Hare Krsna Hare Rama

Date 31 07 2017

Paradigms of Programming

Course Outline

Imperative : A set of statements / commands / instructions each of which changes the current state of system.

Eg: C, Pascal, Fortran, Basic, Assembly, Machine Code

C → Recursion, type-safe

Fortran → Scientific (w/o recursion), Fast HLL

D Why C?

01-08-2017 Imperative paradigm has the concept of time. At any time, the machine is in some state S_t. Any instruction changes the state.

$$S_{t-1} \xrightarrow{I_t} S_t$$

C language was first used to write the UNIX OS. It was useful because it helps to optimize code which is very essential while developing OS.

Eg: while $(i++ >> 2) > (j++ << 2)$.

2 times slower than Assembly.

Object orientation

Classes introduce data types. Data & their associated functions are clubbed together to form "objects".

Eg: C++, Objective C, Simula, Java.

Pure Object Oriented : Only objects Eg messages b/w them. Eg. Java, Objective C

Functional

No notion of time or state $O = f(I)$ (output) (input)

Typically recursion. Eg. LISP (developed in 1959) : List Processing, Python, Haskell, Matlab

5-10 times slower than C.

Logic Programming

Specify a set of rules. Eg. PROLOG.
2 times slower than functional

Imperative Paradigm

Assignment statement :-

$x \leftarrow e$ → expression.

x → variable, ← → assignment

int x; → variable declaration

A data type defines the storage of variables of that type and the associated operations that can be performed on data of this type.

In built data type : int, long, short, float, double, char

User defined data type : class, complex{float r, float i}

Eg-
```
complex operator + (complex a, complex b)
{
    complex c;
    c.r = a.r + b.r;
    c.i = a.i + b.i;
    return c;
}
```

complex a = {1.0, 2.0}, b = {1.5, 2.5}
complex c = a+b; polymorphism helps it to detect to perform +(a,b)

Alternative Statement

if (boolean expression) → { statements }.

General: if B1 → S1
B2 → S2
⋮ ⋮
Bn → Sn.
fi // conclusion

Loops / Iterative Statement

do
B1 → S1
B2 → S2
⋮ ⋮
Bn → Sn
od.

for i ← 1 to 10 step 2. do{S}
↳ used for a fixed number of iterations for a given set of values of an iterator.

while:
while B
do S

→ flag = 0
repeat
if(flag=0 && ¬B)
break;
else flag = 1
S; S
while(¬B)
S;

Repeat_until:
repeat S
until B

→ until B
S;

7-08-2017
Monday

We cannot write a program to generate another program. Because the no. of options available at each step are many (exponential) indefinitely.

Functions

Global declarations (type declarations & variable declarations)

```
function 1 ( ) { }
function 2 ( ) { }
    :
main ( ) { }
```

Scope: int x = 2;

```
f1 ( ){
    int x = 3;
    print(x);      // prints 3    nested scope
}
```

Pascal:

```
Global declaration {

f1 ( ) {
nested func.  square ( ) {
              }
    square (2);
}.
```

```
square ( )
{ square ( ); }.
}
```

Recursion :-

(i) f1 → f1

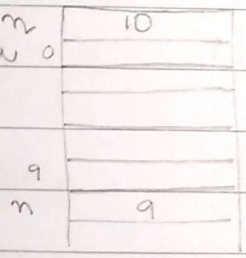
(ii) f1 → f2 → f1

(iii) f1 → f2 → f3 → f1

In recursion, at a given point of time, only one version of the function is active. In FORTRAN at compile time only, the fns, global & local variables are statically allocated. Hence it cannot support recursion - Very fast execution.

Recursion in C is implemented using a "call stack".

2 types of memory in C: heap memory & stack memory
call stack malloc → → fn variables.



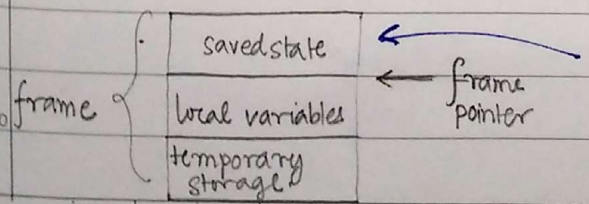

```
int fact (int n) {
    int a[10];
    if (n==0) return 1;
    else return (n * fact (n-1));
}
```

// returning a here would give error because activation record is popped from stack once fn returns

If we want to return array we could do:

```
int * fact (int n) {
    int *a = malloc( );
    return a;
}
```

Activation record.



