Programming Language Qualifying Exam

January, 2000

There are five major questions and 6 pages to this exam. Each question is equally weighted. You have 2.5 hours to complete the exam. Please write all of your answers in the provided exam booklets. Also, please be sure to put your secret code number on all of the exam books that you turn in. Good luck!

1. **Semantics (20 points):** In a simple imperative language, we can interpret statements as partial functions from states to states, where states are mappings from variables to integers ($\sigma = \text{Var} \rightarrow \text{Int}$). For example:

$$\begin{aligned} \texttt{skip} &= \lambda\sigma.\sigma \\ x := E &= \lambda\sigma.\sigma[x \mapsto E(\sigma)] \\ \texttt{if } B \texttt{ then } S_1 \texttt{ else } S_2 &= \lambda\sigma.\text{if } B(\sigma) \text{ then } S_1(\sigma) \text{ else } S_2(\sigma) \end{aligned}$$

We assume that statement composition "$S_1; S_2$" is strict. That is, $(S_1; S_2)(\sigma) = S_2(S_1(\sigma))$ when $S_1(\sigma) \neq \bot$, and otherwise $(S_1; S_2)(\sigma) = \bot$. We also assume that expressions (E) and tests (B) have no side effects.

The interpretation of the statement "$\texttt{while } B \texttt{ do } S$" can be defined as the least (with respect to the usual ordering on $\sigma \rightarrow \sigma_\bot$) solution of the equation:

$$\texttt{while } B \texttt{ do } S = \texttt{if } B \texttt{ then} \{S \,;\, \texttt{while } B \texttt{ do } S\} \texttt{ else skip}$$

or equivalently, as the least fixed-point of the monotone map:

$$\lambda f.\texttt{if } B \texttt{ then } \{S \,;\, f\} \texttt{ else skip}$$

(a) (5 points) Give a similar definition for the statement "$\texttt{repeat } S \texttt{ until } B$". Intuitively, the semantics of repeat-until is as follows: the body S is executed, then the test B is evaluated; if the test evaluates to false, then the body is executed again, and so on until the test becomes true.

(b) Define the repeat-until construct in terms of the while-do construct and vice-versa. Prove that your definitions are correct using the following proof rules. For ϕ a monotone map with least fixed-point f,

$$f = \phi(f)$$

$$\frac{\phi(g) \sqsubseteq g}{f \sqsubseteq g}$$

2. **Object-oriented programming (20 points):**

Alyssa P. Hacker is developing a package of immutable list values in a simple Java-like object-oriented language. So far she has written an interface that defines the type `List`:

```
interface List {
  // A "List" is a possibly empty immutable list of objects.

  Object first() throws Empty; // first : unit -> Object
      // Return the first object in the list or throw the exception "Empty"
      // if the list is empty.
  List rest() throws Empty;    // rest : unit -> List
      // Return the list consisting of the second and following items or
      // throw the exception "Empty" if the list is empty.
}
```

She can't use the special value `null` to represent an empty list, ala Lisp, because it won't support the `first` and `rest` methods properly. Alyssa wants to implement this type using a Java class, but she quickly realizes that using only one class will be inefficient. For example, here is a possible implementation that she has rejected because it wastes both space and time:

```
class BadList {
  bool empty;
  Object item;
  List next;

  Object first() { if (empty) throw new Empty(); else return item; }
  List rest() { if (empty) throw new Empty(); else return next; }
}
```

(a) Show how Alyssa can implement `List` using just two classes, neither of whose code uses the value `null` or an `if` statement.

(b) Alyssa also has defined an interface for doubly-linked lists as follows:

```
interface DList {
  // A "DList" is a possibly empty ordered list of objects with a
  // current position located somewhere in the list.

  Object first() throws Empty;
      // return the object at the current position of the list
  DList rest() throws Empty;
      // return a doubly-linked list identical to the current list, but
      // whose current position is advanced by one position. If there are no
      // following entries in the list, throw "Empty".
  DList prev() throws Empty;
      // Like "rest", but the current position is moved to the preceding
      // position in the list. If there is no preceding object in
      // the list, throw "Empty".
}
```

It would be nice to be able to apply operations that operate on singly-linked lists to doubly-linked lists; in Java terminology, to declare that `DList extends List` or in more conventional terminology, that the interface `Dlist` is a subtype of `List`. However, Java would reject such a definition.

