CORE EXAMINATION with Solutions May 1994

Department of Computer Science

New York University

May 20, 1994

Programming Languages & Compilers

PL&C1. Consider a procedure `swap(a,b)` that interchanges the values of the objects passed as its actual parameters `a` and `b`.

(a) Write the pre-conditions and post-conditions for `swap`.
(b) Condsider a language that allows arbitrary expressions in array subscripts. The following implementation of swap was written in this language.

```
proc swap(a: int, b: int);
      var t: int;
      t := a; a:=b; b:=t;
end swap
```

Suppose that all parameters in this language are "in-out" — that is, the procedure may read the initial value of the actual parameter and changes to the formals are visible at the call site. There are at least four binding classes that might support this:

(1) NAME (as in Algol 60)
(2) REFERENCE (as in Pascal **var**)
(3) COPY-IN-COPY-OUT with the actual's L-values computed once per call (as in Ada)
(4) COPY-IN-COPY-OUT, with the actual's L-values computed twice per call.

Which of the binding classes will produce a correct implementation of `swap`? Explain your answer briefly.

Solution: (a) For `swap(x, y)` the precondition is {x=a, y=b} and the postconditions are {x=b, y=a}. The comma is an "and" (&) condition.

(b) The binding classes (2) and (3) work correctly, but binding classes (1) NAME and (4) COPY-IN-COPY-OUT with actual's L-values computed twice per call, work incorrectly.

In NAME binding, suppose we have an array of two integers A[0..1], then:

```
A[0] := 1;  A[2] := 2;  i := 0;
swap(i, A[i]);
```

produces the following results: `i = 1, A[0] = 1, A[0] = 1`, while the precondition

```
{i = 0, A[0] = 1}
```

is expected to lead to the postconditions

```
      {i = 1; A[0] = 0 }
```

Clearly, `A[0]` is wrong for NAME binding.

In the case of COPY-IN-COPY-OUT, with the actual's L-values computed twice per call, the same example produces the results `i = 1, A[0] = 1, A[1] = 0`. The reason is that when the L-values are computed for the second time (at the time of the COPY-OUT), `i` gets the value 1 and `A[i]` (now equal to `A[1]`) gets the value 0.

Here's another example for the failure of swap using COPY-IN-COPY-OUT, with actual's L-values computed twice. Consider

```
      func succ(n: int): int;
            n := (n+1) mod 2;
            succ := n;
      end succ;

      A[0] := 1; A[1] := 2; i := 0; j := 1;
      swap(A[succ(i)], A[succ(j)]);
```

This results in `A[0] = 1` and `A[1] = 2`, because each time the L-values are computed, the function `succ` changes the values of the indices `i` and `j` as a side effect.

PL&C2. The following Common Lisp program `REVERSE` will reverse the top level of a list:

```
      (DEFUN REVERSE (X) (REVERSE1 X NIL))
      (DEFUN REVERSE1 (REST RESULT)
            (COND ((NULL REST) RESULT)
                  (T (REVERSE1 (CDR REST) (CONS (CAR REST) RESULT)))))
```

Thus we have the following:

```
(REVERSE '(A B (C (D E) F) G (H I))) ==>  ((H I) G (C (D E) F) B A)
```

(This is actually a built in function.)

You are to define a function `REVERSE-ALL` that will reverse a list at all levels. For example, we would like to have

```
(REVERSE-ALL '(A B (C (D E) F) G (H I))) ==>  ((I H) G (F (E D) C) B A)
```

Without using the procedure `REVERSE` in your program, write the function `REVERSE-ALL`.

