Computer Science 15–312: Programming Languages
Midterm Examination

March 2, 2000

Name

Name: ____

Andrew ID: ____

Instructions

- This is an open-book, open-notes examination. You have 1hr 20mins to complete this examination.
- There are 14 pages in this examination, including two worksheets.
- This examination consists of 3 questions worth a total of 90 points.
- Read each question completely before attempting to solve any part.
- Write your answers legibly *in the space provided* on the examination sheet. If you use the back of a sheet, indicate clearly that you have done so on the front.
- Write your name and Andrew id on this page and on the top of each successive page in the sapce provided.
- Two worksheets are attached to the end of this examination for your use.

Grading

Question	1	2	3	Total
Score				
Maximum	30	30	30	90

2 Contextual Semantics

30 Points

The syntax for MinML is given below.

$$
\begin{array}{rcll}
\tau & ::= & \texttt{int} & \text{integer} \\
 & \mid & \texttt{bool} & \text{boolean} \\
 & \mid & \tau \to \tau & \text{function} \\
 \\
e & ::= & x & \text{variable} \\
 & \mid & n & \text{numeral} \\
 & \mid & \texttt{true} \mid \texttt{false} & \text{boolean constant} \\
 & \mid & +(e, e) & \text{primitive operation} \\
 & \mid & \texttt{if}\, e \,\texttt{then}\, e \,\texttt{else}\, e \,\texttt{fi} & \text{conditional expression} \\
 & \mid & \texttt{fun}\, f\ \texttt{(}x \texttt{:} \tau\texttt{)}\texttt{:}\tau \,\texttt{is}\, e \,\texttt{end} & \text{function} \\
 & \mid & \texttt{apply}\texttt{(}e, e\texttt{)} & \text{application} \\
 \\
v & ::= & n & \text{numeral} \\
 & \mid & \texttt{true} \mid \texttt{false} & \text{boolean constant} \\
 & \mid & \texttt{fun}\, f\ \texttt{(}x \texttt{:} \tau\texttt{)}\texttt{:}\tau \,\texttt{is}\, e \,\texttt{end} & \text{function}
\end{array}
$$

The contextual semantics is a particular technique for giving an operational semantics that is based on *evaluation contexts*. The evaluation context C is defined recursively as follows:

$$
\begin{array}{rcl}
C & ::= & [] \\
 & \mid & +(C, e) \\
 & \mid & +(v, C) \\
 & \mid & \texttt{if}\, C \,\texttt{then}\, e \,\texttt{else}\, e \,\texttt{fi} \\
 & \mid & \texttt{apply}\texttt{(}C, e\texttt{)} \\
 & \mid & \texttt{apply}\texttt{(}v, C\texttt{)}
\end{array}
$$

An evaluation context C contains a single hole $[]$, which indicates the place where the current computation is being done. For example, $+([], e)$ says that the first argument of the primitive operator $+$ is currently being evaluated. After we obtain the value v of the first argument, the evaluation context changes to $+(v, [])$. In this way, the evaluation context represents the natural order of computation in order to evaluate an expression. Note that $+(e, [])$ is not a valid evaluation context, which means that the second argument cannot be evaluated first.

1. (5 points)

   Given an expression e, we can produce a new expression e' from an evaluation context C by filling the hole $[]$ in C with e. We write this relation as $C[e] \equiv e'$. For example, we have

$$+([], e_1)[e] \equiv +(e, e_1)$$

   Please define the relation $C[e] \equiv e'$ inductively by a set of rules of the following form:

$$\frac{\ldots}{C[e] \equiv e'}$$

2. (5 points)

   Now we consider a relation R between e and (C, e) defined as follows:

$$e\ R\ (C, e') \quad \Longleftrightarrow \quad C[e'] \equiv e$$

   True or false: R is a partial function. Circle one; you do not need to justify your answer.

3. (5 points)

   The *instruction* i is defined as follows:

$$\begin{array}{rcl} i & ::= & +(v, v) \\ & | & \texttt{if}\, v\, \texttt{then}\, e\, \texttt{else}\, e\, \texttt{fi} \\ & | & \texttt{apply}(v, v) \end{array}$$

   An instruction is an expression that is ready to execute in place.

