

## Sample Solutions for the Midterm Examination

**(Question 1)** [20 points] A two-dimensional array can be represented in ML as a list of lists. For example, a 3 by 3 array of integers can be represented as `[ [1,2,3], [4,5,6], [7,8,9] ]`.

(a) [4 points (revised)] Write a recursive ML function that sums the components of a two-dimensional array of integers represented in this way.

One simple way to solve this problem is to write a helper function (`sum1d`) that sums a flat list of integers, and then to write a main function (`sum2d`) that sums the whole array:

```
fun sum1d [] = 0
  | sum1d (h::t) = h + (sum1d t);

fun sum2d [] = 0
  | sum2d (h::t) = (sum1d h) + (sum2d t);
```

This divide-and-conquer approach to programming using small helper functions can be good style in any programming language, not just in ML.

(b) [3 points (revised)] Write a recursive ML function that sums the components of an array of integers with any number of dimensions that is represented by nested lists, i.e. as a list of lists of lists etc.

This question is difficult because a list in ML must be homogeneous. No list can contain integers and sublists together. Therefore we must define our own recursive union type to represent lists with unlimited nesting of sublists.

```
datatype flex = Base of int | Nest of flex list;

fun sumflexlist [] = 0
  | sumflexlist (h::t) = (sumflex h) + (sumflexlist l);

fun sumflex (Base x) = x
  | sumflex (Nest l) = sumflexlist l;
```

Here the helper function and the main function are mutually recursive. There is some ML syntax using the keyword `and` for defining mutually recursive functions.

(c) [3 points] Pure ML does not have standard arrays. In addition to nested lists, explain an alternative way of representing an array in ML.

Abstractly, an array is a mapping. A mapping can also be represented by a function, in ML and in other programming languages.

(d) [4 points] In numerical computing, assigning a new value to one component of an array is a very common operation. Given this fact, explain as precisely as possible why nested lists are *not* an efficient way of implementing arrays. Is the alternative you suggested in part (c) efficient?

Assigning a new value to one component of a nested list requires sequential search through the nested list for the component. This search is not a constant-time operation. Accessing a conventional array is constant-time: the program computes a memory location arithmetically and then loads from or stores to this location directly.

Updating a single value of an array represented as a function requires creating a whole new function to represent the new array, as in this example:

```
fun update old k v =  
fn j => (if j = k then v else (old j))
```

Given an array represented as a function named `old`, the higher-order pure function named `update` returns a new array. The new array is an anonymous function that yields `v` if its argument equals `k`. Otherwise the anonymous function passes its argument to `old`.

(e) [3 points] Based on your answer to part (d), explain why functional programming languages like pure ML are not well-designed for numerical computing.

Because although there are several methods for implementing updatable arrays in a pure functional language, each of these methods is much less efficient than the standard method.

(f) [3 points] Professor Sussman of MIT has argued that functional PLs are good for numerical computing, because they provide higher-order functions. Based on your experience with the first CSE 130 project, explain Prof. Sussman's opinion.

Many mathematical operators are naturally implemented as higher-order functions. For example, the definite integral operator studied in the first project is a higher-order function. Going further, the indefinite integral is a function that returns a function. A language such as C may provide a way of having a function as an argument, but it does not allow a function to be generated as a return-value. Functional PLs do allow this, as seen in part (d) above for example. Therefore (from this perspective) functional PLs are better suited for coding numerical mathematical operators in a natural way.

**(Question 2)** [20 points] According to Dershem and Jipping (*Programming Languages: Structures and Models*, PWS Publishing Co., 1995, p. 42):

An *abstraction* is a representation of an object that includes only the relevant attributes of the original object, ignoring those attributes that are irrelevant to the purpose at hand. ... With data, the programmer is able to work more effectively by using simpler abstractions that do not include many irrelevant details of data objects. With procedures, abstractions facilitate good design practices and modularity.

(a) [4 points] Consider a dictionary data structure implemented as a binary tree. Sketch a typical C representation for this data structure. Which attributes of the data structure are ignored by the C representation? Are these attributes always irrelevant?

