UNIVERSITY OF EDINBURGH

COLLEGE OF SCIENCE AND ENGINEERING

SCHOOL OF INFORMATICS

INFORMATICS 1A

Tuesday 5 December 2006

09:30 to 11:30

Convener: M O'Boyle
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 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 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.

As an aid to memory, arithmetic and comparison operators are listed in Figure 1 and some library functions are listed in Figure 2. You will not need all the functions listed.

```
(+), (*), (-), (/) :: Num a => a -> a -> a
(<), (<=), (>), (>=) :: Ord => a -> a -> Bool
(==), (/=) :: Eq a => a -> a -> Bool
```

Figure 1: Arithmetic and comparison

```
isAlpha, isUpper, isLower, isDigit :: Char -> Bool
toUpper, toLower :: Char -> Char

error :: String -> a

sum, product :: (Num a) => [a] -> a
sum [1.0,2.0,3.0] == 6.0
product [1,2,3,4] == 24

and, or :: [Bool] -> Bool
and [True,False,True] == False
or [True,False,True] == True

(:) :: a -> [a] -> [a]
'g' : "oodbye" == "goodbye"

(++) :: [a] -> [a] -> [a]
"good" ++ "bye" == "goodbye"

(!!) :: [a] -> Int -> a
[9,7,5] !! 1  =  7

length :: [a] -> Int
length [9,7,5]  =  3

head :: [a] -> a
head "goodbye" == 'g'

tail :: [a] -> [a]
tail "goodbye" == "oodbye"

take :: Int -> [a] -> [a]
take 4 "goodbye" == "good"

drop :: Int -> [a] -> [a]
drop 4 "goodbye" == "bye"

splitAt :: Int -> [a] -> ([a],[a])
splitAt 4 "goodbye" = ("good","bye")

reverse :: [a] -> [a]
reverse "goodbye" == "eybdoog"

elem :: (Eq a) => a -> [a] -> Bool
elem 'd' "goodbye" == True

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

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

zip :: [a] -> [b] -> [(a,b)]
zip [1,2,3,4] [1,4,9] == [(1,1),(2,4),(3,9)]

unzip :: [(a,b)] -> ([a], [b])
unzip [(1,1),(2,4),(3,9)] == ([1,2,3], [1,4,9])
```

Figure 2: Some library functions

1. (a) Write a function `f :: [Int] -> Int` that takes a list of integers, and returns the sum of the cubes of the positive numbers in that list. For example, `f [-1,1,3,-5,2]` returns `36`. Your definition should use *list comprehension*, but not recursion. You may also use arithmetic and comparison and library functions. [*6 marks*]

   (b) Write a second definition of `f`, this time using *recursion* and not a list comprehension. You may use arithmetic and comparison, but no other library functions. [*6 marks*]

   (c) Write a third definition of `f`, this time using the following higher-order library functions.

```
map    :: (a -> b) -> [a] -> [b]
filter :: (a -> Bool) -> [a] -> [a]
foldr  :: (a -> b -> b) -> b -> [a] -> b
```

   You may use arithmetic and comparison, but do not use list comprehension, recursion, or any other library functions. [*6 marks*]

2. A pattern consists of letters and underscores, where a letter matches the given letter, and an underscore matches any single letter. For example, `"_e__o"` matches `"hello"` and `"pesto"`, but does not match `"hallo"` or `"helloa"`.

   (a) Write a function `match :: Char -> Char -> Bool` that returns true if the first character is an underscore or is the same as the second. The function should indicate an error if the first character is not an underscore or a letter, or if the second character is not a letter. You may use arithmetic and comparison and any library functions. [*6 marks*]

   (b) Using `match`, write a function `matches :: String -> String -> Bool` that returns true if the first string matches the second. Your definition should use *list comprehension*, but not recursion. You may also use arithmetic and comparison and any library functions. [*6 marks*]

   (c) Again using `match`, write a second definition of `matches`, this time using *recursion* and not a list comprehension. You may use arithmetic and comparison, but no other library functions. [*6 marks*]

3. (a) Write a function `insert :: Char -> Int -> String -> String` that takes a character, an integer, and a string, and inserts that character after the given number of places in the string. For example,

```
insert 'x' 0 "abcd"  returns  "xabcd",  and
insert 'x' 2 "abcd"  returns  "abxcd",  and
insert 'x' 4 "abcd"  returns  "abcdx".
```

You may use arithmetic and comparison and any library functions, but not recursion. [*7 marks*]

(b) Write a second definition of `insert`, this time using *recursion*. You may use arithmetic and comparison, but no other library functions. [*7 marks*]

Part B COMPUTATION AND LOGIC

4. 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	$\mathcal{F} \vdash C$	$A \text{ or } B \in \mathcal{F},\ [A\|\mathcal{F}] \vdash C,\ [B\|\mathcal{F}] \vdash C$
imp_elim	$\mathcal{F} \vdash B$	$A \to B \in \mathcal{F},\ \mathcal{F} \vdash A$
imp_intro	$\mathcal{F} \vdash A \to B$	$[A\|\mathcal{F}] \vdash B$

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 $\mathcal{F}$; $A \to B$ means that A implies B; $A \text{ and } B$ means that A and B both are true; and $A \text{ or } B$ means that at least one of A or B is true.

