ECM2418

(with Answers)

UNIVERSITY OF EXETER

COLLEGE OF ENGINEERING, MATHEMATICS AND PHYSICAL SCIENCES

COMPUTER SCIENCE

Examination, January 2021

Computer Languages and Representations

Module Leader: Dr David Wakeling

Duration: TWO HOURS + 30 MINUTES UPLOAD TIME

Answer all questions.

Questions 1 and 2 are worth 40 marks each; question 3 is worth 20 marks.

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.

This is an OPEN BOOK examination.

Question 1

(a) Explain what the software crisis is, and why functional programming languages might provide a solution to it.

> The software crisis is that it has long been difficult to produce software on time and to specification **[1 mark]**. Functional programming languages might provide a solution to the software crisis because they eliminate the notion of sequence and state, making it easier to reason about programs **[1 mark]**. The notion of sequence means that statements must be written in just the right order **[1 mark]**. The notion of state means that changes must be made in just the right way **[1 mark]**. Mistakes in either the order of statements or in the changes to state lead to errors **[1 mark]**.

(5 marks)

(b) Write a simpler definition of the Haskell function "`both`" below using a single application of "`map`", and show how you would use it to add 1 to every number in a list.

```
both f g xs
   = (map f . map g) xs
```

> A simpler definition of the function "`both`" using a single application of "`map`" is
>
> ```
> both2 f g
>    = map (f . g)
> ```
>
> One mark for composition, one for eta-reduction **[2 marks]**.
>
> One way to use it to add 1 to every number in a list is
>
> ```
> both2 (\x -> x + 1) (\x -> x) [1,2,3]
> ```
>
> One mark for each argument **[3 marks]**.

(5 marks)

(c) Simplify this λ-expression, showing your working

$$(\lambda x.\ \lambda y.\ \lambda z.\ z\ y\ x)\ 6\ (\lambda x.\ x\ +\ 1)\ (\lambda x.\ x)$$

$$
\begin{aligned}
& (\lambda x.\ \lambda y.\ \lambda z.\ z\ y\ x)\ 6\ (\lambda x.\ x\ +\ 1)\ (\lambda x.\ x) \\
\Rightarrow\ & (\lambda y.\ \lambda z.\ z\ y\ 6)\ (\lambda x.\ x\ +\ 1)\ (\lambda x.\ x) \\
\Rightarrow\ & (\lambda z.\ z\ (\lambda x.\ x\ +\ 1)\ 6)\ (\lambda x.\ x) \\
\Rightarrow\ & (\lambda x.\ x)\ (\lambda x.\ x\ +\ 1)\ 6 \\
\Rightarrow\ & (\lambda x.\ x\ +\ 1)\ 6 \\
\Rightarrow\ & 6 + 1 \\
\Rightarrow\ & 7
\end{aligned}
$$

One mark for each reduction (except last trivial one). Partial marks awarded for partially-correct simplifications **[5 marks]**.

(5 marks)

(d) Given the Haskell data type

```
data Mystery
   = Many Mystery Char Mystery
   | One Char
   | Zero
```

write a function "`toString`" that converts values of this data type to strings.

The "`toString`" function is as follows

```
toString ::  Mystery -> String
toString (Many t x u)
   = toString t ++ [ x ] ++ toString u
toString (One x)
   = [ x ]
toString Zero
   = []
```

[5 marks]

(5 marks)

(e) What are the advantages of a strongly-typed functional programming language over a weakly-typed one?

The advantages of a strongly-typed functional programming language over a weakly-typed one are that:

- types succinctly describe a program design **[1 mark]**;
- types catch program errors at compile-time **[1 mark]**;
- types prevent program errors at run-time **[1 mark]**;
- types speed program development **[1 mark]**;
- types provide useful program documentation **[1 mark]**.

(5 marks)

(f) Rewrite the function "`strange`" without using a higher-order function, add its polymorphic type, and say what it computes.

```
strange
   = map (\(x,y) -> (y,x))
```

The "`strange`" function may be written without using a higher-order function as follows.

```
strange ::  [ (a,b) ] -> [ (b,a) ]
strange []
   = []
strange ((x,y):xys)
   = (y,x) :  strange xys
```

For the type **[1 mark]**. For the definition, **[3 marks]**. This function takes a list of pairs and returns a the list of pairs with the left and right elements of each pair swapped **[1 mark]**.

(5 marks)

(g) Rewrite the Haskell function "`stuff`" below using pattern-matching and guards.

```
stuff xs y
  = if null xs then
          [ y ]
       else if y < head xs
          y :  xs
       else
          head xs :  stuff (tail xs) y
```

The "`stuff`" function may be written using guards function as follows.

```
stuff [] y
   = [ y ]
stuff (x:xs) y
   | x < y     = y :  x :  xs
   | otherwise = x :  stuff xs y
```

For first clause **[2 marks]**. For second (guarded) clause **[3 marks]**.

