

CS 331 – Assignment 7  
Due: Tuesday, April 29<sup>th</sup>, 2008 - 8:00AM  
Total points: 100

This assignment breaks down into two parts. In Part 1, you will be introducing new features into our interpreted language. In Part 2, you will be working with infinite sequences directly in ML, *not* within the interpreter.

Start by picking up the tarball `a7.tar.gz` from `/shared/furcyd/cs331/` and expand it using the command: `tar -zxvf a7.tar.gz`. The resulting directory, called `a7`, contains the file `a7.sml` (the file you need for Part 2) and a directory called `appliedLC` containing these files (which you need for Part 1):

- The file `absyn.sml` describes the structure of your language's abstract syntax as an ML datatype.
- The file `env.sml` defines the data structures and utilities needed to maintain the runtime environment.
- The file `lambda.lex` describes the lexical structure of your language to be fed to ML-Lex.
- The file `lambda.grm` describes the grammatical structure of your language to be fed to ML-Yacc.
- The file `interpreter.sml` implements the main functions of our interpreter.
- The file `sources.cm` controls ML's `CM.make()` "compilation" process. You need not look at this file.
- The files `lambda_calc.sml`, `errormsg.sml`, and `top.sml` deal with user-interface issues. You need not look at these files.
- The files `slide15_10.pl`, `slide16_5.pl`, `rec1.pl`, and `rec2.pl` are the test files mentioned in this handout.

To submit your work for this assignment, deposit **exactly** the following files in a directory called lowercase `a7` at the top level of your dropbox:

- `a7.sml` – with your modifications
- a sub-directory called `appliedLC` containing exactly 13 files, namely:
  - `absyn.sml` – with your modifications
  - `lambda.lex` – with your modifications
  - `lambda.grm` – with your modifications
  - `interpreter.sml` – with your modifications
  - `env.sml` – with your modifications
  - all other files in the tarball – unchanged

I will test your `appliedLC` interpreter at the SML prompt by typing `CM.make "sources.cm" ;` and then interpreting test cases with the concrete syntax illustrated in the samples given below. Be sure that your project compiles successfully when I do this. *Important:* If there are problems that you did not successfully complete, turn in your project for those problems where the interpreter does work and submit a hard copy statement in which you explain what goes wrong when you try to run the interpreter on the problems where it doesn't work. In that situation you should also submit a hard copy of your source code for those problems, but do not electronically submit that non-working code in your dropbox.

You must also submit, **in hard copy**, the file `a7.sml`. **If your code for Part 1 is working properly, you need NOT submit your code for the interpreter. However, everyone must submit a single page stating which option you chose for problem 7 (see below).**

**Part 1: Interpreter for the "applied" lambda calculus**

The version of the interpreter in the tarball that you downloaded handles several new features of the interpreted language, most of which you had to implement for assignment 6 (the others were discussed in class). These new features include:

- **Conditional expressions** of the form `if ... then ... else ...`.
- **Boolean values**, together with equality testing (`==`) and comparators (`<` and `>`). Note that, like in the previous assignment, the boolean literals `true` and `false` are *not* part of the language. Instead, booleans are used as values for the first argument of conditional expressions.
- **Primitive operators** are now split into unary (`add1` and `~`) and binary operators (`+`, `*`, `-`, `div`, and `mod`). Binary operators use parenthesized infix notation, e.g. `(~(x) div (y * 2))`.
- **Let expressions**. Note that, unlike in the previous assignment, `let` blocks:
  - contain one or more bindings,
  - have their own abstract syntax in the form of a `let_exp` data constructor (see the file `absyn.sml`), and
  - use static scoping.
- **Sequencing expressions**. In order to handle sequences of expressions, a `block_exp` data constructor was added to `absyn.sml`. The corresponding non-terminal in the grammar is called `BlockExp`. Such a block expression contains a sequence of one or more expressions separated by semi-colons. As you can see in the file `lambda.grm`, there are no semi-colon after the last expression in a block. Furthermore, a block expression may only appear as the body of a `let` expression.

```

let
  p = 3
in
  let
    foo = fn (h,o,m) =>
      let
        dummy = 1
      in
        print "in_foo";
        set o = p;
        set h = (m + p);
        set p = m;
        set m = (p + m);
        print "h" h;
        print "o" o;
        print "m" m
      end
    in
      let
        main = fn () =>
          let
            b = 5
            c0 = 6
            c1 = 1
            c2 = 3
            c3 = 5
          in
            (foo b b c0);
            print "in_main";
            print "b" b;
            print "c0" c0;
            print "c1" c1;
            print "c2" c2;
            print "c3" c3
          end
        in
          (main);
          print "after_main";
          print "p" p
        end
      end
    end
  end
end