Solution: The program `REVERSE-ALL` below takes a list and reverses the list at all levels, making use of a helper program, called `REVERSE2`. The program `REVERSE2` takes as its two arguments two lists, `REST` and `RESULT`, and returns a list made by reversing the list `REST` at all levels and then concatenating it to the list `RESULT`. Hence if the list `RESULT` is empty `(NIL)`, then `REVERSE2` returns the list `REST` reversed at all levels.

```
      (DEFUN REVERSE-ALL (X) (REVERSE2 X NIL))
      (DEFUN REVERSE2 (REST RESULT)
            (COND ((NULL REST) RESULT)
                  ((ATOM (CAR RESULT)
                        (REVERSE2 (CDR REST) (CONS (CAR REST) RESULT)))
                  (T (REVERSE2 (CDR REST)
                              (CONS (REVERSE-ALL (CAR REST)) RESULT)))))
```

An incorrect solution, because it uses the function `REVERSE`, allows a much easier definition of `REVERSE-ALL`:

```
      (DEFUN REVERSE-ALL (X)
            (COND ((NULL X) NIL)
                  ((ATOM X) X)
                  (T (REVERSE (MAPCAR 'REVERSE-ALL X)))))
```

PL&C3. Code Generation.
Given the following Pascal-like program

```
      program P;
            procedure one(a: integer);
                  var b: integer;
                  function four(h: integer; i: integer): integer;
                  var j: integer;
             begin {four}
                  j := 23;
                  return (b * i) + h + j;
             end {four};
             function two( c: integer): integer;
                  function three(e: integer; f: integer): integer;
                       var g: integer;
                  begin {three}
                       g := e + f -b;
                       return four(g,e);
                  end; {three}
             begin {two}
                  return three(c, c+1);
             end; {two}
        begin {one}
             b := 4;
             writeln(two(a-b));
        end {one}
   begin {program P}
        one(12);
   end.
```

A compiler for the (fictional) Courant Systems 4000 processor will generate the following assembly code for the function `four`:

```
PUSH FP                   ; save dynamic link
MOV FP, SP                ; update frame pointer
SUB SP 4                  ; make room for j
MOV [FP-4],23             ; j <- 23
MOV R2, [FP+8]            ; R2 <- static link
SUB R2, 4                 ; R2 <- address of b
MOV R1, [R2]              ; R1 <- b
MUL R1, [FP+16]           ; R1 <- R1 * i
ADD R1, [FP+12]           ; R1 <- R1 + h
ADD R1, [FP-4]            ; R1 <- R1 + j
ADD SP,4                  ; remove j
POP FP                    ; restore FP to point to previous frame
```

NB: A bug in the assembly code for function `four` appeared in the exam as given on May 20. An extra instruction MOV R2, [R2] appeared in the code, resulting in one extra level of indirection following the static link. There was no penalty assessed for answers in which a similar extra level of indirection occurred.

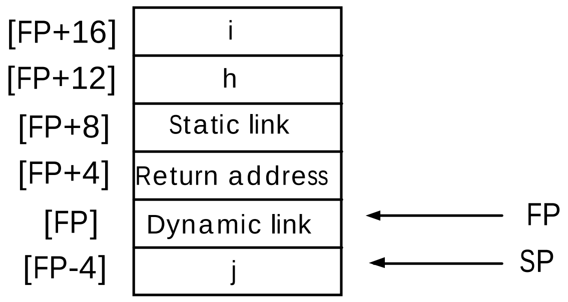
The instructions, registers, and addressing modes for the Courant Systems 4000 are described below.

(a) Draw the stack frame (activation record) for function `four` when it is executing. Show where each parameter, the local variable `j`, the static and dynamic links, and the return address are located in the frame. Also show what **FP** and **SP** would contain while function four is executing.

(b) Describe the calling convention (i.e., how procedures are called, parameters passed, and values returned) that this compiler uses.

(c) Write the assembly code that the compiler might generate for function `three`.

Solutions: (a)



(b) Procedures are passed by pushing the actual parameters in reverse order, pushing the static link, and then using the CALL instruction (which pushes the return address) to enter the procedure. On return from the procedure, the return value is left in register R1. The caller must remove the actual parameters from the stack upon return.

