(VERSION WITH ANSWERS)
CORE EXAMINATION
Department of Computer Science
New York University
September 28, 2007

This is the common examination for the M.S. program in CS. It covers core computer science topics: Programming Languages and Compilers, Operating Systems, and Algorithms. The exam has two parts. The first part (PL&C) lasts three hours and covers the first two topics. The second part (Algo), given this afternoon, lasts one and one-half hours, and covers algorithms.

You will be assigned a seat in the examination room.

Use the proper booklet or answer sheet for each question. Each booklet is marked with the Area and Question number, in the form PL&C1, PL&C2, PLC&C3, OS1, OS2, ALGS1 and ALGS2. But ALGS3 question has an answer sheet and not a booklet. DO NOT put your name on the exam booklet or answer sheet. Instead, your exam number must be on every booklet.

You will be graded according to your exam number, shown on the envelope containing the booklets. Remember your exam number: when grades are given out, they will be published according to this number, not by name.

Make sure your name and signature are on the envelope. This is the only place where your name appears. Please include all the booklets inside the envelope. You can keep the exam.

Good luck!

Programming Languages, Compilers and Operating Systems Questions

Question 1 – please use the Exam Booklet labeled PL&C1

Concurrency: For this question, you need to know either Java or Ada95, but not both.

There are many modern programming languages, including Java and Ada, that have constructs for expressing concurrency.

1. What is meant by "concurrency" in this context, particularly if programs are executed on a single-processor machine?

Answer: In this context, concurrency means the simultaneous execution – from a logical perspective, not necessarily actual simultaneity – of multiple portions of a program. Each portion is typically referred to as a "task" (as in Ada) or "thread" (as in Java). Because tasks or threads might not actually execute simultaneously, as is the case on a single-processor machine, concurrency means that the relative order of events (e.g. instructions) in two concurrent tasks or threads is left unspecified.

2. Write a simple example in Java or Ada of a program that exhibits concurrency.

Answer: Java:

```
class MyThread extends Thread {

    int n;
    MyThread(int j) {
        n = j;
    }

    public void run() {
        for(int i = 1; i <= 5; i++) {
            System.out.println(n + i);
        }
    }
}


public class test {
```

```
    public static void main(String[] args) {
        MyThread t1 = new MyThread(10);
        MyThread t2 = new MyThread(100);
        t1.start();
        t2.start();
    }
}
```

Ada:

```
with Text_Io;
use Text_Io;

procedure F is

   package Int_Io is new Integer_Io(Integer);
   use Int_Io;

   task One; -- spec
   task body One is
      X: Integer:=6;
   begin
      for I in 1..10 loop
         Put(X+I);
      end loop;
   end One;

begin
   for J in 90..100 loop
      Put(J);
   end loop;
end F;
```

□

---

3. Both Java and Ada allow variables to be accessed by portions of a program that are running concurrently. Thus, some synchronization between these portions is required in order to avoid bugs due to race conditions. In Java or Ada, briefly describe and give simple examples of two different ways to express synchronization (that is, using two different syntactic constructs provided by the language).

Answer:

Java (any two of the following is fine):

(a) Class-wide Synchronization: Attaching "synchronized" to a static method f of a class C means that, if f is running, then no other synchronized static method of class C can be running.

```
class C {

  static synchronized void f() { ... }
   ...
}
```

(b) Statement/Block synchronization: Using the construct

```
synchronized(o) { ... }
```

where o is an object, means that while the block ... is being executed, no other block that is also synchronized on the same object can be executing.

```
class C {

  int[] a = new int[20];

  void f() {

    synchronized(a) { ....}
    ...
   }

  ...
}
```

(c) Object Synchronization: Attaching "synchronized" to a non-static method G of a class C means that if, for a given object o of class C, if o.G is running, then no other synchronized method of o can run. Writing

```
synchronized void G { .... }
```

is equivalent to writing

```
void G { synchronized(this) { ... } }
```

An example is:

```
class C {

  synchronized void G() { ... }

  synchronized int H(..) { ... }
}
```

Ada:

(a) Using a task to control access to a variable: Have a single task that "manages" a variable, providing other tasks with access to the desired data through entry calls or procedure calls.

```
task stack_call is
   entry push(x:integer);
   entry pop(y:out integer);
end stack_call;

task body stack_call is
 begin
   loop
     select
       accept push(x: integer) do
         ...  -- push x onto the stack
       end push;
     or
       accept pop(y:out integer) do
         ...  -- pop y off the stack
      end pop;
     or
       terminate;
    end select;
   end loop;
end stack_call;
```

