CS 312 Problem Set 1: An Introduction to SML

Assigned: September 1, 2003
Revised: September 4, 2003 **Due:** 11:59PM, September 10th, 2003

---

1 Introduction

The goal of this problem set is to expose you to as much of the SML language as possible while learning a bit about functional style programming and a few data structures. If you are not already familiar with SML, it may be a fair amount of new material, so please begin this problem set early.

Instructions: You will do this problem set by modifying the source files found in *ps1.zip* and submitting the program that results. The program that you submit must compile. Programs that do not compile will receive an automatic zero. If you are having trouble getting your assignment to compile, please visit consulting hours. If you run out of time, it is better to comment out the parts that do not compile and hand in a file that compiles, rather than handing in a more complete file that does not compile. Also, please pay attention to style. Refer to the CS 312 style guide and lecture notes.

The SML Basis Library will provide you functions for simplifying many of these problems. Part of the goal of this assignment is to make you familiar with the basis library, so read over it to look for functions which are useful to you.

For each of the problems, you will modify an included source file. For functions that are not implemented, you should see something of the form

```
raise Fail ``Not implemented''
```

You should remove this text and replace it with the body of the function. We will test some of your code using an automated test script. Do not change published variable names, function names and types. Do not remove our delimiters. Pay careful attention to types.

2 Expressions and Types

1. Provide the types of each of the following expressions, by modifying the included source file *types.txt*.

   (a) `5`

   (b) `(5, ``3'')`

   (c) `(5, (5, true), ``42'')`

   (d) `fn x => x + 3`

   (e) `fn x => case x of [] => ``true'' | 1::L => ``false'' | x::L => ``false''`

   (f) `(fn y => y+50=3)::[fn x => false, fn x => (x > 2)]`

   (g) `(fn y => y+1) ((fn x:int => x*12) 5)`

   (h) `(fn y => (fn x => y+x+1))`

2. Give expressions which have the following types.

   (a) `char list -⟩ string`

   (b) `int list * char * (int -⟩ real)`

   (c) `int list -⟩ int -⟩ string`

   (d) `(int list * int list) -⟩ int list`

   (e) `((real * (char list) -⟩ int) list list) * int`

   These are defined in the included file *expressions.sig* signature. Implement them by modifying the Expression structure found in the included file *expressions.sml*.

3 Improving Style

The following functions can be found in the included file *style.sml*. Find parts which violate the style guide and fix them. Do not change function names or function types.

```
fun sumArgs(x:int*int*int):int = let
  val A = #1 x
  val B = #2 x
  val C = #3 x
in
  (A + B) + C
end


fun test(x:int, y:int):bool =
  (case x of
    0 =⟩ true
  | 1 =⟩ true
  | _ =⟩ (case y of
            5 =⟩ false
          | b =⟩ true))


fun test2(x:bool, y:bool):bool =
  if x then if y then true else false else true


fun reverse(l:int list):int list =
  if null(l) then [] else
    reverse(tl(l))@[hd(l)]
```

4 Symbolic Datatypes

The file *datatypes.sml* defines the type vehicle as

```
datatype vehicle = CAR | MINIVAN | MOTORCYCLE | BIKE
```

You are to modify the file *datatypes.sml* as described below.

(a) Finish the following function. For our purposes, assume bicycles seat one person, motorcycles seat up to two, cars can seat four, and minivans can seat six.

```
fun maxPassengers(ride:vehicle):int = raise Fail ‘‘Not implemented’’
```

(b) Finish this function which, given a list of vehicles, find the maximum number of passengers that can ride in all of them together. Your solution is required to use *foldl* from the basis library.

```
fun maxPassengersMulti(rides:  vehicle list):int = raise Fail ‘‘Not
implemented’’
```

(c) Finish this function, which should evaluate to true if the number of passengers is no more than the maximum allowed for vehicle `ride`.

```
fun isSafe(ride:vehicle, numPassengers:int):bool = raise Fail ‘‘Not
Implemented’’
```

(d) Finish the function `heavier`. We assume minivans are heavier than cars, cars are heavier than motorcycles, and motorcycles are heavier than bikes. If you are not familiar with the *order* type, look through the lecture notes or read up on it in the basis library.

(e) Finish the `sortVehicles` function which sorts a list of vehicles by increasing weight. You should use the sorting function from the SML/NJ library (i.e., do not write your own sorting function).

The Windows version of SML/NJ does currently support automatically loading the library. In a later problem set you will learn how to load libraries using the compilation manager. For now you may simply use the following code to simulate the appropriate library.

```
signature LIST_SORT =
sig
  val sort : (('a * 'a) -> bool) -> 'a list -> 'a list
  val uniqueSort : (('a * 'a) -> order) -> 'a list -> 'a list
  val sorted : (('a * 'a) -> bool) -> 'a list -> bool
end
```

```
structure ListMergeSort : LIST_SORT =
struct
  fun sort(f: ('a*'a)->bool) (l: 'a list) =
    let
      val split = foldl (fn (x:'a,(a,b):'a list*'a list) => (b,x::a))
                        ([],[])
      fun merge(l: 'a list, l': 'a list) =
        case (l, l') of
          (hd::tl, hd'::tl') => if f(hd,hd') then hd'::merge(l,tl')
                                             else hd::merge(tl,l')
        | ((l',[]) | ([],l') ) => l'
    in
      case l of
        [] => []
      | [a] => [a]
      | _  => let
                val (a,b) = split l
              in
                merge (sort f a, sort f b)
              end
    end

  fun uniqueSort (x) = raise Fail ``Unimplemented''
  fun sorted(x) = raise Fail ``Unimplemented''
end
```

5 Data Carrying Datatypes

In *datatypes2.sml* the measurement datatype is defined. Finish the unimplemented functions below:

```
datatype measurement = MILLIMETER of int | CENTIMETER of int | METER of int |
KILOMETER of real

fun numMillimeters(x:measurement):int = raise Fail ``Not implemented''

fun millimetersArea({width:measurement, height:measurement}):int =
raise Fail ``Not implemented''
```

The first function should return the number of millimeters that are in x. The second function should return the area, in millimeters, of the space given. If rounding is needed, round down. Look through the standard library for useful functions.

6 Recursive Datatypes

The mylist datatype is defined in *datatypes3.sml*. Implement the functions below, whose behavior is described in the comments in the SML file:

```
datatype mylist = END | ITEM of (int * mylist)

fun length(l:mylist):int = raise Fail ``Not implemented''

fun reverse(l:mylist):mylist = raise Fail ``Not implemented''

fun map(convert:int->int) (l:mylist):mylist = raise Fail ``Not implemented''

fun find(l:mylist) (v:int):int = raise Fail ``Not implemented''
```

7 The Stack Data Structure

The file *stack.sml* contains the skeleton implementation of a stack data structure. The abstract data type Stack normally has the functions `push` and `pop` defined on it. Stacks provide last-in-first-out (LIFO) access: the last item to be *push*ed onto the stack will be the first item to be taken off the top when *pop*ed. The semantics of these and all stack operations are defined in *stack.sig*. Implement the missing functions.

8 The Queue Data Structure

The file *queue.sml* contains the skeleton implementation of a queue data structure. The `enqueue` and `dequeue` operations are defined on queues. Queues are first-in-first-out (FIFO) data structures, in that the first item to be *enqueued* will be the first returned when you *dequeue* from the queue. The semantics of these and all queue operations are defined in *queue.sig*. Implement the missing functions. [Note: this problem is hard if you don't think about it right. Don't spend infinite effort on it, as it is only worth 5% of the grade for this homework.]