 = count!}
answer := 1;
I = {answer = count!}
while count != n do
    {answer = count!}
    count := count + 1;
    {answer = (count-1)!}
    answer := answer * count;
    {answer = count!}
end;
{answer = count!, count = n}
{answer = n!}
```

Axiomatic semantics

- can prove small programs correct.
- complex control structures are hard to model.
  - break, concurrency
- designing correct pre- and post- conditions for code is as hard as writing correct code.
- does not prove termination
- led to a fad of proving programs correct to teaching students to explicitly show loop invariants.

Denotational (Scott-Strachey) semantics

Components

- Abstract syntax
- Semantic domains : mathematical sets representing values
- Semantic functions : take syntax and yield values in semantic domains.

Language 1: Binary literals.

Abstract syntax:

$BN \in BinLit$

$BN \rightarrow Seq$

$Seq \rightarrow 0 \mid 1 \mid Seq\ 0 \mid Seq\ 1$

Semantic domain

$N = \{0, 1, 2, \ldots\}$

Semantic function

$E : BinLit \longrightarrow N$

$E[0] = 0$

$E[1] = 1$

$E[Seq\ 0] = 2 \cdot E[Seq]$

$E[Seq\ 1] = 1 + 2 \cdot E[Seq]$

Language 2: Simple expressions

Abstract syntax

$T \in Exp$

$T \rightarrow T + T$

$T \rightarrow T - T$

$T \rightarrow T * T$

$T \rightarrow Seq$

Semantic domain:

$N = \{0, 1, 2, \ldots, -1, -2, -3 \ldots\}$

Semantic function $E : Exp \to N$

$$E[Seq] = \text{old}E[Seq]$$

$$E[T_1 + T_2] = E[T_1] + E[T_2]$$

similarly for $-$, $*$

Language 3: range checks, error conditions.

Abstract syntax

$$T \to T \,/\, T$$

Function

“bottom”
“error”

range: $N \to$ ~~{minInt~~

$$\{\text{minInt} \ldots \text{maxInt}\} \oplus \{\bot\}$$

Semantic domain

$$R = N \oplus \{\bot\}$$

Semantic function $E : Exp \to R$

$$E[0] = 0$$

$$E[1] = 1$$

$$E[Seq\ 0] = E[Seq] \overset{\text{is a member of}}{?} N \Rightarrow \text{range}(2 \cdot E[Seq]),\ \bot$$

$$\underline{\text{if}}\ E[Seq] ? N\ \underline{\text{then}}\ \text{range}(2 \cdot E[Seq])\ \underline{\text{else}}\ \bot$$

$E[Seq\ 1]$ is similar

$$E[T_1 + T_2] = \text{if } E[T_1] ? N \wedge E[T_2] ? N$$
$$\underline{\text{then}}\ \text{range}(E[T_1] + E[T_2])$$
$$\underline{\text{else}}\ \bot$$

$-$, $*$ are similar

$$E[T_1 / T_2] = \underline{\text{if}}\ E[T_1] ? N \wedge E[T_2] ? N \wedge E[T_2] \neq 0$$
$$\underline{\text{then}}\ E[T_1] / E[T_2]$$
$$\text{else}\ \bot$$

Language 4: initialized constants

Abstract syntax

$P \in Pr$ (program)

$T \in Exp$ (expression)

$I \in Id$ (identifier)

$Def \in Decls$ (declaration)

[BNF]

Backus-Naur Form

$P \to Def \;\; T$

$Def \to \varepsilon \mid I = T \mid Def \; Def$

$T \to$ (all previous ones) $\mid I \mid T = T$

Semantic Domains

$R = N \oplus Bool \oplus \{\bot\}$ (result)

$V = N \oplus Bool \oplus \{udef, \bot\}$ (lookup value)

$U = Id \to V$ ($\overset{\text{universe}}{\text{environment}}$)

function from Id to V

Semantic functions

$E : Exp \to U \to R$

$D : Decls \to U \to U$

$M : Pr \to R$ (meaning of program)

$M[Def \;\; T] = E[T]u$ where $u = D[Def]u_0$

$u_0[I] = udef$

$$D[\varepsilon]u = u$$

$$D[Def_1\ Def_2]u = D[Def_2](D[Def_1]u) \qquad f(I)=e$$

$$D[I = T]u = \underline{\text{let}}\ e = E[T]u\ \underline{\text{in}}\ u\underset{\substack{\uparrow\\ \text{upgrade}}}{[}I \leftarrow \overset{\downarrow}{e}]$$

except in bad cases.

$e = \bot$: put e in u anyway

$!(u(I)\ \cancel{\#}?\ \{udef\})$ (already declared)

could allow overriding declaration.

could return u (new declaration has no effect)

✓ could return $u[I \leftarrow \bot]$

$$E[0]u = 0$$

similarly for many possible T values.

$E[I]u = u(I)$ except $\underline{\text{if}}$ $u(I)=udef$ $\underline{\text{then}}$ return $\bot$

$E[T_1 = T_2]u = E[T_1]u == E[T_2]u$

except: $\underline{\text{if}}$ $E[T_1]u = \bot$ $\underline{\text{or}}$ $E[T_2]u = \bot$

$\underline{\text{return}}$ $\bot$.

$\underline{\text{if}}$ $E[T_1]u?N \wedge\ !E[T_2]u?N$ $\underline{\text{or}}$

$E[T_1]u?Bool \wedge\ !E[T_2]u?Bool$

$\underline{\text{return}}$ $\bot$.

Language 5: Variables, statements

Abstract syntax

⋮

$St \in Stm$ (statement)

$P \to$ <u>program</u> (I) Def St <u>end</u>

Def $\to$ as before

$\to I$: integer;

$\to I$: bool;

$St \to \varepsilon$ (empty)

$St \to St\ St$ (list)

$St \to I := T$

Semantic domains

R, (as before) $= N \oplus Bool \oplus \{\bot\}$

$V = N \oplus Bool \oplus \{\bot, udef, redef\}$ (lookup value)

$\mathcal{U} = Id \to V \otimes \{var, const, uninit\}$

Semantic functions

$$\text{as before}\left\{\begin{array}{l} E : Exp \to \mathcal{U} \to R \\ D : Decls \to \mathcal{U} \to \mathcal{U} \\ M : Pr \to R \end{array}\right.$$

$S : Stm \to \mathcal{U} \to \mathcal{U} \oplus \{\bot\}$

$$u_0[I] = \langle udef, \underset{\uparrow\ \text{arbitrary}}{var} \rangle$$

