ECM2418

(with Answers)

UNIVERSITY OF EXETER

COLLEGE OF ENGINEERING, MATHEMATICS AND PHYSICAL SCIENCES

COMPUTER SCIENCE

Examination, January 2018

Computer Languages and Representations

Module Leader: Dr David Wakeling

Duration: TWO HOURS

Answer question 1, and any two of the three other questions.

Question 1 is worth 40 marks. Other questions are worth 30 marks each.

The marks for this module are calculated from 70% of the percentage mark for this paper plus 30% of the percentage mark for associated coursework.

No electronic calculators of any sort are to be used during the course of this examination.

This is a CLOSED BOOK examination.

SECTION A (Compulsory Question)

Question 1

(a) Write the type of the Haskell function "`f`" below, and redefine it using pattern matching instead of guards.

```
f a
  |  length a == 1  =  head a
  |  otherwise      =  f (tail a)
```

BOOKWORK

```
f ::  [a] -> a
f [x]
  =  x
f (x:xs)
  =  f xs
```

For type **[1 mark]**. For equations **[3 marks]**.

(4 marks)

(b) Explain what is meant by lazy evaluation in Haskell, and give an example of lazy evaluation.

BOOKWORK

In Haskell, lazy evaluation evaluates an expression only when it is needed to compute the overall result **[1 mark]**, and then only as much as is needed to compute the overall result **[1 mark]**. An example of lazy evaluation might be an expression that represents an infinite list of integers, only some leading portion of which may be used **[1 mark]**.

```
ints ::  [Int]
ints
  =  [1..]
```

For an example **[1 mark]**.

(4 marks)

(c) Write a Haskell data type declaration for polymorphic binary trees that can be printed.

BOOKWORK

```
data Tree a
  =  Leaf
  |  Node (Tree a) a (Tree a)
  deriving Show
```

For data declaration **[1 mark]**. For constructors **[2 marks]**. For deriving Show **[1 mark]**.

(4 marks)

(d) Write the type of the Haskell function "g" below and using an example argument, explain what it computes.

```
g (w:ws)
  =  g ws ++ [ w ]
g []
  =  []
```

BOOKWORK

```
g ::  [a] -> [a]

g [5,4,3,2,1]
===> g [4,3,2,1] ++ [5]
===> g [3,2,1] ++ [4] ++ [5]
===> g [2,1] ++ [3] ++ [4] ++ [5]
===> g [1] ++ [2] ++ [3] ++ [4] ++ [5]
===> g [] ++ [1] ++ [2] ++ [3] ++ [4] ++ [5]
===> [] ++ [1] ++ [2] ++ [3] ++ [4] ++ [5]
===> [1,2,3,4,5]
```

For type **[1 mark]**. For trace **[2 marks]**. For saying this is reverse **[1 mark]**.

(4 marks)

(e) What is meant by logic programming and what is the idea behind it? Why is Prolog a logic programming language?

BOOKWORK

Logic programming is essentially programming computers via logic, in particular via a form of predicate calculus **[1 mark]**. The idea behind logic programming is that the programmer gives to the computer the information on how to solve a problem via predicate calculus formulae, and the process that the computer uses to solve the problem is a form of logical deduction **[1 mark]**.

Prolog is a logic programming language because it allows the programmer to specify the program via logical clauses **[1 mark]**, and uses a logical reasoning engine based on resolution to give the answers to the queries issued by the user **[1 mark]**.

(4 marks)

(f) What are the four types of equality in Prolog? Briefly explain the meaning of each.

BOOKWORK

The four kinds of equality in Prolog are:

- `X = X`. This is true if `X` and `X` can be unified syntactically (no arithmetical evaluation is involved).
- `X is Y`. Here `X` must be a variable and `Y` an arithmetical expression; `X` is instantiated to the value of `Y`.
- `X =:= Y`. This is true if `X` and `Y` are arithmetical expressions and their values are equal.
- `X == Y`. This is true if `X` and `Y` are syntactically identical; (again, no arithmetical evaluation is involved).

(1 mark each = **4 marks**)

(4 marks)

(g) What is the meaning and use of the "cut" predicate '`!`' in Prolog?

BOOKWORK

The symbol '`!`' is the "cut" symbol. As a goal, it always succeeds, but

only once, and then acts as a barrier to backtracking.

[2 marks]

In the query

```
?- a, b, !, c, d
```

if `d` fails, we can backtrack to `c`, but if `c` fails we cannot backtrack to `b`. The cut symbol is used to prune unwanted branches from the resolution tree.