```

slide15\_10.pl

```

let
  m0 = 2   m1 = 3   m2 = 8   m3 = 5
  b = 8
in
  let
    foo = fn (m,p) =>
      let
        d = 8
      in
        print "in_foo";
        set p = (b + m);
        set b = p;
        set m = (b + p);
        set d = (m + p);
        print "d" d;
        print "m" m;
        print "p" p
      end
    in
      let
        b = 5
        j0 = 7   j1 = 9   j2 = 10   j3 = 9
      in
        (foo b j3);
        print "in_main";
        print "b" b;
        print "j0" j0;
        print "j1" j1;
        print "j2" j2;
        print "j3" j3
      end
    end;
    print "globals";
    print "m0" m0;
    print "m1" m1;
    print "m2" m2;
    print "m3" m3;
    print "b" b
  end
end

```

slide16\_5.pl

- **References.** The runtime environment now binds variables to references (to values), instead of binding variables directly to values. The file `env.sml` was modified accordingly, as discussed in class.
- **Assignment expressions.** These expressions correspond to standard assignment statements and return the value of their right-hand side.
- **Print expressions.** There are three types of `print` expressions in our language. While all three of them use the “print” keyword and print their argument(s) on one line of output, they are distinguished by the number and types of their argument(s).
  - A `print1_exp` expression is followed by a single expression that must evaluate to an integer literal.
  - A `print2_exp` expression is followed by a single expression, namely a double-quoted variable, that is printed literally (that is, *not* looked up in the environment). Recall that a variable in our language is a letter followed by zero or more letters, digits, or underscores.
  - A `print3_exp` expression is followed by a double quoted string and an expression that must evaluate to an integer, and prints both of them separated by a white-space character.

Most of the new features just described are illustrated in the two sample programs shown at the top of this page. Each program is a rendition in our interpreted language of the sample problem on the corresponding slide. Before you start working on this assignment, make sure to study the code handout fully so that you fully understand:

- How the new features are implemented,
- Why function calls use the call-by-value mechanism, and
- Why function definitions may *not* be recursive.

Your task in this assignment is to implement the other five parameter-passing mechanisms we discussed in class, as well as recursive functions.

The output produced when interpreting each of these sample programs is shown on the next page. Make sure to understand each line of output given that the parameter-passing mechanism used in the code handout is pass-by-value.

```

- Top.eval "slide15_10.pl" "byvalue";
in_foo
h 9
o 3
m 12
in_main
b 5
c0 6
c1 1
c2 3
c3 5
after_main
p 6
val it = Num 6 : Env.value

```

```

- Top.eval "slide16_5.pl" "byvalue";
in_foo
d 39
m 26
p 13
in_main
b 5
j0 7
j1 9
j2 10
j3 9
globals
m0 2
m1 3
m2 8
m3 5
b 13
val it = Num 13 : Env.value

```

Every time you implement a new parameter-passing mechanism, you will add it to the language without modifying anything else. In order to tell the interpreter which parameter-passing mechanism to use, you now need to pass one more argument to the `Top.eval` function, as shown in the sample sessions. Its second argument is a string whose possible values are given in the file `interpreter.sml` as well as in problems 1, 2, 4, 5, and 6 below.

