CS 131: Programming Languages, Spring 2003

Assignment 1
Programming in Standard ML

Due: Wednesday, January 29

Goals for this Assignment

1. To practice programming in SML, including the use of datatypes.
2. To see some very simple (but concrete) examples of several general ideas which we will return to later in the class: fixed points, abstract syntax, interpreters, compilers, and stack machines.

Office Hours

The purpose of office hours is to provide you with a chance to discuss class-related topics, to ask questions about material you find particularly interesting or particularly confusing, and to provide comments about the class.

You can drop by Olin 1253 any time to see if the professor is around (if not, look at the schedule posted by the door). The times specifically set aside by the professor are

Thursdays 2:30–4pm
Tuesdays 3–5pm
Tuesdays 7–9pm

and by appointment. Tutors are available in the Terminal Room

Sundays 3–5pm
Sundays 7–9pm

E-mailed questions about this assignment should be sent to `cs131help@cs.hmc.edu` .

Instructions

This assignment consists of 3 problems. You should get the file `assign1.sml` from the Assignments web page and modify this file to include your solutions.

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 (though 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 Warmup (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 uniquely determines the value of F_n for all integer n, not just the non-negative integers. (For example, what must F_{-1} be to satisfy this specification? What does this mean for 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%)

Define the function

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

which takes a list 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.). 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, not to mention the empty list (in which case the output should be a pair of empty lists).

Hint: You can use `let` and pattern-matching to give names to the components of a pair.

3 Stack Machines (60%)

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. In such an arithmetic computation is expressed as a sequence of operations, which we will represent using 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]
   ```

The first list of instructions requires that the stack be able to hold at least three numbers simultaneously, while the second list 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 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. Provide a small collection of well-commented test inputs for these three `RPN` functions. For each, say what the output *should* be.

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, that the stack operations really do correspond to some arithmetic expression.)