Argue that either the Java type system is too conservative in that `DList` could be safely considered a subtype of `List`, or else provide a counter-example demonstrating that we could get a run-time type error.

(c) Suppose we define an interface `Holder` as follows:

```
interface Holder {
  Object get();
  void set(Object x);
}
```

We might define a class `IntHolder` as follows:

```
class IntHolder implements Holder {
  private Integer item;
  Integer get() { return item; }
  void set(Integer x) { item = x; }
}
```

Note that in Java, `Integer` is a sub-type of `Object`. However, Java will reject the definition `IntHolder` claiming that it does not meet the interface requirements. Argue that either Java's type system is too conservative in that `IntHolder` can be safely considered a `Holder` or else provide a counter-example demonstrating that we could get a run-time type error.

3. **Programming Methodology (20 points):**

Consider the following SML code for regular expressions over strings of characters:

```
datatype regexp =
   Zero                       (* R[0] = {} (the empty set of strings) *)
 | One                        (* R[1] = {""} = {epsilon} *)
 | Char of char               (* R[c] = {"c"} *)
 | Concat of (regexp * regexp)(* R[r1 r2] = {s1^s2 | si in R[ri]} *)
 | Or of (regexp * regexp)    (* R[r1|r2] = R[r1] U R[r2] *)
 | Star of regexp             (* R[r*] = R[1|rr*] *)

fun match (r: regexp) (k: char list -> bool) (s: char list) : bool =
    case r of
      Zero          => false
    | One           => k(s)
    | Char c1       => (case s of
                          nil => false
                        | c2::rest => if c1 = c2 then k(rest) else false)
    | Concat(r1,r2) => match r1 (match r2 k) s
    | Or(r1,r2)     => (match r1 k s) orelse (match r2 k s)
    | Star(r')      => k(s) orelse match r' (match (Star(r')) k) s

fun accept (r: regexp) (s: string) : bool =
    match r (fn nil => true | _ => false) (String.explode s)
```

The function `accept` is intended to return true if the given string `s` is in the set denoted by the regular expression `r` and false otherwise. Running the code seems to indicate that this is the case:

```
- accept (Star(Or(Char #"a",Char #"b"))) "abababaabbbba";
val it = true : bool
- accept (Star(Or(Char #"a",Char #"b"))) "";
val it = true : bool
- accept (Star(Or(Char #"a",Char #"b"))) "ccc";
val it = false : bool
```

The heart of the algorithm is the function `match` which is written in a continuation-passing style. Conceptually, `match` breaks the string (represented as a list of characters) into two components, the first to be matched by the regular expression `r`, and the second to be matched by the continuation `k`. For example, in the case for `Concat(r1,r2)`, we recursively call `match r1` to match the beginning of `s` against `r1`, and to match the rest of `s` using `r2` and the continuation `k`. We use an initial continuation that succeeds iff the entire string `s` is consumed by the match against `r`.

(a) Though simple and elegant, the algorithm is extremely inefficient. However, there is a more serious problem: Give a regular expression and string that causes `accept` to loop forever. (Hint: see the next question.)

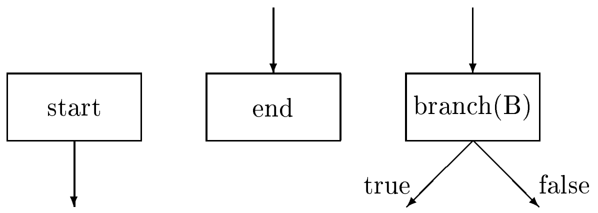
(b) We can fix the implementation by transforming the regular expression to *standard form* before calling `match`. A standard form regular expression has the property that if it contains a sub-expression of the form $r*$, then the language denoted by r does not contain the empty string. Prove that "`match r k s`" always terminates when `r` is standard and `k` is a total function by establishing a suitable metric and (informally) arguing that this metric strictly decreases for each recursive call to `match`.

