Assignment 1

Getting Started with ML

Code Due: 10:00 pm, Wednesday, September 15, 2004

Instructions

This assignment consists of four problems. You should get the file `assign1.sml` from the assignments web page and modify it to include your solutions.

The largest part of the assigment grade is based on satisfying the specifications and documentation (comments). Excessively convoluted or inefficient (unless otherwise specified) code will not receive full credit, however.

To submit code, run the command

```
cs131submit assign1.sml
```

You may submit as often as you wish (to provide a backup of your code in case of catastrophe) but *your final submission* must not *have syntax or type errors in order to get credit for the assignment*. If you know code does not work, explain how or why in a comment. Code that does not even compile must be commented out. Functions must have the exact names and types given in the assignment (although you are free to define other helper functions as well), and should be reasonably well commented in order to help the graders understand your code.

1 Warm-up (20%)

1. Define the function

   ```
   fib : int -> int
   ```

   which computes the Fibonacci number F_n when given $n \geq 0$. The Fibonacci numbers are defined by

   $$\begin{aligned} F_0 &= 0 \\ F_1 &= 1 \\ F_n &= F_{n-1} + F_{n-2} \qquad \text{when } n \geq 2. \end{aligned}$$

   Your code need not be efficient; the most obvious definition turns out to take time exponential in n. You need not check for negative inputs.

2. It is occasionally useful to consider a more general specification,

   $$\begin{aligned} F_0 &= 0 \\ F_1 &= 1 \\ F_n &= F_{n-1} + F_{n-2} \qquad \textbf{for all integers } n. \end{aligned}$$

This specification uniquely determines the value of F_n for all integers n, not just the non-negative integers. (For example, what must F_{-1} be to satisfy this specification? What about F_{-2}?) Define the function

```
intfib : int -> int
```

that computes the value of F_n given an integer n. Again, efficiency is secondary.

2 Lists (20%)

In class, we discussed how to define a function

```
split : 'a list -> 'a list * 'a list
```

which takes a list (of elements of an arbitrary type `'a`) and separates out those elements in odd positions (that is, the first, third, fifth, etc.) from those in even positions (that is, the second, fourth, etc.).

Now write a function

```
merge : 'a list * 'a list -> 'a list
```

that does the opposite (i.e., interleaves the elements of the two list arguments), so that for any list `l` we have

```
merge (split l) = l
```

Be sure to carefully document in your source code what your function will do if the input lists are not of equal length.

3 Higher-Order Functions (10%)

There are two ways to define a two-argument function in SML: the arguments can either be supplied simultaneously (as a pair) or they can be supplied separately through successive applications. For example, a function having two integer inputs could be written to have either the type

```
int * int -> int
```

or, written slightly differently, to have the type

```
int -> int -> int
```

In different circumstances, one or the other type may be more convenient. But a function defined in either fashion may be converted to the other.
Define the functions

```
curry   : ('a * 'b -> 'c) -> 'a -> 'b -> 'c
uncurry : ('a -> 'b -> 'c) -> ('a * 'b -> 'c)
```

to do the conversion. For example, if we define

```
val g = curry f
```

then g x y and f(x, y) should yield the same answer for all x and y. Similarly, if we say

```
val h = uncurry k
```

then h(x, y) and k x y should yield the same answer for all x and y.

Note that the type of `curry` is equivalent to the type

```
(('a * 'b) -> 'c)  ->  ('a -> ('b -> 'c))
```

and the type of `uncurry` is equivalent to the type

```
('a -> ('b -> 'c))  ->  (('a * 'b) -> 'c).
```

4 Stack Machines (50%)

For this problem, don't worry about any inefficiencies introduced by the list-append function.

