CSE 341, Winter 2005, Assignment 3
Due: Thursday, February 3, 10:00 PM

Last updated: 01-29-05

This assignment consists of two parts; the first deals with a simple list manipulation language, and the second uses closures in the context of calculus. In total, there are 7 functions to implement that aren't trivial uses of other functions. Start early and make sure you are comfortable with closures and higher-order functions. The sample solution is about 100 lines long.

Part I
List manipulation language

We will define a small language for performing operations on lines of dominoes. Each operation acts on a *pair* of lines. You will use the following definitions in your solutions. A bone is represented as it was in homework 1 and 2:

```
type bone = int * int
```

A line is simply a list of bones:

```
type line = bone list
```

There are several operations we will define on a pair of lines. We can take the first bone of the left line and push it onto the front of the right line and vice versa (think "stack"), we can swap the first bones in the lines, and we can apply some arbitrary function to the first bone in either the left or right line and replace it with the result. Formally:

```
datatype lineop =
          PushRight (*Take element from left, push it onto right *)
        | PushLeft (*Take element from right, push it onto left*)
        | SwapFirst (* Take first bone from each line and swap them*)
        | ApplyRight of bone -> bone (*Apply function to first bone on the right*)
        | ApplyLeft of bone -> bone (*Apply function to first bone on the left*)
```

A program is a list of operations, applied in order to the two lines:

```
type lineprog = lineop list
```

Using these definitions, do the following:

1. Write a function `lineop_func` that takes a lineop `lop` and returns a function which, when invoked with a pair of lines, returns the pair of lines that result from applying the operation `lop`. An operation that would attempt to manipulate (or remove) the first bone in an empty line should raise the exception `Empty`. For example, `PushRight` on the pair `([],[(1,0)])` should raise an exception, because there is no bone in the left line to move to the right. `PushLeft` on the same pair is legal.

2. Write a function `lineprog_func` that takes a lineprog `prog` and returns a function which, when invoked with a pair of lines, returns the pair of lines that result from applying the operations in `prog` in order. For an elegant solution, use SML higher-order library functions such as `List.map` and `List.foldl` and the built-in function composite operator `o` (that's a small letter 'o'). The sample solution is only 1 line. Solutions that work but do not make use of higher order functions *will lose style points*.

3. Write a function `pushn_right_safe` that takes an integer `n` and returns a lineprog which pushes the first `n` bones in the left line onto the right line, but also ensures that the sides of the `n` bones that were originally touching are still touching afterwards. The bones that are moved will end up in reverse order; you do not have to attempt to prevent this from happening.

4. Write a function `pushn_right_block` that takes an integer `n` and returns a lineprog which pushes the first `n` bones in the left line onto the right line *as a block*. That is, the bones should end up in the same order they started in. Keep in mind that simply doing `n` `PushRight` operations will reverse the order of the bones, so you must somehow prevent this. Sitting down with pencil and paper and working out an algorithm before you begin coding is advisable. Use helper functions and the list append operator `@` to make your life easier.

5. Write a function `lineprog_mirror` which takes a lineprog `prog` and returns another lineprog which performs the mirror image of the operations in `prog`; that is, an operation that operates on the left line must instead operate on the right, and vice versa.

6. Use `lineprog_mirror` to define two more functions, `pushn_left_safe` and `pushn_left_block`, which work as you would expect. Use `val` bindings instead of `fun` bindings, and don't introduce anonymous functions with `fn`. *Hint: you will need to use a higher-order function.*

7. (**Extra credit**) Write a function `lineprog_optimize` which takes a lineprog `prog` and returns another lineprog which performs the same transformation on a pair of lines, but contains fewer operations. It may return a lineprog of the same size if no optimization is possible, but the size of the lineprog must not increase. Try to detect and simplify as many optimizable patterns as possible.

Sample output

```
- lineprog_func (pushn_left_block 3) ([],[(1,2),(2,3),(3,4),(0,0)]);
val it = ([(1,2),(2,3),(3,4)],[(0,0)]) : bone list * bone list
```

Part II
Calculus

You will write two functions (and a third trivial one) that calculate the derivative or integral of a function operating on real numbers. The algorithms to do this numerically will be presented here in case your calculus is rusty.

