The University of Melbourne
Department of Computing and Information Systems

Semester 2, 2015 Sample Assessment

COMP30020/COMP90048
Declarative Programming

[redacted] Sample Answers Included [redacted]

Reading Time: 15 minutes Total marks for this paper: 100

Writing Time: 2 hours

This paper has 7 pages.

Authorised Materials:

Writing instruments (e.g., pens, pencils, erasers, rulers). No other materials and no electronic devices are permitted.

Instructions to Invigilators:

Each student should receive a script book.

The exam paper must remain in the exam room and be returned to the subject coordinator.

Instructions to Students:

Answer each question in the script book provided. **Begin each question on a fresh page, and write the question number in the upper right corner of the page.** Any unreadable answers will be considered wrong.

The marks for each question are listed at the beginning of the question. You should attempt all questions. Use the number of marks allocated to a question as a rough indication of the time to spend on it.

This paper should *not* be lodged with Baillieu Library.

This page intentionally left blank

Question 1 [12 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 (<)`

(b) `:t map (+3)`

(c) `:t zip [True,True,False]`

(d) `map (length < 3) [[1],[1,2,3]]`

(e) `filter (not.(==3)) [1,2,3]`

(f) `let e = head [] in 3`

Sample Answer to Question 1

(a) `(<) :: Ord a => a -> a -> Bool`

(b) `map (+3) :: Num b => [b] -> [b]`

(c) `zip [True,True,False] :: [b] -> [(Bool, b)]`

(d) *Type error*

(e) `[1,2]`

(f) `3`

Question 2 [18 marks]

In Haskell, the less than operator (`<`) can be applied to many different data types, including new types created by the programmer. Briefly (in two or three paragraphs) explain the feature of the Haskell type system which allows this. We would normally expect `<` to obey certain laws such as transitivity (`x<y` and `y<z` implies `x<z`). Does the Haskell implementation enforce such laws? If so, explain how. If not, explain whether it is important for the programmer to make sure they are obeyed.

Sample Answer to Question 2

In Haskell, *type classes* support a disciplined form of function *overloading*. A type class is a generalisation of a type: any number of types may be declared to be instances of a given type class, and all instances of a type class must define the functions (sometimes called *methods*) specified by the type class. Then a function may be defined to take any instance of a type class as input, knowing that all instances of the class will define those methods.

Haskell's less than function (<) is one of the methods of the `Ord` type class, the class of types whose values can be compared to decide if one value is smaller than another. Any type the programmer might define can then be defined to be an instance of `Ord` by defining the comparison functions for that type. Once this is done, then functions written to expect ordered values can be applied to values of that type. For example, if type `Foo` is declared to be an instance of `Ord`, then a list of `Foo` could be passed to `sort`.

Haskell does not enforce laws that might be expected to be respected for instances of a type class, such as transitivity of <. It is important that the programmer take care that such laws are obeyed by any < function they define for their own types, because functions written to be applied to instances of the class may count on such laws being obeyed. For example, if < were defined not to be transitive for some type, the behaviour of `sort` on lists of this type would be unreliable.

Question 3 [30 marks]

Consider the following Haskell type for ternary trees:

```
data Ttree t = Nil | Node3 t (Ttree t) (Ttree t) (Ttree t)
```

Suppose we have a `Ttree` of Doubles and we want a function to find the average of the numbers in the tree. Write a Haskell function which performs this task. If the `Ttree` is empty, your function should return 0.0. Include type declarations for all your functions. To obtain maximum marks, your code should use a single traversal over the tree and have $O(N)$ worst case time complexity.

Sample Answer to Question 3

Solution using custom fold function for Ttrees

```
ttreeAverage1 :: Ttree Double -> Double
ttreeAverage1 Nil = 0.0
ttreeAverage1 tt =
```

```
  let (sum,count) = foldTtree plusCount (0.0,0.0) tt
  in  sum / count


plusCount :: Double -> (Double, Double) -> (Double, Double)
plusCount n (sum,count) = (sum+n, count+1)


foldTtree :: (a -> b -> b) -> b -> Ttree a -> b
foldTtree _ acc Nil = acc
foldTtree f acc (Node3 n left mid right)
  = let acc1 = foldTtree f acc  right
        acc2 = foldTtree f acc1 mid
        acc3 = foldTtree f acc2 left
    in  f n acc3
```

