Enrolment Number:

The University of Melbourne
School of Computing and Information Systems

Semester 2, 2018 Assessment

COMP30020 Declarative Programming

[redacted] Sample Answers Included [redacted]

Reading Time: 15 minutes

Writing Time: 2 hours

Total marks for this paper: 70

This paper has 6 pages, including this title page.

Authorised Materials:

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

Instructions to Invigilators:

Students will write their answers in the exam paper itself.

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

Instructions to Students:

Write your enrolment (student) number in the box above. Answer questions directly on this exam paper in the box(es) provided. Use the flip sides of pages for rough work. The last 2 pages are provided in case you need more space for any answers. If you use this overflow space, put a note where the answer belongs saying where the rest of the answer is.

This exam is worth 70% of your mark for this subject. 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. We have tried to provide ample space for your answers; do not take the amount of space provided for an answer as an indication of how much you need to write.

This paper must *not* be lodged with the university library.

Question 1 [16 marks]

For each of the following Haskell expressions, give its **type** (which may be a function type, may include type variables, and may include type class constraints) or indicate that it represents a type error. You need not write anything other than the type, or that it is an error.

Marking Scheme for Question 1

2 marks per part. No partial credit for almost correct answers, except: 1 mark for saying "error" without saying what kind or saying the wrong kind.

(a) `(==)`

`Eq a => a -> a -> Bool`

⇛

(b) `putStr`

`String -> IO ()`

⇛

(c) `map Just`

`[a] -> [Maybe a]`

⇛

(d) `(++ [True])`

`[Bool] -> [Bool]`

⇛

(e) `\x -> x []`

`([a] -> b) -> b`

⇛

(f) `(+1)`

`Num a => a -> a`

⇛

(g) `map (>"xyz")`

`[String] -> [Bool]` —**or**— `[[Char]] -> [Bool]`

⇛

(h) `flip filter`

`[a] -> (a -> Bool) -> [a]`

⇛

Question 2 [7 marks]

For each of the following Haskell expressions, give its **value**, or explain why it will produce an error or fail to terminate.

(a) `map length ["All", "mimsy", "were", "the", "borogoves"]`

`[3,5,4,3,9]`

⇛

(b) `filter ((>4) . length) ["All", "mimsy", "were", "the", "borogoves"]`

`["mimsy", "borogoves"]` ⇐

(c) `[reverse x | x<-["All", "mimsy", "were", "the", "borogoves"], length x <= 4]`

`["llA", "erew", "eht"]` ⇐

(d) `foldr (:) [0] [1, 2, 3, 4]`

`[1, 2, 3, 4, 0]` ⇐

(e) `foldl (flip (:)) [0] [1, 2, 3, 4]`

`[4, 3, 2, 1, 0]` ⇐

(f) `length $ take 10 [5..]`

`10` ⇐

(g) `length $ drop 10 [5..]`

infinite loop ⇐

Marking Scheme for Question 2

1 point for each part; no partial credit.

Question 3 [20 marks]

Write a Haskell function `subsequences :: [a] -> [[a]]` that returns a list of all the subsequences of the input list. That is, it returns a list of every list made up of some, all, or none of the elements of the input list, in the order they appear in the input list. For example,

- `subsequences []` = `[[]]`
- `subsequences [True,False]` = `[[],[True],[False],[True,False]]`
- `subsequences "abc"` = `["","a","b","ab","c","ac","bc","abc"]`

You may use any standard Haskell prelude functions you like in your implementation, but no functions from Haskell libraries. Note that the order of the subsequences in the resulting list does not matter, but the elements in each subsequence must appear in the order they appear in the input list.

Sample Answer to Question 3 ⇐

```
subsequences :: [a] -> [[a]]
subsequences [] = [[]]
```

```
subsequences (e:es) = restSeqs ++ map (e:) restSeqs
    where restSeqs = subsequences es
```

Question 4 [12 marks]

List all the solutions to the following Prolog goals (i.e., what will Prolog print if these are given as queries?). If the goal would not succeed, give the reason (failure, infinite loop, or error), and in the case of error, indicate what sort of error will occur.