KOKUYO

No.

Date

S -> M|U
S -> aTbU
M ->

8-08-2017
Tuesday

Dynamic Scope

```
int x = 1;
f1(){
    int x = 2;
    f2();
}
f2(){
    print(x);        // prints 2.
}
main(){
    f1();
}
```

Call Sequence:

main	
f1	int x=2
f2	print(x)

Early implementation of LISP had dynamic scope.

Static / Lexical scope

can be determined at compile time.

Most modern languages (including C) follow static scope.

In previous code, it would've printed 1.

Function parameter passing mechanism

Call-by-value

```
f1(int x, float y)
{
    x = 2;
    y = 3.0f;
}
main(){ int x = 1; float y = 2.0f;
    f1(x,y); print(x,y);    // prints 1,2.0
}
```

Call by refrence

prints 2, 3.0.

```
f1(int &x, int &y)
```

No.

Date

Call by value-result — Copy while calling & returning.
Code prints 2, 30.
Slower than call by reference.

Call by name. (Textual substitution). - Typically used for macros

```
Eg - swap (int x, int y){
        int temp = x;
        x = y;
        y = temp;}
main
swap (int x, int y){
        int x = 1, y = 2;
        swap (w, z);        // w acquires value of x, z → y. Then swapped
        print (w, z);       // prints 2,1
}
```

Functions

- Multiple occurrences of code segments can be put into a function
- Modularity
- Libraries
- Recursion

C program :-

```
Global declarations {

f1(){
}
f2(){
}
```

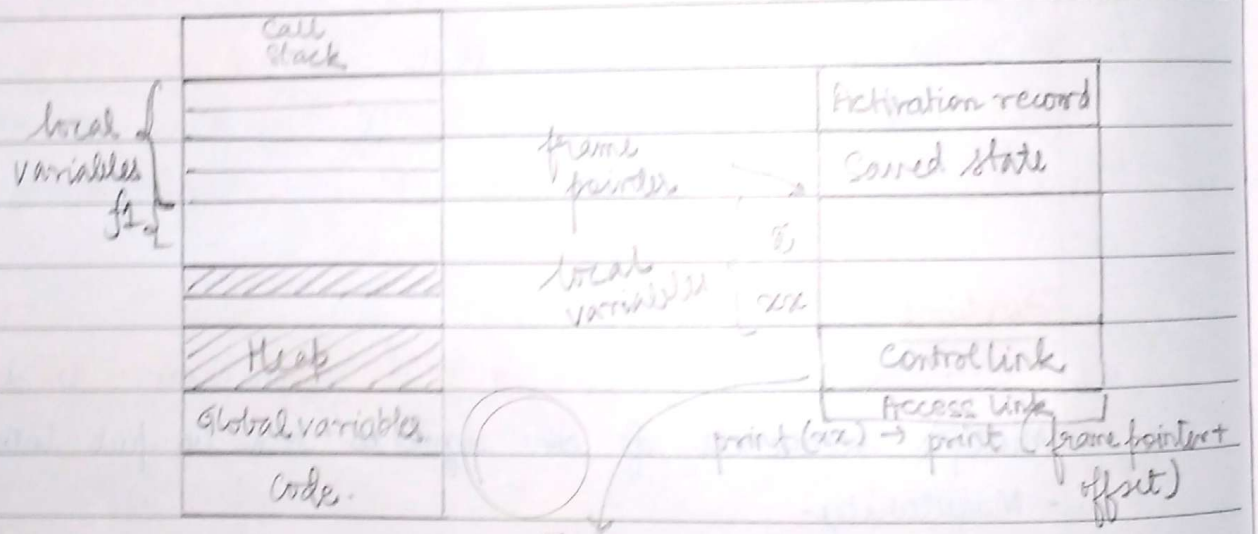
```
int x = 1;
f1() {
    int x = 2;
    {
        int x = 3;
        print(x);
    }
}
```

int y = 3;

⇔ print(y)

C has only 2 levels of scope.
Nested scope is created only in function declaration.

