UNIVERSITY OF EDINBURGH
COLLEGE OF SCIENCE AND ENGINEERING
SCHOOL OF INFORMATICS

Date: 06 December 2004 **Time: 1430-1630**

INFORMATICS 1A

Examiners R. Irving (External)

M. Jerrum (Chair)

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 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. Imagine that in a murder case you have three people as suspects: Phil, Dave and Mary. They make the following statements:

   Phil says : "Dave is guilty and Mary is innocent."

   Dave says : "If Phil is guilty then so is Mary."

   Mary says : "I am innocent but at least one of the others is guilty."

   Let p, d and m mean "Phil is innocent", "Dave is innocent" and "Mary is innocent", respectively.

   (a) Rewrite the statements made by Phil, Dave and Mary, each as an expression in propositional logic.

   [3%]

   (b) If all the suspects are innocent, which people lied? Explain your answer using a truth table.

   [3%]

   (c) If the statements made by Phil, Dave and Mary all are true, who is/are innocent? Explain your answer using a truth table.

   [6%]

2. You are given the following proof rules:

Rule name	Sequent	Supporting proofs
immediate	$\mathcal{F} \vdash A$	$A \in \mathcal{F}$
and_intro	$\mathcal{F} \vdash A \; and \; B$	$\mathcal{F} \vdash A, \; \mathcal{F} \vdash B$
or_intro_left	$\mathcal{F} \vdash A \; or \; B$	$\mathcal{F} \vdash A$
or_intro_right	$\mathcal{F} \vdash A \; or \; B$	$\mathcal{F} \vdash B$
or_elim	$\mathcal{F} \vdash C$	$A \; or \; B \in \mathcal{F}, \; [A \vert \mathcal{F}] \vdash C, \; [B \vert \mathcal{F}] \vdash C$
imp_elim	$\mathcal{F} \vdash B$	$A \rightarrow B \in \mathcal{F}, \; \mathcal{F} \vdash A$
imp_intro	$\mathcal{F} \vdash A \rightarrow B$	$[A \vert \mathcal{F}] \vdash B$

where $\mathcal{F} \vdash A$ means that expression A can be proved from set of axioms $\mathcal{F}$; $A \in \mathcal{F}$ means that A is an element of set $\mathcal{F}$; $[A|\mathcal{F}]$ is the set constructed by adding A to set $\mathcal{F}$; $A \rightarrow B$ means that A implies B; $A \; and \; B$ means that A and B both are true; and $A \; or \; B$ means that at least one of A or B is true.

(a) Using the proof rules above, prove the following sequent:

$$[a, \; (a \; and \; b) \rightarrow c] \; \vdash \; b \rightarrow c$$

Show in your answer precisely how the proof rules are applied.

[7%]

(b) Convert the expression $(a \; and \; (a \rightarrow b)) \rightarrow b$ into an equivalent expression containing only '*or*' and '*not*' logical operators (and no '$\rightarrow$' operator) by applying the following rules of equivalence between expressions:

$(P \rightarrow Q)$	is equivalent to	$not(P) \; or \; Q$
$not(P \; and \; Q)$	is equivalent to	$not(P) \; or \; not(Q)$

Show precisely how the equivalence rules are applied.

[6%]

3. Consider the problem of modelling a Drinks Machine which sells cans of drinks. The machine sells two types of drink - Cola and Irn-Bru. The machine will only accept two types of coins as payment, the 50p coin and the £1 coin. Each of the drinks costs 50 pence.

   The machine operates in two phases. The first phase is the *Coin Entry* phase. In the initial state of this phase, there are no coins in the machine. The customer inserts a coin into the machine. If the coin is any coin other than 50p or £1, then the machine will return the coin that was input (performing `oth-coin-return`), and move back into its initial state. Otherwise if the coin is either a 50p coin or a £1 coin, then the machine moves into its *Can Selling* phase. This is because the machine never accepts more than one coin at a time - it is not physically possible to insert a second coin into the machine if there is already a coin in there.