Alternative full-credit solution

```
ttreeAverage2 :: Ttree Double -> Double
ttreeAverage2 Nil = 0.0
ttreeAverage2 tt =
  let (sum,count) = ttAverage' tt
  in  sum / count

ttAverage' :: Ttree Double -> (Double,Double)
ttAverage' Nil = (0.0,0.0)
ttAverage' (Node3 n left mid right) =
  let (suml,countl) = ttAverage' left
      (summ,countm) = ttAverage' mid
      (sumr,countr) = ttAverage' right
  in  (n+suml+summ+sumr, 1+countl+countm+countr)
```

Partial credit solution using two passes

```
-- Partial credit, two-pass solution.  This answer would still get
-- most of the marks.
ttreeAverage3 :: Ttree Double -> Double
ttreeAverage3 Nil = 0.0
ttreeAverage3 tt = ttSum tt / ttCount tt

ttSum :: Ttree Double -> Double
ttSum Nil = 0.0
ttSum (Node3 n left mid right) =
```

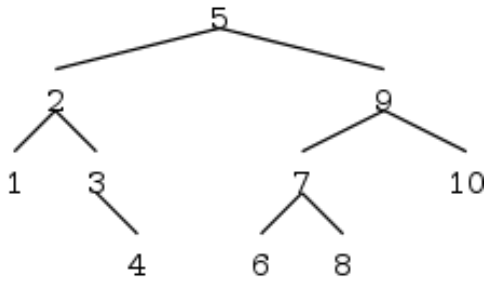
```
  n + ttSum left + ttSum mid + ttSum right

ttCount :: Ttree Double -> Double
ttCount Nil = 0.0
ttCount (Node3 n left mid right) =
  1 + ttCount left + ttCount mid + ttCount right
```

Question 4 [30 marks]

For this question, we will represent a set of integers as a binary tree in Prolog, using the atom `empty` to represent an empty tree or node, and `tree(L,N,R)` to represent a node with label `N` (an integer), and left and right subtrees `L` and `R`. Naturally, we also insist that `N` be strictly larger than any label in `L` and strictly smaller than any in `R`. We do not require that the tree be balanced. For example,

```
tree(tree(tree(empty, 1, empty),
          2,
          tree(empty, 3, tree(empty, 4, empty))),
     5,
     tree(tree(tree(empty,6,empty),
               7,
               tree(empty,8,empty)),
          9,
          tree(empty, 10, empty)))
```

is one possible representation of the set of numbers from 1 to 10. It might be visualized as



Write a predicate `intset_insert(N, Set0, Set)` such that `Set` is the same as `Set0`, except that `N` is a member of `Set`, but may or may not be a member of `Set0`. That is, either `N` is a member of `Set0` and `Set = Set0`, or `N` is not a member of `Set0` and is a member of `Set`, and other than that, `Set` is the

same as `Set0`. This predicate must work as long as `N` is bound to an integer and `Set0` is ground.

Hint: Prolog's arithmetic comparison operators are `<`, `>`, `=<` (not `<=`), and `>=`. You can also use `=` and `\=` for equality and disequality.

Sample Answer to Question 4

```
intset_insert(N, empty, tree(empty,N,empty)).
intset_insert(N, tree(Left,Val,Right), Result) :-
        (   N = Val
        ->  Result = tree(Left,Val,Right)
        ;   N < Val
        ->  Result = tree(Left1,Val,Right),
            intset_insert(N, Left, Left1)
        ;   Result = tree(Left,Val,Right1),
            intset_insert(N, Right, Right1)
        ).
```

Question 5 [10 marks]

Following is a definition of a Prolog predicate to compute the sum of a list of numbers. Transform this definition to be tail recursive. You need not show the steps of your transformation.

```
sumlist([], 0).
sumlist([N|Ns], Sum) :-
        sumlist(Ns, Sum0),
        Sum is N + Sum0.
```

Sample Answer to Question 5

```
sumlist(List, Sum) :- sumlist(List, 0, Sum).

sumlist([], Sum, Sum).
sumlist([N|Ns], Sum0, Sum) :-
        Sum1 is Sum0 + N,
        sumlist(Ns, Sum1, Sum).
```

— End of Paper —