[2 marks]

(4 marks)

(h) Explain the difference between a deterministic and a non-deterministic finite-automata (FA). In particular:

(i) how the transition function is specified; and

(ii) what the acceptance conditions for these two kinds of automata are.

(iii) Is it true that non-deterministic FAs are more powerful than deterministic FAs (i.e., non-deterministic ones recognise more languages than deterministic ones)? Give your reasons.

BOOKWORK

(i)
- In a deterministic finite-automata (FA), there is at most one possible next step when a given symbol is read by the machine in a given state **[1 mark]**.
- In a non-deterministic FA, there can be more than one possible next step when a given symbol is ready by the machine in a given state **[1 mark]**.

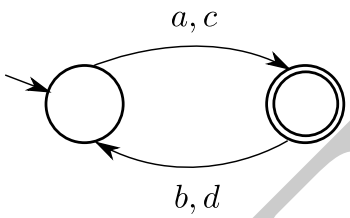
(ii)
- A deterministic FA D accepts a string w if the sequence of transitions performed by D when processing w terminates in an accepting state **[1 mark]**.
- A non-deterministic FA N accepts a string w if there is at least one possible sequence of transitions for N when processing w terminating in an accepting state **[1 mark]**.

(iii) Non-deterministic FAs are *not* more powerful than deterministic

FAs **[1 mark]**, because for any non-deterministic FA it is possible to derive a deterministic one recognising the same language **[1 mark]**.

(6 marks)

(i) Consider the following automaton.



(i) State which of the following strings are accepted, and which are rejected, by the automaton:

$$\Lambda, cda, acbc, abdc, abcda.$$

(2 marks)

(ii) Write down a regular expression for the language accepted by the automaton.

(2 marks)

(iii) Consider the languages L_1 and L_2 defined by the regular expressions $a(bc)^+$ and $a(bdc)^*$ respectively. For L_1 and L_2, state whether they are a subset of the language L of the automaton above giving your reasons.

(2 marks)

BOOKWORK

(i) Accepted strings: $cda, abcda$ **[1 mark]**.
Rejected string: $\Lambda, acbc, abdc$ **[1 mark]**.

(ii) The regular expression representing the language recognised by the automaton is $(a + c)((b + d)(a + c))^*$

[2 marks].

(iii) L_1 is a subset of L. In particular, if w is a string of L_1, then w has to start with an 'a', which is compatible with the initial part $(a \cup c)$ of the expression representing L. The remaining characters of w

have to follow the expression $(bc)^+$, which describes strings that are compatible with $((b+d)(a+c))^*$ **[1 mark]**.

L_2 is not a subset of L, because there exists a string, say $abdc$, which belongs to L_2 but not to L **[1 mark]**.

(Total 40 marks)

SECTION B (Optional Questions)

Question 2

(a) Define two Haskell functions, "`keep`" and "`lose`", along with their types. The behaviour of these functions is described by the following examples.

```
keep 3 [1,2,3,4,5]  ===>  [1,2,3]
keep 3 [1,2]        ===>  [1,2]
keep 3 []           ===>  []
keep 0 [1,2]        ===>  []

lose 3 [1,2,3,4,5]  ===>  [4,5]
lose 3 [1,2]        ===>  []
lose 3 []           ===>  []
lose 0 [1,2]        ===>  [1,2]
```

```
keep ::  Int -> [a] -> [a]
keep n []
  =  []
keep 0 xs
  =  []
keep n (x:xs)
  =  x :  keep (n-1) xs

lose ::  Int -> [a] -> [a]
lose n []
  =  []
lose 0 xs
  =  xs
lose n (x:xs)
  =  lose (n-1) xs
```

For each polymorphic type **[1 mark]**; for each recursion equation **[1 mark]**.

(8 marks)

(b) The maximum value in a list of integers may be found by dividing the list in half, finding the maximum value in each half, and taking the maximum of those two values. Use the functions "`keep`" and "`lose`" of part (a) to express this algorithm as a Haskell function with the type

```
maxlist :: [Int] -> Int
```

```
maxlist ::  [Int] ->Int
maxlist [x]
  =  x
maxlist xs
  =  if ml > mr then ml else mr
     where
     h  =  length xs `div` 2
     l  =  keep h xs
     r  =  lose h xs
     ml =  maxlist l
     mr =  maxlist r
```

For base case, **[1 mark]**. For recursive case: conditional **[1 mark]**; for dividing list **[3 marks]**; for recursive calls **[1 marks]**.

