Homework 4

Due 7 March

Handout 16
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 the last assignment unless you email me (dbarowy@cs.williams.edu) with your partner's name **the evening of Wednesday, Feb 28**.

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 `cs334_hw4_<username>` repository. For example, if your GitHub username is `dbarowy`, then your repository will be called `cs334_hw4_dbarowy`. Be sure that your work is committed and pushed to your repository by Wednesday, March 7 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 `cs334_hw4_dbarowy-wjannen` (usernames will be in alphabetical order). Be sure that your work is committed and pushed to your repository by Wednesday, March 7 at 11:59pm.

Standard MLfiles should have a "`sml`" suffix. For example, the pair programming problem **P1** should appear as the file `p1.sml`. Your code should be documented using comments. Comment lines appear inside `(*` and `*)` braces.

Reading

1. **(Required)** Read Mitchell, Chapter 5.
2. **(As necessary)** Read Ullman or other ML references, as needed for the programming questions.

Self Check

S1. .... Higher-Order ML Types

Explain the ML type for each of the following declarations:

(a) `fun a b c = b c`

(b) `fun h () = "hello"`

(c) `fun d e = map (a h)`

(d) `fun f xs = map (fn x => ()) xs`

What is the output of the expression `d 1 (f [1,2,3,4])`?

Problems

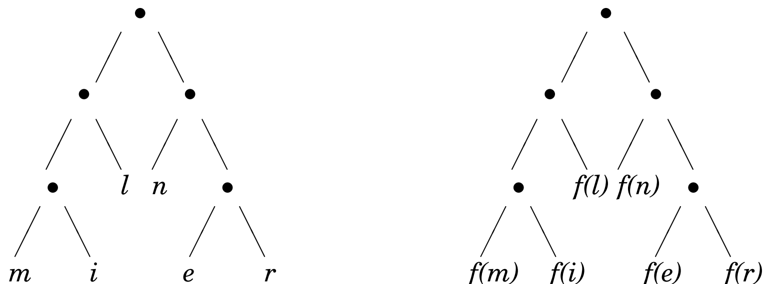
Q1. (10 points) .... ML Map for Trees

(a) The binary tree datatype

```
datatype 'a tree  =  LEAF of 'a |
                     NODE of 'a tree * 'a tree;
```

describes a binary tree for any type, but does not include the empty tree (i.e., each tree of this type must have at least a root node).

Write a function `maptree` that takes a function as an argument and returns a function that maps trees to trees, by mapping the values at the leaves to new values, using the function passed in as a parameter. In more detail, if `f` is a function that can be applied to the leaves of tree `t`, and `t` is the tree on the left, then `maptree f t` should result in the tree on the right.



For example, if `f` is the function `fun f(x)=x+1` then

`maptree f (NODE(NODE(LEAF 1,LEAF 2),LEAF 3))`

should evaluate to `NODE(NODE(LEAF 2,LEAF 3),LEAF 4)`. Explain your definition in one or two sentences.

(b) What is the type ML gives to your function? Why isn't it the expected type `('a → 'a) → 'a tree → 'a tree`?

You may want to include `Control.Print.printDepth:= 100;` at the top of your file, so that datatypes print completely.

Q2. (10 points) .... ML Reduce for Trees

The binary tree datatype

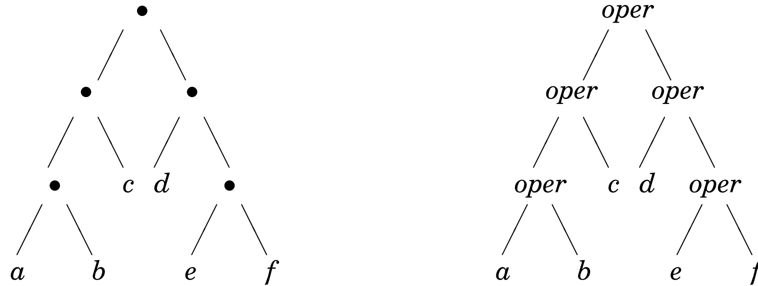
```
datatype 'a tree  =  LEAF of 'a |
                     NODE of 'a tree * 'a tree;
```

describes a binary tree for any type, but does not include the empty tree (i.e., each tree of this type must have at least a root node).