(c) Here's assembly code for `three`:

```
PUSH FP           ; save dynamic link
MOV FP, SP        ; update frame pointer
SUB SP, 4         ; make room for g
MOV R1, [FP+12]   ; R1 <- e
ADD R1, [FP+16]   ; R1 <- R1 + f
MOV R2, [FP+8]    ; R2 <- static link, points to two's frame
ADD R2, 8         ; R2 now points to two's static link
MOV R2, [R2]      ; R2 now points to one's frame
SUB R2, 4         ; R2 now points to b
SUB R1, [R2]      ; R1 <- R1 - b
MOV [FP-4], R1    ; g <- R1
;; Getting ready to call four(g,e)
PUSH [FP+12]      ; push e
PUSH [FP-4]       ; push g
;; Now need to compute four's static link, i.e., pointer to one's frame
MOV R2, [FP+6]    ; R2 <- static link, points to two's frame
ADD R2, [R2]      ; R2 now points to two's static link
PUSH R2           ; push four's static link
CALL four         ; call four, leave return value in R1
ADD SP, 8         ; remove the two arguments to four
ADD SP, 4         ; restore FP to point to previous frame
RET
```

The Courant Systems 4000

- Registers — all 4 bytes
  - `R1, R2, R3, R4` General registers (used for arithmetic, indexing, etc.)
  - `SP`: Stack pointer
  - `FP`: Frame pointer
  - Other registers....
- Addressing modes
  - `imm`: immediate (e.g., 10)
  - `reg`: contents of register (could be any register listed above)
  - `[reg]`: indirect addressing through register
  - `[imm]`: direct memory addressing (i.e., location at address `imm`)
  - `[FP+i], [FP-i]`: stack access via offset from `FP`.
- Relevant Instructions
  - `MOV` *x,y* : writes the contents of *y* to *x*
  - `PUSH` *x* : pushes *x* onto the stack (i.e., `SP` is decremented by 4 and then *x* is written to `[SP]`)
  - `CALL` *l* : push return address and jump to label *l*
  - `ADD` *x,y* : *x* is replaced by *x* + *y* (where *x* and *y* can be registers, memory locations, or immediates). Other arithmetic instructions include `SUB, MUL, DIV`, etc.
  - `POP` *x* : pop value off of stack into *x*. Here *x* can be a register or memory location.
  - `RET`: return from procedure, i.e., pop return address off stack and jump to that address.

Operating Systems

OS1. (a) [6 pts] Describe a logical data-structure suitable for representing a file system with a hierarchical directory structure on a hard disk. You should assume that the disk is separated into three areas of fixed size:

- An area containing housekeeping information;
- An area containing data segments;
- An area containing i-nodes.

(b) [2 pts] How can such a system be efficient for large and small files?
(c) [2 pts] How can files and directories have multiple parent directories?

Solution: (a) Every file and directory is assigned an i-node, which is identified by a unique number, and contains the following information:

- The length of the data portion of the file or directory;
- Access bits to control permissions;
- Addresses of the first few data segments;
- If required, a pointer to a block of indirect pointers to data segments, where the block occupies a data segment;
- If required, a pointer to a block of indirect pointers to blocks of indirect pointers to data segments.

For a file, the fixed-length data segments contain the data for the file. For a directory, the data contains a list of inode numbers for files and directories contained in the directory, together with the associated names for those files and directories. The housekeeping area contains a pointer to the root directory inode, and pointers to a linked list of free segments in the data area, and information about available inodes.

(b). For files, the data segment addresses consist of a fixed number of directly addressed segments, an indirect tree of segments, and a double indirect tree of segments. (A triple indirect tree of segments may also be included if the file system is to be able to handle very large files.) The scheme is efficient since at most two or three pointers must be tracked to access any segment containing file data. (Incidentally, the nodes of the tree of pointers to data segments are **not** inodes; the inode is the structure located in the inode area containing the data named in section (a) above.)

(c). The inode does not contain a parent inode number. However, inodes for a file contain a count of the number of directories that contain the file. A directory contains the file if the inode number of the file is listed among its list of files and directories that are contained in the directory. When a file is removed from a directory, the count is decremented, and the file is removed when the count goes to zero.

OS2. The message-passing operations

```
send(process* pto, byte* msg)
receive(process* pfrom, byte* msg)
```

between up to *n* processings, are to be implemented using just semaphores for synchronization. You may assume that the communication is synchronous (i.e., a sender `pfrom` issuing the operation `send(pto, msg)` will block until a receiver `pto` issues the operation `receive(pfrom, msg)` and vice-versa).