(6 marks)

(c) How could parallelism annotations be used to implement the "`maxlist`" function of part (b) efficiently on a parallel machine?

The key insight is that maximum of each half of a list can be evaluated in parallel **[1 mark]**. However, it is only necessary to launch a parallel task with "`par`" for one half of the list, while the other is evaluated by the launching task **[1 mark]**. There is no need for "`seq`" here, as the comparison to find the maximum of the two halves does the forcing of evaluation **[1 mark]**.

```
maxlist xs
  =  par ml (if ml > mr then ml else mr)
     where
     h  =  length xs `div` 2
     l  =  keep h xs
     r  =  lose h xs
     ml =  maxlist l
     mr =  maxlist r
```

For the placement of "`par`" here **[1 mark]**.

(4 marks)

(d) What four factors should a functional language implementation consider before choosing to evaluate an expression in parallel?

An implementation should consider whether:

- the expression will be needed **[1 mark]**;
- the expression is large enough to warrant a task **[1 mark]**;
- there is a spare processor **[1 mark]**;
- communication with the spare processor is cheap enough **[1 mark]**.

(4 marks)

(e) How might a function "`maxlist2`" find the maximum value in a list of integers using a higher-order function?

```
maxlist2 ::  [Int] ->Int
maxlist2 (x:xs)
  =  foldr max x xs
```

For identifying "`foldr`" **[1 mark]**. For choosing "`max`" as the binary operator **[1 mark]**, for choosing the head and tail of the list as second **[1 mark]** and third arguments **[1 mark]**.

(4 marks)

(f) Nowadays, most computers are multiprocessors. Does this strengthen the case for functional programming or not? Argue one way or the other.

Either argue yes, as the number of processors, n, becomes large (say, more that 32), it difficult for a programmer to find and exploit opportunities for parallel evaluation in their program fully, except in the case of some highly regular array-based computations. By "letting go" of some control, the programmer can focus of what is to be computed, leaving the implementation to focus of how this should best be done. This separation of concerns makes programming a large number of processors tractable **[4 marks]**.

Or argue no, although the number of processors, n, is becoming large (say, more that 32), it is still possible to write parallel programs for them, especially in the case of some highly regular array-based computations. There is a huge investment in traditional programming that would be "thrown away" completely by using functional programming languages for parallel programming. They lack lack the algorithms, tools, and

programmers necessary to make this practical **[4 marks]**.

(4 marks)

(Total 30 marks)

Question 3

(a) Explain how lists are represented in Prolog, using the "`[X|Y]`" notation. Write down two different ways of denoting a list with one element, and two different ways of denoting a list with three elements.

The empty list is represented by `[]` **[1 mark]**. The list whose head is `X` and tail is `Y` is represented by `[X|Y]` **[1 mark]**. One can also write e.g., `[X1,X2,X3|Y]` instead of `[X1|[X2|[X3|Y]]]` **[1 mark]**, and `[X]` instead of `[X|[]]` **[1 mark]**.

(4 marks)

(b) (i) Define Prolog clauses for a function `append(L1,L2,L)` with the following meaning: `L1`, `L2`, and `L`, are lists, and `L` is the list obtained by concatenating `L1` and `L2` (first `L1` and then `L2`).

(4 marks)

(ii) Give an example of two queries for the function `append` defined above such that one query returns `true` and the other `false`

(2 marks)

(i) `append([],L2,L2).` **(2 marks)**
`append([H1|T1],L2,[H1|REST]):-append(T1,L2,REST).`
(2 marks)

(ii) Query returning `true`: `?- append([],[1],[1]).`

Query returning `false`: `?- append([],[1],[]).`

(**1 mark** each)

(c) Define Prolog clauses for a function `member(X,L)` that is true whenever `X` is an element of the list `L`. Use the cut predicate '`!`' and the anonymous variable '`_`' where possible (to get full marks).

```
member(X,[X|_]):- !.
member(X,[_|T]):- member(X,T).
```
(**4 marks**, of which: **1 mark** if cut is used, and **1 mark** if anonymous variables are used)

(4 marks)

(d) Define Prolog clauses for a function `isSet(L)` that holds whenever the list `L` is a set (i.e., no element of `L` is contained more than once in `L`). (**Hint**: use the function `member` defined above).

```
isSet([]).
isSet([X|T]):- \+member(X,T),isSet(T).
```
(2 marks) **(2 marks)**