Write a function

$$\texttt{reduce} : (\texttt{'a} * \texttt{'a} \to \texttt{'a}) \to \texttt{'a tree} \to \texttt{'a}$$

that combines all the values of the leaves using the binary operation passed as a parameter. In more detail, if oper : 'a * 'a → 'a and t is the nonempty tree on the left in this picture,



then reduce oper t should be the result obtained by evaluating the tree on the right. For example, if f is the function

$$\texttt{fun f(x : int, y : int) = x + y;}$$

then reduce f (NODE(NODE(LEAF 1, LEAF 2), LEAF 3)) = (1 + 2) + 3 = 6. Explain your definition of reduce in one or two sentences.

Q3. (10 points) .... Currying

This problem asks you to show that the ML types 'a → ('b → 'c) and ('a * 'b) → 'c are essentially equivalent.

(a) Define higher-order ML functions

$$\texttt{Curry} : (('a * 'b) \to 'c) \to ('a \to ('b \to 'c))$$

and

$$\texttt{UnCurry} : ('a \to ('b \to 'c)) \to (('a * 'b) \to 'c)$$

(b) For all functions f : ('a * 'b) → 'c and g : 'a → ('b → 'c), the following two equalities should hold (if you wrote the right functions):

$$\texttt{UnCurry(Curry(f))} = \texttt{f}$$

$$\texttt{Curry(UnCurry(g))} = \texttt{g}.$$

Explain why each is true, for the functions you have written. Your answer can be 3 or 4 sentences long. Try to give the main idea in a clear, succinct way. We are more interested in the insight than in the number of words. Be sure to consider termination behavior as well.

Note that you MUST explain why the equations hold to receive full credit.

Hint: One way to do this is to apply both sides of each equation to the same argument(s) and describe how each side evaluates to the same term. For example, show that

$$\texttt{UnCurry(Curry(f))(s, t)} = \texttt{f(s, t)}$$

and

$$\texttt{Curry(UnCurry(g)) s t} = \texttt{g s t}$$

for any s and t.

Pair Programming

P1. (20 points) .... Low-Level Data Representation

The programming languages group in the computer science department at Williams College has recently realized that programming is too complicated and wants to solve the problem. Specifically, programming is too complicated because there are just too many darn letters in a program.

As an alternative, the department PL has whizzed up a new language composed entirely of the letters `e`, `p`, `h`, `r`, `a`, `i`, `m`, and `!` (for extra effect). The department considers this new language, which is called "Breph," such a resounding success that all future introductory programming classes will use it.

Since you've signed up to be a 134 teaching assistant, the department asks you to write a simple variant on `Hello world` in Breph. Specifically, you must write a program that prints

```
Hello <your first name> <your last name>
```

The department gives you its `breph` interpreter (see starter code in repository) and a copy of the language documentation, reproduced below.

Running a Breph Program

Breph programs end in the suffix "`.moo`". To run a program, type the following at the command prompt:

```
$ ./breph <yourprogram>.moo
```

Breph Abstract Machine:

Breph uses simple machine model consisting of

(a) the program text (supplied by the programmer),

(b) a pointer to the current instruction (initialized to point at the first instruction in the program),

(c) an array of 1-byte memory cells of (effectively) unlimited size (all initialized to zero),

(d) a movable data pointer (initialized to point to the leftmost byte of the array)

(e) input, connected to the user's keyboard,

(f) and output, connected to the user's terminal, which prints bytes using their ASCII character encoding.

Initial Abstract Machine State:

Assume that the equivalent of the following Java code is run before the user's program runs to set the machine's initial state.

```
char[] a = new char[∞];
Arrays.fill(a, '\0');
int p = 0;
```

Breph Machine Instructions:

Breph has the following machine instructions. Since the department realizes that TAs took 134 with Java, they've also provided a list of equivalent Java expressions to aid in the transition to Breph.