Recall that the derivative of a function is the slope of the function at a particular point, defined as:

$$f'(x) = \frac{d}{dx} f(x) = \lim_{\Delta x \to 0} \frac{f(x+\Delta x) - f(x)}{\Delta x}$$

To approximate a derivative numerically, you can simply choose some Δx close to 0 instead of taking the limit. Approximating an integral is somewhat more complicated:

For a function $f(x)$ over the interval $[a,b]$, divide the interval into n subintervals of width $\Delta x = \frac{b-a}{n}$, and choose a point from each interval x_i. The definite integral is:

$$\int_a^b f(x)dx = \lim_{n \to \infty} \sum_{i=1}^{n} f(x_i)\Delta x$$

As n approaches infinity, the result approaches the true integral, so we can pick a very large n, iterate through values of x between a and b in increments of Δx, find the value of f at each x and multiply by Δx, and then sum the terms. There are other methods, such as the trapezoid method, that give more accurate results; if you know them, you may use them instead of the algorithm here. The definition of an indefinite integral is simple once the definite integral is defined. Let $F(b)$ be the indefinite integral of $f(x)$. It is defined as:

$$F(b) = \int f(x)dx = \int_0^b f(x)dx$$

We can define some useful constants to make writing the functions easier:

```
val bigNumber = 100000.0
val smallNumber = 1.0 / bigNumber
```

A module to allow you to display graphs of your functions has been provided. This is a useful visualization tool to ensure that your functions are working properly. To use it, you first need to import the provided code:

```
use "hw3-provided.sml"
```

To print a graph to the screen, simply do:

```
Grapher.graph (f)
```

Where `f` is a function of type real -> real.

1. Write a function derivative that takes a function `f` of type `real->real` and returns a function of type `real->real` which evaluates to the derivative of $f(x)$ when invoked with some `real` x. That is, write a curried function `derivative f x`.

2. Write a curried function `def_integral f a b` that returns the integral of $f(x)$ between a and b. You can use the special syntax `fun def_integral f a b =` ... for convenience.

3. Write a function `integral` that takes a function `f` and returns the indefinite integral of $f(x)$. Use `def_integral` in your solution.

Sample output

```
- Grapher.graph (derivative (fn x => 10.0/x));
                                  |
                                  |
                                  |
                                  |
                                  |
                                  |
                                  |
                                  |
                                  |
                                  |
                                  |
                                  |
                                  |
                                  |
oooooooooooooooo------------------+------------------ooooooooooooooo
                oooooooo          |          oooooooo
                        oo        |        oo
                          oo      |      oo
                                  |
                            o     |     o
                                  |
                             o    |    o
                                  |
                                  |
                              o   |   o
                                  |
                                  |
                                  |
                                  |
```

Type Summary

A correct solution will cause these bindings to be printed in the read-eval-print loop:

```
type bone = int * int
datatype lineop
  = ApplyLeft of int * int -> int * int
  | ApplyRight of int * int -> int * int
  | PushLeft
  | PushRight
  | SwapFirst
type lineprog = lineop list
val lineop_func = fn : lineop -> bone list * bone list -> bone list * bone list
val lineprog_func = fn
  : lineop list -> bone list * bone list -> bone list * bone list
val pushn_right_safe = fn : int -> lineop list
val pushn_right_block = fn : int -> lineop list
val lineprog_mirror = fn : lineop list -> lineop list
val pushn_left_safe = fn : int -> lineop list
val pushn_left_block = fn : int -> lineop list
val derivative = fn : (real -> real) -> real -> real
val def_integral = fn : (real -> real) -> real -> real -> real
val integral = fn : (real -> real) -> real -> real
```

Getting these bindings does not necessarily mean your solution is correct. Also, the bindings you get may not look exactly like these due to type synonyms.

Assessment

- Your solution should generate correct bindings and give correct results.
- You must use pattern matching. Do not use the functions `null`, `hd`, `tl`, or anything starting with `#`.
- Do not use mutation, even if you know how.
- Don't worry about gracefully handling non-continuous functions in the calculus section. The grapher will do its best to draw them, but don't expect the integral to look very impressive.

Turn-in Instructions

Use the turn-in form linked from the course website.