The University of Melbourne

Department of Computer Science and Software Engineering

COMP30020

Declarative Programming

Semester 1, 2012

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 etc, 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 are 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 filter`

(b) `:t (+)`

(c) `[1,2] ++ [] ++ [3]`

(d) `map (\x->[x]) [2]`

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

(f) `let x = x+1 in 1+1`

Question 2 [4 marks]

For each of the following pairs of terms, give the most general unifier, or state the unification fails.

(a) `f(a, X, Y)` and `f(Z, b, c)`

(b) `f(a, X, X)` and `f(Z, Y, c)`

(c) `f(a, X, X)` and `f(Z, b, Z)`

(d) `f(a, X, X)` and `f(Z, Z, a)`

Question 3 [10 marks]

Consider the following type declaration for representing statements (from the projects; the definition of types `Var` and `Expr` are not relevant to this question).

```
data Stmt
    = Assign Var Expr
    | ITE Expr [Stmt] [Stmt]
    | While Expr [Stmt]
```

Write a Haskell function

```
num_ass :: [Stmt] -> Int
```

which takes a list of statements and returns the number of assignment statements in it, including those nested within `ITE` and `While`.

Question 4 [3+2=5 marks]

(a) Prolog and Mercury use "`:-`" to separate clause heads and bodies, and "`,`" and "`;`" to separate subgoals within the body. What are the simplest logical meanings given to these three symbols?

(b) Consider the following definition of the list membership predicate:

```
member(X, [X|Z]).
member(X, [U|Z]) :- member(X, Z).
```

One logical meaning given to this predicate (in Prolog) is the following:

$$\forall X \forall Y (member(X, Y) \leftrightarrow \exists U \exists Z (Y = [X|Z] \lor Y = [U|Z] \land member(X, Z)))$$

What could be considered counter-intuitive about this meaning, and what feature of Mercury allows a more intuitive meaning?

Question 5 [5+5+5=15 marks]

Consider the following sections of code written in C, Haskell and Mercury, respectively (assume `cond` and `nextval` are defined elsewhere and the type of `x` is known):

```
while (cond(x))        findx x =                    findx(X) =
    x = nextval(x);        if cond x then               (cond(X) ->
                               findx (nextval x)            findx(nextval(X))
                           else                         ;
                               x                            X
                                                        ).
```

(a) For the code above, compare and contrast how easy it is for a programmer to determine what each section of code does.

(b) For the code above, compare and contrast the likely efficiency, briefly describing reasons for relative inefficiency.

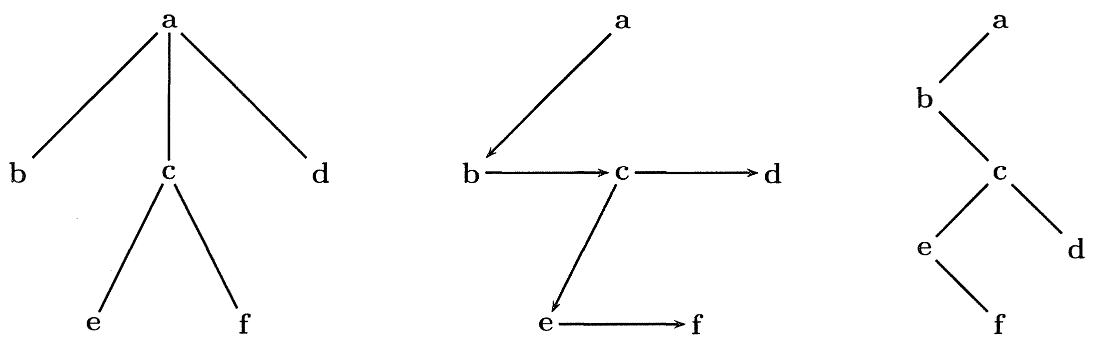
(c) Give one other important aspect/quality of code and briefly compare and contrast the code above with respect to this aspect/quality.

Question 6 [7+3+7+3=20 marks]

For this question you may use Haskell or Mercury. Consider the Haskell type definitions below (or the Mercury equivalent), for representing binary trees and general trees with any number of children per node (represented as a list), respectively.

```
data Btree a = Nil | Bnode a (Btree a) (Btree a)
data Gtree a = Gnode a [Gtree a]
```

There is a standard way of representing (a list of) general trees as a binary tree (and vice versa). The left child of a `Bnode` "points to" the first child of the corresponding `Gnode` and the right child "points to" the next sibling in the `Gtree` (or the next `Gtree` in the list if we are on the rightmost branch of the tree). The following diagram illustrates three equivalent views of a tree (the left is a `GTree`, the middle is a `BTree` drawn to emphasise its equivalence to the `GTree` and the right is the same `BTree` drawn in a more conventional way):



(a) Write a Haskell function

```
btree_gtrees :: Btree a -> [Gtree a]
```

which converts a `Btree` to the equivalent list of `Gtrees` (eg, going from right to left in the diagram above), or Mercury code which can do the same.

(b) What is the expected code structure for functions over `Btrees`, and does your code for part (a) follow this pattern? If not, can you briefly explain why?

(c) Write a Haskell function

```
gtrees_btree :: [Gtree a] -> Btree a
```

which converts a list of `Gtrees` to the equivalent `Btree`, or Mercury code which can do the same. If you wrote a suitable Mercury predicate for part (a), you can just give the mode declaration (no need to repeat the clauses).

(d) What is the expected code structure for functions over lists, and does your code for part (c) follow this pattern? If not, can you briefly explain why?