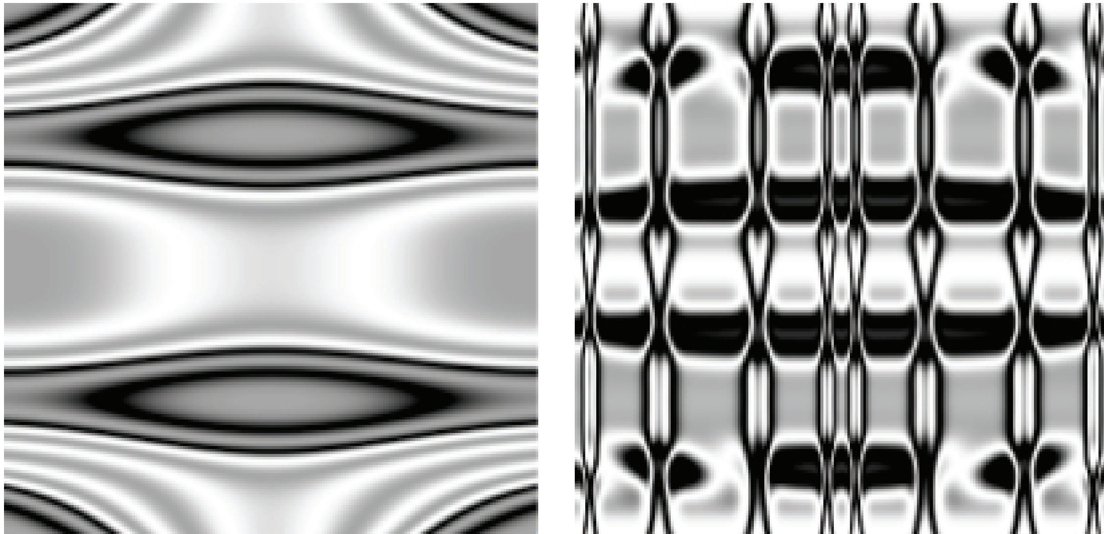
Note that every instruction increments the instruction pointer after performing its function except `i` and `m`, which may jump the pointer to some other location (see table).

Character	Semantics	Java equivalent expression
`e`	Increment the data pointer (to point to the next cell to the right).	`++p;`
`p`	Decrement the data pointer (to point to the next cell to the left).	`--p;`
`h`	Increment (increase by one) the byte at the data pointer.	`a[p] = a[p] + 1;`
`r`	Decrement (decrease by one) the byte at the data pointer.	`a[p] = a[p] - 1;`
`a`	Accept one byte of input, storing its value in the byte at the data pointer.	`a[p] = (char)System.in.read();`
`i`	If the byte at the data pointer is zero, then instead of moving the instruction pointer forward to the next command, jump it forward to the command after the matching `m` command.	`while (a[p] != 0) {`
`m`	If the byte at the data pointer is nonzero, then instead of moving the instruction pointer forward to the next command, jump it back to the command after the matching `i` command.	`}`
`!`	Print the byte (using ASCII encoding) at the data pointer.	`System.out.print(a[p]);`

ASCII Character Encoding

Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
0	00	Null	32	20	Space	64	40	@	96	60	`
1	01	Start of heading	33	21	!	65	41	A	97	61	a
2	02	Start of text	34	22	"	66	42	B	98	62	b
3	03	End of text	35	23	#	67	43	C	99	63	c
4	04	End of transmit	36	24	$	68	44	D	100	64	d
5	05	Enquiry	37	25	%	69	45	E	101	65	e
6	06	Acknowledge	38	26	&	70	46	F	102	66	f
7	07	Audible bell	39	27	'	71	47	G	103	67	g
8	08	Backspace	40	28	(	72	48	H	104	68	h
9	09	Horizontal tab	41	29	)	73	49	I	105	69	i
10	0A	Line feed	42	2A	*	74	4A	J	106	6A	j
11	0B	Vertical tab	43	2B	+	75	4B	K	107	6B	k
12	0C	Form feed	44	2C	,	76	4C	L	108	6C	l
13	0D	Carriage return	45	2D	-	77	4D	M	109	6D	m
14	0E	Shift out	46	2E	.	78	4E	N	110	6E	n
15	0F	Shift in	47	2F	/	79	4F	O	111	6F	o
16	10	Data link escape	48	30	0	80	50	P	112	70	p
17	11	Device control 1	49	31	1	81	51	Q	113	71	q
18	12	Device control 2	50	32	2	82	52	R	114	72	r
19	13	Device control 3	51	33	3	83	53	S	115	73	s
20	14	Device control 4	52	34	4	84	54	T	116	74	t
21	15	Neg. acknowledge	53	35	5	85	55	U	117	75	u
22	16	Synchronous idle	54	36	6	86	56	V	118	76	v
23	17	End trans. block	55	37	7	87	57	W	119	77	w
24	18	Cancel	56	38	8	88	58	X	120	78	x
25	19	End of medium	57	39	9	89	59	Y	121	79	y
26	1A	Substitution	58	3A	:	90	5A	Z	122	7A	z
27	1B	Escape	59	3B	;	91	5B	[	123	7B	{
28	1C	File separator	60	3C	<	92	5C	\	124	7C	|
29	1D	Group separator	61	3D	=	93	5D	]	125	7D	}
30	1E	Record separator	62	3E	>	94	5E	^	126	7E	~
31	1F	Unit separator	63	3F	?	95	5F	_	127	7F	□