C memory allocation

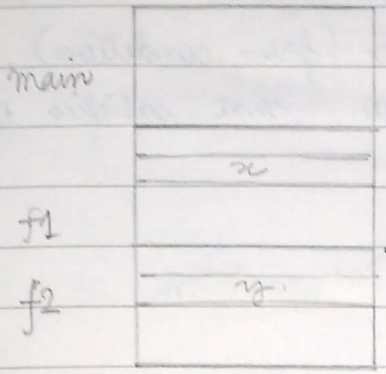


Tells about call sequence. Tells about the last call. Also called dynamic link

PASCAL :

```
function f1()
{
    int x = 2;
    function f2()
    {
        int y = 3;
        print(x, y);
    }
    f2();
}
```

→ frame pointer + fixed offset



Say another fn inside f1 is declared as follows :-

```
f3( ) {
    int x = 4;
    f2( );
}
```

same access link

& we call f3 inside f1 .

Main → f1 → f3 → f2 Dynamic (control) links

Main → f1 → f3 → f2 Access (static) links

14-08-2017
Monday

Analyzing Imperative Programs

Eg 1.

Precondition: $\{ MININT \leq x * x \leq MAXINT \}$ x**x = x*x,

$y \leftarrow x * x$.

Postcondition : $\{ (y = x * x) \wedge (MININT \leq y \leq MAXINT) \}$

Eg 2. swap

Precondition : $\{x = X, y = Y\}$

$\{x = X, y = Y\}$ $t \leftarrow x;$

$\{y = Y, t = X\}$ $x \leftarrow y;$

$\{x = Y, t = X\}$ $y \leftarrow t;$

Postcondition : $\{x = Y, y = X\}$

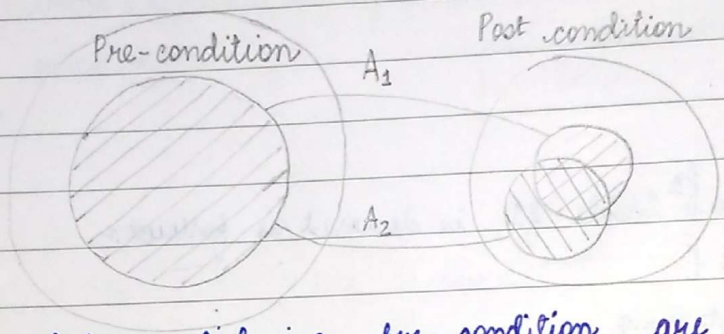
work up the preconditions in reverse order

KOKUYO

No.

Date

Program correctness : If a program starts in a state satisfying the initial condition (pre-condition), then the program terminates in a state that satisfies the post-condition.



Even if states satisfying pre-condition are mapped to a subset of post condition, it's a valid program. Eg. MST
The program should work for a condition weaker than pre-condition & condition stronger or equal to post condition.

Weakest Pre-condition (wp) : $wp(S, R) = P$

Weakest pre-condition such that execution of S in any of the states satisfying S will result in state satisfying R.

$$wp(x \in e, R) = \text{domain}(e) \text{ cand } R_e^x$$

(cand: conditional and; R_e^x: replace x with e in R.)

$$Q \Rightarrow wp(S, R)$$

$$wp(\text{"}x \leftarrow e\text{"}, R) = \text{domain}(e) \text{ cand } R_e^x$$

KOKUYO

Conditional Statements.

IF:

if → guard.

$B_1 \rightarrow S_1$ → Statement

$B_2 \rightarrow S_2$

$\vdots$

$B_n \rightarrow S_n$

fi

$wp(IF, R) =$ domain (BB) cand. BB $\wedge$

$B_1 \Rightarrow wp(S_1, R) \wedge$

$B_2 \Rightarrow wp(S_2, R) \wedge$ $\quad BB \Rightarrow B_1 \vee B_2 \vee \ldots B_n.$

$\vdots$

$B_n \Rightarrow wp(S_n, R)$

$\{P\}$ → Loop invariant

DO:

do

$B_1 \rightarrow S_1$

$B_2 \rightarrow S_2$

$\vdots$

$B_n \rightarrow S_n$

od.

$\{R\}$

(or)

do

BB →

if $B_1 \rightarrow S_1$

$B_2 \rightarrow S_2$

$\vdots$

$B_n \rightarrow S_n$

fi

od.

Find a predicate P called the loop invariant that satisfies ① P is true before the loop.

