

**Honors Programming Languages  
G22.3110 Fall 2004**

**Final Exam**

1. (a) Show how Ada tasks can be used to implement a binary semaphore; the operations P and V should be provided. Your solution should be of the form:

```
task type Semaphore is
    entry P;
    entry V;
end Semaphore;
task body Semaphore is
    ...
end Semaphore;
```

A semaphore is an object used to enforce mutual exclusion among multiple processes. When a process A performs a P operation on a semaphore, the process can only proceed if no other process has already executed a P on the same semaphore. If another process, B, has already executed a P, then process A will suspend until B has executed a V on the semaphore. At that point, A can continue executing (thus all other processes attempting to execute a P on the same semaphore will suspend until A executes a V on the semaphore).

If you need a better explanation of semaphores, don't hesitate to ask me.

- (b) As you can see, above, a semaphore is a task. However, users of semaphores shouldn't know whether it is a task or some other kind of object. It should be an abstract data type. Use the Ada package facility to create a semaphore abstract data type.
2. Reference counting storage reclamation is generally more "incremental" (i.e. a little at a time) than stop-and-collect methods.
- (a) Why is it sometimes desirable to have incremental storage reclamation?
- (b) Reference counting isn't truly incremental, since the deletion of a single pointer could lead to a lengthy delay of processing due to storage reclamation. Under what circumstances could reference counting storage reclamation take a sizeable amount of time?
- (c) Reference counting also cannot guarantee that every dead object (i.e. an object that cannot be subsequently referenced) is reclaimed. Write in Scheme some code that creates a data structure that will not be reclaimed by reference counting.
- (d) Give the algorithm for generational garbage collection.
3. (a) Why are higher order functions a desirable property in a programming language? What are the drawbacks of including higher order functions in a language?
- (b) Write ML functions that implement the operators  $\sum$  (sum),  $\prod$  (product), and  $\circ$  (compose).
- (c) Write one ML function F such that  $\sum$  and  $\prod$  can be defined as values returned by F. Show how  $\sum$  and  $\prod$  are defined using F.
- (d) In ML, formal parameters must be used monomorphically. Thus, the definition

```
fun f g = g 1 + g true
```

is type-incorrect. Suggest an extension to ML that would allow formal parameters to be used polymorphically. You may require the programmer to supply additional declarations if necessary. Once you have done this, describe (informally) how ML's type checking algorithm might have to be modified to handle your extension (Don't spend much time on this question, it's a bit of a research question).

4. In Prolog, write the code to find the maximum element of a list of numbers. That is, implement the predicate `max` such that the result of the query

```
max([1,2,3,0],X).
```

is

```
yes, X=3.
```

Don't worry about error checking (e.g. if an element of the list is not a number). If the list is empty, then `max` should instantiate `X` (the second argument) with 0. Be sure to handle the case correctly, though, when all the elements of the list are negative.

5. In class, we defined the denotational semantics of a simple language with assignment and pass-by-reference. Suppose we wanted to extend the language with explicit pointers, with the following syntax for expressions:

```
e ::= ... | &e | *e
```

where `&` is the "address of" operator and `*` is the dereferencing operator, just like in C (and the "..." is the syntax already given in class).

- (a) Would the definitions of any of the semantic domains (*FUN*, *D*, *ENV*, *STORE*, *LOC*, etc.) have to be modified from those definitions given in class? If so, write out the new definitions of any semantic domains that would have to change.
  - (b) Define the semantics of the two new expressions (`&e` and `*e`), by defining how the semantic function *E* operates on them.
6. The expression `n npow S` in SETL returns the set of all subsets of *S* that have *n* elements.
- (a) Suppose `npow` wasn't already defined for you in SETL. Write a function `my_npow(S,n)` that behaves like `npow`, using as short a definition as possible (you can still use `pow`).
  - (b) Now, write a more efficient version of `my_npow` that doesn't use `pow`.