P2. (50 points) .... Random Art



This problem brings together a number of topics we have studied, including grammars, parse trees, and evaluation. Your job is to write an ML program to construct and plot randomly generated functions. The language for the functions can be described by a simple grammar:

$$e ::= x \mid y \mid \sin \pi e \mid \cos \pi e \mid (e + e)/2 \mid e * e$$

Any expression generated by this grammar is a function over the two variables x and y. Note that any function in this category produces a value between -1 and 1 whenever x and y are both in that range.

We can characterize expressions in this grammar with the following ML datatype:

```
datatype Expr =
    VarX
  | VarY
  | Sine     of Expr
  | Cosine   of Expr
  | Average  of Expr * Expr
  | Times    of Expr * Expr;
```

Note how this definition mirrors the formal grammar given above; for instance, the constructor `Sine` represents the application of the sin function to an argument multiplied by π. Interpreting abstract syntax trees is much easier than trying to interpret terms directly.

Your starter code directory should include the `expr.sml` and `art.sml` starter files.

(a) **Printing Expressions:** The first two parts require that you edit and run only `expr.sml`. First, write a function

```
exprToString : Expr -> string
```

to generate a printable version of an expression. For example, calling `exprToString` on the expression

```
Times(Sine(VarX),Cosine(Times(VarX,VarY)))
```

should return a string similar to "`sin(pi*x)*cos(pi*x*y)`". The exact details are left to you. (Remember that string concatenation is performed with the "^" operator.)

Test this function on a few sample inputs before moving to the next part.

(b) **Expression Evaluation:** Write the function

```
eval : Expr -> real*real -> real
```

to evaluate the given expression at the given (x, y) location. You may want to use the functions `Math.cos` and `Math.sin`, as well as the floating-point value `Math.pi`. (Note that an expression tree represented, e.g., as `Sine(VarX)` corresponds to the mathematical expression $\sin(\pi x)$, and the `eval` function must be defined appropriately.)

Test this function on a few sample inputs before moving on to the next part. Here are a few sample runs:

```
- eval (Sine(Average(VarX,VarY))) (0.5,0.0);
val it = 0.707106781187 : real
- eval sampleExpr (0.1,0.1);
val it = 0.569335014033 : real
```

(c) **Driver Code:** The `art.sml` file includes the `doRandomGray` and `doRandomColor` functions, which generate grayscale and color bitmaps respectively. These functions want to loop over all the pixels in a (by default) 501 by 501 square, which naturally would be implemented by nested for loops. In `art.sml`, complete the definition of the function

```
for : int * int * (int -> unit) -> unit
```

The argument triple contains a lower bound, an upper bound, and a function; your code should apply the given function to all integers from the lower bound to the upper bound, inclusive. If the greater bound is strictly less than the lower bound, the call to `for` should do nothing. Implement this function using imperative features. In other words, use a `ref` cell and the while construct to build the `for` function.

Note: It will be useful to know that you can use the expression form (e1 ; e2) to execute expression e1, throw away its result, and then execute e2. Thus, inside an expression a semicolon acts exactly like comma in C or C++. Also, the expression "`()`" has type `unit`, and can be used when you want to "do nothing".

Test your code with a call like the following:

```
for (2, 5, (fn x => (print ((Int.toString(x)) ^ "\n"))));
```

It should print out the numbers 2,3,4, and 5.

Now produce a grayscale picture of the expression `sampleExpr`. You can do this by calling the `emitGrayscale` function. Look at `doRandomGray` to see how this function is used.

If you get an `uncaught exception Chr` error while producing a bitmap, that is an indication that your `eval` function is returning a number outside the range [-1,1].