(b) Using a protected type, which ensures that only one procedure declared within the protected type can be running at any one time.

```
protected P_Stack is
  procedure Protected_Push(X:Integer);
  function  Protected_Pop return Integer;
end P_Stack;

protected body P_Stack is

    procedure Protected_Push(x:integer) is
     begin
       ...
     end;

    function Protected_Pop return integer is
    begin
      ...
    end;

end P_Stack;
```

□

Question 2 – please use the Exam Booklet labeled PL&C2

Fun with Functional Lists: Remember that both Scheme and ML have a built-in list data structure, where a list is either a NULL or a pair comprising of a head and a tail. The tail is (recursively) a pointer to a list. Given this structure, it's straight-forward to write a function `length` that counts the number of elements in a (non-circular) list.

Please keep your answers within the specified limits; LONGER ANSWERS WILL BE IGNORED.

1. (2 Points) Write the (recursive) definition of `length` in Scheme. Keep your answer to 5 lines of code or less.

Answer:

```
(define (length lst)
   (if (null? lst)
       0
       (+ 1 (length (cdr lst)))))
```

□

---

2. (2 Points) Write the (recursive) definition of `length` in ML. Keep your answer to 5 lines of code or less.

Answer:

One version in Standard ML:

```
fun length nil    = 0
   | length (_::t) = 1 + length t ;
```

Another version in O'Caml:

```
let rec length = function
   | []   -> 0
   | _::t -> 1 + length t ;;
```

□

---

3. (2 Points) What is the static Scheme type of `length` as determined by a Scheme compiler? Keep your answer to one sentence or less.

Answer: There is none; Scheme is dynamically typed.

4. (2 Points) What is the static ML type of `length` as determined by an ML compiler? Keep your answer to one sentence or less.

Answer: `'a list -> int`

5. (2 Points) For each of the following Scheme or ML programs indicate (a) whether the program will compile, (b) if so, what happens when you run it, and (c) why? Keep each answer to one sentence: "The program does x because of y".

   (a) Scheme: `(length 1)`

   Answer: The program compiles and terminates with an error becasue `1` is neither null nor a pair.

   (b) ML: `length 1;`

   Answer: The program does not compile because ML's type checker detects that `1` is not a list.

   (c) Scheme: `(length '(1 "two"))`

   Answer: The program compiles and evaluates to `2` because Scheme lists may contain any (type of) value.

   (d) ML: `length [1, "two"];`

   Answer: The program does not compile because ML lists must contain elements of a single type but `1` is of type `int` and `"two"` is of type `string`.

   (e) ML: `length [1, 2];` or `length ["one", "two"];`

   Answer: Both programs compile and evaluate to `2` since the lists are well-typed.

Question 3 – please use the Exam Booklet labeled PL&C3

Consider the following five context-free grammars (A)—(E) below. For each grammar, we have corresponding string.

	Grammar	String
A)	$S \rightarrow 0S1 \mid 01$	"000111"
B)	$S \rightarrow +SS \mid *SS \mid a$	"+*aaa"
C)	$S \rightarrow S+S \mid S*S \mid (S) \mid a$	"(a+a)*a"
D)	$S \rightarrow (L) \mid a$ $L \rightarrow L,S \mid S$	"((a,a),a,(a))"
E)	$S \rightarrow aSbS \mid bSaS \mid \epsilon$	"aabbab"

For each grammar and its corresponding string, answer the following questions:

1. Give a leftmost derivation for the string.
2. Is the grammar ambiguous or unambiguous? Justify your answer.
3. Describe the language generated by this grammar.
4. Is this grammar LL(1)? If so give the predictive parsing table.

Answer:

A) 1. A leftmost derivation of the string "000111" is given by

$$S \Longrightarrow 0S1 \Longrightarrow 00S11 \Longrightarrow 000111$$

2. The grammar is unambiguous. It can be shown to be an LL(2) grammar, and such grammars are always unambiguous.
3. The grammar defines the language of all strings of the form $\{0^n 1^n \mid n > 0\}$.
4. The grammar is not an LL(1) grammar. The two production rules for S start with the same terminal symbol 0.

