Problems and Solutions to the January 1994 Core Examination

Programming Languages

PL 1. Consider the following program written in an Algol-like language:

```
begin
      integer i,j;
      integer array A[0..10];

      Procedure swap(x,  y);  Integer x,y;  begin
            integer t;
            t := y;
            y := x;
            x := t;
      end

      for i := 0 step 1 until 10 do begin
            A[i} := 2 + i;
      end

      j := 3;
      swap( j , A[j] )

      print j;

      for i := 0 step 1 until 10 do begin
            print i, A[i];
      end

end.
```

What is printed if parameters are passed using call-by-value?
What is printed if parameters are passed using call-by-reference?
What is printed if parameters are passed using call-by-name?

Solutions:

Pass by value	Pass by reference	Pass by name
3	5	5
0 2	0 2	0 2
1 3	1 3	1 3
2 4	2 4	2 4
3 5	3 3	3 3
4 6	4 6	4 6
5 7	5 7	5 7
6 8	6 8	6 8
7 9	7 9	7 9
8 10	8 10	8 10
10 12	10 12	10 12
11 13	11 13	11 13

PL 2. Write a scheme (or LISP) function called POS that accepts a list of integers as input and produces as output the same list with negative elements replaced by zero. For example,

```
(POS '(0 4 -7 -9 9))  ==> (0 4 0 0 9)
```

You may assume that all the elements are integers; in particular, there are no sublists. Recall that scheme has a predicate called `positive?` that returns `#t` if given a positive integer and **`nil`** for zero or negative integers. That is,

```
(positive? 5) ==> #t
(positive? 0) ==> nil
(positive? -2) ==> nil
```

Solution: Here are two solutions:

```
(define (POS lis)
  (if (null? lis)
      nil
      (if (positive? (car lis))
      (cons (car lis)   (pos (cdr lis)))
      (cons 0           (pos (cdr lis))))))


(define (POS lis)
    (cond
        ( (null? lis)           nil)
        ( (positive? (car lis)) (cons (car lis)   (pos (cdr lis))))
        ( else                  (cons 0           (pos (cdr lis))))))
```

Note that from a programming language standpoint, a LISP function operating on a list must use **cons**, **car**, and **cdr**, and should not process the list using a loop.

Operating Systems

Each question has ONE or MORE correct answers. You are required to find ALL the correct answers. The grading scheme is 1 point for each correct answer and –0.5 (minus half) for each wrong answer that you write down.

Solution: ***Correct answers are shown with check marks ✔.***

1. Non-preemptive CPU schedulers have the following shortcomings:

A. Chances of process starvation increases.
B. The system is more prone to deadlocks.
✔C. An erroneous infinite loop in a user process can crash the system.
D. Priority scheduling schemes are impossible to implement.
✔E. CPU bound processes cause response time to degrade.

2. A process can be in one of three states, namely waiting, running and ready. Which of the following statements about uniprocessor operating system are true? (Assume round robin scheduling.)

✔A. Several processes can be in their waiting states.
✔B. Several processes can be in their ready states.
C. Several processes can be in their running states.
D. It is possible under some circumstances for a process to be in its wait state for ever.
E. It is possible for a process to be in its ready state forever.
F. It is possible for a process to be in its running state forever.

3. A process changes its state from the wait state to the ready state when:

✔A. I/O is completed.
✔B. A requested resource gets allocated.
C. The CPU becomes idle.
D. An I/O device being used by another process becomes idle.
E. A job is aborted.

4. The multi-level feedback queue scheme for CPU scheduling has a potential for starvation. A CPU intensive job may be starved by smaller jobs. To avoid starvation, a job should be promoted to a higher level queue than the one it currently occupies. This promotion should be based on:

A. The length of time the job has been in the system.
B. The length of time the job has been in a particular queue.
✔C. The length of time the job has not received service.
D. The ratio of CPU usage to I/O usage of the job is lower than 50%.

