UNIVERSITY OF EDINBURGH

COLLEGE OF SCIENCE AND ENGINEERING

SCHOOL OF INFORMATICS

INFORMATICS 1A

Tuesday 23 August 2005

09:30 to 11:30

Convener: M Jerrum
External Examiner: R Irving

INSTRUCTIONS TO CANDIDATES

1. **Candidates in the third or later year of study for the degrees of MA(General), BA(Relig Stud), BD, BCom, BSc(Social Science), BSc(Science) and BEng should put a cross (X) in the box on the front cover of the script book.**
2. **Answer Part A and Part B in SEPARATE SCRIPT BOOKS. Mark the question number clearly in the space provided on the front of the book.**
3. **Note that THERE ARE 9 QUESTIONS AND ALL QUESTIONS ARE COMPULSORY.**
4. **DIFFERENT QUESTIONS MAY HAVE DIFFERENT NUMBERS OF TOTAL MARKS. Take note of this in allocating time to questions.**

Write as legibly as possible.

THIS EXAMINATION WILL BE MARKED ANONYMOUSLY

PART A: COMPUTATION AND LOGIC

1. You are given the following argument in English:

   "If the temperature and air pressure remain constant then it does not rain. The temperature is constant. Therefore if it has rained then the air pressure has not remained constant."

   (a) Represent the argument above using propositional logic. [*2 marks*]

   (b) Show whether or not the argument is a tautology using a truth table. [*5 marks*]

2. Suppose that you are given three boxes. One contains gold and the other two are empty. Upon each box is written a message:

   - On box 1 it says "The gold is not here."
   - On box 2 it says "The gold is not here."
   - On box 3 it says "The gold is in box 2."

   Only one of the three messages above is true; the other two are false. Write each of the messages as an expression in propositional logic. Then show, using a truth table, how you would prove which box contains the gold. [*8 marks*]

3. You are given the following proof rules:

Rule name	Sequent	Supporting proofs
immediate	$\mathcal{F} \vdash A$	$A \in \mathcal{F}$
and_intro	$\mathcal{F} \vdash A \text{ and } B$	$\mathcal{F} \vdash A,\ \mathcal{F} \vdash B$
or_intro_left	$\mathcal{F} \vdash A \text{ or } B$	$\mathcal{F} \vdash A$
or_intro_right	$\mathcal{F} \vdash A \text{ or } B$	$\mathcal{F} \vdash B$
or_elim	$\mathcal{F} \vdash C$	$\mathcal{F} \vdash (A \text{ or } B),\ [A\|\mathcal{F}] \vdash C,\ [B\|\mathcal{F}] \vdash C$
imp_elim	$\mathcal{F} \vdash B$	$A \to B \in \mathcal{F},\ \mathcal{F} \vdash A$
imp_intro	$\mathcal{F} \vdash A \to B$	$[A\|\mathcal{F}] \vdash B$

   Using the proof rules above, prove the following sequent:

$$[p \to (a \text{ or } b),\ a \to x,\ b \to x] \ \vdash\ p \to x$$

   Show in your answer precisely how the proof rules are applied. [*10 marks*]

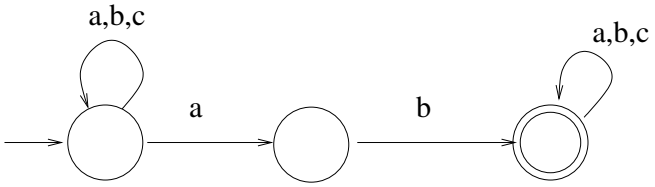
4. In this question we are concerned with modelling the behaviour of a gas cooker with two burners, a small burner and a large burner. The cooker has a gas tap for each burner, to turn on and turn off the gas for that burner. The cooker also has an ignite button for the cooker: when the ignite button is pressed, it lights

any burner whose gas is turned on (if both taps are on it lights both burners; if both taps are off it does nothing). When the tap for a burner is used while the tap is on or the burner is lit, it switches off that burner. We say that the cooker is in a *safe* state when both taps are switched off.

