Name: ______________________________

ID number: ______________________________

CS 312, Fall 2002

Exam 2

November 19, 2002

Problem	1	2	3	4	5	Total
Grader						
Grade	$\overline{20}$	$\overline{20}$	$\overline{20}$	$\overline{20}$	$\overline{20}$	

There are 5 problems on this exam. Please check now that you have a complete exam booklet with ??? numbered pages. *Write your name or id number on each page of the exam.* Be sure to try all of the problems, as some are more difficult than others (i.e., don't waste a lot of time on a problem that is giving you a hard time – move on to another problem and then return to it later).

This exam is closed book. No papers, books or notes may be used. However, a handout containing excerpts of the evaluator will be provided.

There is space provided to answer each question. You may request extra paper if necessary.

To help ensure that you don't accidentally miss any of the questions, we have marked those sections where an answer is requested with a $\Longleftarrow$ in the right margin.

The staff reserves the right to ignore illegible answers. "Pretty printing" (proper indentation) of your code will aid in the grading process.

Note: you may *only* use side effects (i.e., `ref`, `!` and `:=`) in problem #1. You are not allowed to use them in any other problem.

1. (20 Points)

   Consider the definition of a binary tree given below:

   ```
   datatype 'a rtree = Leaf | Node of 'a rtree ref * 'a * 'a rtree ref
   ```

   Note that the left and right subtrees are given as references to trees, not as actual trees. The use of references allows us to define data structures other than regular trees on this datastructure. Consider, for example, the (`int rtree`) below. Note that ref cells are shown with an (R) inside them.

   ```
           +---+---+---+
           |   | 1 |   |
           +---+---+---+
             |       |
         +---+       |
         |           |
         V           V
       +---+       +---+
       |(R)|       |(R)|
       +---+       +---+
         |           |
         V           V
   +---+---+---+   +----+
   |   | 2 |   |   |Leaf|
   +---+---+---+   +----+
     |       |
     V       V
   +---+   +---+
   |(R)|   |(R)|
   +---+   +---+
     |       |
     V       V
   +----+  +----+
   |Leaf|  |Leaf|
   +----+  +----+
   ```

   We can modify the tree so that references to Leaf nodes are overwritten to point to the root of the tree:

```
           +---+---+---+
+--------->|   | 1 |   |<--+---+
|          +---+---+---+   |   |
|            |       |     |   |
|        +---+       |     |   |
|        |           |     |   |
|        V           V     |   |
|      +---+       +---+   |   |
|      |(R)|       |(R)|   |   |
|      +---+       +---+   |   |
|        |           |     |   |
|        V           +-----+   |
|  +---+---+---+               |
|  |   | 2 |   |               |
|  +---+---+---+               |
|    |       |                 |
|    V       V                 |
|  +---+   +---+               |
|  |(R)|   |(R)|               |
|  +---+   +---+               |
|    |       |                 |
+----+       +-----------------+
```

Implement the SML function

`closeIt: 'a rtree -> unit`

that performs this operation on a tree. Do not forget to handle special cases. Note that because the function not return any value, the transformation has to occur in place: you must not design a function that builds a tree with the required characteristics, you have to modify the input in place. This means that you will not build new Nodes, you may only change the references that they contain.

```
fun closeIt(tree: 'a rtree):unit =
  let
    fun doClose(t: 'a tree ref):unit =
      case !t of
          Leaf => t := tree
        | Node(l,_,r) => (doClose(l); doClose(r))
  in
    doClose(ref tree)
  end
```

2. (20 Points)

   Recall that SML is applicative order (also called "eager") and uses static scoping. But there can also be variants of ML that are normal order ("lazy"), or that use dynamic scoping, or that use both, for a total of 4 different ML variants.

   Consider the following code:

```
val x = 5
fun f(n) =
  let
    fun g(n) = n + x
    val x = g(n)
  in
    g(n)
  end
```

   In a variant of ML that is eager/static (like SML), what is the value of `f(3)`?

   Ans: 8 $\Longleftarrow$

   In a variant of ML that is eager/dynamic, what is the value of `f(3)`?

   Ans: 11 $\Longleftarrow$

   In a variant of ML that is lazy/static, what is the value of `f(3)`?

   Ans: 8 $\Longleftarrow$

   In a variant of ML that is lazy/dynamic, what is the value of `f(3)`?