5. Two processes share the variable `I`. Process A increments it 10 times. Process B decrements it 25 times. The initial value of `I` is 0. Process A and B run concurrently, without using any critical sections. Which of the following are true statements?

A. The final value of `I` will be –15.
B. The final value of `I` can be any integer.
✔C. The final value of `I` will be an integer between –25 and 10, inclusive.
D. The final value of `I` will not be 0.
E. The final value of `I` can be as small as –20 or as large as 5, but will not fall outside this range.

6. Which of the following statements concerning static and dynamic relocation are true?

✔A. Memory Management is easier under dynamic relocation.

✔B. Linkers have to generate relocation bits in the object code for static relocation systems.
C. Linkers have to generate relocation bits for dynamic relocation systems.
✔D. The loader does the relocation in static relocation systems.
E. The operating system kernel does the relocation in static relocation systems.
F. The operating system kernel does the relocation in dynamic relocation systems.
✔G. The hardware does the relocation in dynamic relocation systems.

7. Some operating systems use *base* and *bound* registers to protect memory. The bound registers are updated (or changed) by:
A. Memory management hardware.
✔B. Memory management software.
C. Inline code in the processes (generated by compilers).
D. Array bound checking routines.
E. Language debugging support tools.

8. Swapping is a method of increasing the capacity of the main memory. Note that swapping is not the same as demand paging. Swapping is achieved by:
A. Using the same part of the main memory to run several processes at the same time.
B. Keeping only the relevant parts of a process in main memory and the rest on disk.
✔C. Keeping some running processes in main memory and some waiting processes on disk.
D. Moving processes from memory to disk every time a process time-slice is over.

9. A page table for a running process may need to be updated under some circumstances. Find the true statements about updating page tables.
A. The page table of a process has to be modified when the process accesses a page.
✔B. The page table of a process has to be modified when the process asks for more memory to be allocated.
✔C. The page table of a process has to be modified when a page is paged-out to disk.
D. The page table should be modified by the paging hardware.
✔E. The page table is modified by the memory management routines.
F. Once created, the size of a page table never changes.

10. A computer system has a segmented paging scheme. The segments are paged and the pages are demand-paged.
A. This scheme needs occasional compaction of main memory due to external fragmentation.
✔B. There is no internal or external fragmentation in main memory.
C. The segment tables are allocated in frames.
D. The page tables are allocated inside segments.
E. If the cache hit ratio is 90%, then a logical memory access on the average needs about 2 physical memory accesses.
F. If the cache memory is not used, the speed of execution of this scheme is about twice as slow as a system not using virtual memory.

Compilers

Comp1.

a) The type model of Pascal supports mixed-mode (integer-float) multiplication, but requires the arguments of addition and subtraction to have the same arithmetic types. Write a small grammar of expressions and assignments for Pascal, and write the semantic rules for type-checking each production. Is type-checking done in bottom-up or top-down fashion?

b) Some versions of Pascal define array compatibility in terms of structural equivalence. Give the semantic rules for type-checking array assignment in such a language.

Solution: Let us use the following conventional grammar for assignments and simple expressions. To perform type checking, it is best to define an attribute -type- on every non-terminal, and give rules for computing that attribute for each production in the grammar. For identifiers we can assume that the type comes from some symbol table. For literals the type is delivered by the scanner. For the non-terminals we give the following rules to take into account mixed-mode arithmetic:

```
S => Id := E           if Id.type /= E.type then
                          error ("Invalid types for assignment");
                          end if;

E => E + T                if E2.type /= T.type then
                             error("incompatible types for addition");
                          else
                             E1.type := E2.type;
                          end if;

E => T                    E.type := T.Type;

T => T * F                if T2.type = F.type then
                             T.Type := T2.Type;
                           elsif (T2.Type = Type_Float
                                            and F.Type = Type_Integer)
                           or else (T2.Type = Type_Integer
                                            and F.Type = Type_Float)
                           then
                              T1.type := T2.Type;
                            else
                               error("incompatible types for mult");
                            end if;

F => id                     F.type := defined_type (id);
F => literal                F.type := literal.type;
F => (E)                    F.type := E.type;
```

