Computer Science 131, Fall 2000
Assignment 1: SML Programming

Out: Thursday, August 31

Due: Friday, September 8, 5:00pm

This assignment consists of 4 problems, each of which requires defining two or more functions. All solutions should be placed into a single file named `assign1.sml`, which should contain no syntax or type errors. (If you know the code does not work, explain why in a comment; code that does not even compile should be commented out.) Functions should have the exact names and types given in the problem below, and should be labeled with comments specifying the corresponding problem number. Feel free to define any other functions you find useful, however.

The submission process should be familiar; when you are ready, run the command

```
cs131submit assign1.sml
```

This assignment uses material covered in class, which appears in more detail in Chapters 1–3, 5, and 7–9 of the course textbook. DO NOT USE `ref` OR `:=` IN YOUR SOLUTIONS!

1 Defining Functions

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:

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

   This uniquely determines the value of F_n for all integer n, not just the non-negative integers. (What is F_{-1}? F_{-2}?) Define the function

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

   that computes the value of F_n given an arbitrary integer n. Again, this need not be efficient (or even particularly elegant).

2 Lists and Polymorphism

1. Define the function

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

   to separate the list elements in odd positions (that is, the first, third, fifth, etc.) from those in even positions (that is, the second, fourth, etc.). More formally, we want `split` [x_1,x_2,...] to return the pair ([x_1,x_3,...], [x_2,x_4,...]). Your function should work for both inputs of even length and inputs of odd length (as well as for the empty list, in which case the output should be a pair of empty lists).

   Hint: our example solution is written as a function having three cases: one for an empty input, one for an input of length one, and one for an input of length two or more.

2. Define the function

   ```
   merge : ('a * 'a -> bool) -> ('a list * 'a list) -> 'a list
   ```

   which takes an order relation and a pair of lists (which can be assumed to be increasing with respect to to the given order) and merges the two lists into a single list, also in increasing order.

3. The mergesort algorithm works by splitting the input into half, sorting each half recursively, and then merging the results. Use the `split` and `merge` functions to define the function

   ```
   mergesort: ('a * 'a -> bool) -> 'a list -> 'a list
   ```

   which, given an ordering relation, sorts its input list. (Be careful to include the right base cases for the recursion.) Then, for example, we can define a sorting routine for lists of integers:

   ```
   val intmergesort : int list -> int list = mergesort (op <)
   ```

   (The type annotation here is necessary for the compiler because `<` is overloaded to work on integers, characters, etc., and there would otherwise not be enough context here to determine which was meant.)

3 Higher-Order Functions

There are two ways to define a two-argument function; the arguments can either be supplied simultaneously (as a pair) or 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. That is, `curry` f should be the function g where $(g\,x)y = f(x,y)$, and `uncurry` g which be the function f such that $(g\,x)y = f(x,y)$.

Note that because of the parenthesization conventions 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 Datatypes

Consider the following datatype for representing arithmetic expressions.

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

Then `3.0 + (9.0 / 2.5)` is represented as

```
Opn(Num 3.0, Add, Opn(Num 9.0, Divide, Num 2.5))
```

Certain HP calculators and certain programming languages evaluate expressions using a stack; then an arithmetic computation is expressed as a sequence of operations, which will be represented with the following 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 to replace the top two numbers on their stack with their sum, difference, etc.; the `Swap` operation swaps the top two numbers on the stack:

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

(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
   ```

   which 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`;

   ```
   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. Correct code may therefore generate non-exhaustive match warnings.

2. Write a function

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

which converts an arithmetic expression to a list of stack operation instructions which 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 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]
   ```

   Note that the first set of instructions requires that the stack be able to hold at least three numbers simultaneously, while the second sequence never requires more than two numbers on the stack at any one time.

   Define the function

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

   which 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 having the least possible maximum number of values on the stack at any given time. (In other words, you are trying to minimize the maximum stack depth during evaluation.)

   This function can be computed inductively for each binary operation, using the optimal instruction sequence for the first operand, the optimal sequence for the second operand, and the stack sizes they each require.

4. **Extra Credit (i.e., not required)** 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, that the stack operations really do correspond to some arithmetic expression.)