   In the *Can Selling* phase, the customer presses any of the following buttons: the "Coin Return" button, the "Cola" button, or the "Irn-Bru" button. If the `CR` input is given (for Coin Return), then the machine returns a coin equal to the credit remaining (which will be either 50p or £1); If the `Cola` input is given, the machine vends a Cola; If the `Irn` input was given, the machine vends an Irn-Bru. After each of these three cases, the machine either returns to its initial state of the Coin Entry phase (if there is no credit remaining) or gives the customer the chance to buy another can (if there is 50p credit remaining).

   We assume that the machine never runs out of cans, or 50p coins, or £1 coins.

   The inputs for the machine are: 50p, £1, `oth-coin`, `CR`, `Cola`, `Irn`.

   The outputs for the machine are: `oth-coin-return`, `return-`50p, `return-`£1, `vend-Cola` and `vend-Irn`.

   (a) Draw a transducer-style Finite State Machine with 3 states to Model the Drinks Machine described above.

   [6%]

   (b) Give a sequence of inputs which ensures that you will visit all the states of the transducer you have constructed. You do not necessarily have to give all inputs (or see all outputs).

   [2%]

4. In this question we consider Finite State Machines (FSMs) for recognising natural numbers (the counting numbers, that is, $0, 1, 2, 3, 4, \ldots$) with certain properties. We will work with two different representations of natural numbers:

   In *unary* notation, the natural number n is represented by a string of n "1"s. The natural number 0 is a special case which we represent by the empty string ϵ.

   In *binary* notation, a number is represented by a string of 0s and 1s. If $b = b_k \ldots b_1 b_0$ is a binary string (of length $k+1$) then the number represented by that binary string is the number $n = b_0 + 2b_1 + \ldots + 2^k b_k$, also written as $\sum_{j=0}^{k} b_j 2^j$.

   (a) First consider the question of designing FSMs over the binary alphabet $\Sigma = \{0, 1\}^*$. Assume that a binary string $b \in \{0, 1\}^*$ is input to an FSM with its *least significant digit first* (b_0, then b_1, then b_2, $\ldots$).

   (i) Draw a deterministic FSM to recognise the set of binary strings which represent *even* natural numbers.

   [2%]

   (ii) Is it possible to design an FSM to recognise the set of binary strings which represent numbers which are *powers of* 2 (the numbers $1, 2, 4, 8, \ldots$ and so on)? Justify your answer, *either* by drawing the machine *or* by explaining why you think no such machine exists.

   [2%]

   (b) Now consider the question of designing FSMs over the unary alphabet $\Sigma = \{1\}^*$.

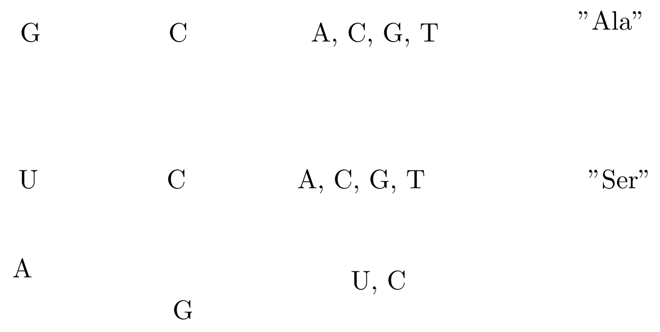
   (i) Draw a deterministic FSM to recognise the set of unary strings which represent *even* natural numbers.

   [2%]

   (ii) Is it possible to design an FSM to recognise the set of unary strings which represent numbers which are *powers of* 2 (the numbers $1, 2, 4, 8, \ldots$ and so on)? Justify your answer, *either* by drawing the machine *or* by explaining why you think no such machine exists.

   [3%]

5. Consider the following two Finite State Machines over the alphabet $\Sigma = \{A, C, T, G\}$, which accept the languages $L(\text{Ala})$ and $L(\text{Ser})$ respectively:



(a) Give a regular expression (r.ex.) for the language $L(\text{Ala})$ accepted by the first machine "Ala".
Give an r.ex. for the language $L(\text{Ser})$, accepted by the second machine "Ser".
[2%]

(b) Use the *rules* of *Kleene's Theorem* (using ϵ-transitions) to come up with a non-deterministic Finite State Machine to accept the language $L(\text{Ala}) \cup L(\text{Ser})$.
[3%]

(c) Now, without using the rules of Kleene's Theorem, draw a simple FSM with 6 states to accept the language $L(\text{Ala}) \cup L(\text{Ser})$.
[3%]

Part B FUNCTIONAL PROGRAMMING

Unless otherwise stated, you may use either list comprehension or recursion in your answers (or both).