② $\{P \wedge B_i\}\ S_i\ \{P\} \quad \forall\ 1 \le i \le n$

(or) $P \wedge B_i \Rightarrow wp(S_i, P)$

③ $P \wedge \neg BB \Rightarrow R$

④ $P \wedge BB \Rightarrow t > 0$

⑤ $\{P \wedge B_i\}\ t1 = t\ ;\ S_i\ \{t < t1\} \quad \forall\ 1 \le i \le n$

Date

Eg: Given array $b[1...n]$, $n \geq 0$

$i \leftarrow 0$ $s \leftarrow 0$

$\{ P: 0 \leq i \leq n, \quad S = (\Sigma k : b[k], 1 \leq k \leq i) \}$ $t : n - i$

do

$i < n \longrightarrow$ ~~skip~~ $i \neq n$, $i \leftarrow i+1$

$S \leftarrow S + b[i]$

$\{P\}$

od

$\{ R: S = (\Sigma k : b[k] ; 1 \leq k \leq n)$

$\neg BB : i = n \Rightarrow BB \Rightarrow i \neq n$

bound function : t (integer valued) - upper bound on the # of iterations still to be done.

① P satisfied before loop

② $(P \wedge \neg BB) S \rightarrow (P)$

$S = \Sigma k : b[k]$ $1 \leq k \leq i$

$i \neq n$ $i \leftarrow i+1$

$S = \Sigma k : b[k]$ $1 \leq k \leq i+1$

$0 \leq i \leq n$ ✓ P is satisfied

③ $P \wedge \neg BB \Rightarrow S = \Sigma k : b[k]$ $1 \leq k \leq n$

$\Rightarrow R$

④ $P \wedge BB \Rightarrow t = n - i > 0$

⑤ $(P \wedge BB)\ S\ t = n - (i+1) < t$

$t = n - i$ $i = i+1$; True

1

$wp(S \leftarrow S + b[i], (S = \Sigma k : b[k] \quad 1 \leq k \leq i))$

$wp(i \leftarrow i+1, (S + b[i] = \Sigma k : b[k] \quad 1 \leq k \leq i))$

$= (S + b[i+1] = \Sigma k : b[k] \quad 1 \leq k \leq i+1)$

Final program :

$i \leftarrow 0$; $s \leftarrow 0$

do

$i \neq n \rightarrow i \leftarrow i+1$;

$S = S + b[i]$;

od

$x \leftarrow a$

$y \leftarrow b$

$z \leftarrow 0$ $\{P: z + x*y = a*b\}$ $\{t = y\}$

do $((y > 0) \wedge even(y)) \rightarrow$ $y \leftarrow y/2$; (Right shift)

$x \leftarrow x + x$; (Left shift)

$odd(y) \rightarrow$ $y \leftarrow y - 1$;

$z \leftarrow z + x$;

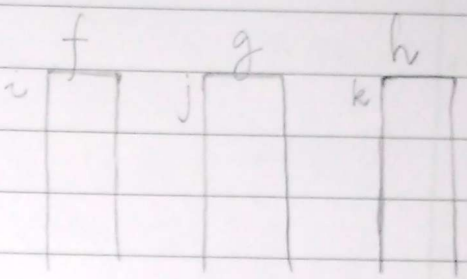
od

KOKUYO

{R: Z = a*b}

Welfare Crook.

Given: 3 sorted lists



Problem: Find the first common entry in the 3 lists.

$i \leftarrow 0,\ j \leftarrow 0,\ k \leftarrow 0$

do

$f[i] < g[j] \vee f[i] < h[k]$ $\quad i \leftarrow i+1;$

$g[j] < f[i] \vee g[j] < h[k]$ $\quad j \leftarrow j+1;$

$h[k] < f[i] \vee h[k] \vee g[j]$ $\quad k \leftarrow k+1;$

od

$\{R: f[i] = g[j] = h[k]\}$

$P \wedge \neg BB = R$

↳ vague.

