The University of Melbourne

Department of Computer Science and Software Engineering

COMP30020

Declarative Programming

Semester 1, 2011

Exam Duration: Two hours.

Reading Time: Fifteen minutes.

Length: This paper has 4 pages including this cover page.

Authorised Materials: None.

Instructions to Invigilators:

Instructions to Students: Answer all of the questions in the script book provided. Start your answer for each question on a separate page and write the question number at the top of the page. The number of marks for each question (totaling 60) is given; where no breakdown is given for multi-part questions, each part is worth the same number of marks. Minor errors in Haskell and Mercury syntax will be tolerated but you should add comments if you feel it will aid understanding of what you have written.

Calculators: Calculators are not permitted.

Baillieu Library: Exam paper to be held by the library.

Question 1 [6 marks]

What would be printed if the following expressions were typed into `ghci`? Recall `:t` tells `ghci` to just print the type of the expression. If any expressions would result in errors, just give the general nature of the error, for example "type error", rather than detailed error messages.

(a) `:t (++)`

(b) `:t filter`

(c) `:t take`

(d) `map (:[4]) [1,2]`

(e) `((drop 7).(take 10)) [1..]`

(f) `foldl (++) [4] [[1,2],[]]`

Question 2 [4+2+4 marks]

Most programs in imperative programming languages like C and Java spend much of their time updating existing data structures. In declarative programming languages like Haskell and Mercury, all data structures are *immutable*; they cannot be updated.

1. What do programs written in declarative languages do instead of updating existing data structures?
2. Briefly describe one programming task that is simpler to do if all data structures are immutable.
3. Immutability of data structures has an impact on the performance of programs. Briefly describe two effects of immutability on performance, one negative and one positive.

Question 3 [4+2 marks]

1. Briefly describe two advantages of Haskell's module system compared to C's module system.
2. Briefly describe one advantage of Haskell's module system compared to Mercury's module system.

Question 4 [8 marks]

What are *accumulators*? What purpose do they serve? Give an example of their use, and show how they help in that case.

Question 5 [6 marks]

In Mercury, I/O actions operate on a data structure that represents the state of the world. How does Mercury ensure that at any point in time, there is *exactly one* reference to the current state of the world?

Question 6 [8 marks]

Project 2 used these data types to represent terms:

```
data Var = VarName String
    deriving (Eq, Ord, Read, Show)
data FSym = FSym String
    deriving (Eq, Ord, Read, Show)

data Term
    =   TermVar Var
    |   TermNum Int
    |   TermFSym FSym [Term]
    deriving (Eq, Ord, Read, Show)
```

Write a Haskell function to print values of type Term so that

- each variable, number or function symbol is on a line of its own;
- each argument term of a function symbol occupies a consecutive sequence of one or more lines;
- every argument is intended four columns more than the function symbol it is an argument of;
- there are no parentheses, commas or other punctuation, with the structure of the term being shown by indentation only.

For example, the term `TermFSym (FSym "f") [TermVar (VarName "X"), TermFSym (FSym "g") [TermVar (VarName "Y"), TermNum 5]]`, which is the Haskell representation of the logic term `f(X, g(Y, 5))`, should be printed out as

```
f
    X
    g
        Y
        5
```

The signature of your function should be

```
print_term :: Term -> IO ()
```

Question 7 [8+8=16 marks]

A *heap* is a tree that satisfies the *heap property*: if node A is the parent of node B, then the value in node A is at least as big as the value in node B.

Heaps can be represented in Haskell using the following type.

```
data Heap a = Empty | Node a (Heap a) (Heap a)
```

(a) Write a Haskell function

```
remove_biggest :: (Ord a) => Heap a -> (a, Heap a)
```

Given a heap of items, this function should remove the biggest item in the given heap, and return a tuple containing the removed item and the heap that remains after its removal.

When you remove the biggest item in the heap, which must be at the top, your algorithm should replace it with its bigger child. That item should in turn be replaced with its bigger child, and so on, until you come to a node that has no children, whose node should be replaced with the empty tree.

For example, removing the biggest item (7) from this heap

```
    7
  5   6
 1 3 4 2
```

should yield this heap:

```
    6
  5   4
 1 3   2
```

(b) Assume you have access to these two Haskell functions:

```
remove_biggest :: (Ord a) => Heap a -> (a, Heap a)
insert_into_heap :: (Ord a) => Heap a -> a -> Heap a
```

The meaning of the `remove_biggest` function is described above. The meaning of the `insert_into_heap` function is that `insert_into_heap oldheap item` should insert `item` into `oldheap` returning `newheap`, which contains all the items in `oldheap` as well as `item`, and which also has the heap property.

Write a Haskell function

```
heapsort :: (Ord a) => [a] -> [a]
```

that implements heapsort. Heapsort starts with an empty heap, and then inserts all the items in its input list into that heap. It then keeps removing the biggest item from the heap until the heap is empty. Obviously, the first item removed is the largest item in the input, and the last item removed is the smallest item in the input. Your function should return the list of items in the input in ascending order.