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"

(!!) :: [a] -> Int -> 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
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]]
```

6. A date is a triple, representing day, month, and year.

```
type Day = Int
type Month = String
type Year = Int
type Date = (Day, Month, Year)
```

A month is represented by the first three letters of its name.

(a) Write a function `leapYear :: Year -> Bool` that returns true if the year is a leap year, and false otherwise. The following rules decide which years are leap years:

- Every year divisible by 4 is a leap year, *except*
- Every year divisible by 100 is *not* a leap year, *except*
- Every year divisible by 400 is still a leap year.

For example, 2004 is a leap year (divisible by 4), 1900 is not a leap year (divisible by 100), and 2000 is a leap year (divisible by 400).

[5%]

(b) Write a function `daysInMonth :: Month -> Year -> Day` that returns the number of days in the given month, according to the following table: Jan 31, Feb 28 (29 if it is a leap year), Mar 31, Apr 30, May 31, Jun 30, Jul 31, Aug 31, Sep 30, Oct 31, Nov 30, Dec 31. Your function should return 0 if passed a month name other than the above.

[5%]

(c) Write a function `validDate :: Date -> Bool` that returns true if given a date in which the day is between one and the number of days in the month, and false otherwise.

[5%]

(d) Write a function `showDate :: Date -> String` that converts a date to a string. For instance, the first date of this year should return the string `"1 Jan 2004"`. This function should signal an error if it is passed an invalid date.

[5%]

7. The *vowels* are the letters AEIOU and the remaining letters are *consonants*.

   (a) Write a function `vowel :: Char -> Char` that converts every vowel to the letter ‘a’ and every consonant to the letter ‘b’, while preserving case. 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 ‘B’, ‘a’, ‘A’, ‘b’, ‘a’ ‘!’.

   [6%]

   (b) Write a function `vowels :: String -> String` that applies the function vowel 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 “BaAba”. Your definition should use a *list comprehension*, not recursion.

   [6%]

   (c) Write a second definition of `vowels`, this time using *recursion*, and not a list comprehension.

   [6%]

8. A point can be represented as a pair of numbers.

```
type Point = (Float, Float)
```

You may use the function `dist` to compute the distance between two points.

