Name _____

Student Number _____

HAND IN
answers recorded
on question paper

BISHOP'S UNIVERSITY

DEPARTMENT OF COMPUTER SCIENCE

CS 403

MID-TERM EXAMINATION

31 June 1900

Instructor: Stefan D. Bruda

Instructions

- This examination is *85 minutes in length* and is *open book*. You are allowed to use any kind of printed documentation. Electronic devices are *not* permitted. *Any violation of these rules will result in the complete forfeiture of the examination.*
- There is no accident that the total number of marks add up to the length of the test in minutes. The number of marks awarded for each question should give you an estimate on how much time you are supposed to spend answering the question.
- *To obtain full marks provide all the pertinent details*. This being said, do not give unnecessarily long answers. In principle, all your answers should fit in the space provided for this purpose. If you need more space, use the back of the pages or attach extra sheets of paper. However, if your answer is not (completely) contained in the respective space, clearly mention within this space where I can find it.
- The number of marks for each question appears in square brackets right after the question number. If a question has sub-questions, then the number of marks for each sub-question is also provided.

When you are instructed to do so, turn the page to begin the test.

1		5	/	5				
2		5	/	5				
3		10	/	10				
4	a–d	20	/	20				
5	a–b	10	/	10				
6	a–b	15	/	15				
7	a–c	15	/	15				
	Total:	80	/	80	=	40	/	40

1. [5] Using a list comprehension, define a HASKELL function `multiples` that, given two integers `m` and `n`, returns a list of all the common multiples of `m` and `n` in increasing order. Note that in HASKELL `mod x y` is zero if and only if `x` is a multiple of `y`.

ANSWER:

```
multiples m n = [x | x <- [1..], mod x m == 0, mod x n == 0]
```

---

2. [5] Using `multiples` from the previous question define a function `lcm` such that `lcm x y` returns the least (or smallest) common multiple of the integers `x` and `y`.

ANSWER:

```
lcm x y = head (multiples x y)
```

---

3. [10] How will the expression `lcm 6 9` be evaluated if HASKELL used eager evaluation (or innermost reduction) and how does this differ from the way the HASKELL interpreter actually evaluates the expression?

ANSWER:

`multiples` generates an infinite list. In a lazy evaluated language that is not a problem since we only need the first element in that list (and thus the rest of the list is not evaluated). Therefore HASKELL will readily produce the least common multiplier (18). If eager evaluation is used however, the argument of `head` will have to be completely evaluated before the call to `head` takes place. Since the argument is an infinite list, this evaluation will result in a non-terminating computation.

---

4. [20] The following HASKELL code introduces a type suitable for storing arbitrary trees and provides an incomplete instantiation of that type from `Eq`.

```
data Tree a = Pair a [Tree a]

instance {-1-} Eq (Tree a) where
    {-2-}
```

(a) [5] Use `foldr` to define a function `concat :: [[a]] -> [a]` that receives a list of lists and returns all the lists in the argument concatenated together.

ANSWER:

```
concat = foldr (++) []
```

(b) [5] Define a function `breadthFirst` that receives a tree `t` and returns all the values from `t` listed in a breadth-first order (where a node's value is listed before all the values stored in that node's children). Include a type signature for your function.

ANSWER:

```
breadthFirst :: Tree a -> [a]

breadthFirst (Pair v ts) = v : concat (map breadthFirst ts)
```

(c) [5] Provide a replacement for `{-2-}` so that two trees are deemed equal if and only if their breadth-first traversals are the same.

ANSWER:

```
t1 == r2 = (breadthFirst t1) == (breadthFirst t2)
```

(d) [5] Provide a suitable replacement, if any for `{-1-}`. Explain your choice (even if nothing is needed you still need to explain why).

ANSWER:

In order to compare `Tree a` we compare their values (or type `a`) which as a consequence need to be comparable. The replacement is therefore `Eq a =>` .