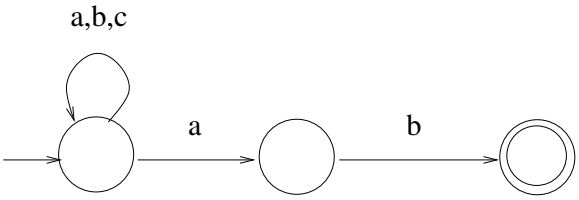
(a) Draw a Finite State Machine to model the behaviour of this cooker. The FSM should have 9 states in total, because there are 3 possible behavioural states for each burner (off, gas on, and lit), and these can hold independently for each of the two burners. The only accepting state is the safe state. The input alphabet for the FSM is {TAP-S, TAP-L, IGN}. The input TAP-S indicates that the tap for the small burner has been used; the input TAP-L indicates that the tap for the large burner has been used; IGN indicates that the ignite button has been pressed. [*8 marks*]

(b) Give a string of inputs which visits every state of the FSM. [*2 marks*]

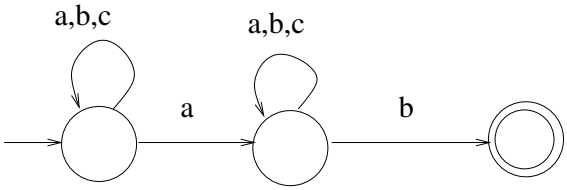
5. Consider the following set of Finite State Machines over the alphabet {`a`, `b`, `c`}. Let L_{I} represent the language accepted by the FSM (I), L_{II} represent the language accepted by the FSM (II), and L_{III} represent the language accepted by the FSM (III).



(I)



(II)



(III)

(a) Give an example of a string which is in L_{I} but not in L_{II}. Give an example of a string which is in L_{III} but not in L_{I}. [*2 marks*]

(b) Use the rules of Kleene's theorem to draw a Finite State Machine to accept $L_{\mathrm{I}} \cup L_{\mathrm{II}}$. [*2 marks*]

(c) Draw an FSM with three states (not using Kleene's theorem) to accept the language $L_{\mathrm{I}} \cup L_{\mathrm{II}}$. [*2 marks*]

(d) Draw an FSM with three states (not using Kleene's theorem) to accept the language $L_{\mathrm{II}} \cup L_{\mathrm{III}}$. [*2 marks*]

6. For this question we are concerned with Finite State Machines to recognize languages over the decimal numbers, that is, languages over the alphabet $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$.

(a) Is there a Finite State Machine to recognize the language of all decimal numbers ending with the digit 4? Justify your answer, either by describing how to construct an FSM to accept the language **or** by showing that no such machine can exist. [*3 marks*]

(b) Is there a Finite State Machine to recognize the language of all decimal numbers which are divisible by 4? Justify your answer, either by describing how to construct an FSM to accept the language **or** by showing that no such machine can exist. [*4 marks*]

PART B: FUNCTIONAL PROGRAMMING

In the answer to any part of any question, you may use any function specified in an earlier part of that question. You may do this whether or not you actually provided a definition for the earlier part; nor will you be penalized in a later part if your answer to an earlier part is incorrect.

Unless otherwise stated, you may use any operators and functions from the standard prelude and from the libraries Char, List, and Maybe. You need not write import declarations.