**Important note:** For each of the five parameter-passing mechanisms you have to implement, you can assume that each function call has the form:  $(f\ a_1\ a_2\ \dots)$ , where  $f$  is an expression that evaluates to a closure, and  $a_1, a_2, \dots$ , are always variable expressions bound to integer values. You need not consider any other types of arguments (such as function calls or variables bound to closures, etc.), nor should you worry about the “match nonexhaustive” warnings that ML may report.

- (10 points) **Pass by reference.** To implement this mechanism, add the pattern “byref” to the `Absyn.app_exp` case in the `eval_exp` function. When this pattern is matched, simply call a new function called `apply_byref` that you must implement below the `apply_byvalue` function. Note that you must connect your new function’s definition to the previous ones with the `and` keyword. Your new function should first evaluate the function expression, then look up each of the variable expressions, and finally evaluate the body of the function in the appropriate environment. For this problem, you only need to modify the `interpreter.sml` file as was just described. Sample sessions:

```

- Top.eval "slide15_10.pl" "byref";
in_foo
h 9
o 9
m 12
in_main
b 9
c0 12
c1 1
c2 3
c3 5
after_main
p 6
val it = Num 6 : Env.value

```

```

- Top.eval "slide16_5.pl" "byref";
in_foo
d 39
m 26
p 13
in_main
b 26
j0 7
j1 9
j2 10
j3 13
globals
m0 2
m1 3
m2 8
m3 5
b 13
val it = Num 13 : Env.value

```

- (10 points) **Pass by copy-restore.** To implement this mechanism, add the pattern “bycopy” to the `Absyn.app_exp` case in the `eval_exp` function. When this pattern is matched, simply call a new function called `apply_bycopy` that you must implement below the `apply_byref` function. Note that you must connect your new function’s definition to the previous ones with the `and` keyword. Your new function should first evaluate the function expression, look up each of the variable expressions and make copies of their values (copy-in phase), and finally evaluate the body of the function in the appropriate environment and copy-out the resulting values. For this problem, you only need to modify the `interpreter.sml` file as was just described. Sample sessions:

```

in_foo
h 9
o 3
m 12
in_main
b 3
c0 12
c1 1
c2 3
c3 5
after_main
p 6
val it = Num 6 : Env.value

```

```

- Top.eval "slide16_5.pl" "bycopy";
in_foo
d 39
m 26
p 13
in_main
b 26
j0 7
j1 9
j2 10
j3 13
globals
m0 2
m1 3
m2 8
m3 5
b 13
val it = Num 13 : Env.value

```

3. (10 points) Write, in the file `interpreter.sml`, a new curried function called *substitute* that takes in a list of expressions (these will be the arguments in a function call), a list of strings (these will be the names of the formal parameters in a function definition), and an expression (the body of a function), and returns the body in which each argument has been substituted for the corresponding parameter. Sample usage:

```

- Interpreter.substitute [Absyn.var_exp "x", Absyn.var_exp "y"]
  ["a", "b"]
  (Absyn.prim2app_exp( Absyn.add,
                      Absyn.var_exp "a",
                      Absyn.var_exp "b"));

val it = prim2app_exp (add, var_exp "x", var_exp "y") : Absyn.exp

```

Note that the body of the function (in which the substitutions take place) could be any kind of expression. Finally, the *substitute* function will be used in the next two problems.

4. (10 points) **Pass by macro-expand.** To implement this mechanism, add the pattern “`bymacro`” to the `Absyn.app_exp` case in the `eval_exp` function. When this pattern is matched, simply call a new function called `apply_bymacro` that you must implement below the `apply_bycopy` function. Note that you must connect your new function’s definition to the previous ones with the `and` keyword. Your new function should first evaluate the function expression, and then evaluate the body of the function in the caller’s environment after having performed the required substitutions. Note that this implementation of macro expansion only performs one substitution, since it simulates the second substitution that we discussed in class using dynamic scoping. For this problem, you only need to modify the `interpreter.sml` file as was just described. Sample sessions:

```

in_foo
h 9
o 9
m 12
in_main
b 9
c0 12
c1 1
c2 3
c3 5
after_main
p 6
val it = Num 6 : Env.value