```
dist :: Point -> Point -> Float
dist (x,y) (u,v) = sqrt (sqr (x-u) + sqr (y-v))
                   where  sqr x = x*x
```

(a) Write a function `total :: [Point] -> Float` that given a list of points, returns the distance traveled if one went from the first point to the second point, from the second point to the third, and so on. Thus, the total for the list [(1,2), (4,6), (10,-2), (6,1)] is 20, since the distance from (1,2) to (4,6) is 5, the distance from (4,6) to (10,-2) is 10, and the distance from (10,-2) to (6,1) is 5. If the list contains less than two points, the total should be zero. Your definition should use a *list comprehension* and library functions, not recursion.

[6%]

(b) Write a second definition of `total`, this time using *recursion*, and no list comprehension or library functions (other than `(+)`).

[6%]

ANSWERS
Part A COMPUTATION AND LOGIC ANSWERS

1. (a) The appropriate formulations are:

   Phil says : $not(d)\ and\ m$

   Dave says : $not(p)\ \rightarrow\ not(m)$

   Mary says : $m\ and\ (not(d)\ or\ not(p))$

   [3%]

   (b) Draw up a truth table with one row (where p, d and m each is true) for each of the expressions above. This is the first row of the truth table below. The resulting truth values give the conclusion that Phil and Mary lied but what Dave says is true.

   [3%]

   (c) Draw up the same truth table as for the previous question but (unless the student is very insightful) all eight of the rows will need to be done, so as to consider all possible truth values for p, d and m. The full truth table is below. There is only one row (the third one) in which all the expressions corresponding to people's statements are true. This occurs when Phil and Mary are innocent and Dave is guilty.

p	d	m	$not(d)$	$not(d)\ and\ m$	$not(p)$	$not(m)$	$not(p)\ \rightarrow\ not(m)$	$not(d)\ or\ not(p)$	$m\ and\ (not(d)\ or\ not(p))$
t	t	t	f	f	f	f	t	f	f
t	t	f	f	f	f	t	t	f	f
t	f	t	t	t	f	f	t	t	t
t	f	f	t	f	f	t	t	t	f
f	t	t	f	f	t	f	f	t	t
f	t	f	f	f	t	t	t	t	f
f	f	t	t	t	t	f	f	t	t
f	f	f	t	f	t	t	t	t	f

[6%]

2. (a) Apply the proof rules in the following order:

$[a,\ (a\ \mathit{and}\ b) \to c]\ \vdash\ b \to c$
Applying *imp_intro*
$[b,\ a,\ (a\ \mathit{and}\ b) \to c]\ \vdash\ c$
Applying *imp_elim* using $(a\ \mathit{and}\ b) \to c$
$[b,\ a,\ (a\ \mathit{and}\ b) \to c]\ \vdash\ a\ \mathit{and}\ b$
Applying *and_intro* first sub-proof
$[b,\ a,\ (a\ \mathit{and}\ b) \to c]\ \vdash\ a$
Applying *immediate* using a
Done
Applying *and_intro* second sub-proof
$[b,\ a,\ (a\ \mathit{and}\ b) \to c]\ \vdash\ b$
Applying *immediate* using b
Done

[7%]

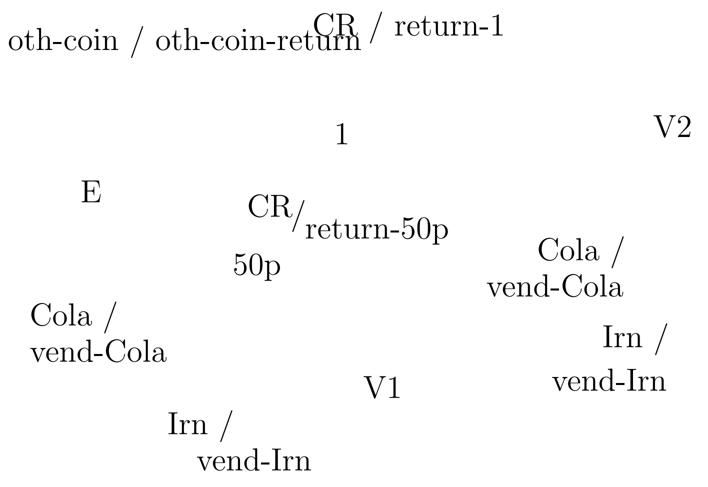
(b) One answer is to apply the equivalences in the following order:

$(a\ \mathit{and}\ (a \to b)) \to b$	is equivalent to	$\mathit{not}(a\ \mathit{and}\ (a \to b))\ \mathit{or}\ b$
	giving the formula	
	$\mathit{not}(a\ \mathit{and}\ (a \to b))\ \mathit{or}\ b$	
	then	
$\mathit{not}(a\ \mathit{and}\ (a \to b))$	is equivalent to	$\mathit{not}(a)\ \mathit{or}\ \mathit{not}(a \to b)$
	giving the formula	
	$\mathit{not}(a)\ \mathit{or}\ \mathit{not}(a \to b)\ \mathit{or}\ b$	
	then	
$a \to b$	is equivalent to	$\mathit{not}(a)\ \mathit{or}\ b$
	giving the formula	
	$\mathit{not}(a)\ \mathit{or}\ \mathit{not}(\mathit{not}(a)\ \mathit{or}\ b)\ \mathit{or}\ b$	

