

# Final Exam

CSC 254

19 December 2004

## Directions

This exam comprises 8 short-answer questions (4 points each), 4 multiple-choice questions (2 points each), and 4 brief exercises (6 points each), for a total of 64 points. There are also two extra credit exercises, worth up to 5 points each. These won't factor into your exam score, but may help to raise your final semester grade.

This is a *closed-book* exam. You must put away all books and notes. Please confine your answers to the space provided. For multiple choice questions, darken the circle next to the best answer. Be sure to read all candidate answers before choosing.

Put your name on every page. Scratch paper is available if you need it, but **the proctor will collect only the exams**. No partial credit will be given on the multiple-choice questions.

In the interest of fairness, the proctor has been instructed not to answer questions during the exam. If you are unsure what a question is asking, make a reasonable assumption and state it as part of your answer.

You have a maximum of three hours to complete the exam, though it shouldn't take anywhere near that long. The proctor will collect any remaining exams promptly at 10:15 pm. Good luck!

## Short Answer (4 points each)

1. Under what circumstances should a `switch` statement be compiled as (a) a sequential series of `if`-like tests; (b) a jump table (indexed array); (c) a hash table; (d) binary search in an array?

**Answer:** (a) Use a sequential series of `if`-like tests if the number of arms in the `switch` statement is small. (b) Use a jump table if the overall set of labels on arms is dense, and too large for sequential testing. (c) Use a hash table if the set of labels is large but not dense. (d) Use binary search if there are many arms and the labels include large ranges.

2. Explain the difference between the *static link* and the *dynamic link* in a stack frame.

**Answer:** The dynamic link is the saved frame pointer; it points to the frame of the calling subroutine. In a language with lexical scope and nested subroutines, the static link points to the frame of the lexically surrounding subroutine.

3. Summarize the tradeoffs between reference counts and tracing (e.g. mark-and-sweep) as implementations of garbage collection.

**Answer:** Reference counting is generally cheaper. It's also incremental: it doesn't require us to pause the program in order to collect. On the other hand, it can fail to collect circular structures, and it can't be used for conservative collection in a non-type-safe language.

4. Explain why it may be desirable to write multithreaded code for a uniprocessor.

**Answer:** For servers in particular, multithreaded code allows us to maintain a separate context for every logically separate task. If tasks must sometimes block (e.g. while waiting for messages or I/O) then multithreaded code is likely to be much easier to understand than code that explicitly interleaves the execution of the tasks.

5. ML and Lisp both provide implicit parametric polymorphism, but in very different ways. Explain.

**Answer:** ML performs all type checking at compile time. It uses extensive type inference to figure out exactly what a subroutine needs to know about its arguments, and what it doesn't. It then allows the routine to be applied to the broadest possible range of types. Lisp performs all type checking at run time. ML collections must be homogeneous; Lisp collections can be heterogeneous.

6. In C, which uses a value model of variables, the semantics of  $a = b + c$  can be stated informally as “give  $a$  the value obtained by adding the values of  $b$  and  $c$ .”

- (a) Give similar informal semantics for the same assignment in Clu or Lisp, which have a reference model of variables. (In Lisp the assignment would be written `(set! a (+ b c))`.)
- (b) Give similar informal semantics for the same assignment in Smalltalk or Ruby, which have both a reference model of variables and an object model of data. (In Smalltalk the assignment would be written `a <- b + c`.)

**Answer:**

- (a) Make  $a$  refer to the value obtained by adding the values to which  $b$  and  $c$  refer.
- (b) Send an “add” message to  $b$ , with a reference to  $c$  as a parameter. Make  $a$  refer to the object indicated by  $b$ 's response.

7. Under what circumstances might it be better to use a test-and-set spin lock for mutual exclusion, rather than a binary semaphore?

**Answer:** A spin lock makes sense if the expected wait time is shorter than twice the context switch time, or if we have nothing else productive to do with the current processor. A semaphore is better if you want to give the processor to another thread while waiting. A spin lock probably doesn't make sense if you have more threads than processors. It definitely doesn't make sense on a uniprocessor. (Also, a test-and-set lock won't work well on more than a few processors; larger machines need fancier spin locks. And a test-and-set lock isn't fair: threads don't necessarily get it in order. A semaphore is fair, as are many of the fancier spin locks.)