---

5. [10] The following HASKELL function receives a list and returns a copy of it:

```
copy [] = []
copy (x:xs) = x : copy xs
```

(a) [5] Consider the following definition:

```
copy [] = []
copy xs = xs
```

Is this equivalent to the first definition? Explain.

ANSWER:

This will return any list it is applied to (without making a copy) but this result is indistinguishable from a copy for all practical purposes because HASKELL is purely functional.

(b) [5] Consider the following definition:

```
copy xs = xs
```

Is this equivalent to the first definition? Explain.

ANSWER:

The answer to the previous question also applies. In addition, this is more general; its type is `a -> a` and so it is applicable to anything, not just lists.

---

6. [15] Consider the following predicates: `food(X)` (true when `X` is food), `alive(X)` (true when `X` is alive), and `eats(X,Y)` (true when `X` eats `Y`).

   (a) [5] Represent in PROLOG the following statements using only the above predicates and maintaining the order in which the statements are given.

   > Apples, vegetables, and peanuts are food. Anything that someone eats and does not kill them is food. Adam eats peanuts and is still alive. John eats any kind of food.

ANSWER:

```
food(apples).                    % 1

food(vegetables).                % 2

food(peanuts).                   % 3

food(X) :- eats(Y,X), alive(Y). % 4

eats(adam,peanuts).              % 5

alive(adam).                     % 6

eats(john,X) :- food(X).         % 7
```

   (b) [10] Based on the knowledge base developed in the previous question what would be the answer to the query:

   ```
   ?- food(peanuts).
   ```

   Assume that you are willing to press the semicolon for as many times as necessary and so explain the *complete behaviour* of the PROLOG interpreter on this query including all the answers given and why they are given. You may want to draw (part of) the proof tree of the query but you are not required to do so.

   (*provide your answer on the next page*)

ANSWER:

Peanuts are food for two reasons, so the interpreter will say for a starter `true` two times: First peanuts are food according to Fact #3. Peanuts are food a second time because of Rule #4. Indeed, Adam eats peanuts (Fact #5) and is alive (Fact #6), thus satisfying both premises of that rule.

The third attempt at proving that peanuts are food turn into a circular argument, as follows: Upon a request for resatisfaction at the prompt, the resatisfaction of `alive(X)` fails (according to the knowledge base Adam is the only one alive), and so a redo request is issued for `eats(Y,peanuts)`. Fact #5 was already used so we use the second match which is Rule #7. This rule in turn requires the satisfaction of `food(peanuts)` which brings us to the starting point, meaning that we get another `true` according to Fact #3, yet another one by Rule #4, and yet another match with Rule #7. Rinse and repeat.

In all, the resatisfaction of the query succeeds for as many times as we care to type that semicolon.

7. [15] Recall that the predicate `append/3` is predefined in PROLOG in such a way that `append(X,Y,Z)` succeeds whenever `Z` can be bound to the concatenation of `X` and `Y`.

   (a) [5] Using `append/3` (two times), define a PROLOG predicate `append/4` such that `append(L1,L2,L3,Z)` succeeds whenever `Z` can be bound to the concatenation of `L1`, `L2`, and `L3` (in this order).

   ANSWER:

   ```
   append(L1,L2,L3,Z) :- append(L1,L2,R), append(R,L3,Z).
   ```

   (b) [5] Define a PROLOG predicate `sum/2` such that `sum(L,N)` receives a list `L` of numbers and binds `N` to the sum of all the values in `L`.

   ANSWER:

   ```
   sum([],0).

   sum([H|T],N) :- sum(T,N1), N is H+N1.
   ```

   (c) [5] Define the PROLOG predicate `maxsum/3` which solves a modified version of the maximum sum problem as follows: Given a list `L` of numbers and another number `M`, `maxsum(L,M,T)` binds `T` to a sub-list of `L` such that the sum of all the values in `T` exceeds `M`.

   ANSWER:

   Trivial generate and test:

   ```
   maxsum(L,M,T) :- append(_,L,_,T), sum(T,N), N >= M.
   ```