(a) `length([1,2,3],4).`

⇛ `false` —**or**— `fail`

(b) `append(X,[Y|Z],[a,b,c]), append(X,[d|Z],List).`

⇛
```
List = [d, b, c], X = [], Y = a, Z = [b, c] ;
List = [a, d, c], X = [a], Y = b, Z = [c] ;
List = [a, b, d], X = [a, b], Y = c, Z = []
```
If they just show the values for `List`, that's good enough

(c) `length(List,Len), Len < 0.`

⇛ Infinite loop (runs out of stack eventually). It will not fail.

(d) `X = 6*7.`

⇛ `6*7` (not 42)

(e) `X = f(g(2,Y),Y,3), X = f(_,A,A).`

⇛
```
X = f(g(2, 3), 3, 3), Y = 3, A = 3   ---or---  equivalent
```
If they just show the value for `X`, that's good enough

(f) `f(g(3),Y,Y,g(3)) = f(A,g(A),B,B).`

⇛ `false` —**or**— `fail`

Marking Scheme for Question 4

2 marks per part.

Question 5 [7 marks]

Give the formal semantics (meaning) of the following Prolog program. Recall that the formal semantics of a logic program is the set of *ground unit clauses* that have the same meaning as the program itself. English descriptions of the meanings of the programs will receive no credit.

```
p(a,b).   p(b,c).   p(c,d).

q(X,Y) :- p(X,Y).
q(X,Y) :- p(Y,X).
q(X,Z) :- p(X,Y), q(Y,Z).

r(X,Z) :- r(X,Y), p(Y,Z).
r(X,Z) :- p(X,Y), r(Y,Z).
```

Sample Answer to Question 5 ⇐

```
p(a,b). p(b,c). p(c,d).

q(a,b). q(b,c). q(c,d).
q(b,a). q(c,b). q(d,c).
q(a,c). q(b,d). q(a,d).
q(a,a). q(b,b). q(c,c).
```

There should be no clauses for `r`/2. Order of clauses does not matter.

Marking Scheme for Question 5

1 for all correct `p`/2 clauses, and no extras

6 for all correct `q`/2 clauses, and no extras

-1 for each missing or extra clause

Question 6 **[8 marks]**

Fill in the **four** blanks in the following Prolog code to create correct implementations of these predicates as documented.

```
% same_length(L1, L2)
% List L1 has the same length as list L2.  This works in any mode, and
% L1 and L2 are always proper lists when this succeeds.
same_length([], []).
same_length( [_|Xs], [_|Ys] ) :-                                    ⇐
        same_length(Xs, Ys).
```

```
% select(Elt, List, Rest)
% Elt is an element of List, and Rest is all the other elements of
% List except Elt, in the same order as in List.  In other words,
% List is the same as Rest, except that it has Elt inserted somewhere.
% This works in any mode.
select(X, [X|Xs], Xs).
select( X, [A|Xs], [A|Ys] ) :-
⇛       select(X, Xs, Ys).


% tree_list(Tree, List)
% List is a list of all the labels in tree Tree in an inorder traversal.
% A tree is either 'empty' or a term 'tree(Left,Elt,Right)' where Elt
% is the root label of the tree and Left and Right are the left and
% right subtrees.
tree_list(Tree, List) :- tree_list( Tree, List, [] ).
⇛
% tree_list(Tree, List, List0)
% Equivalent to 'tree_list(Tree, List1), append(List1, List0, List)', but
% computes the result without appending.
tree_list(empty, List, List).
tree_list(node(Left,Elt,Right), List, List0) :-
        tree_list( Left, List, [Elt|List1] ),
⇛       tree_list(Right, List1, List0).
```

— End of Exam —