```

```

- Top.eval "slide16_5.pl" "bymacro";
in_foo
d 30
m 20
p 10
in_main
b 20
j0 7
j1 9
j2 10
j3 10
globals
m0 2
m1 3
m2 8
m3 5
b 8
val it = Num 8 : Env.value

```

5. (10 points) **Pass by name.** To implement this mechanism, add the pattern “byname” to the `Absyn.app_exp` case in the `eval_exp` function. When this pattern is matched, simply call a new function called `apply_byname` that you must implement below the `apply_bymacro` function. Note that you must connect your new function’s definition to the previous ones with the `and` keyword. Your new function should first evaluate the function expression, then freeze all the arguments (see below for a discussion of the thunk-related utilities), and finally evaluate the body of the function in the appropriate environment.

For this parameter-passing mechanism (and the next), the runtime environment needs to handle a new type of values, namely *thunks*. A thunk is a value tagged *Frozen*. To build and thaw thunks, two utility functions, called *freeze* and *thaw* respectively, were added to the file `env.sml`. A thunk is simply an anonymous function, together with the environment in which the thunk was built (that is, the caller’s environment).

For this problem, you need to modify both the `interpreter.sml` file as described above and the `env.sml` file, since the `lookupReference` function now needs to consider two types of looked-up values: thunks, that must be thawed, and other values, that are returned directly, as before. Sample sessions:

```

in_foo
Thawing b
Thawing c0
Thawing b
Thawing c0
Thawing c0
Thawing c0
Thawing b
Thawing b
h 9
Thawing b
o 9
Thawing c0
m 12
in_main
b 9
c0 12
c1 1
c2 3
c3 5
after_main
p 6
val it = Num 6 : Env.value

- Top.eval "slide16_5.pl" "byname";
in_foo
Thawing b
Thawing j3
Thawing j3
Thawing j3
Thawing b
Thawing b
Thawing j3
d 39
Thawing b
m 26
Thawing j3
p 13
in_main
b 26
j0 7
j1 9
j2 10
j3 13
globals
m0 2
m1 3
m2 8
m3 5
b 13
val it = Num 13 : Env.value

```

6. (10 points) **Pass by need.** One disadvantage of the call-by-name mechanism is that the same thunk may be thawed multiple times, more precisely, as many times as the corresponding parameter appears in the body of the function. This inefficiency is illustrated (using a “print” statement inserted in the *thaw* function) in the sample sessions for the previous problem. To eliminate this inefficiency, another parameter-passing mechanism called “pass-by-need” only thaws each thunk once and then caches (i.e., remembers) its value in case the thunk is needed again later. To implement call-by-need, I have added a new type of value tagged “Thawed” in the file `env.sml`. All you have to do to implement call-by-need is to:

- add the pattern “byneed” to the `Absyn.app_exp` case in the `eval_exp` function,
- call the existing function `apply_byname` when this pattern is matched (no new function is needed here since call-by-need is built on top of call-by-name), and
- change the `lookupReference` function so that the first time a thunk is thawed, it is replaced in the environment by a thawed reference to its value.

The last step is as simple as adding an assignment statement. To determine when the `lookupReference` function is called under call-by-name versus call-by-need, simply use the boolean value “!ByNeed” which is automatically set to true/false when using call-by-need/name. As a result of your changes for this part, the interpreter should give you the same result as for call-by-name, but it should only thaw each thunk once, as in the following sample sessions:

```

- Top.eval "slide15_10.pl" "byneed";
in_foo
Thawing b
Thawing c0
Thawing b
h 9
o 9
m 12
in_main
b 9
c0 12
c1 1
c2 3
c3 5
after_main
p 6
val it = Num 6 : Env.value

```

```

- Top.eval "slide16_5.pl" "byneed";
in_foo
Thawing b
Thawing j3
d 39
m 26
p 13
in_main
b 26
j0 7
j1 9
j2 10
j3 13
globals
m0 2
m1 3
m2 8
m3 5
b 13
val it = Num 13 : Env.value

```

7. (10 points + 5 bonus points) In this problem, you must add a new version of the “let block” that handles recursive functions. The new concrete syntax will be identical to the regular “let block” but for the use of the `letrec` keyword (instead of plain `let`). After modifying the `lambda.lex` and `lambda.grm` files accordingly, you must add a new `letrec_exp` to the abstract syntax of the language. Finally, you will add the corresponding pattern and code in the interpreter.