(5 marks)

(h) Rewrite the Haskell function "`mash`" below, making the minimum changes to the right-hand side of each clause required for it to have the type given.

```
mash ::  ([a],[b]) -> [(a,b)]
mash []
   = ([], [])
mash ((a:as), (b:bs))
   = (a, b) ++ mash (as, bs)
```

The first clause should be:

```
mash ([], [])
   = []
```

[2 marks]

The second clause should be:

```
mash ((a:as), (b:bs))
   = (a, b) :  mash (as, bs)
```

[3 marks]

(5 marks)

(Total 40 marks)

Question 2

(a) Table 1 shows the Morse Code representation of three letters.

Letter	Representation
A	Dot Dash
B	Dash Dot Dot Dot
C	Dash Dot Dash Dot

Table 1: The Morse Code representation of three letters.

Write Prolog predicates to express the information in Table 1, and show how these predicates may be used in queries for Morse Code encoding, and for Morse Code decoding.

```
morse( 'A', [ dot, dash ]).
morse( 'B', [ dash, dot, dot, dot ]).
morse( 'C', [ dash, dot, dash, dot ]).
```

[3 marks]

```
morse( 'A', X ) % encode
morse( X, [ dot, dash ]) % decode
```

[2 marks]

(5 marks)

(b) Write a Prolog predicate that may be used to naively translate a sequence of English words directly into their French counterparts. For example,

```
translate( [ he, is, an, idiot ], F ).
F = [ il, est, un, imbecile ]
```

```
dictionary( he, il ).
dictionary( is, est ).
dictionary( an, un ).
dictionary( idiot, imbecile ).

translate( [], []).
translate( [X|XS], [Y|YS] )
   :- dictionary( X, Y ), translate( XS, YS ).
```

For dictionary **[2 marks]**; for translation **[3 marks]**.

(5 marks)

(c) If it is possible to unify the terms:

- $g(a, U, h(T), V)$;
- $g(T, b, h(a), h(b))$

show the substitution that unifies them; otherwise say why such a substitution cannot be produced.

It is possible to unify the two terms **[1 mark]**. The substitution is a set **[1 mark]**. In this case, { T = a, U = b, V = h(b) } **[3 marks]**.

(5 marks)

(d) What was the Japanese Fifth Generation Project, and why was it based on Prolog?

The Japanese Fifth Generation Project was Japan's effort to become a leader in the computer industry by developing large computer systems for Artificial Intelligence **[1 mark]**. It was based on Prolog because:

(i) Logic programs provide a suitable formalism for encoding specifications **[1 mark]**;

(ii) Logic programs can be automatically transformed into more efficient ones **[1 mark]**;

(iii) Logic programs can be efficiently implemented on massively-parallel computers **[1 mark]**;

(iv) Logic programs incorporate the rule-based inference engine needed for artificial intelligence **[1 mark]**.

(5 marks)

(e) Write a Prolog predicate that may be used to intersperse a separator among the elements of a list. For example,

```
intersperse( [1,2,3], 'A', X ).
X = [1,'A',2,'A',3]
```

```
intersperse( [],    X, [] ).
intersperse( [H],   X, [H]).
intersperse( [H|T], X, [H,X|W])
   :- intersperse( T, X, W ).
```

For first clause **[1 mark]**, for second clause **[1 mark]**, and for final clause **[3 marks]**.

(5 marks)

(f) A rushed Prolog programmer has written the following code. By tracing an example query, briefly explain what it computes.

```
tonto( [X|Y], Z, W )
   :- member( X, Z ), tonto( Y, Z, W ).
tonto( [X|Y], Z, [X|W] )
   :- \+ member( X, Z ), tonto( Y, Z, W ).
tonto( [], Z, Z ).
```

```
tonto( ['A', 'B', 'C'], [1, 2, 3 ], R )
R ⇒ tonto( ['B', 'C'], [1,2,3], ['A'|W1] )
W1 ⇒ tonto( ['C'], [1,2,3], ['B'|W2] )
W2 ⇒ tonto( [], [1,2,3], ['C'|W3] )
W3 ⇒ [1,2,3]
W2 ⇒ ['C',1,2,3]
W1 ⇒ ['B','C',1,2,3]
R ⇒ ['A','B','C',1,2,3]
```

List union.

[5 marks]

(5 marks)

(g) Does Prolog need the predicates `number(` T `)` and `string(` T `)`, for some term T? Give your reasons.

The predicate `number(` T `)` is true if T is bound to a number **[1 mark]**, and the predicate `string(` T `)` is true if T is bound to a string **[1 mark]**. Prolog does not need the `number` and `string` predicates **[1 mark]** — one could write programs that would pass a static type-checker **[1 mark]**, and these tests would be unnecessary

because operations would always be applied to operands of the right type **[1 mark]**.

(5 marks)

(h) Among the most dangerous creatures in the world are crocodiles, lions, tigers and wolves. Write Prolog predicates to capture this information, and explain what the "closed world assumption" means in the context of these predicates.

```
dangerous( crocodile ).
dangerous( lion ).
dangerous( tiger ).
dangerous( wolf ).
```

For these clauses **[2 marks]**.

The "closed world" assumption means that animals not mentioned in these predicates are not considered to be dangerous **[1 mark]**. Thus, a computer would not consider a rhino to be dangerous **[1 mark]**, although it is **[1 mark]**.

