Homework 3

Due 28 Feb

Handout 13
CSCI 334: Spring 2018

Notes

This homework has three types of problems:

Self Check: You are strongly encouraged to think about and work through these questions, and may do so with a partner. Do not submit answers to them.

Problems: You will turn in answers to these questions.

Pair Programming: This part involves writing ML code. You are required to work with a partner for this section. I will assume that you plan to work with the same partner you worked with on Homework #2 unless you email me (dbarowy@cs.williams.edu) with your partner's name **by the evening of Wednesday, Feb 21**.

Turn-In Instructions

You must turn in your work for this assignment in two parts, one for the Problems part and one for the Pair Programming part. You will be assigned two GitHub repositories.

Note that this assignment should be anonymized: neither your name nor your partner's name should appear in any of the files submitted with this assignment except the file "`collaborators.txt`" (see below).

Problems: Problem sets should be typed up using LaTeX and submitted using your `hw3-<username>` repository. For example, if your GitHub username is `dbarowy`, then your repository will be called `hw3-dbarowy`. Be sure that your work is committed and pushed to your repository by Wednesday, February 28 at 11:59pm.

If you discuss the problem set with your partner or with a study group, please be sure to include their names in a `collaborators.txt` file in your repository.

Pair Programming: Programming solutions should be typed up and submitted using your partner repository. For example, if your GitHub username is `dbarowy` and you are working with a partner whose username is `wjannen`, look for a repository called `hw3-dbarowy-wjannen` (usernames will be in alphabetical order). Be sure that your work is committed and pushed to your repository by Wednesday, February 28 at 11:59pm.

Standard ML files should have a "`.sml`" suffix. For example, the pair programming problem **P1a** should appear as the file `p1a.ml`. Your code should be documented using comments. Comment lines appear inside "`(*`" and "`*)`" braces.

Reading

1. **(Required)** Read Mitchell, skim 4.4–4.5, 5, skim 6.1.
2. **(As necessary)** Read *A Gentle Introduction to ML* by Andrew Cumming.
3. **(Optional)** J. Backus, *Can Programming be Liberated from the von Neumann Style?*, Comm. ACM 21, 8 (1978) 613-641. You can find this on the CSCI 334 web site.

Self Check

S1. ..... ML Types

Explain the ML type for each of the following declarations:

(a) `fun a(x,y) = x+2*y;`

(b) `fun b(x,y) = x+y/2.0;`

(c) `fun c(f) = fn y => f(y);`

(d) `fun d(f,x) = f(f(x)));`

(e) `fun e(x,y,b) = if b(y) then x else y;`

Since you can simply type these expressions into an ML interpreter to determine the type, be sure to understand why the interpreter derives a given type.

Problems

Q1. (20 points) ..... Lazy Evaluation and Parallelism

In a "lazy" language, a function call `f(e)` is evaluated by passing the unevaluated argument to the function body. If the value of the argument is needed, then it is evaluated as part of the evaluation of the body of `f`. For example, consider the function `g` defined by

```
fun g(x,y) =
  if
    x = 0
    then 1
  else if
    x + y
    = 0
    then 2
  else
    3;
```

Assume, in the above function, that each separate line can be entirely performed in a single step (i.e., in a single tick of the CPU's clock).

In a lazy language, the call $g(3, 4+2)$ is evaluated by passing some representation of the expressions 3 and $4+2$ to `g`. The test `x = 0` is evaluated using the argument 3. If `x = 0` were `true`, the function would return 1 without ever computing $4+2$. Since the test is `false`, the function must evaluate `x + y`, which now causes the actual parameter $4+2$ to be evaluated. Some examples of lazy functional languages are Miranda, Haskell and Lazy ML; these languages do not have assignment or other imperative features with side effects. In other words, they are "pure" functional languages.

