Programming Languages and Compilers
Comprehensive Exam

May 1996

Please answer each question in a separate booklet.

1. (a) Given the SML declaration

   ```
   exception foo of int
   ```

   what is the type of `foo`?

   (b) Write a legal expression that involves the raising and handling of `foo`.

   (c) The raise operator is not a first-class function but, if it were, what would its type be?

   (d) Type `exn` is not defined via the use of the `datatype` construct, but rather via the `exception` construct. Describe, and give examples of, how `exn` is similar to a type defined using datatype (it's very similar). Then describe how it is different.

2. (a) For Pascal, prove formally that when a non-local variable x is referenced in a procedure f, the activation record containing x is currently on the stack and in the static chain of the current invocation of f.

   (b) Write an example in Scheme that demonstrates that this property would not hold in Scheme (assuming the implementation used a stack).

   (c) In a compiled Scheme implementation with heap allocation of activation records and mark/sweep garbage collection, prove that the same property as in part (a) holds, except that the activation record containing x will be in the heap (and not have been garbage collected).

3. Consider the following interprocedural analysis problem, where we assume unnested procedure definitions, no procedure parameters, call-by-reference parameter passing, and exactly one parameter per procedure. The language has assignments $x := e$ under copy/value semantics, conditionals **if** e **then goto** L, and subroutine calls $f(e)$, where x is a variable, e is an expression with no side effects, and L is a label. All variables other than formal parameters are global. A **goto** statement S must only branch to statements within the procedure or main program that contains S.

   Let $vars$ be the set of global variables, and $procs$ be the set of procedures. For each procedure f, let $param(f)$ be the name of the formal parameter, and let $calls\{f\}$ be the set of **call**-instructions occurring within f. For each **call**-instruction $i \in calls\{f\}$, let $called(i)$ be the procedure called by i, and let $arg(i)$ be the argument passed in procedure call i.

   (a) With wanton disregard for efficiency show how to compute the set $assoc\{f\}$ of variables that at runtime might be associated with the formal parameter of procedure f for each such f.

   (b) Assume a control flow graph, where each node represents a single instruction. If node n represents procedure call $f(e)$, then the successor of n in the control flow graph is the node representing the first statement of procedure f. The successors of the last statement of procedure f are the nodes that represent those statements in the program that come just after a call to procedure f.

   Given such a control flow graph, and taking aliasing into account, what does it mean for a use of a variable at a program point to be live at another point? What does it mean for a definition to a variable at a point to reach another point? Formulate how to determine the *usetodef* relation (which relates each use u with every definition v that might reach u, taking aliasing into account).

4. (a) Give an example of
   i. a context free grammar that's not regular.
   ii. an LL(2) grammar that's not LL(1)
   iii. an LR(1) grammar that's not LR(0)

   (b) Show that the following grammar is LR(0) but not LL(k) for any k:

```
E = B
E = A
B = true
B = ( B )
A = 1
A = ( A )
```

   (c) Draw the LR(0) automaton for the grammar shown in part (b), clearly indicating the productions used in each reduce state.

5. (a) Show how Ada (83 or 95) tasks can be used to implement semaphores; 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;
```

   (b) How could you extend your solution to (a) to provide initialization of the semaphore count?

   (c) *Monitors* are an alternative synchronization mechanism which provides hidden data, mutually exclusive procedures which operate on that data, and condition variables. Operations on condition variables include `await` and `signal`; if a process does an `await` on a condition variable, another process is allowed to execute a monitor procedure. You may assume that the `signal` operation can only be used as the last operation executed by a monitor procedure. Show how to implement monitors using Ada (83 or 95) tasks. (Note that Ada 95 also provides *protected records* which are similar to monitors - these should not be used). Hint: Don't try to implement the monitor as a task; implement the monitor as a package, and use a task to implement mutual exclusion and the condition variables.

6. The abstract syntax for a Scheme-like language without continuations is given below:

```
datatype value = ...            --omitted
datatype id = ...               --omitted
datatype pfun1 = ...            --omitted
datatype pfun2 = ...            --omitted
datatype decl = ...             --omitted
datatype environment = ...      --omitted
datatype expr =    constant of value
                 | variable of id
                 | primfun1 of pfun1*expr
                 | primfun2 of pfun2*expr*expr
                 | if of expr*expr*expr
                 | let of decl*expr
                 | lambda of id*expr
                 | application of expr*expr
```

An interpreter for this language is the following:

```
fun     eval (constant(v), environment) = v
|       eval (variable(v), env) = lookup(v,env)
|       eval (if(x,y,z),env) =
                if eval(x,e) then eval(y,e) else eval(z,e)
|       eval (let(d,x), env) = eval(x, extendenv(d,env))
|       eval (lambda(v,x), env) = closure(v, x, env)
|       eval (application(e1, e2), env) =
                let val closure(param,body,e) = eval(e1, env)
                in  eval(body, extendenv(bind(param,eval(e2,env)), e))
                end
|       eval (primfun1(f,x), env) = applyprim1(f, eval(x,env))
|       eval (primfun2(f,x,y), env) =
                applyprim2(f, eval(x,env), eval(y,env))
```

(a) Rank, with a grade of 0 (bad) through 10 (good), the following languages as candidates for (i) specifying the semantics of the language and (ii) implementing the interpreter: (1) Pascal (2) Ada83 (3) Scheme (4) C++ (5) Prolog (6) ML. Write 1 sentence of explanation on each (i.e. 12 in all).

(b) Sketch how to modify the interpreter to use continuations, and implement call/cc. (Hint: Eval will need an additional parameter).