   The *execution* rules for instructions state how to execute these instructions. Specifically, an execution rule $i \Rightarrow e$ states that the result of executing instruction i is the expression r. We have four execution rules:

$$\begin{array}{lcl} +(n_1, n_2) & \Rightarrow & n_1 + n_2 \\ \texttt{if}\, \texttt{true}\, \texttt{then}\, e_1\, \texttt{else}\, e_2\, \texttt{fi} & \Rightarrow & e_1 \\ \texttt{if}\, \texttt{false}\, \texttt{then}\, e_1\, \texttt{else}\, e_2\, \texttt{fi} & \Rightarrow & e_2 \\ \texttt{apply}(v_1, v_2) & \Rightarrow & [v_1, v_2/f, x]e \end{array}$$

where in the last rule $v_1 = \texttt{fun}\, f\ \texttt{(}x\texttt{:}\tau_1\texttt{)}\texttt{:}\tau_2\ \texttt{is}\ e\ \texttt{end}$.

Now we are going to define the steps of evaluation in terms of evaluation contexts and instructions. The evaluation of e consists of three steps: first, we identify the context and the instruction for e; second, we execute the instruction to produce a new expression r; finally, we replace the instruction by r.

Thus we have a single evaluation rule defined in terms of the instruction execution rules and the context filling relation with the following form:

$$\frac{\cdots}{e \mapsto e'}$$

Please complete the above evaluation rule by filling its premises.

4. (15 points)

The *progress theorem* may be stated using evaluation contexts as follows: if $e : \tau$ then either e value or there exists unique evaluation context C and instruction i such that $C[i] \equiv e$.

The proof is by induction on typing. Please complete the missing parts of the following proof for the case $e = \texttt{apply}(e_1, e_2)$.

By assumption, $e : \tau$. Therefore by inversion

(a) ______________________________

(b) ______________________________

By the inductive hypothesis we have

(a) *either* ______________________________

or ______________________________

(b) *either*

---

or

We proceed by cases on the first inductive hypothesis as follows:

(a) If ____________________ then we proceed by cases on the second inductive hypothesis:

  i. If ____________________ then

  ---

  ii. Otherwise

(b) Otherwise

---

3 Records $\boxed{\textbf{30 Points}}$

In this question we investigate *records*: those are tuples whose components, called *fields*, have labels. Actually tuples can be seen as records with numeric fields $1, 2 \ldots$; while in tuples elements are identified via their position, in records the fields can appear in any order. A record is enclosed in braces and every field has the form `l=e`. For example `type id ={name:string,born:int}`; given a record, we can access one of its component via its label: if `someid` is of type `id`, you can access the value corresponding to the `name` field with `someid.name`. In SML this operation is denoted by `#name(someid)`. This yields the following language, where we assume we are given a denumerable set of labels (not to be confused with locations).

$$\begin{array}{rrcl} \textit{Types} & \tau & ::= & \cdots \mid \{l_1 : \tau_1, \ldots, l_n : \tau_n\} \\ \textit{Expressions} & e & ::= & \cdots \mid \{l_1 = e_1, \ldots, l_n = e_n\} \mid e.l \\ \textit{Values} & v & ::= & \cdots \mid \{l_1 = v_1, \ldots, l_n = v_n\} \end{array}$$

Note that $n \geq 0$, which accounts for the empty record. Records are first-class objects and are evaluated eagerly as any other expression.

1. Part 1 (10 points). Give typing and SOS evaluation rules for this language.

2. Part 2 (15 points) We now want to add the following *operations* on records: record *extension*, $+$, *overriding*, $\oplus$, *subtraction*, $\setminus$, and *concatenation* @ with the following semantics:

   - $r + \{l = e\}$ is defined as the addition of $\{l = e\}$ to r provided that l is a new label.
   - $r \oplus \{l = e\}$ is defined as the overwriting of r with $\{l = e\}$ provided that l is one of the labels in r and its type is consistent.
   - $r \setminus l$ is defined as r minus $\{l = e\}$ provided that l is one of the labels in r and its type is consistent.
   - $r@r'$ the concatenation of r with r', provided no common labels exist, which can be defined in terms of record *extension*.