As an aid to memory, a summary of relevant functions follows. You will not need all of the functions listed.

```
(<), (<=), (>=), (>) :: (Ord a) => a -> a -> Bool
(1 < 2) == True      ('a' < 'b') == True     (False < True) == True

(==), (/=) :: (Eq a) => a -> a -> Bool
(2 == 2) == True     (2 /= 2) == False

min, max :: (Ord a) => a -> a -> a
min 1 2 == 1     min 'a' 'z' == 'a'
max 1 2 == 2     max 'a' 'z' == 'z'

(+), (-), (*), (/) :: (Num a) => a -> a -> a
3+4 == 7     3-4 == -1    3*4 == 12

fromIntegral :: (Integral a, Num b) => a -> b
fromIntegral 3 / fromIntegral 4 == 0.75

div, mod :: (Integral a) => a -> a -> a
14 `div` 3 == 4     14 `mod` 3 == 2

(&&), (||) :: Bool -> Bool -> Bool
(True && False) == False     (True || False) == True

(:) :: a -> [a] -> [a]
1 : [2,3,4] == [1,2,3,4]
'h' : "ello" == "hello"

(++) :: [a] -> [a] -> [a]
[1,2,3] ++ [4,5] == [1,2,3,4,5]
"Hello" ++ " " ++ "world" == "Hello world"

(!!) :: Int -> [a] -> a
[3,1,4,1,2] !! 2 == 4     "world" !! 2 == 'r'

otherwise :: Bool
otherwise == True

sum :: (Num a) => [a] -> a
sum [1,2,3,4] == 10

product :: (Num a) => [a] -> a
product [1,2,3,4] == 24

concat :: [[a]] -> [a]
concat ["con","cat","en","ate"] == "concatenate"
```

```
minimum, maximum :: (Ord a) => [a] -> a
minimum [1,2,3] == 1       maximum [1,2,3] == 3
minimum "world" == 'd'     maximum "world" == 'w'

show :: (Show a) => a -> String
show 344 == "344"   show "hello" == "\"hello\""    show 'a' == "\'a\'"
```

```
elem :: (Eq a) => a -> [a] -> Bool
elem "rat" ["fat","rat","sat","flat"] == True

lookup :: (Eq a) => a -> [(a,b)] -> Maybe b
lookup 'c' [('a',1),('b',2),('c',3)] == Just 3
lookup 'd' [('a',1),('b',2),('c',3)] == Nothing

fromMaybe :: a -> Maybe a -> a
fromMaybe 0 (lookup 'c' [('a',1),('b',2),('c',3)])  == 3
fromMaybe 0 (lookup 'd' [('a',1),('b',2),('c',3)])  == 0

isAlpha, isUpper, isLower, isDigit :: Char -> Bool
isAlpha 'a' == True      isAlpha 'A' == True      isAlpha '0' == False
isUpper 'a' == False     isUpper 'A' == True      isUpper '0' == False
isLower 'a' == True      isLower 'A' == False     isLower '0' == False
isDigit 'a' == False     isDigit 'A' == False     isDigit '0' == True

toUpper, toLower :: Char -> Char
toUpper 'a' == 'A'       toUpper 'A' == 'A'       toUpper '0' == '0'
toLower 'a' == 'a'       toLower 'A' == 'a'       toLower '0' == '0'

words, lines :: String -> [String]
words "the quick brown fox" == ["the", "quick", "brown", "fox"]
lines "the quick\nbrown fox\n" == ["the quick", "brown fox"]

unwords, unlines :: [String] -> String
unwords ["The", "quick", "brown", "fox"] == "The quick brown fox"
unlines ["The quick", "brown fox"] == "The quick\nbrown fox\n"

head :: [a] -> a
tail :: [a] -> [a]
head [1,2,3,4] == 1        tail [1,2,3,4] == [2,3,4]
head "goodbye" == 'g'      tail "goodbye" == "oodbye"

zip :: [a] -> [b] -> [(a,b)]
zip [1,2,3] [1,4,9] == [(1,1),(2,4),(3,9)]
zip [1..] "abcd" == [(1,'a'),(2,'b'),(3,'c'),(4,'d')]

take, drop :: Int -> a -> [a]
take 4 "goodbye" == "good"        drop 4 "goodbye" == "bye"
take 5 [1,2,3,4] == [1,2,3,4]     drop 5 [1,2,3,4] == []

replicate :: Int -> a -> [a]
replicate 5 '*' == "*****"
```

```
map :: (a -> b) -> [a] -> [b]
map (*2) [1,2,3] == [2,4,6]
map tail ["the","quick","brown","fox"] == ["he","uick","rown","ox"]

zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
zipWith (+) [1,2,3] [1,4,9] == [2,6,12]

filter :: (a -> Bool) -> [a] -> [a]
filter isUpper "The Quick Brown Fox" == "TQBF"
filter (<5) [1,3,5,6,4,2] == [1,3,4,2]

takeWhile, dropWhile :: (a -> Bool) -> [a] -> [a]
takeWhile (<5) [1,3,5,6,4,2] == [1,3]
dropWhile (<5) [1,3,5,6,4,2] == [5,6,4,2]

iterate :: (a -> a) -> a -> [a]
iterate (*2) 1 == [1,2,4,8,16,...]

foldr :: (a -> b -> b) -> b -> [a] -> b
foldr op a [x,y,z] == x `op` (y `op` (z `op` a))
foldr (&&) True [False,True] == False
foldr (+) 0 [1,2,3] == 6
foldr (++) [] ["con","cat","en","ate"] == "concatenate"

foldl :: (b -> a -> b) -> b -> [a] -> b
foldl op a [x,y,z] == ((a `op` x) `op` y) `op` z
foldl (\ xs x -> x:xs) [] [1,2,3]  == [3,2,1]

foldr1, foldl1 :: (a -> a -> a) -> a -> [a] -> a
foldr1 op [x,y,z] == x `op` (y `op` z)
foldl1 op [x,y,z] == (x `op` y) `op` z
foldr1 max [1,2,3] == 3

sort :: (Ord a) => [a] -> [a]
sort [3,1,4,1,2,2] == [1,1,2,2,3,4]
sort ["con","cat","en","ate"] == ["ate","cat","con","en"]

nub :: (Eq a) => [a] -> [a]
nub [3,1,4,1,2,2] == [3,1,4,2]

group :: (Eq a) => [a] -> [[a]]
group [1,1,1,2,2,3,2,2] == [[1,1,1],[2,2],[3],[2,2]]
```