If we are working in a pure functional language, then for any function call $\texttt{f}(e_1, e_2)$, we can evaluate $\texttt{e}_1$ before $\texttt{e}_2$ or $\texttt{e}_2$ before $\texttt{e}_1$. Since neither can have side-effects, neither can affect the value of the other. However, if the language is lazy, we might not need to evaluate both of these expression. Therefore, if something goes wrong in the evaluation of both expressions (e.g., one expression does not terminate), the entire program itself my still terminate.

As Backus argues in his Turing Award lecture, an advantage of pure functional languages is the possibility of parallel evaluation. For example, in evaluating a function call $\texttt{f}(e_1, e_2)$ we can evaluate both $\texttt{e}_1$ and $\texttt{e}_2$ in parallel. In fact, we could even start evaluating the body of `f` in parallel as well.

In the questions below, I suggest that you draw a table representing all concurrent threads of execution. Each row should represent what the CPU executes in a single step. Each column should represent a separate thread. When a thread halts (because there is nothing left to compute), simply repeat the value in the thread's column until all other threads have halted. For example,

g	e_1	e_2
`if`	3	$4+2$
...	...	...

(a) (5 points) Assume we evaluate g(e_1, e_2) by starting to evaluate g, $\texttt{e}_1$, and $\texttt{e}_2$ in parallel, where g is the function defined above. Is it possible that one process will have to wait for another to complete? How can this happen? If it can happen, demonstrate by showing a sample table.

(b) (5 points) Suppose the value of $\texttt{e}_1$ is zero and evaluation of $\texttt{e}_2$ terminates with an error. In the normal (i.e., eager) evaluation order that is used in C, Java, and other common languages, evaluation of the expression g(e_1, e_2) will terminate in error. What will happen with lazy evaluation? Parallel evaluation? Again, draw sample tables, keeping in mind that lazy evaluation only has a single thread (i.e., your table should only have one column).

(c) (6 points) Suppose you want the same value, for every expression, as lazy evaluation, but you want to evaluate expressions in parallel to take advantage of your new pocket-sized multiprocessor. What actions should happen, if you evaluate g(e_1, e_2) by starting g, $\texttt{e}_1$, and $\texttt{e}_2$ in parallel, if the value of $\texttt{e}_1$ is zero and evaluation of $\texttt{e}_2$ terminates in an error?

(d) (4 points) Suppose, now, that the language contains side-effects. What if $\texttt{e}_1$ is z, and $\texttt{e}_2$ contains an assignment to z. Can you still evaluate the arguments of g(e_1, e_2) in parallel? If you think you cannot, provide a table with a counterexample.

Q2. (5 points) ..... Algol 60 Procedure Types

In Algol 60, the type of each formal parameter of a procedure must be given. However, proc is considered a type (the type of procedures). This is much simpler than the ML types of function arguments. However, this is really a type loophole; since calls to procedure parameters are not fully type checked, Algol 60 programs may produce run-time type errors.

Write a procedure declaration for `Q` which causes the following program fragment to produce a run-time type error.

```
proc P (proc Q)
   begin Q(true) end;
P(Q);
```

where `true` is a boolean value. Explain why the procedure is statically type correct, but produces a run-time type error. (You may assume that adding a boolean to an integer is a run-time type error.)

Q3. (10 points) ..... Translation into Lambda Calculus

A programmer is having difficulty debugging the following Python program. In theory, on an "ideal" machine with infinite memory, this program would run forever. In practice, this program crashes because it runs out of memory, since extra space is required every time a function call is made.

```
def f(g):
    g(g)

def mymain():
  x = f(f)
  print x
```

Explain the behavior of the program by translating the definition of `f` into lambda calculus and then reducing the application `f(f)`. Note that an equivalent program in a statically typed language like Java or ML would not compile.

Pair Programming

For these problems, use the ML interpreter on the Unix machines in the computer lab. You can run ML on the file "example.sml" as follows:

```
sml < example.sml
```

at the command line. As with Lisp, the ML compiler will process the program in the file and print the result. For example, if "example.sml" contains

```
(* double an integer *)
fun double (x) = x * x;

(* return the length of a list *)
fun listLength (nil) = 0
  | listLength (l::ls) = 1 + listLength ls
  ;

double (10);
listLength (1::[2,3,4]);
```

the command "`sml < example.sml`" will produce the following:

```
val double = fn : int -> int
val listLength = fn : 'a list -> int
val it = 100 : int
val it = 4 : int
```

You can also run "sml" and enter in declarations and expressions to evaluate at the prompt.