```
M[program (I) Def St end] =
      let  u = D[Def]u₀            | except if ! c?u then ⊥
           c = S[St]u              |
      in                           |
           E[I]c                   |
      end                          |

D[Def₁ Def₂]u    as before
                                                     | except
D[I = T]u =  let  e = E[T]u                          | if ! u[I] =
                  f = <e, const>                     |    <udef, _>
             in   u[I ← f]                           |      .
             end                                     |      .
                                                     |      .
D[I: integer]u =  u[I ← <0, uninit>]
      except if  u[I] = <_, uninit>
                        <_, const>  ...
D[I: bool]u = u[I ← <false, uninit>]

E[I]u =  let  u[I] = <e, Kind>   | except
         in   e                  |  if Kind =
         end                     |  uninit
                                 |  then ⊥
```

(46)

$S[\varepsilon] u = u$

$S[St_1\ St_2] u =$ let $w = S[St_1] u$
in
$S[St_2] w$
end

except
if $w = \bot$
then $\bot$

$S[I := T] u = u[I \leftarrow \langle e, var \rangle]$

where ~~$\langle e, var \rangle = u[T]$~~ $E[T] u = \langle e, \_ \rangle$

except . . .

Language 6: conditionals, limited iterations

~~Sem~~ Abstract syntax

$St \rightarrow$ if T then St else St

$St \rightarrow$ do T times St

Semantic domains: as before

Semantic functions: as before, with:

$S[$if T then St_1 else $St_2] u =$

let ~~b~~ $E[T] u = \langle b, \_ \rangle$
in if b then $S[St_1] u$ else $S[St_2] u$
end;

$S[$do T times $St] u =$

let $E[T] u = \langle e, \_ \rangle$;
$v_0 = u$; $v_{i+1} = S[St] v_i$
in ~~v_e end;~~ $v_{\max(0,e)}$

Language 7 : include while

Abstract syntax :

$$St \rightarrow \underline{while}\ B\ \underline{do}\ St$$

Semantic domains : as before

Semantic functions : as before, with :

$S[\underline{while}\ B\ \underline{do}\ St]u =$

~~$\underline{let}\ v = S[st]u\ \underline{in}$~~

~~$p_{i+1}(u$ $\underline{if}\ v = \bot\ \underline{then}\ \bot\ \underline{else}$~~

~~$p_{i+1}(u) =$~~

~~$\underline{if}\ v = \bot\ \underline{then}\ \bot\ \underline{else}\ p_i(v)$~~

⋮

$p_0 = \bot$

$p_{i+1}u = \underline{let}$
$e = E[B]u$
$v = S[st]u$
$\underline{in}$
$\underline{if}\ e$
$\underline{then}\ p_i\ v$
$\underline{else}\ u$
$\underline{end}$
$\underline{end}$

$S[\underline{while}\ B\ \underline{do}\ ST] = \lim\limits_{i\to\infty} p_i\ u$