7. An angle is given in degrees and minutes.

```
type Deg = Int
type Min = Int
type Angle = (Deg, Min)
```

For example, the angle 3°11′ (3 degrees, 11 minutes) is represented as `(3,11)`.

A longitude consists of an angle and a letter, for instance `((3, 11), 'W')`

```
type Longitude = (Angle, Char)
```

The angle should be between 0°0′ and 180°0′ and the letter should be W (for west) or E (for east).

(a) Write a function `value :: Angle -> Float` that converts an angle to its decimal equivalent. For example, `value (51,30) == 51.5`. There are 60 minutes in a degree (hence 30 minutes are half of a degree). [*5 marks*]

(b) Write a function `ok :: Longitude -> Bool` that returns true if the degrees are between 0 and 180 (inclusive), and the minutes are between 0 and 59 (inclusive), and the value of the angle is at most 180, and the letter is W or E. For example, `ok ((3,11),'W')` and `ok ((0,0),'E')` are true, but `ok ((-3,11),'W')` and `ok ((2,71),'W')` and `ok ((180,1),'W')` are false. [*5 marks*]

(c) Write a function `location :: String -> Longitude` that returns the longitude of a city, according to the following table: Edinburgh 3°11′W, Glasgow 4°17′W, London 0°0′E, Paris 2°29′E. This function should signal an error if passed a city name other than these four. [*5 marks*]

(d) Write a function `showLongitude :: Longitude -> String` that converts a longitude to a string. The degree symbol is written as a period. For example, `showLongitude ((3,11),'W')` returns `"3.11'W"`. This function should signal an error if passed an invalid longitude. [*5 marks*]

8. (a) Write a function `swap :: Char -> Char` that converts every lower case letter to upper case and vice versa. If given a character that is not a letter, it is returned unchanged. Thus, applying this function to 'G', 'o', 'E', 'd', 'i', '!' respectively returns 'g', 'O', 'e', 'D', 'I' '!'. [*6 marks*]

   (b) Write a function `swaps :: String -> String` that applies the function `swap` to each letter in a string to yield a new string; characters in the initial string that are not letters are omitted. Thus, applying this function to "GoEdi!" returns "gOeDI". Your definition should use a *list comprehension*, not recursion. [*6 marks*]

   (c) Write a second definition of `swaps`, this time using *recursion*, and not a list comprehension. [*6 marks*]

9. (a) Write a function `same :: (Eq a) => [a] -> Bool` that given a list of values, returns true if two adjacent values in the list are equal, and false otherwise. Thus `same [1,2,2,3]` returns true, while `same [1,2,3]` and `same [1,2,1,2]` return false. If the list contains less than two values, the result should be false. Your definition should use a *list comprehension* and library functions, not recursion. [*6 marks*]

   (b) Write a second definition of `same`, this time using *recursion*, and no list comprehensions or library functions (other than equality and logical operators). [*6 marks*]

ANSWERS

1. If t represents "Temperature remains constant"; a represents "Air pressure remains constant" and r represents "It rains" then an appropriate expression in propositional logic is:

$$((t \wedge a \rightarrow \neg r) \wedge t) \rightarrow (r \rightarrow \neg a)$$

An appropriate truth table is:

t	a	r	$\neg a$	$\neg r$	$t \wedge a$	$t \wedge a \rightarrow \neg r$	$(t \wedge a \rightarrow \neg r) \wedge t$	$r \rightarrow \neg a$	$((t \wedge a \rightarrow \neg r) \wedge t) \rightarrow (r \rightarrow \neg a)$
t	t	t	f	f	t	f	f	f	t
t	t	f	f	t	t	t	t	t	t
t	f	t	t	f	f	t	t	t	t
t	f	f	t	t	f	t	t	t	t
f	t	t	f	f	f	t	f	f	t
f	t	f	f	t	f	t	f	t	t
f	f	t	t	f	f	t	f	t	t
f	f	f	t	t	f	t	f	t	t

so the expression is a tautology.