B) 1. A leftmost derivation of the string "+∗aaa" is given by

$$S \Longrightarrow +SS \Longrightarrow +*SSS \Longrightarrow +*aSS \Longrightarrow +*aaS \Longrightarrow +*aaa$$

2. The grammar is unambiguous. As we show below, it is an LL(1) grammar, and such grammars are always unambiguous.
3. The grammar defines the language of all arithmetical expressions over "a" with the operations + and $*$, written in Polish (prefix) notation. An equivalent characterization is that in any word in the language the number of operations (+, $*$) is smaller by 1 than the number of operands (a) and, in any proper prefix of a word, the number of operations is smaller by more than 1 than the number of operands.
4. The grammar is an LL(1) grammar with the following LL(1) parsing table:

	$+$	$*$	a
S	$S \to +SS$	$S \to *SS$	$S \to a$

Answer:

C) 1. A left derivation of the string "(a+a)∗a" is given by

$$S \Longrightarrow \; SS \Longrightarrow S*S \Longrightarrow (S)*S \Longrightarrow (S+S)*S \Longrightarrow (a+S)*S \Longrightarrow (a+a)*S \Longrightarrow (a+a)*a$$

2. The grammar is ambiguous. The string aaa has the following two different leftmost derivations:

$$S \Longrightarrow SS \Longrightarrow aS \Longrightarrow aSS \Longrightarrow aaS \Longrightarrow aaa$$
$$S \Longrightarrow SS \Longrightarrow SSS \Longrightarrow aSS \Longrightarrow aaS \Longrightarrow aaa$$

3. The grammar defines the language of all regular expressions over the single symbol "a".
4. The grammar is not LL(1) because it is ambiguous and also left-recursive.

D) 1. A left derivation of the string "((a,a),a,(a))" is given by

$$S \Longrightarrow \; (L) \Longrightarrow (L,S) \Longrightarrow (L,S,S) \Longrightarrow (S,S,S) \Longrightarrow ((L,S),S,S) \Longrightarrow ((S,S),S,S) \Longrightarrow ((a,S),S,S) \Longrightarrow ((a,a),S,S) \Longrightarrow ((a,a),a,S) \Longrightarrow ((a,a),a,(L)) \Longrightarrow ((a,a),a,(S)) \Longrightarrow ((a,a),a,(a))$$

2. The grammar is unambiguous. It can be shown to be SLR(0) and is, therefore unambiguous.
3. The grammar defines the language of all list expressions over the atom "a".
4. The grammar is not an LL(1) grammar since it is left recursive.

E) 1. A leftmost derivation of the string "aabbab" is given by:

$$S \Longrightarrow \; aSbS \Longrightarrow aaSbSbS \Longrightarrow aabSbS \Longrightarrow aabbS \Longrightarrow aabbaSbS \Longrightarrow aabbabS \Longrightarrow aabbab$$

2. The grammar is ambiguous. Following are two different leftmost derivations of the string "abab":

$$S \Longrightarrow aSbS \Longrightarrow abS \Longrightarrow abaSbS \Longrightarrow ababS \Longrightarrow abab$$
$$S \Longrightarrow aSbS \Longrightarrow abSaSbS \Longrightarrow abaSbS \Longrightarrow ababS \Longrightarrow abab$$

3. The grammar defines the language of all strings over $\{a, b\}$ which have equal numbers of a's and b's.
4. The grammar is not LL(1) because it is ambiguous.

Question 4 – please use the Exam Booklet labeled OS1

Demand Paging: Recall that one millisecond (1ms) is 1/1000 seconds, one microsecond (1μs) is 10^{-6} seconds and one nanosecond (1ns) is 10^{-9} seconds. For this question, assume that one kilobyte (1KB) is 1,000 bytes (not 1024 bytes), and one megabyte (1MB) is 1,000,000 bytes.

Consider a system with demand paging but no segmentation, which preloads programs before they begin execution. All instructions that do not page fault require the same amount of time.

1. (1 point). When the system is equipped with a huge memory, a certain program P requires 50 seconds to complete, during which time P executes exactly 10 billion instructions and encounters no page faults. How long does each instruction take in this case.

   Answer: 50 seconds for 10 billion instructions gives 5ns per instruction.

2. (1 point). When the system is equipped with a modest memory and the page size is 4KB, the page fault rate is 1 per million instructions, with one tenth of the victim pages being dirty.

   How many faults occur with a modest memory, and how many I/Os are required to service all these faults.

   Answer: 1 fault per million for 10 billion instructions gives 10,000 faults.
   1/10 dirty gives 1000 writebacks and 11,000 total I/Os.

3. (6 points). EACH page fault requires executing an additional 1000 instructions that are guaranteed NOT to page fault (they are not counted in the page fault rate, and each takes the same time as you calculated for part 1).