Of course, you will implement recursion using the technique called “tying the knot” that we discussed in class. Recall that for each recursive function, you must first create a reference to a dummy closure, then create the actual closure containing a reference to the dummy one, and finally tie the knot by making the reference to the dummy closure point to the actual closure.

For full credit (i.e., 10 points) you may complete this problem under the simplifying assumption that the `letrec` expression is used to declare a single recursive function, as in the case of *factorial* below.

Alternatively, **instead of solving the problem as just described**, you have the option of tackling a more interesting problem, for an additional 5 bonus points. We say that  $n$  functions are *mutually recursive* if there is a chain of function calls along which each function ends up calling itself. In this case, the recursion is indirect since neither function calls itself directly. Hint: To implement “tying-the-knot” in this scenario will require you to create a list of references to a dummy closure. An example with  $n = 2$  is shown in the second session below, where the contents of the file `rec1.pl` are:

```

letrec
  fact = fn (n) => if (n == 0) then 1 else (n * (fact (n-1)))
in
  (fact 5)
end

```

and the contents of the file `rec2.pl` are:

```

letrec
  fact = fn (n) => if (n == 0) then 1 else (n * (fact (n-1)))
  odd  = fn (n) => if (n == 0) then 0 else (even (n-1))
  even = fn (n) => if (n == 0) then 1 else (odd  (n-1))
in
  print (fact 5);
  print (odd  11);
  print (even 10);
  print (odd  10);
  print (even 11)
end

```

```

- Top.eval "rec1.pl" "byvalue";
val it = Num 120 : Env.value

```

```

- Top.eval "rec2.pl" "byvalue";
120
1
1
0
0
val it = Num 0 : Env.value

```

### Important notes:

- This problem is independent of the previous problems and I will only test your solution under the call-by-value semantics.
- Your hard copy submission must clearly indicate which option you chose for this problem: option 1 (single recursive function) or option 2 (mutually recursive functions).

## Part 2: Manipulating infinite sequences in ML

8. (10 points) Write an ML function called *add\_adj* that takes in an infinite sequence  $[x_1, x_2, x_3, x_4, \dots]$  and returns the infinite sequence  $[x_1 + x_2, x_3 + x_4, \dots]$  formed by adding consecutive pairs of elements in the input sequence. Sample usage:

```
- Seq.take (add_adj (Seq.from 1)) 10;  
val it = [3,7,11,15,19,23,27,31,35,39] : int list
```

9. (10 points) Write an ML function called *interleave* that interleaves two input sequences to produce a single sequence, as shown in the following session:

```
- Seq.take (interleave (Seq.from 100) primes) 16;  
val it = [100,2,101,3,102,5,103,7,104,11,105,13,106,17,107,19] : int list
```

10. (10 points) Consider the following ML data type for infinite binary trees:

```
datatype 'a inf_bintree =  
  Node of 'a * (unit -> 'a inf_bintree) * (unit -> 'a inf_bintree);
```

Write an ML function called *make\_tree* that takes in an integer  $n$  and returns an infinite binary tree whose root node has value  $n$ , and whose left and right subtrees are *make\_tree*( $2n$ ) and *make\_tree*( $2n+1$ ), respectively. Finally, write an ML function called *preorder* that takes in a tree and returns the infinite sequence of the node values when the tree is visited in preorder. Sample usage:

```
- Seq.take (preorder (make_tree 1)) 15;  
val it = [1,2,3,4,6,5,7,8,12,10,14,9,13,11,15] : int list
```

Hint: Use your *interleave* function from the preceding problem. In preorder traversal, each node is always visited before its children are. This remains true here. However, in this variant of preorder traversal, the nodes at a given level of the tree are not necessarily visited from left to right.