Marking guide: 2 marks for the correct propositional expression. 1 mark for knowing what a tautology is. 4 marks for working out the truth table.

2. If p represents "The gold is in box 1" and q represents "The gold is in box 2" then the three messages (in order) are: $\neg p$, $\neg q$ and q.

An appropriate truth table is:

p	q	$\neg p$	$\neg q$	$p \rightarrow \neg q$	$q \rightarrow \neg p$	$(p \rightarrow \neg q) \wedge (q \rightarrow \neg p)$
t	t	f	f	f	f	f
t	f	f	t	t	t	t
f	t	t	f	t	t	t
f	f	t	t	t	t	t

The only row in which the uniqueness condition is true and in which only one box has the gold is the second row (counting from the top) so the gold is in box 1.

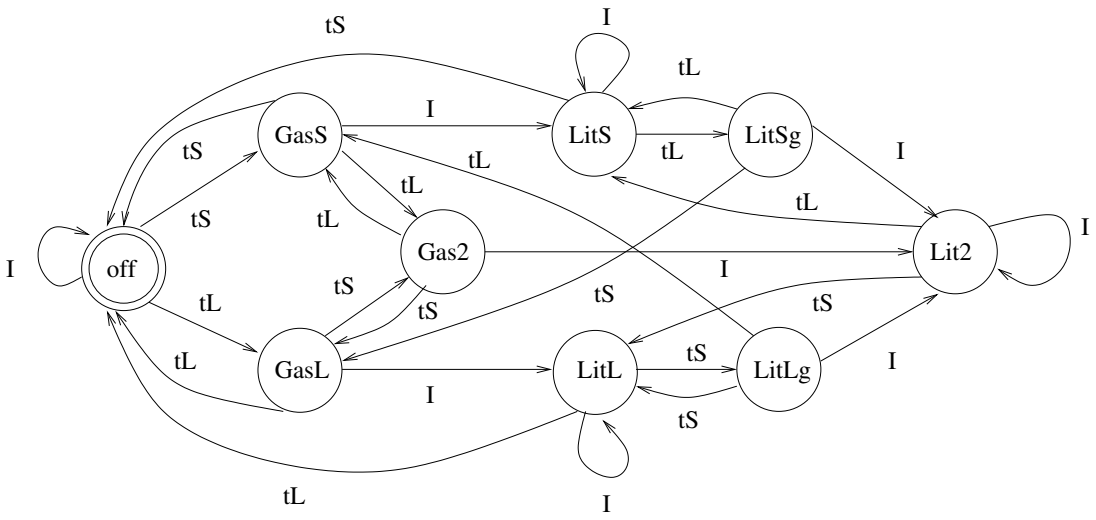
Marking guide: 2 marks for the formalisation of the English statements. 2 marks for an appropriate uniqueness condition. 4 marks for the truth table.

3. Let S be the initial set of axioms $[p \rightarrow (a \; or \; b),\; a \rightarrow x,\; b \rightarrow x]$. Apply the proof rules in the following order:

$S \vdash p \to x$
Applying *imp_intro*
$[p|S] \vdash x$
Applying *or_elim*
$[p|S] \vdash (a \text{ or } b)$
Applying *imp_elim*
$p \to (a \text{ or } b) \in [p|S],\ [p|S] \vdash p$
Applying *immediate*
$p \in [p|S]$
$[a|[p|S]] \vdash x$
Applying *imp_elim*
$a \to x \in [a|[p|S]],\ [a|[p|S]] \vdash a$
Applying *immediate*
$a \in [a|[p|S]]$
$[b|[p|S]] \vdash x$
Applying *imp_elim*
$b \to x \in [b|[p|S]],\ [b|[p|S]] \vdash b$
Applying *immediate*
$b \in [b|[p|S]]$

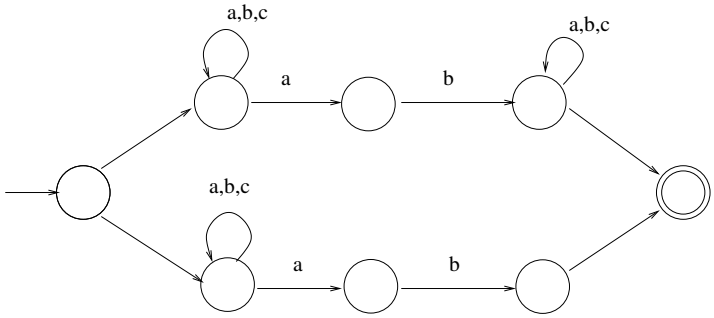
Marking guide: 4 marks for showing a basic idea of how to build a sequent proof. 6 marks for getting the details right.