(a) [3 points] What information must be stored in the process table for each process?
(b) [1 point] How many times is each message copied?
(c) [2 points] How many semaphores are needed for *n* processes?
(d) [4 points] Give the algorithms for send and receive.

Solution: (a) In addition to the normal information in the process table, the process table must contain the following fields in order to support the communications:

```
process*    recver;     receiver process that is waited for
process*    sender;     sender being waited for
semaphore   sem;        semaphore of the process for waiting;
byte*       msgbuf;     address of the buf for blocked operation
```

(b) Each message is copied once; no extra space other than space in processes for messages; buffering is not required. Indeed, by passing pointers to shared memory, messages do not have to be copied at all.
(c) $n+1$ are required (see below).
(d) When a process sends a message to another, one of the two will execute the send or receive first, and whichever goes first must wait for the other procedure to be invoked. A global semaphore is required in order to determine which process goes first, and each process requires a semaphore in case it must wait for the partner in communication. In the code below, we use nonzero values in the sender and receiver fields in the process tables to determine whether the send or receive has preceeded the partner. We also assume that each process has a variable `me` that is a pointer to the process table.

```
send(pto, msg)
{     P(mutex);
      if (pto->sender == me) {
            copy(msg, pto->msgbuf);
            pto->sender = 0;
            V(mutex);
            V(pto.sem); }
      else {
            me->recver = pto;
            V(mutex)
            P(me.sem); }
}
receive(pfrom, msg)
{     P(mutex);
      if (pfrom->recver == me) {
            copy(pfrom->msgbuf, msg);
            pfrom->recver = 0;
            V(mutex);
            V(pfrom.sem); }
      else {
            me->sender = pfrom;
            V(mutex);
            P(me.sem); }
}
```

ALGORITHMS

Algs1. [**12 points**] Let $G=(V,E)$ be a directed acyclic graph, where a number *numb*(*v*) is stored at each vertex *v*. Our problem is to find the maximum *numb*(*v*) over all *v*.

We say that *w* is reachable from *v* if there is a directed path from *v* to *w* in *G*. In the procedures below, `v.numb` is the value of *numb*(*v*), and `v.max` is the maximum value discovered so far among nodes that are reachable from *v*. There is also a field `v.unseen,` which can be either true or false. The function `edges(v)` gives the set of all vertices that are adjacent to *v* by a directed edge from *v* (which might be empty). Two students have submitted the following programs, Driver and Zriver:

```
Global V, E;
Procedure Driver
      foreach vertex v in V do
            v.unseen := True
      foreach vertex v in V do
            if v.unseen then
                  Mmax(v) endif
Procedure Mmax( var T: vertex)
      T.unseen := False;
      T.max := T.numb;
      foreach x in edges(T) do
            begin
            Mmax(x);
            if T.max < x.max then
                  T.max := x.max endif
            end;
```

```
Global V, E;
Procedure Zriver
      foreach vertex v in V do
      v.unseen := True
      foreach vertex v in V do
            if v.unseen then
                  Zmax(v) endif
Procedure Zmax( var T: vertex)
      T.unseen := False;
      T.max := T.numb;
      foreach x in edges(T) do
            begin
            if x.unseen then Zmax(x) endif;
            if T.max < x.max then
                  T.max := x.max endif
            end;
```

Answer these questions, **under the assumption that *G* is a tree.**
a. Explain why each of the two procedures halts.
b. State if they compute the desired result.
c. Compare the asymptotic runtimes for the two procedures.

Solutions (a)-(c): (a) Halting of the procedure call Mmax(*v*) is immediate by induction on the height of *v*. The height of *v* is the length of the longest path from *v*. This is clear if *v* is a sink (height=0). If *v* is not a sink, then each recursive call is to a node of height less than *v*. By induction, these recursive calls will halt. Then Mmax(*v*) halts, as desired.

Exactly the same argument applies to Zmax(*v*).

(b) Both compute the correct result: that is, Mmax(*v*) and Zmax(*v*) will finally end with `v.max` equal to the largest value of `u.numb`, where `u` ranges over all nodes that are reachable from `v`. Again, we can do this by induction on the height. The base case when v is a sink is immediate. Althogh Mmax(*v*) and Zmax(*v*) have different bodies, it is immediate that v.max in both cases will be equal to the maximum of

v.numb, u.max

where u ranges over all nodes in edges(v). Since u.max is correctly computed (inductively), v.max will be correct.