   Give typing and SOS evaluation rules for those operations.

3. Part 3 (5 points) State the Canonical Form Lemma for record types and the inversion lemma for the record extension operation.

4 Continuations, References and Exceptions **30 Points**

After learning the degree of control that continuations give us over the control stack, it should come as no surprise to learn that we can implement failures and handlers using continuations and mutable storage. In this Standard ML programming problem, we investigate this approach.

You are asked to define two functions, `tryow` and `fail` that will implement failures and handlers in Standard ML *without using the exception mechanism built into SML.* Your code should implement this signature:

```
signature FAILURES = sig
  val tryow : (unit -> 'a) -> (unit -> 'a) -> 'a
  val fail : unit -> 'a
end
```

The semantics of `tryow` is such that `tryow f1 f2` evaluates to the result of calling `f1` on `()` unless a `fail ()` is called during this execution, in which case `tryow f1 f2` evaluates to the result of evaluating `f2 ()`.

As an example, if we define:

```
fun justfail () = fail ()
fun yield4 () = 4
fun failthen4 () = tryow justfail yield4
fun failthenfail () = tryow justfail justfail
```

Then at toplevel we would have the following behavior:

```
val n = tryow justfail yield4     (* and get n = 4 *)
val m = tryow justfail justfail   (* raises UncaughtException *)
val p = tryow failthen4 justfail  (* and get p = 4 *)
val q = tryow failthenfail yield4 (* and get q = 4 *)
```

Your implementation of this signature is to be based on the (H, K, e) machine. (Please consult your notes for the definition of this machine.) You are to maintain a mutable stack of handlers corresponding to the handler stack H of the abstract machine. If `tryow` is called, a handler is pushed onto the stack and the first argument is evaluated. If it returns normally, that value is returned and top handler is removed from the stack. If instead `fail` is called, the top handler is removed from the top of the stack and invoked.

A handler is represented by a `unit cont`, a continuation that, when invoked, performs the following actions:

1. Applies the handler function given by the "otherwise" clause of the call to *tryow* that installed this handler.
2. Returns the result of this clause as the value of the call to `tryow`.

A handler is invoked by throwing `()` to it. The invoked continuation may assume that the top handler has been popped from the stack by the call to `fail` that invoked it!

You may assume that you are given an implementation of handler stacks with the following signature:

```
signature HANDLER_STACK = sig
  (* raised by pop_hdlr if the handler stack is empty *)
  exception UncaughtException
  (* push a handler onto the handler stack *)
  val push_hdlr : unit cont -> unit
  (* remove the top handler from the stack and return it *)
  val pop_hdlr : unit -> unit cont
end

structure HS :> HANDLER_STACK = ...  (elided) ...
```

All manipulation of the handler stack must be through this interface. Note that the handler stack module takes care of uncaught failures arising from attempting to pop an empty handler stack.

1. Part 1 (10 points).

   Define `fail : unit -> 'a`:

   Remember that it is the responsibility of `fail` to pop the handler stack so that if a failure occurs during evaluation of the otherwise function control passes to the nearest enclosing handler (rather than causing a loop!)

   ```
   fun fail () : unit =
   ```

2. Part 2 (20 points).

   Define `tryow : (unit -> 'a) -> (unit -> 'a) -> 'a`:

   Be sure to maintain the invariant that the top of the handler stack is the nearest enclosing handler for the code being executed. Make sure that the handler stack is properly restored after a *tryow* is executed.

   Specifically, `hdlr` should be the continuation that throws the result of evaluating `ow ()` to `ans`. You have been shown code for capturing this kind of continuation in class.

```
fun tryow (try : unit -> 'a) (ow : unit -> 'a) : 'a =
    callcc
    (fn ans : 'a cont =>        (* final answer *)
     let
         (* create the appropriate handler continuation *)
         val hdlr : unit cont =









     in
        (* push the handler continuation, run first argument,
           pop handler on normal return *)









     end)
```

Worksheet

Worksheet