4. (a) Here is the FSM (I have used `I` to represent IGN, `tS` as shorthand for TAP-S, `tL` as shorthand for TAP-L.

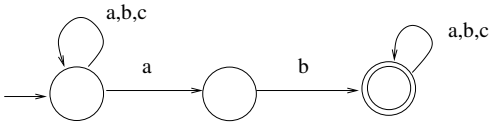


(b) A string of inputs which visits every state of the FSM is the following: `tS`, `I`, `tL`, `tS`, `tS`, `tS`, `I`, `tS`, `I`.

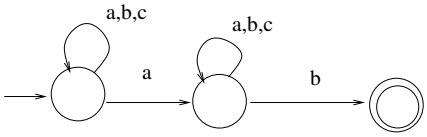
5. (a) A string accepted by L_{I} but not by L_{II} is *abaa*. A string accepted by L_{III} but not by L_{I} is *acb*.



(b)



(c)



(d)

6. For this question we are concerned with Finite State Machines to recognize languages over the decimal numbers, that is, languages over the alphabet $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$.

   (a) Yes, there is a Finite State Machine to recognize the language of all decimal numbers ending with the digit 4.

   It is easy to construct such a machine (with 3 states) - just read the string in order of least-significant bit. If the first digit is 4, then take a transition to an accepting sink state which can read any subsequent digits. If the first digit is not 4, then take a transition to a non-accepting sink state.

   Alternatively, a student might read the string in order of most-significant bit. This solution is fine too (but a bit more difficult for them to do).

   (b) Yes, there is a Finite State Machine to recognize the language of all decimal numbers which are divisible by 4.

   The reason is that any multiple of 100 is divisible by 4. Therefore if we want to work out if a decimal number is divisible by 4, we only need to look at the two least-significant digits. If the least-sig digit is $0, 4$ or 8, then the 2nd-least-sig digit must be $0, 2, 4, 6$ or 8. If the least-sig digit is $2, 6$, then the 2nd digit must be $1, 3, 5, 7, 9$. All other combinations for the two least-sig digits mean the decimal number is not divisible by 4. We can use these facts to draw a FSM with 4 states to recognise all numbers divisible by 4 (we will have two sink-states (one accept, one non-accept) to read the digits after the first two.

Part B FUNCTIONAL PROGRAMMING ANSWERS

7. (a)
```
value :: Angle -> Float
value (d,m) = fromIntegral(d) + fromIntegral(m)/60
```
[*5 marks*]

(b)
```
ok :: Longitude -> Bool
ok ((d,m),ew) =   0 <= d && 0 <= m && m < 60 && value (d,m) <= 180
                && (ew=='E' || ew=='W')
```
[*5 marks*]

(c)
```
location :: String -> Longitude
location "Edinburgh" = ((3,11),'W')
location "Glasgow"   = ((4,17),'W')
location "London"    = ((0,0),'E')
location "Paris"     = ((2,29),'E')
```
[*5 marks*]

(d)
```
showLongitude :: Longitude -> String
showLongitude ((d,m),ew)
  | ok ((d,m),ew)   =  show d ++ ". " ++ show m ++ "'" ++ [ew]
```
[*5 marks*]

8. (a)
```
swap :: Char -> Char
swap c
  | isAlpha c && isLower c  =  toUpper c
  | isAlpha c && isUpper c  =  toLower c
  | otherwise  =  c
```
[*6 marks*]

(b)
```
swaps :: String -> String
swaps s  =  [ swap c | c <- s, isAlpha c ]
```
[*6 marks*]

(c)
```
swaps :: String -> String
swaps []  =  []
swaps (c:s) | isAlpha c  =  swap c : swaps s
            | otherwise  =  swaps s
```
[*6 marks*]

9. (a)
```
same :: (Eq a) => [a] -> Bool
same xs  =  or [ x==y | (x,y) <- zip xs (tail xs) ]
```
[*6 marks*]

(b)
```
same :: (Eq a) => [a] -> Bool
same []  =  False
same [x]  =  False
same (x:y:ys)  =  x==y || same (y:ys)
```
[*6 marks*]