(4 marks)

(e) Define Prolog clauses for a function `intersection(S1,S2,S)` with the following meaning: `S1`, `S2`, and `S`, are sets represented via lists, and `S` is the set obtained by intersecting `S1` and `S2`. (**Hint**: use the function `member` defined above). Use the cut predicate '`!`' if possible to improve the efficiency of your function.

```
intersection([],S2,[]).
intersection([X1|REST1],S2,[X1|REST]):-
    member(X1,S2),!,intersection(REST1,S2,REST).
intersection([X1|REST1],S2,REST):-intersection(REST1,S2,REST).
```
[1 mark] **[2 marks]** **[2 marks]**

(An additional **1 mark** if cut is used)

(6 marks)

(f) Define Prolog clauses for a function `union(S1,S2,S)` with the following meaning: `S1`, `S2`, and `S`, are sets represented via lists, and `S` is the set obtained by the union of `S1` and `S2`. (**Hint**: use the function `member` defined above). Use the cut predicate '`!`' if possible to improve the efficiency of your function.

Since the result must be a set, when we add the elements of `S1` to `S2` we have to check that those elements are not already in `S2`.

```
union([],S2,S2).
union([X1|REST1],S2,REST):-
    member(X1,S2),!,union(REST1,S2,REST).
union([X1|REST1],S2,[X1|REST]):-union(REST1,S2,REST).
```
[1 mark] **[2 marks]** **[2 marks]**

(An additional **1 mark** if cut is used)

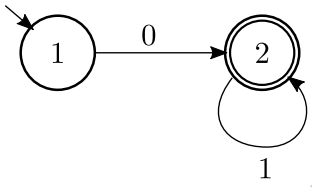
(6 marks)

(Total 30 marks)

Question 4

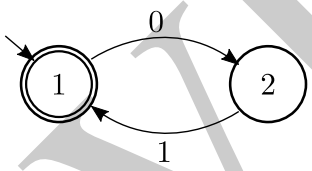
(a) Define deterministic finite-state automata accepting the regular languages 01^* and $(01)^*$.

The following automaton recognises 01^*:



(3 marks)

The following automaton recognises $(01)^*$:



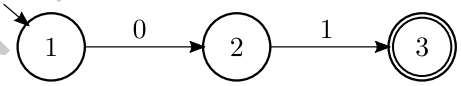
(3 marks)

(6 marks)

(b) What is the intersection of the regular languages considered in part (a)? Define a deterministic finite-state automaton recognising this intersection.

The intersection is just one string: 01 **(2 marks)**.

The automaton recognising the intersection is:



(2 marks)

(4 marks)

(c) Consider the language L whose regular expression is $(ab + aba)^*$.

(i) Describe in words the language L.

(2 marks)

(ii) Write two strings belonging to L and two strings not belonging to L,

respectively.

(4 marks)

(iii) Define a non-deterministic finite-state automaton recognising L. State where the non-determinism occurs.

(6 marks)

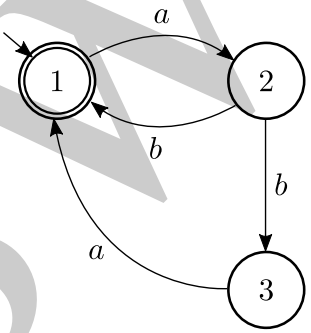
(iv) Transform the non-deterministic finite-state automaton recognising L above into a deterministic one.

(8 marks)

(i) L is the language whose strings are formed by sequences of ab or aba. The empty string belongs to L **(2 marks)**.

(ii) Strings belonging to L: Λ, aba (**(1 mark)** each). Strings not belonging to L: a, b (**(1 mark)** each).

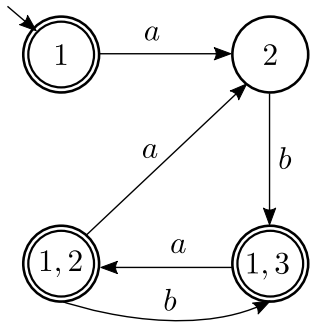
(iii) A non-deterministic automaton recognising L is:



The non-determinism occurs in the state 2 where the machine, when reading b, may move to state 1 or 3.

(6 marks), of which **1 mark** for stating correctly where the non-determinism is.

(iv) The deterministic automaton derived from the automaton above is:



(8 marks)

(Total 30 marks)