(c) Since *G* is a tree, $|E| = \Theta(|V|)$, so we just focus on the parameter $n = |V|$. Mmax(*v*) takes $O(n)$ time since it spends $O(1)$ per node *u* reachable from *v*. It follows that Driver takes $O(n^2)$ time. This bound is the best possible, as seen for the case *G* is a linear graph (1→2→3→...→n) and Driver visits the nodes in the order: Mmax(*n*), Mmax(*n*-1), ... , Mmax(1).

Similarly, Zmax(*v*) takes $O(n)$ time. However, Zriver takes $O(n)$ time overall if we use a more careful analysis: Zmax(v) takes $O(1+h)$ time where *h* is the number of reachable nodes that are still unseen.

Answer the following questions, **under the assumption that *G* is a DAG with vertices $\{1, \cdots n\}$ and edges *E* containing every edge from *i* to *j* for $i < j$.**
d. Explain why each of the two procedures halts.
e. State of they compute the desired result.
f. Compare the asymptotic runtimes for the two procedures.

Solutions (d)-(f):
(d) Both procedures halt for exactly the same reason as before (height is well-defined for DAGS).
(e) Both procedures give the correct answer for the same reason as before.
(f) Zriver still takes $O(n)$ time, using the same argument as before.

In contrast, we show that Driver can be exponential**.**
Let $T(i)$ be the time taken for Mmax(*i*) where $i = 1, \cdots, n$. Clearly $T(i) = O(1) + T(1) + T(2) + \cdots T(i-1)$.
This is easily expanded and shown to be

$$T(i) = O(2^i).$$

Hence Driver takes time

$$T(1) + T(2) + \cdots + T(n) = O(2^n).$$

This bound is tight.

Algs2. [**10 points**] Consider the problem of sorting a set of *n* numbers, where $n / \log n$ are reals, and the remaining $n - {}^{n}\!/_{\log n}$ numbers are integers in the range $[0, n^3 - 1]$. For each of the following proposed algorithms, state:

1. Is it correct? (I.e., does it sort the *n* numbers?)
2. What is its running time?

Justify your answers briefly. Standard results may be assumed. In addition, you may assume that each number can be identified as a real or an integer in constant time and that a real and an integer can be compared in constant time.

Method A.
1. Radix sort the integers.
2. Quicksort the reals.
3. Merge the sorted sets resulting from steps 1 and 2.

Method B.
1. Radix sort the reals.
2. Quicksort the integers.
3. Merge the sorted sets resulting from steps 1 and 2.

Method C.
1. Mergesort the reals.
2. Radix sort the integers.
3. Merge the sorted sets resulting from Steps 1 and 2.

Method D.
1. Radix sort the integers.
2. Concatenate the reals to the end of the sorted set of integers obtained in Step 1.
3. Insertion sort the set created in Step 2.

Method E.
1. Radix sort the integers.
2. Concatenate the reals to the start of the sorted set of integers obtained in Step 1.
3. Insertion sort the set created in Step 2.

Solution.

Facts: On *n* elements, Quicksort takes $O(n \log n)$ on average, $O(n^2)$ worst case; Mergesort takes $O(n \log n)$ worst case. Merging two sorted lists of total size *n* takes $O(n)$. Insertion sort works from front to back, and on *n* items takes $O(n^2)$.

To radix sort the integers, we use base *n*. Then the number of digits is 3, so total time is $O(n)$. (Use of base 10 or 2 would take time $O(n \ln n)$, which is less efficient. Use of base $b > n$ takes time $O(b)$ since have to look at *b* bins so less efficient.)

All methods work except B; you can't radix sort on reals without a predetermined bound on the number of digits. To Quicksort (average) or Mergesort $n / \log n$ reals takes time $O\left({}^{n}\!/_{\ln n} \ln\left({}^{n}\!/_{\ln n}\right)\right) = O\left({}^{n}\!/_{\ln n} \cdot \ln n\right) = O(n)$.