   Ans: 11 $\Longleftarrow$

Now consider:

```
val x = ref 0
fun inc(y) = (x := !x + 1; x)
val x = ref 1
fun f(x) = if (!x)*(!x) < 2 then !x else f(x)
```

In a variant of ML that is eager/static (like SML), what (if anything) is the value of `f(inc(x))`?

Ans: 1 $\Longleftarrow$

In a variant of ML that is eager/dynamic, what (if anything) is the value of `f(inc(x))`?

Ans: infinite loop $\Longleftarrow$

In a variant of ML that is lazy/static, what (if anything) is the value of `f(inc(x))`?

Ans: infinite loop $\Longleftarrow$

In a variant of ML that is lazy/dynamic, what (if anything) is the value of `f(inc(x))`?

Ans: infinite loop $\Longleftarrow$

3. (20 points)

Inside the evaluator, the function `evaluate` contains the following code to handle tuples:

```
| Tuple_e elist         =>
    let
      val (v, t) = foldr (fn ((v, t), (vl, tl)) => (v::vl, t::tl))
                         ([],[])
                         (map (fn(e) => evaluate (e, en)) elist)
    in
     (Tuple_v v, Tuple_t t)
    end
```

Suppose that we change the old code for tuples to the new code, which is:

```
| Tuple_e elist         =>
  let
    type acc_t = env * value list * typ list
    fun evaltuple(explst: exp list, (en, vals, types): acc_t): value * typ =
      case explst of
        []     => (Tuple_v vals, Tuple_t types)
      | hd::tl =>
          let
            val en2 = insertBinding("prevs",(Tuple_v vals,Tuple_t types),en)
            val (v, t) = evaluate(hd, en2)
          in
            evaltuple(tl, (en2, vals @ [v], types @ [t]))
          end
  in
    evaltuple(elist, (en, [], []))
  end
```

Give an expression which, when typed into Mini-ML, will give one value with the old code for handling tuples, and a different value with the new code for handling tuples. No other parts of the evaluator have been changed from the handout you have. Also, recall that you may not use side effects.

To get full credit for this problem, you needed to use a tuple where at least after the first element you somehow used the "prevs" variable. Note that you also needed to create something that actually returned a *value* in both cases. For example:

```
let
    val prevs = 1
in
    (1,2,prevs)
end
```

was ok but

```
(1,2,prevs)
```

gives an error when using the old code. If you did something like this, you got partial credit. ⟸

Provide the value that your expression gives with the old code, and the value that your expression gives with the new code.

This will be based on what your answer to the above was. From what we have above (the let example), it will evaluate to

```
(1,2,1)
```

in the old model and

```
(1,2,(1,2))
```

in the new model. ⟸

4. (20 points)

   In this problem you must give the evaluator the capability to count the arguments to a function. `nargs` takes as input a function and returns an integer that tells how many arguments that function takes. Below are some examples of how the Mini-ML evaluator should behave after you have made your changes

```
Mini-ML >> fun sqr(x:int):int = x*x
Mini-ML >> fun foo(x:int,y:int):int = x*y
Mini-ML >> fun bar(x:int,y:int,z:int):int = x*y-z
Mini-ML >> nargs(foo)
2: int
Mini-ML >> nargs(bar)
3: int
Mini-ML >> nargs(fn(x:int,y:int):int => x)
2: int
Mini-ML >> let val f:int->(int->int) = (fn(x:int):int->int => sqr) in
nargs(f) end
1: int
Mini-ML >> let val f:int->((int*int)->int) = (fn(x:int):(int*int)->int
=> (fn(y:int,z:int):int => x+y*z)) in nargs(f(1)) end
2: int
Mini-ML >> let val sqr:int = 3 in nargs(sqr) end
Error: nargs takes a function
Mini-ML >> nargs(fn()=>42)
0: int
```

Be sure to specify **exactly** where you are changing the code, using the line numbers we have supplied on the handout. Also, recall that you may not use side effects.

Add the following code to function 'predefined' in evaluator.sml:

```
| ("nargs", Fn_v(lst, _, _, _), _ )  => (Int_v (List.length lst),
                                          Int_t)
| ("nargs", _,             _    )  => err "narg takes a function"
```

Add this to top_level in environment.sml:

```
("nargs",     Predef_v "nargs",        Fn_t(Fn_t(Undef_t, Undef_t),     ⟸
                                            Int_t)),
```

5. (20 points)

In this problem, you must add the special forms `eval` and `eval3` to Mini-ML. These special forms take as their first inputs expressions that evaluate to strings, which are themselves Mini-ML expressions. To simplify matters, you may assume that the inputs will always be valid, i.e. the string is syntactically valid Mini-ML which will always return a value.