8. Ada, which has parameter modes similar to those of this year's PL/0, allows `in out` parameters to be implemented either by value/result (copying, as we did in PL/0) or reference (passing the address of the argument). Which implementation should a compiler use when? Explain. (Note that I am asking about `in out` parameters only; not `in` or `out`.)

**Answer:** In general, small arguments should be passed by value, because copying them is cheaper than indirecting through a reference all the time. Large arguments should generally be passed by reference, particularly if the callee doesn't read their entire contents, to avoid the expense of copying.

**Multiple Choice** (2 points each)

9. *Closures*, which combine a subroutine pointer and a referencing environment, are not needed

- a. in a language with dynamic scope.
- b. for shallow binding.
- c. for second-class subroutines.
- d. None of the above—closures are needed for all of these.

10. What does it mean for a method in C++ to be *virtual*?

- a. It overloads a base class method of the same name.
- b. It employs dynamic dispatch (i.e. is polymorphic).
- c. It must be defined explicitly in every derived class.
- d. Only its declaration appears in the `.h` file; its definition is in a separate `.cc` file.

11. What formal system provides the semantic foundation for Prolog?

- a. Predicate calculus.
- b. Lambda calculus.
- c. Milner's Calculus of Communicating Systems (CCS).
- d. Propositional logic.

12. Consider the following two programs in C++:

```
// program A
#include <iostream>
using std::cout;
main() {
    struct foo {
        int i;
    };
    foo x = {3};
    foo y = x;
    x.i = 4;
    cout << y.i << "\n";
}

// program B
#include <iostream>
using std::cout;
main() {
    struct foo {
        int i;
    };
    foo x = {3};
    foo& y = x; // note use of ref
    x.i = 4;
    cout << y.i << "\n";
}
```

What do programs A and B print?

- a. A prints 3; B prints 4.
- b. A prints 4; B prints 3.
- c. They both print 3.
- d. They both print 4.

**Problem Solving** (6 points each)

13. Suppose  $A$  is a  $10 \times 10$  array of (4-byte) integers, indexed from  $[0][0]$  through  $[9][9]$ . Suppose further that the address of  $A$  is currently in register  $r1$ , the value of integer  $i$  is currently in register  $r2$ , and the value of integer  $j$  is currently in register  $r3$ .

Give pseudo-assembly language (C-like syntax is fine) for a code sequence that will load the value of  $A[i][j]$  into register  $r1$  (a) assuming that  $A$  is implemented using (row-major) contiguous allocation; (b) assuming that  $A$  is implemented using row pointers. Each line of your pseudocode should correspond to a single instruction on a typical modern machine. You may use as many registers as you need. You need not preserve the values in  $r1$ ,  $r2$ , and  $r3$ . You may assume that  $i$  and  $j$  are in bounds, and that addresses are 4 bytes long.

Which code sequence is likely to be faster? Why?

**Answer:**

|                  |                 |
|------------------|-----------------|
| (a) $r2 \ll= 40$ | (b) $r2 \ll= 2$ |
| $r3 \ll= 2$      | $r1 += r2$      |
| $r1 += r3$       | $r1 = *r1$      |
| $r1 += r2$       | $r3 \ll= 2$     |
| $r1 = *r1$       | $r1 += r3$      |
|                  | $r1 = *r1$      |

The left shifts effect multiplication by 4. The code sequence on the left is likely to be faster on a modern machine, not because it is one instruction shorter, but because it performs only one load instead of two.

14. The following implements an iterator object for a simple linked list in Java.

```
class List implements Iterable {
    class node {
        public Object val;
        public node next;
        node(Object v) { val = v; }
    }
    private node head;
    ...
    public Iterator iterator() { return new ListIterator(); }
    private class ListIterator implements Iterator {
        private node cursor;
        ListIterator() { cursor = head; }
        public boolean hasNext() { return cursor != null; }
        public Object next() {
            node n = cursor; cursor = cursor.next; return n.val;
        }
    }
}
```

If Java supported true iterators (which of course it doesn't), how might you rewrite this? (I'm not going to be picky about syntax here, since you have to invent your own. I simply want to know that you understand the concept of true iterators.)