Consider the following datatype for representing arithmetic expressions:

```
datatype opn  = Add | Sub | Mult | Divide
datatype aexp = Num  of  real
              | Opn  of  aexp * opn * aexp
```

Certain HP calculators and certain programming languages evaluate expressions using a stack. An arithmetic computation is expressed as a sequence of operations, which we will represent using the datatype

```
datatype sopn = Push   of  real
              | DoOpn  of  opn
              | Swap
```

The operation `Push r` means push the number r onto the stack; the operations `DoOpn Add`, `DoOpn Sub`, `DoOpn Mult`, and `DoOpn Divide` mean "replace the top two numbers on the stack with their sum", "... their difference", and so forth. The `Swap` operation swaps the top two numbers on the stack. In summary,

If the stack looks like	and the operation is	afterwards the stack should be
...	`Push` r	r ...
$a\ b$ ...	`DoOpn Add`	$(b + a)$ ...
$a\ b$ ...	`DoOpn Sub`	$(b - a)$ ...
$a\ b$ ...	`DoOpn Mult`	$(b * a)$ ...
$a\ b$ ...	`DoOpn Divide`	(b/a) ...
$a\ b$ ...	`Swap`	$b\ a$ ...

Note: The top of the stack is shown on the left

1. We will represent the stack as a list,

   ```
   type stack = real list
   ```

   with the front of the list corresponding to the top of the stack.

   Write a recursive function,

   ```
   evalRPN : sopn list * stack -> real
   ```

   that returns the number at the top of the stack after performing the given operations in order, starting with the given stack. Be careful to get the order right for `Sub` and `Divide`. For example,

   ```
   evalRPN ([Push 2.0, Push 1.0, DoOpn Sub], [])
   ```

   should return `1.0`, not `~1.0`.

   At this point in the course you need not worry about stack underflow, numeric overflow, or divide-by-zero errors. Correct code may therefore generate nonexhaustive match warnings.

2. Write a function

   ```
   toRPN : aexp -> sopn list
   ```

   that converts an arithmetic expression to a list of stack operation instructions that compute the same expression. There should be an `DoOpn Add` stack operation for every `Add` in the input, and so on; evaluating the input expression to a number r and then returning `[Push` r`]` is not acceptable.

   Hint: this function corresponds exactly to a postfix traversal of the input expression viewed as a tree.

3. The same expression can be computed several ways in the stack machine, because, for each subexpression, you can choose to evaluate the left side first or the right side first. (Because subtraction and division are not commutative, evaluating the right side first and then the left will require a `Swap` to fix things up.)

   For example, $1.0 - (2.0 + 3.0)$ can be computed either by the sequence

   ```
   [Push 1.0, Push 2.0, Push 3.0, DoOpn Add, DoOpn Sub]
   ```

   or by

   ```
   [Push 2.0, Push 3.0, DoOpn Add, Push 1.0, Swap, DoOpn Sub]
   ```

The first list of instructions requires a stack that can hold at least three numbers simultaneously, whereas the second list never requires more than two numbers on the stack at any one time.

Define the function

```
toRPNopt : aexp -> sopn list * int
```

that returns a pair containing (1) an optimal sequence of operations to evaluate the given arithmetic expression, and (2) the maximum number of values simultaneously on the stack during the execution of this sequence. Optimal here is defined to mean "requiring the smallest amount of stack space", which means having the smallest possible *maximum* number of values on the stack at once.

Hints: (1) This function can be computed inductively, using the optimal instruction sequence for the first operand, the optimal sequence for the second operand, and the stack sizes they each require. (2) You can decide whether to evaluate the left side first or the right side first just by looking at the stack depth each requires (and without looking at the particular operations!).

4. Define the arithmetic expressions

```
aexp3 : aexp
aexp4 : aexp
```

that, even when evaluated optimally, require the stack to contain simulataneously at least 3 or 4 values, respectively.

Extra Credit (5%)

Write the function

```
fromRPN : sopn list -> aexp
```

that converts a list of stack operations to the corresponding arithmetic expression. You may assume that evaluation of these stack operations starting with an empty stack would yield a stack containing only a single number, the answer. (That is, you may assume that the stack operations really do correspond to some arithmetic expression.)