`eval` takes a single argument, and will evaluate the Mini-ML expression in the input string in the current environment, and return its value. The correct behavior of `eval` is shown below.

```
Mini-ML >>let val x: int = 3 in eval("x*(x + 7)") end
30: int
Mini-ML >>let val s:string = "x+39" val x: int = 3 in eval(s) end;
42: int
Mini-ML >> eval("3+(5*7)")
38: int
Mini-ML >> let val x:int = 27 in eval("3+x") end
30: int
```

`eval3` is like `eval`, only it takes 3 arguments, all of which must evaluate to strings. The 2nd argument evaluates to the name of an identifier; let us call that `id`. The 3rd argument is another Mini-ML expression. First of all, we will evaluate this expression in the input environment; let us call the resulting value `v`. Then the 1st argument is evaluated in an environment which contains the input enviroment plus the binding of `id` to `v`.

```
Mini-ML >> eval3("x+3","x","2*5")
13: int
Mini-ML >> let val s:string = "x+5" val x:int = 5 in eval3("x*3","x",s)
end
30: int
```

Be sure to specify **exactly** where you are changing the code, using the line numbers we have supplied on the handout. Also, recall that you may not use side effects.

Eval and Eval3:

Add to top_level environment in environment.sml:

```
(* Evaluates its string argument. *)
("eval",      SpecForm_v "eval",       Fn_t(String_t, Undef_t)),
(* Evaluates the *)
("eval3",     SpecForm_v "eval3",      Fn_t(Tuple_t([String_t,
                                                     String_t,
                                                     String_t]),
                                            Undef_t)),
```

Define function exp2str2exp:

```
fun exp2str2exp (expr: exp, en: env): exp =
  case evaluate(expr, en) of
    (String_v s, String_t) =>
      (case Parser.parseString s of
         NONE => err "incorrect SML expression in 'eval' or 'eval3'"
        | SOME (Exp_t ex) => ex
        | SOME _ => err "no declarations at top level in 'eval' or
'eval3'")
     | _ => err "'eval' and 'eval3' only take string arguments"
```

Add to special forms in evaluator.sml:

```
      (*
         eval(s) - returns the value of string s, considered an SML
         expression, in the current environment.
      *)
    | "eval" => evaluate(exp2str2exp(expr, en), en)
      (*
        eval(s1, s2, s3)
        1. Evaluates s2 in the current environment to a string s
        representing a variable name.
        2. Evaluates s3 in the current environment to value v.
        3. Evaluates s1 in the current environment extended w/ the
        binding (s, v).
      *)
    | "eval3" =>
        (case expr of
           Tuple_e([s1, s2, s3]) =>
             (case evaluate(s2, en) of
                (String_v s, String_t) =>
                  evaluate(exp2str2exp(s1, en),
                           insertBinding(s,evaluate(exp2str2exp(s3,en),en),en))
              | _ => err "second argument of 'eval3' must evaluate to
string")
         | _ => err "'eval3' needs three arguments")
```

One other solution for eval3 acutally is to treat it in terms of eval by doing a let, basically translating "eval3(e1,e2,e3)" into "eval("let val " ^s2 ^"=" ^s3 ^" in " ^s1 ^" end")". Or in other words,

```
| "eval3" =>
    (case evaluate(expr, en) of
      (Tuple_v [String_v s1, String_v s2, String_v s3], _) =>
         (*
           Alternatively, we could call parseString on "let val ...
end",
           extract the AST that the parser returns, then evaluate the
AST
           in the current environment.
         *)
         specialForm("eval",
                     String_c("let val " ^ s2 ^ ": undefined = "
                                       ^ s3 ^ " in " ^ s1 ^ " end"),
en)
     | _ => err "'eval3' takes three strings")
```

Notes:

(a) We can't use string or character constants in eval: we can't escape quotes. It does not matter for the basic idea, but I thought I would point it out.

(b) Function exp2str2exp must have evaluate in scope. It can be defined after evaluate, or using 'and.'

(c) There is nothing conceptually that says that we cannot write eval and eval3 as a predefined function since eval and eval3 must evaluate their arguments regardless. However, since eval and eval3 need the current environment to evaluate the resulting strings, we would also need to modify the predefined function to take in an environment. $\Longleftarrow$