**Answer:** Eliminate the nested `ListIterator` class and rewrite method `iterator` as follows.

```
public Object iterator() {
    private node cursor = head;
    while (cursor != null) {
        yield cursor.val;
        cursor = cursor.next;
    }
}
```

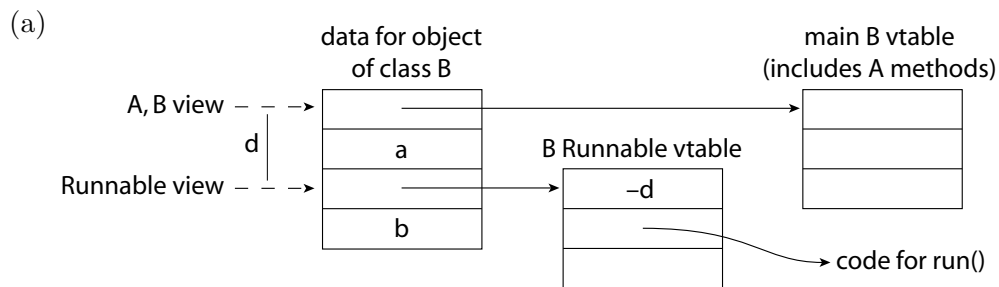
15. Consider the following code, written in a language like Java, with mix-in inheritance:

```
class A {
    private int a;
    ...
}
class B extends A implements Runnable {
    private int b;
    ...
}
...
Runnable r = new B();
```

Suppose our language is compiled, using the implementation techniques described in class.

- (a) Draw a diagram of the implementation of a B object, showing fields (data members) and vtable(s).
- (b) Show (in pseudo-assembly language) the calling sequence for `r.run()` (caller side only). You may assume that `r` has been loaded into register `r1`, and that `run` is a method defined by the `Runnable` interface. You may use as many registers as you need. You need not preserve `r1`. Assume that the `this` parameter is to be passed in register `r3`.

**Answer:**



```
(b) r2 = *r1    // Runnable vtable
     r3 = *r2    // B view offset
     r3 += r1    // B view of r
     call *(r2+4) // run
```

This assumes that `run` is the first method in the B-Runnable vtable (after the `this` correction, `-d`).

16. Consider the following function in Scheme:

```
(define min
  (lambda (l)
    (cond
      ((null? l) '())
      ((null? (cdr l)) (car l))
      (#t (let ((a (car l))
                 (b (min (cdr l))))
             (if (< b a) b a))))))
```

Explain how to convert this to proper tail recursive form (no computation after the recursive call). You can write code if you want, or just explain the transformation well.

**Answer:** Create a nested “helper function” that carries the minimum-so-far along with it:

```
(define min2
  (lambda (l)
    (letrec ((min-helper
              (lambda (l m)
                (if (null? l) m
                    (let* ((a (car l))
                           (n (if (or (null? m) (< a m)) a m)))
                      (min-helper (cdr l) n))))))
      (min-helper l '()))))
```

Alternatively,

```
(define min3
  (lambda (l)
    (cond
      ((null? l) '())
      ((null? (cdr l)) (car l))
      (#t (let ((a (car l))
                 (b (cadr l)))
             (min3 (cons (if (< b a) b a) (cddr l)))))))
```

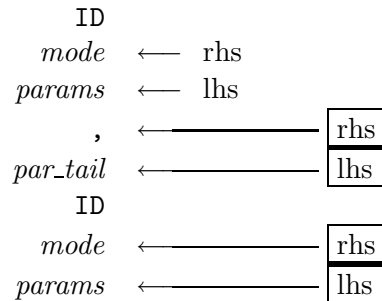
17. (Extra Credit up to 5 points)

Consider the following grammar with action routines.

```
params  → mode ID par_tail [[ params.list = insert(⟨mode.val, ID.name⟩, par_tail.list) ]]
par_tail → , params [[ par_tail.list = params.list ]]
          → [[ par_tail.list = nil ]]
mode   → IN [[ mode.val = IN ]]
          → OUT [[ mode.val = OUT ]]
          → IN OUT [[ mode.val = IN OUT ]]
```

Suppose we are parsing the input IN *a*, OUT *b*, and that our compiler, like the PL/0 compiler, uses an automatically maintained attribute stack to hold the active slice of the parse tree. Show the contents of this attribute stack immediately *before* the parser predicts the production *par\_tail*  $\rightarrow \epsilon$ . Be sure to indicate where *lhs* and *rhs* point in the attribute stack. Also show the stack of saved *lhs* and *rhs* values, showing where each points in the attribute stack. You may ignore the *ssf* (seen so far) pointer.

**Answer:** *par\_tail*                      saved values



18. (Extra Credit up to 5 points)

Discuss what you consider to be one of the most interesting interactions between language design and language implementation.

**Answer:** (No solution offered; see the book!)