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

$$[b \to c] \quad \vdash \quad (a \text{ or } b) \to (a \text{ or } c)$$

Show precisely how the proof rules are applied. [*7 marks*]

(b) Using only the proof rules above it is not possible to perform the following proof:

$$[(a \text{ and } b)] \quad \vdash \quad a$$

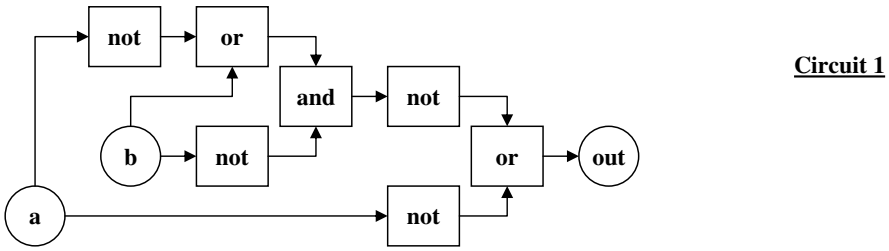
Briefly explain why the proof rules fail to support this proof. [*3 marks*]

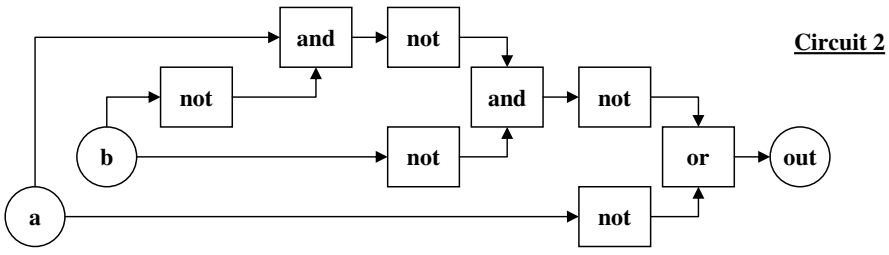
(c) Briefly explain, in general terms, the concepts of soundness and completeness of a proof system. [*2 marks*]

5. The diagram below gives two different logic circuits (Circuit 1 and Circuit 2) constructed from the following components:

   - An "and" connective (drawn in the diagram as a box labelled "and") sends as output the signal "true" only if both its inputs are signalling "true", otherwise it signals "false".
   - An "or" connective (drawn in the diagram as a box labelled "or") sends as output the signal "false" only if both its inputs are signalling "false", otherwise it signals "true".
   - A "not" connective (drawn in the diagram as a box labelled "not") sends as output the signal "true" if its input signals "false", otherwise it signals "false" if its input signals "true".

   Signals are propagated through the circuit in the direction indicated by the arrows connecting components. Each circuit has two inputs (the circles labelled "a" and "b" in the diagram) and each has a single output (the circle labelled "out" in the diagram).





   Suppose that you have been given these circuits by an engineer who has two concerns: he is unsure whether or not the two circuits give the same responses to inputs, and he has a suspicion that at least one of the circuits shows little change in output regardless of input.

(a) Describe each of the two circuits as an expression in propositional logic, where the signal produced as the output of the circuit should correspond to the truth value of the expression. [*8 marks*]

(b) For each of the two propositional expressions construct a truth table that calculates the truth value of the expression for all combinations of truth values for its two inputs. [*10 marks*]

(c) Write a brief analysis of the circuits to answer the engineer's concerns (given above) using your truth tables as evidence. [*2 marks*]

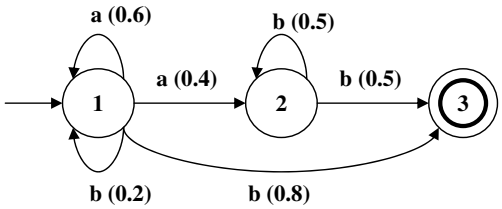
6. The genetic information contained in DNA is often expressed as a sequence of the characters `a`, `c`, `g` and `t`. Only certain sequences of these characters are permitted; valid sequences being restricted (for the purposes of this question) to those satisfying the following conditions:

   - A *gene* sequence consists of a *catbox* sequence, followed by a *base* sequence, followed by a *tatabox* sequence.
   - A *catbox* sequence consists of a *pyrimidine* followed by the sequence `[c,a,a,t]`.
   - A *base* sequence can EITHER be empty (consisting of no characters) OR can contain a *purine* followed by a further *base* sequence OR can contain a *pyrimidine* followed by a further *base* sequence.
   - A *purine* is EITHER the character `a` OR the character `g`.
   - A *pyramidine* is EITHER the character `c` OR the character `t`.
   - A *tatabox* sequence consists of the sequence `[t,a,t,a]` followed by a *purine*, followed by the character `a`.

   (a) Draw a Finite State Machine (FSM) that will accept gene sequences that are valid according to the definition above and will reject those that are invalid. Indicate which parts of your FSM diagram correspond to the catbox, base and tatabox components. Your machine can be non-deterministic. [*10 marks*]

   (b) Suppose that we want to construct an alternative FSM acceptor. In this acceptor a *gene* sequence consists of a *catbox* sequence, followed by a *base* sequence, followed by a *tatabox* sequence, followed by a second *base* sequence. The acceptor must also ensure that the number of occurrences of the character "`a`" is the same in the first and second *base* sequences. Describe briefly how you would extend your original FSM to deal with this new problem or, if it is not possible to construct the FSM, explain why. [*3 marks*]

7. The diagram below describes a Probabilistic Finite State Machine that accepts strings over the alphabet `{a,b}`. Each transition between states has a probability of occurrence (for example the transition between states 1 and 2 accepting character "`a`" has probability 0.4).



Explain how you would calculate the probability that the string `aabb` would be accepted by this FSM. [*5 marks*]