The word “sketch” does not always refer to a drawing. It can also mean any sort of an outline where not all details are specified. Here, it means you should provide an overview of the C code needed to implement a dictionary as a binary tree.

When a question uses words that appear in a quotation given as part of the question, you can assume that the words mean the same as in the quotation. Unless stated otherwise, you can also assume that what the quotation says is compatible with claims made in the course.

The C implementation ignores attributes (i.e. features or aspects) of the data structure that are machine-dependent. These attributes would be specified by the programmer, or at least visible to the programmer, if s/he used a lower-level language such as machine code. A specific example of one of these attributes is the exact address in memory at which the data structure is stored, and its size in bytes.

The low-level attributes are usually irrelevant, but not always. For example, to write the data structure in binary to a file, one needs to know how many bytes of storage it occupies.

(b) [4 points] Sketch a recursive type representing the same data structure. Which attributes are ignored by this representation? Are these attributes always irrelevant?

A dictionary is a collection of pairs where each pair is a key and an associated value. One of these pairs has type

```
type pair = keytype * valtype
```

Each node of a tree implementing a dictionary contains a pair as above, and pointers to left and right subtrees, if it is not a leaf, which is the base case. The recursive type is

```
type treedict = pair + (treedict * pair * treedict)
```

Here + is the disjoint union type operator.

(c) [4 points] Sketch a pure abstract data type (ADT) representing the same data structure. Which attributes are ignored by this representation? Are these attributes always irrelevant?

As explained in class, an ADT in principle has four parts: a public signature part, a public specification part, a private concrete type, and a private function implementation part. Your sketch should at least indicate the existence of each of these parts.

From the point of view of the programmer using the ADT, it ignores (i.e. hides) the concrete type and the implementations of the functions supplied by the ADT. These attributes (i.e. aspects of the ADT) are not always irrelevant. They are important to the programmer writing the code for the ADT, of course. Also, choices in the implementation part determine the time and space efficiency of the operations, which can be very important to the ADT user.

(d) [4 points] Explain these two claims:

- (i) A procedure with static scoping is more abstract than the same procedure with dynamic scoping.
- (ii) A pure function with no side-effects is more abstract than a subroutine with side-effects.

To show that  $X$  is more abstract than  $Y$ , we just need to identify some aspect of  $X$  and  $Y$  that is important to the user of  $Y$ , but hidden (i.e. unimportant) to the user of  $X$ . Equivalently, we just need to show that the interface of  $X$  visible to the user is a subset of the interface of  $Y$ .

(i) Under static scoping the meaning of a function (i.e. its behavior) depends only on its compile-time context, and the runtime context cannot change the meaning of the function. With dynamic scoping, the runtime context can change the meaning of the function, so this aspect is not hidden from the user of the function (or procedure).

(ii) The behavior of a pure function with no side-effects depends only on what actual argument values it is passed, and the only way its behavior affects the rest of the program is through its result value. The behavior of a subroutine with side-effects can depend on global variables that are not passed as actual arguments, and the subroutine can change these global variables. Hence the visible interface of the pure function is smaller, so it is more abstract.

(e) [4 points] Explain how “abstractions facilitate good design practices and modularity.”

This question asks you to explain how abstractions, as defined above, help a programmer to use at least one good design practice, and to use modularity.

Modularity is easier to achieve when modules have public interfaces that are small and defined in a single place. The entire interface of a pure function is defined in the header of the function. This is not true for subroutines with side-effects. Hence, pure functions facilitate modularity.

One good design practice is to make programs as portable as possible. A low-level implementation of a data structure where specific memory addresses and data structure sizes are explicit, is not portable. Hence more abstract data structure implementations, as described in parts (a), (b), and (c), facilitate a good design practice.

Another good design practice is to have the PL implementation check for as many errors as possible, and do as much memory allocation and deallocation as possible. When using a recursive union type, as in part (b), the compiler generates all pointer-manipulation code automatically, which prevents many bugs that are common when using languages like C that are less abstract, i.e. lower-level.