Regardless of the parsing strategy, the computation of the type proceeds bottom-up, from leaves of the tree to internal nodes. The attribute is synthesized. Computation of semantic attributes is described in great detail in chapter 5 of the text by Aho, Sethi and Ullman.

b) If the language specifies structural equivalence, arrays are conformant if they have the same component type and the same size. If the language specifies name equivalence, two arrays are compatible only if their types are one and the same. In the case of structural equivalence, the assignment

```
A1 := B1;
```

is valid if

```
     (A1.type.component_type = B1.type.component_type)
 and (A1.high_bound - A1.low_bound) = (B1.high_bound - B1.low_bound)).
```

Comp 2. For a one-dimensional array with a non-zero lower bound in a simple language like Pascal, a standard optimization called **virtual origins** is used to simplify addressing in the array. In the one-dimensional case, you can think of the virtual origin of the array as the location in memory where the first element of the array would have appeared had the array been declared with a lower bound at zero.

Consider a Pascal function:

```
function F(b, c : integer) : integer;
  var i, j : integer;
      a : array[2..4] of integer;
begin
  ...
  a[i] := 0;
  ...
end;
```

(a) Draw a picture of the stack frame, showing how all variables in `F` are allocated, and indicate where the virtual origin of the array `A` is located.

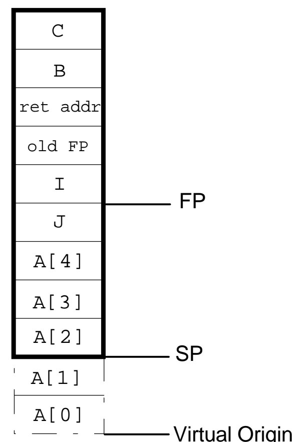
(b) Show the assembly language code (for any real assembly language) that a compiler would generate taking advantage of the use of virtual origins. Indicate how this code would have been different had virtual origins *not* been used.

(c) What if the array above was declared as a two-dimensional array:

```
a : array[2..4,2..4] of integer;
```

Is the virtual origin of a two-dimensional array still the same as the memory location of the first element (`a[0,0]` in this case) if the array had been indexed at 0? Either explain why you believe this is so or prove (with a counterexample) why not.

Solution:
Part (a). The stack frame for the procedure is shown below, along with the virtual origin of the array A. Note the order of the elements of the array A in the stack frame. Also note that the elements A[0] and A[1] are *not* part of the stack frame for the procedure.



Part (b). The 8086/88 assembly language code for the statement a[i] := 0 if virtual origins are used would be:

```
mov   si, i
shl   si, 1
mov   [bp-14][si], 0
```

Without the use of virtual origins the 8086/88 assembly language code would be:

```
mov   si, i
sub   si, 2
sh    si, 1
mov   [bp-10][si], 0
```

Note that without the use of virtual origins an extra instruction is required.

Part (c). The answer is no. The questions asks for an explanation using the array `a[2..4,2..4]` as an example. If the array were index at 0 (meaning it was declared as `a[0..4,0..4]`, then it's actual origin would be -54 (two bytes each for i and j and 25 * 2 for the 25 elements of array). On the other hand, if we do the address calculation for the virtual origin we get:

$$\text{Addr}(\ a[i,j]\ ) = \text{base}(a) + [\ ((i - low_1) * dim_1) + (j - low_2)\ ] * \text{stride}(a)$$

For this specific example we have:

```
Addr( a[i,j] )  = -22 + [ ((i - 2) * 3) + (j - 2) ] * 2
                = -22 + [ 3i - 6 + j - 2 ] * 2
                = -22 + [ 3i + j  - 8 ] * 2
                = -38 + [ 3i + j ] * 2
```

The correctly computed virtual origin is `-38`.