$f(i) \geq g(j) \wedge f(i) \geq h(k)$

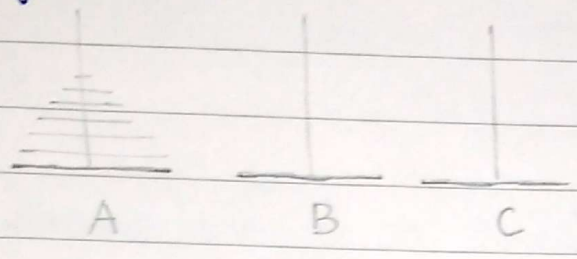
$\wedge\ g(j) \geq f[i] \wedge g[i] \geq h(k)$

$\wedge\ h[k] \geq f[i] \wedge h[k] \geq g[j]$

KOKUYO

Recursive vs Iterative

Tower of Hanoi




```
ToH(n, A, B, C)  : Move n disks from A to C using B as spare.
{
    TOH(n-1, A, C, B);
    move a disk from A to C;
    TOH(n-1, B, A, C);
}
```

Recursive is tough in this case. (It uses stack through)

Mergesort

```
mergeSort (list)
{
    mid ← size(list)/2;
    leftlist ← sublist( list[:mid] );
    rightlist ← sublist( list[mid+1:] );
    merge (mergeSort(leftlist), mergeSort(rightlist));
}
```

Iterative :

```
set s ← split the list into n singleton lists
while ( size(s) > 1 ) {
        pick any two lists from s & merge. Put it back into s.
}
```

Note: Set works because ordering doesn't matter of merge

No.

Date

Most efficient merge: always pick the 2 smallest sets & merge

P: all lists in s are sorted, union of all lists = original list

t: size(s).

Quicksort

```
quicksort (b[], i, j) {
    if (j-i+1 > 1) {
        k ← partition (b, i, j);
        quicksort (b, i, k-1);
        quicksort (b, k+1, j);
    }
}
```

Iterative:

$s \leftarrow \{(1, n)\}$

P: Those elements not included in any interval are correctly placed.

t: sum of interval lengths

```
while (size(S) > 0)
{
    pull out any interval (i, j) ∈ s;
    k ← partition (b, i, j);
    put back (i, k-1) into s : S = S ∪ {(i, k-1), (k+1, j)}
}
```

Developing loops

Initialization statements

{P}

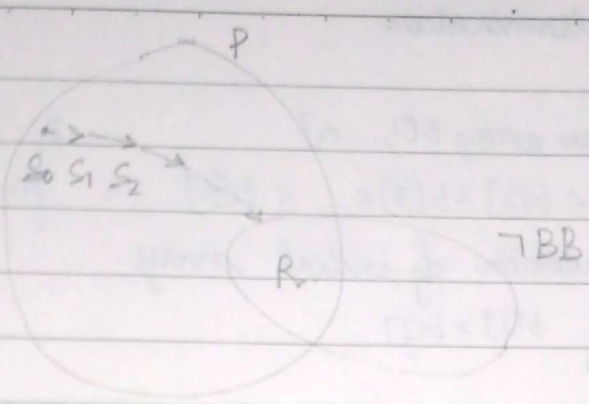
do

$B_1 \rightarrow S$

od

{R}

KOKUYO



P is obtained from R by weakening R.

Eg: R: $z = a * b$

By introducing variables, P: $z + x * y = a * b. \quad x, y \geq 0$

① Develop the guard first (By looking at which condition P reduces to R).

Eg-1

do

$y \neq 0 \rightarrow z \leftarrow z + x$
$\quad\quad\quad\quad\quad\quad y \leftarrow y - 1$

$y > 0 \wedge y \in \text{even} \quad$ or $y \leftarrow y^2/2$

or.

wp $(y \leftarrow y - 1, \; z + xy = ab)$.

$= z + x(y-1) = ab.$

$= z + xy - x = ab.$

wp $(z \leftarrow z + x, \; z - x + xy = ab)$.

$= z + xy = ab.$

wp $(y \leftarrow y/2, \; z + xy = ab)$

$= z + x(y/2) = ab.$

wp $(x \leftarrow x + x, \; z + x(y/2) = ab)$

$= z + xy = ab \; \wedge \; (xy \text{ is even} \wedge y > 0)$

$\quad\quad P \quad\quad\quad\quad\quad\quad B_i$

KOKUYO

(2) Write guarded statements to Make progress towards termination

Eg 2. Bubble Sort

Precondition Q: Given an array $b[1,\ldots,n]$ P:

Post condition R: $b[1] \le b[2] \le b[3] \le \ldots \le b[n]$ $\wedge$ final array is permutation of initial array.

Bound function: $\sum_{\substack{(i,j) \\ i<j}} b[i] > b[j]$

Start by making progress towards termination.

$b[i] > b[j] \longrightarrow$ swap $(b[i], b[j])$.