Start early on this part so you can see the TA or me if you have problems understanding the language. There are many valuable resources available to help you learn ML:

- The examples in the Mitchell book and in your notes.
- Ullman's Elements of ML Programming book. I will leave several copies in the Unix lab for your reference. **Do not remove these books from the lab.**
- Several very good tutorials listed on the course web page.

A few additional details:

- Emacs on the Unix machines will provide auto-formatting and syntax highlighting while editing ML files. **Be sure your file names end with ".sml" so Emacs can recognize them as containing ML code.**
- Comments in ML are delineated by `(*` and `*)`.
- Put the following line at the top of your ML files to ensure that large data types and lists are fully printed:

  ```
  Control.Print.printDepth := 100;
  Control.Print.printLength := 100;
  ```

- Unless otherwise specified, you should use pattern matching where possible.
- There are several thought questions in the descriptions below. Please answer these questions with your partner in comments in the code.

P1. (10 points) ..... Basic Functions in ML

(a) Define a function `sumSquares` that, given a nonnegative integer n, returns the sum of the squares of the numbers from 1 to n:

```
- sumSquares(4)
val it = 30 : int
- sumSquares(5)
val it = 55 : int
```

(b) Define a function `listDup` that takes an element, e, of any type, and a non-negative number, n, and returns a list with n copies of e:

```
- listDup("moo", 4);
val it = ["moo","moo","moo","moo"] : string list
- listDup(1, 2);
val it = [1,1] : int list
- listDup(listDup("cow", 2), 2);
val it = [["cow","cow"],["cow","cow"]] : string list list
```

(c) Your `listDup` will have a type like `'a * int -> 'a list`. What does this type mean and why is it the appropriate type for your function? Answer this question as a comment in the code.

P2. (10 points) ..... Zipping and Unzipping

(a) Write the function `zip` to compute the product of two lists of arbitrary length. You should use pattern matching to define this function:

```
- zip [1,3,5,7] ["a","b","c","de"];
val it = [(1,"a"),(3,"b"),(5,"c"),(7,"de")]: (int * string) list
```

Note: This is the curried version with type `'a list -> 'b list -> ('a * 'b) list`. Be sure to define it to match this type.

When one list is longer than the other, repeatedly pair elements from the longer list with the last element of the shorter list.

```
- zip [1,3,5,7] ["a","b"];
val it = [(1,"a"),(3,"b"),(5,"b"),(7,"b")]: (int * string) list
```

In the event that one or both lists are completely empty, return the empty list.

```
- zip [1,3,5,7] [];
stdIn:27.1-27.17 Warning: type vars not generalized because of
   value restriction are instantiated to dummy types (X1,X2,...)
val it = [] : (int * ?.X1) list
```

In this last case, you will receive a warning like the one above since ML cannot determine the type of the element of an empty list.

(b) Write the inverse function, `unzip`, which behaves as follows:

```
- unzip [(1,"a"),(3,"b"),(5,"c"),(7,"de")];
val it = ([1,3,5,7], ["a","b","c","de"]): int list * string list
```

(c) Write `zip3`, to zip three lists.

```
- zip3 [1,3,5,7] ["a","b","c","de"] [1,2,3,4];
val it = [(1,"a",1),(3,"b",2),(5,"c",3),(7,"de",4)]: (int * string * int) list
```

You must use `zip` in your definition of `zip3`.

(d) Why can't you write a function `zip_any` that takes a list of any number of lists and zips them into tuples? From the first part of this question it should be pretty clear that for any fixed n, one can write a function `zipn`. The difficulty here is to write a single function that works for all n. In other words, can we write a single function `zip_any` such that `zip_any [list1,list2,...,listk]` returns a list of k-tuples no matter what k is? Answer this question as a comment in the code.