ALGORITHMS

Algs1. An algorithm *A* has this behavior: on inputs of size *n*, it takes 1 unit of time if $n \le 3$. If $n \ge 4$, it calls itself recursively on an input of size at most *n*/4. On return from the recursive call, *A* performs some more actions and then halts. The non-recursive part of *A* takes at most *n* units of time on inputs of size *n*. Let *T*(*n*) be the worst case time of this algorithm on inputs of size *n*.

a. State the recurrence for *T*(*n*).

$$T(n) = \begin{cases} 1 & \text{if} \quad n \le 3 \\ T\left(\left\lfloor n/4 \right\rfloor\right) + n & \text{if} \quad n > 3 \end{cases}$$

b. Solve as exactly as you can for *T*(*n*), assuming *n* is a power of 4 (i.e., $n = 4^k$).

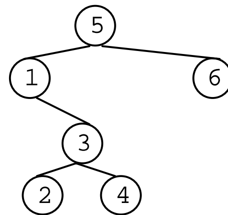
Set $S(k) = T(4^k)$. Then $S(k) = S(k-1) + 4^k$ for $k \ge 1$ and $S(0) = 1$. Thus $S(k) = \sum_{i=0}^{k} 4^i = \frac{4^{k+1}-1}{4-1} = \frac{4n-1}{3}$, so $T(n) = ⅓(4n-1)$.

c. What can you say about *T*(*n*) for **all** *n*?

If *n* is not a power of 4, then $T(4^{i-1}) \le T(n) \le T(4^i)$ where $i = \lceil \log_2 n \rceil$. So $⅓(4^i - 1) \le T(n) < ⅓\left(4^{i+1}\right)$, and $T(n) = \Theta(n)$.

Algs2.

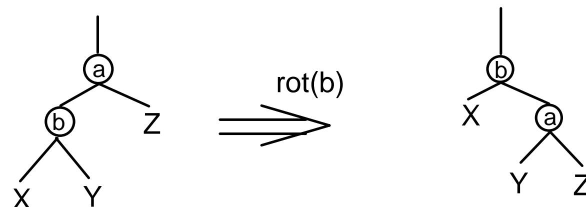
a. Draw the binary search tree T with keys 1,2,3,4,5,6 such that the preorder traversal of T yields the sequence 5,1,3,2,4,6.



b. For this question, first choose either red-black trees or 2-3 trees as your data structure. State your choice. Then answer the following question.

We want to modify the data structure so that each node of the tree maintains the *size* information; this information simply records the number of descendants of that node. We want to achieve this while preserving the $O(\log n)$ performance bound for insertions and deletions. Explain why this can be done.

Solution: For red-black trees, it suffices to consider only the rotations, since insertions and deletions reduce to a sequence of rotations. Suppose we rotate at a node b, as in the following diagram:



Here X, Y, Z are subtrees, and the size information at every node in these subtrees remains the same.

Then all the nodes in the tree preserve their size information EXCEPT at nodes b and its parent a. The new size information at a and b can easily be deduced from the old one in constant time. In fact, a gets the old size information at b, and a's size information is reduced by 1+size(X). Therefore, the time needed to update the size information in one rotation or one "double rotation" is constant, i.e., a rotation or double rotation is still constant time. Thus the asymptotic complexity does not change.

2-3 TREES: Insertion and deletion into a 2-3 tree modifies the tree either by the merging two nodes or splitting a node into two, potentially at every level from the root to the leaves. However, the operation of updating size information when we merge two nodes takes constant time: namely, the size of their merger is the sum of the sizes of the leaves minus 1.

When we split a node into u and v, the sizes of u and v are respectively the sum of the sizes of their children plus 1. Thus updating size information for a split also adds a constant amount of time. Clearly this information can be maintained without changing the $O(\log n)$ bound for insertion or deletion, since at most a constant amount of time is added per level.