Other orders of application are permitted as long as they are applied correctly and the result is correct.

[6%]

3. (a) The student should draw a transducer-style Finite State Machine with 3 states similar to the one below.



[6%]

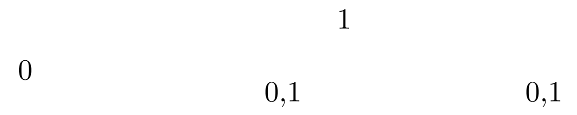
(b) For the transducer above, one sequence of inputs which ensures that you will visit all the states is: £1, `Cola`, `CR`.

Depending on the particular machine the student has constructed, the list may be different.

[2%]

4. (a) To answer this question the student should draw FSMs over the binary alphabet.

(i) First machine is:



[2%]

(ii) The answer to this question is also a machine (students had to decide if a machine existed). Here it is:

[2%]

(b) This questions asks about machines over a unary alphabet.

(i) Here is the FSM that recognises even numbers.

[2%]

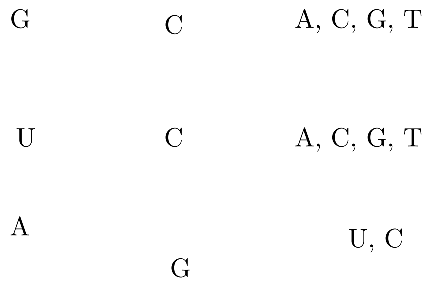
(ii) No, There is no FSM to recognise powers of 2 written in the unary alphabet. The reason is as follows (but I will accept less formal answers talking about the fact that FSMs can't "count" to arbitrarily high numbers):
No. If there was one with k states we consider n with $k \leq 2^n < 2k$. Then the FSM would revisit one of its states q in accepting 2^n. Suppose the "loop" has length $r \geq 1 \leq k$. Then the machine must also accept $2^n - r$, $2^n + r$, and $2^n + 2r$. They can't all be powers of 2 - so the machine cannot really exist.

[3%]

5. (a) The regular expressions are as follows: $L(\text{Ala}) = GC(A+C+G+T)$. $L(\text{Ser}) = UC(A+C+G+T) + AG(U+C)$.

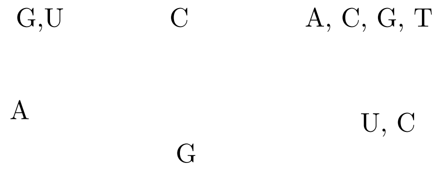
[2%]

(b) Here is the FSM obtained using Kleene' rules.



[3%]

(c) Here is the FSM with just 6 states.



[3%]

Part B FUNCTIONAL PROGRAMMING ANSWERS

6. (a)
```
leapYear :: Year -> Bool
leapYear y  =     ((y ‘mod‘ 4 == 0) && (y ‘mod‘ 100 /= 0))
               || (y ‘mod‘ 400 == 0)
```
[5%]

(b)
```
daysInMonth :: Month -> Year -> Day
daysInMonth m y  =  fromMaybe 0 (lookup m
                     [("Jan", 31),
                      ("Feb", if leapYear y then 29 else 28),
                      ("Mar", 31),
                      ("Apr", 30),
                      ("May", 31),
                      ("Jun", 30),
                      ("Jul", 31),
                      ("Aug", 31),
                      ("Sep", 30),
                      ("Oct", 31),
                      ("Nov", 30),
                      ("Dec", 31)])
```
[5%]

(c)
```
validDate :: Date -> Bool
validDate (d,m,y)  =  1 <= d && d <= daysInMonth m y
```
[5%]

(d)
```
showDate :: Date -> String
showDate (d,m,y)
  | validDate (d,m,y)  =  show d ++ " " ++ m ++ " " ++ show y
```
[5%]

7. (a)
```
vowel :: Char -> Char
vowel c
  | isAlpha c  =  sameCase c (if isVowel c then ’a’ else ’b’)
  | otherwise  =  c
  where
  isVowel c  =  toLower c ‘elem‘ "aeiou"
  sameCase c d  =  if isLower c then toLower d else toUpper d
```
[6%]

(b)
```
vowels :: String -> String
vowels s  =  [ vowel c | c <- s, isAlpha c ]
```
[6%]

(c)
```
vowels :: String -> String
vowels []  =  []
vowels (c:s) | isAlpha c  =  vowel c : vowels s
             | otherwise  =  vowels s
```

[6%]

8. (a)
```
total :: [Point] -> Float
total ps  =  sum [ dist p q | (p,q) <- zip ps (tail ps) ]
```

[6%]

(b)
```
total :: [Point] -> Float
total []  =  0
total [p]  =  0
total (p:q:qs)  =  dist p q + total (q:qs)
```

[6%]