   EACH I/O triggered by a page fault takes 15ms + PageSize/(20MB/second)

   How long does P require to complete when run with a modest memory.

   Answer: 1000 extra instructions per fault for 10,000 faults gives 10 million extra instructions at 5ns per instruction gives 50ms extra compute time.
   Each I/O takes 15ms + 4KB/(20MB/sec) = 15ms + 0.2ms = 15.2ms.
   Total time = 50sec + 50ms + 15.2ms = 50.0652 seconds.

4. (2 points). An analyst predicts that every time the page size is doubled (within a certain range) the fault rate would be multiplied by 0.7.

   If this prediction is correct, what is the run time as a function of the page size? That is, how long would P require to complete when run with a modest memory and a page size of S kilobytes?

Answer: Fault rate is multiplied by $0.7^{log(SKB/4KB)}$.
Let F be the number of faults $= 0.7^{log(S/4)} * 10,000$.
Number of I/Os $= 1.1 * F$
Number of extra instructions $= 1000 * F$, which requires $5 * F$ μs (microseconds)
Time for each I/O is $15ms + S/20ms = (15 + S/20)$ ms.
Time for all I/Os is $1.1 * F * (15 + S/20)$ ms.
Total time is $50s + 5F\mu s + 1.1F(15 + S/20)ms = (50 + .000005F + .0011(15 + S/20)F)$ seconds.

Question 5 – please use the Exam Booklet labeled OS2

Memory Management: Consider a memory management system that uses paging.

1. Typical page size range from 512 Bytes to 16 MBytes. What are the trade-offs in choosing smaller versus larger page sizes? List at least 3 considerations for full credit.

   Answer: Smaller pages means proportionally less internal fragmentation. Larger pages means smaller page tables. Larger pages leads to more efficient disk I/O.

2. What are the advantages of making page sizes a power of two?

   Use a concrete example to illustrate.

   Answer: This allows bits in a physical memory address to be divided into two disjoint parts: page address and page offset, with the property that the page offset occupies the lower order bits of the addresses. Example: if the physical address has n bits, and the page size is 2^m, then the page offset forms the the lower-order m bits of address.

3. Consider the following scenario:

   (a) Physical memory is 8 *gigabytes (GB)*

   (b) Logical address space is 64*-bit where each logical address refers to a byte of memory*

   (c) Page size is 128 *kilobytes (KB)*

   (d) Each entry in the page table stores a frame number, together with a byte of protection information (valid bit, read-write permission, etc).

   Use this scenario to compute the size of a page table, illustrating the following remark: "In 64-bit computer systems, the size of the page table is a serious issue." Assume that processes use the entire 64-bit logical address space. In this question, the prefixes kilo-, mega-, giga-, etc, refers to powers of two: $2^{10}, 2^{20}, 2^{30}$, etc.

Answer: Page table size is 3×2^{47} bytes. Hence a page table needs about 400 terabytes. This is a serious issue because each process is associated with one such table, and normally, the page table would be loaded in memory with the process.
Calculations: The number of frames is (physical memory size)/(page size) $= 2^{33}/2^{17} = 2^{16}$.
So each frame number is a 16-bit number. Thus each page table entry needs $16 + 8$ bits or 3 bytes.
The number of page table entries is (logical memory size)/(page size) $= 2^{64}/2^{17} = 2^{47}$.

4. "To reduce the memory requirements for a process table in the above scenario, we could use hierarchical page tables. But this would use 3 levels of page hierarchies, and it becomes essential to use hardware techniques to make it practical."

   Justify this remark about 3 levels of page hierarchies, and describe the hardware techniques needed.

Answer: Typically, the page table hierarchy is a tree whose internal nodes are pages. Since each page has 2^{17} bytes, and each frame number uses 2 bytes, it follows that each node of the hierarchy has $2^{17}/2 = 2^{16}$ children. Since the page table has 2^{47} entries, we need $\lceil 47/16 \rceil = 3$ levels in the tree hierarchy. Assuming the root of the page hierarchy is always in memory, each address reference requires up to 3 disk accesses: 2 disk accesses to get to nodes of the page hierarchy, and one more disk access to obtain the referenced data.
This factor of 3 slowdown is unacceptable. The usual solution is to use caching to store some of the recently used pages in memory. Moreover, we need a TLB (translation look-aside buffer) to quickly detect if a referenced page is in the cache. Assuming a high cache hit-ratio, the expected access time can go down to just a small fraction of a single disk access.