Note: The type assigned to your `for` function may be more general than the type described above. How could you force it to have the specified type, and why might it be useful to do that? (You don't need to submit an answer to this, but it is worth understanding.)

(d) **Viewing Pictures:** You can view pgm files, as well as the ppm files described below, on our Linux computers with the `eog` program. To view the output from a non-Unix machine, or to post them on a web, etc., you might need to first convert the file to jpeg format with the following command:

```
convert art.pgm art.jpg
```

The `convert` utility will work for both .ppm and .pgm files.

(e) **Generating Random Expressions:** Your next programming task is to complete the definition of

```
build(depth, rand) : int * RandomGenerator -> Expr
```

The first parameter to `build` is a maximum nesting depth that the resulting expression should have. A bound on the nesting depth keeps the expression to a manageable size; it's easy to write a naive expression generator which can generate incredibly enormous expressions. When you reach the cut-off point (i.e., `depth` is 0), you can simply return an expression with no sub-expressions, such as `VarX` or `VarY`. If you are not yet at the cut-off point, randomly select one of the forms of `Expr` and recursively create its subexpressions.

The second argument to `build` is a function of type `RandomGenerator`. As defined at the top of `art.sml`, the type `RandomGenerator` is simply a type abbreviation for a function that takes two integers and returns an integer:

```
type RandomGenerator = int * int -> int
```

Call `rand(l,h)` to get a number in the range `l` to `h`, inclusive. Successive calls to that function will return a sequence of random values. Documentation in the code describes how to make a `RandomGenerator` function with `makeRand`. You may wish to use this function while testing your `build` function.

Once you have completed `build`, you can generate pictures by calling the function

```
doRandomGray : int * int * int -> unit
```

which, given a maximum depth and two seeds for the random number generator, generates a grayscale image for a random image in the file `art.pgm`. You may also run

```
doRandomColor : int * int * int -> unit
```

which, given a maximum expression depth and two seeds for the random number generator, creates three random functions (one each for red, green, and blue), and uses them to emit a color image `art.ppm`. (Note the different filename extension).

A few notes:

- The build function should not create values of the `Expr` datatype directly. Instead, use the build functions `buildX`, `buildY`, `buildSine`, etc. that I have provided in `expr.sml`. This provides a degree of modularity between the definition of the `Expr` datatype and the client.
- A depth of 8 – 12 is reasonable to start, but experiment to see what you think is best.
- If every sort of expression can occur with equal probability at any point, it is very likely that the random expression you get will be either `VarX` or `VarY`, or something small like `Times(VarX,VarY)`. Since small expressions produce boring pictures, you must find some way to prevent or discourage expressions with no subexpressions from being chosen "too early". There are many options for doing this— experiment and pick one that gives you good results.
- The two seeds for the random number generators determine the eventual picture, but are otherwise completely arbitrary.

(f) **Extensions:** Extend the `Expr` datatype with at least three more expression forms, and add the corresponding cases to `exprToString`, `eval`, and `build`. The two requirements for this part are that:

  i. these expression forms must return a value in the range [-1,1] if all subexpressions have values in this range, and
  ii. at least one of the extensions must be constructed out of 3 subexpressions, ie. one of the new `build` functions must have type `Expr * Expr * Expr -> Expr`.

There are no other constraints; the new functions need not even be continuous. Be creative!

Make sure to comment your extensions.

P3. (10 points) .... Challenge Problem

This question explores an alternative way of representing expressions in the random art program.

Create a new file `expr-func.sml` which, like the file `expr.sml`, defines the `Expr` representation and basic operations on it. In this version, the definition of the type `Expr` should be not a datatype, but:

```
type Expr =  real * real -> real
```

That is, instead of the symbolic representation used by `expr.sml`, this implementation will represent each function in x and y directly as an SML function of two `real` arguments. Redefine the following operations on the new type:

- `exprToString`
- `eval`
- `buildX`, `buildY`, `buildSine`, etc.

The `eval` function in particular becomes much, much simpler than in `expr.sml`, but the `exprToString` function cannot be written successfully, since there is no way to convert an ML function to a string. Thus, your implementation of this function can return something like `"<function>"` or `"unknown"`. To test your code, replace

```
use "expr.sml";
```

at the top of `art.sml` with

```
use "expr-func.sml";
```