(5 marks)

(Total 40 marks)

Question 3

(a) Consider the Deterministic Finite State Automaton (DFA) shown in Figure 1

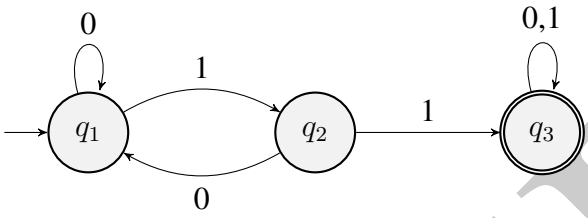


Figure 1: A DFA.

Show the transition table for this DFA and describe (in words) the strings that it accepts.

The transition table is

		0	1
$\rightarrow$	q_1	q_1	q_2
	q_2	q_1	q_3
*	q_3	q_3	q_3

[3 marks]

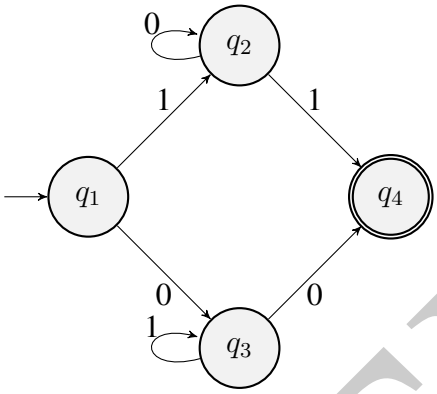
This DFA accepts strings that contain at least two successive '1's.

[2 marks]

(5 marks)

(b) Show a Deterministic Finite State Automaton (DFA) that accepts strings from the alphabet { '0', '1' } that are either:

- a '1', followed by zero or more '0's, followed by a '1';
- a '0', followed by zero or more '1's, followed by a '0'.



For states **[2 marks]**; for transitions **[3 marks]**.

(5 marks)

(c) Consider the Nondeterministic Finite State Automaton (NFA) shown in Figure 2.

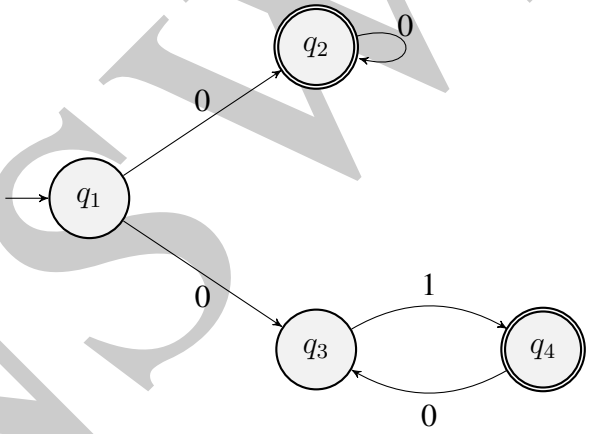


Figure 2: An NFA.

Show the transition table for this NFA, and write the regular expression describing the strings that it accepts.

The transition table is:

	0	1	ϵ
$\rightarrow q_1$	$\{q_2, q_3\}$	$\emptyset$	$\emptyset$
$* q_2$	$\{q_2\}$	$\emptyset$	$\emptyset$
q_3	$\emptyset$	$\{q_4\}$	$\emptyset$
$* q_4$	$\{q_3\}$	$\emptyset$	$\emptyset$

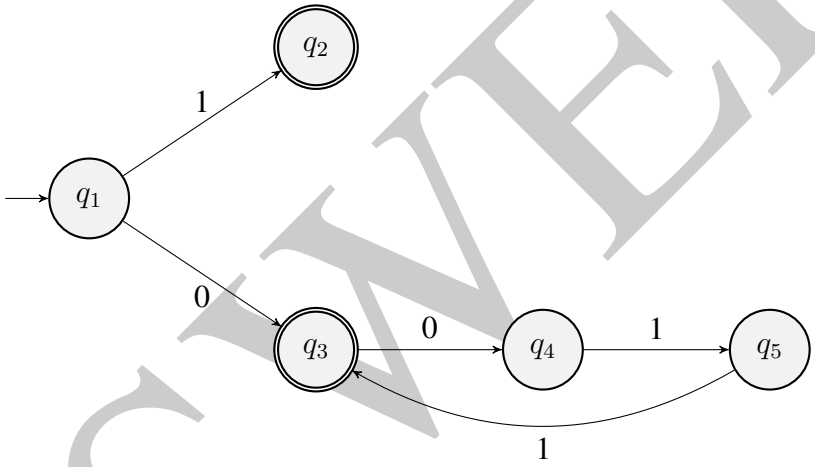
[3 marks]

This NFA accepts the strings described by the regular expression "`00* ∪ 01(01)*`".

[2 marks]

(5 marks)

(d) Show a Deterministic Finite State Automation (DFA) that accepts strings described by the regular expression "`0(011)* ∪ 1`".



For states **[2 marks]**; for transitions **[3 marks]**.

(5 marks)

(Total 20 marks)