4. **Dataflow: (20 points)**

A variable is *live* at a given control-flow point if there exists a path from the point to a use of the variable before any definition of the variable.

(a) Formulate the problem of finding the set of variables that are live at a given control-flow point using a dataflow schema. Your answer must specify clearly whether the problem is a forward or backward flow problem, what the transfer function for a node is, and what the confluence operator is.

You may assume that the control-flow graph has the types of nodes pictured below and that there is a path from every node to end.



(b) Explain how your analysis can be used by an optimizer to eliminate some useless computations.

(c) Explain briefly how a garbage collector might take advantage of your analysis.

5. **Exceptions and Type Systems (20 points):**

Languages such as Clu, Modula-3, ML, and Java all provide facilities for raising and catching exceptions in order to signal and recover from unexpected situations. Clu, Java, and Modula-3 have the additional nice feature that the set of exceptions that might be raised within a procedure or method is explicitly mentioned in interfaces.

(a) What is the advantage of explicitly mentioning the exceptions that may be raised in an interface?

(b) Unfortunately, the ML type system does not provide any information about whether functions raise exceptions. This is primarily due to interactions with higher-order functions, as we will demonstrate below.

Suppose we decided to add exception annotations to function types in a simply-typed, *call-by-value* functional language. For instance, we might give the function "div" (integer division) the type:

```
div : (int*int -> int){Div}
```

reflecting the fact that it will raise the exception `Div` when the second argument is zero. In contrast, a function like + (integer addition) might have the type

```
+ : (int*int -> int){}
```

indicating that addition raises no exceptions.

Let us define the abstract syntax for expressions as follows. Let x range over variables, c over constants including integers and primitive functions, and e over a finite set of exception names. Define types and expressions using the following BNF grammar:

$$\begin{array}{rcl} \tau & ::= & \texttt{int} \mid (\tau_1 \to \tau_2)\{e_1, \ldots, e_n\} \\ E & ::= & x \mid c \mid \lambda x.E \mid E_1\, E_2 \mid \texttt{raise}\ e \mid \texttt{try}\ E_1\ \texttt{catch}\ e \Rightarrow E_2 \end{array}$$

The expression "`raise` e" raises the exception e by transfering control to the nearest (dynamically) enclosing `catch`-phrase. The form "`try` E_1 `catch` $e \Rightarrow E_2$" establishes a new exception context and evaluates as follows: First, the expression E_1 is evaluated. If E_1 returns a value v, then the old exception context is restored and v is returned as the result of the entire expression. If an exception e' is raised during evaluation of E_1 (and not caught by an intervening exception handler), then control transfers to the `catch`-phrase. If $e' = e$ then the exception is caught, the old exception context is restored, and the expression E_2 is evaluated, yielding the value for the expression or else another exception raise. If $e' \neq e$, then the exception is propagated to the next enclosing `catch` (if any).

Let Γ range over typing assumptions (finite maps from variables to types.) Formulate an appropriate set of inference rules that allows one to conclude judgments of the form:

$$\Gamma \vdash E : \tau\{e_1, \ldots, e_n\}$$

with the meaning that, under the assumptions of Γ, the expression E either evaluates to a value of type τ or raises one of the exceptions in $\{e_1, \ldots, e_n\}$. (You may ignore the constants.)

(c) Treating "`let` $x = E_1$ `in` E_2" as syntactic sugar for "$(\lambda x.E_2)E_1$", assuming that we added tuples to the language, and assuming that + and div have the types given above, what is the best type your rules can assign to the function below?

$$\begin{array}{l} \texttt{let apply} = \lambda f.\lambda x.(f x)\ \texttt{in} \\ \texttt{let divide} = \texttt{apply div in} \\ \texttt{apply} + \end{array}$$