To Quicksort (worst time) $n / \log n$ reals would take $O\left({}^{n^2}\!/_{\ln^2 n}\right)$.

Method A. Step (1) $O(n)$ (average), Step (2) $O(n)$, Step (3.) $O(n)$. Total: $O(n)$. If we instead consider worst case, then we have Step (1) $O\left({}^{n^2}\!/_{\ln^2 n}\right)$, Step (2) $O(n)$, Step (3) $O(n)$, Total: $O\left({}^{n^2}\!/_{\ln^2 n}\right)$

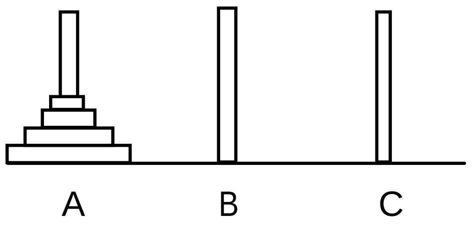
Method B. Doesn't work.

Method C. Same as average case for A, but worst case is the same as average case for this method..

Method D. Step (1) is $O(n)$, and the concatenation of the reals in step (2) takes time $O(n/\ln n)$. The insertion sort begins with the integers in the front, which are already sorted. It takes only time $O(n)$ to check that they are in correct order. The reals each take time $O(n)$ to be inserted, but there are only $n/\ln n$ of them so total time is $O\left(n^2/\ln n\right)$.

Method E. Steps (1) and (2) of Method E take the same time as Method D, namely $O(n)$ total. The insertion sort begins at the front with the reals, which are sorted in time $O\left(n^2/\ln^2 n\right)$. Now for a real subtlety. In inserting the already sorted integers, each integer will only be compared with at most one other integer, the one it to its left, since the new integer will necessarily be bigger, and so that will end the insertion. Since each integer will be compared to at most $n/\ln n$ reals (and one integer), the total time to insert all n integers is $O\left(n^2/\ln n\right)$. This dominates the total time. If you disregard the subtlety (the fact that each successive integer is compared with no more than one integer during the insertion sort), then the apparent complexity of the algorithm is $O(n^2)$.

Algs3. [8 points] Recall the Tower of Hanoi problem. In this problem, you are given three pegs, named A, B, and C, and a collection of n disks, of radii 1, 2, ⋯ n, respectively. Initially, the disks are piled on peg A, in order of decreasing radius, with the largest on the bottom and successively smaller disks immediately on top.



The task is to move all of the disks from peg A to peg B. There are two contraints:

- Only the disk at the top of a peg may be moved, and when it is moved, it must be moved to the top of the stack of another peg;
- When moved, a disk may be placed either on an empty peg, or on a disk with a larger radius, so at no time is a larger disk on top of a smaller disk.

a. [3 pts] Write a recursive procedure that lists the sequence of moves performed in moving the *n* disks from peg A to peg B. HINT: One way of doing this is to give the pseudo-code for a procedure Move(x,y,z,k) which outputs the moves needed to move k disks from peg x to peg y, where peg z is the third peg.

b. [3 pts] Write, but do not solve, a recurrence equation for the running time of your procedure from part a.

c. [2 pts] How many times does your procedure move the disk of radius *n* (i.e., the largest disk). HINT: Consider the case *n* = 1.

Solution: (a). Let Put(x,y) be the primitive operation of moving the top disk of Peg x to Peg y. Then:

```
Procedure Move(x,y,z,k)
If k=1 then
    Put(x,y);
Otherwise
    Move(x,z,y,k-1);
    Put(x,y);(or Move(x,y,z,1))
    Move(z,y,x,k-1);
End;
```

(b). Let $T(k)$ be the time for Move(x,y,z,k). Then $T(1)=O(1)$. The procedure uses $O(1)$ time plus two recursive calls to Move(-,-,-,k-1), so the recurrence is $T(k)=2T(k-1)+O(1)$. If we count peg moves then, more precisely, $T(1)=1$ and $T(k)=2T(k-1)+1$.

(c). The largest peg is not moved in the recursive calls, but only with the Put(x,y) in the middle. Thus, for any k, the largest peg is moved precisely once.