P3. (10 points) ..... Find

Write a function `find` with type `'a * 'a list -> int` that takes a pair of an element and a list and returns the location of the first occurrence of the element in the list. For example:

```
- find(3, [1, 2, 3, 4, 5]);
val it = 2 : int
- find("cow", ["cow", "dog"]);
val it = 0 : int
- find("rabbit", ["cow", "dog"]);
val it = ~1 : int
```

Suggestion: first write a definition for `find` where the element is guaranteed to be in the list. Then, modify your definition so that it returns `~1` when the element is not in the list.

P4. (10 points) ..... Trees

Here is the datatype definition for a binary tree storing integers at the leaves:

```
datatype IntTree = LEAF of int | NODE of (IntTree * IntTree);
```

(a) Write a function `sum:IntTree -> int` that adds up the values in the leaves of a tree:

```
- sum(LEAF 3);
val it = 3 : int
- sum(NODE(LEAF 2, LEAF 3));
val it = 5 : int
- sum(NODE(LEAF 2, NODE(LEAF 1, LEAF 1)));
val it = 4 : int
```

(b) Write a function `height: IntTree -> int` that returns the height of a tree:

```
- height(LEAF 3);
val it = 1 : int
- height(NODE(LEAF 2, LEAF 3));
val it = 2 : int
- height(NODE(LEAF 2, NODE(LEAF 1, LEAF 1)));
val it = 3 : int
```

(c) Write a function `balanced: IntTree -> bool` that returns true if a tree is balanced (ie, both subtrees are balanced and differ in height by at most one). You may use your `height` function in the definition of `balanced`.

```
- balanced(LEAF 3);
val it = true : bool
- balanced(NODE(LEAF 2, LEAF 3));
val it = true : bool
- balanced(NODE(LEAF 2, NODE(LEAF 3, NODE(LEAF 1, LEAF 1))));
val it = false : bool
```

(d) What is non-optimal about using the `height` function in the definition of `balanced`? Can you suggest a more efficient implementation? You need not write code, but describe in a sentence or two how you would do this. Answer this question as a comment in the code.

P5. (8 points) ..... Challenge Problem

Certain programming languages (and HP calculators) evaluate expressions using a stack. We are going to implement a simple evaluator for such a language. Computation is expressed as a sequence of operations, which are drawn from the following data type:

```
datatype OpCode =
    PUSH of real
  | ADD
  | MULT
  | SUB
  | DIV
  | SWAP
  ;
```

The operations have the following effect on the operand stack. (The top of the stack is shown on the left.)

OpCode	Initial Stack	Resulting Stack
`PUSH(r)`	...	r ...
`ADD`	a b ...	(b + a) ...
`MULT`	a b ...	(b * a) ...
`SUB`	a b ...	(b - a) ...
`DIV`	a b ...	(b / a) ...
`SWAP`	a b ...	b a ...

The stack may be represented using a list for this example, although we could also define a stack data type for it.

```
type Stack = real list;
```

Write a recursive evaluation function with the signature

```
eval : OpCode list * Stack -> real
```

It takes a list of operations and a stack. The function should perform each operation in order and return what is left in the top of the stack when no operations are left. For example,

```
eval([PUSH(2.0),PUSH(1.0),SUB],[])
```

returns 1.0. The `eval` function will have the following basic form:

```
fun eval (nil,a::st) = (* ... *)
  | eval (PUSH(n)::ops,st) = (* ... *)
  | (* ... *)
  | eval (_,_) = 0.0
  ;
```

You need to fill in the blanks and add cases for the other opcodes.

The last rule handles illegal cases by matching any operation list and stack not handled by the cases you write. These illegal cases include ending with an empty stack, performing addition when fewer than two elements are on the stack, and so on. You may ignore divide-by-zero errors for now (or look at exception handling in Ullman– we will cover that topic in a few weeks).