UNIVERSITY OF EDINBURGH

COLLEGE OF SCIENCE AND ENGINEERING

SCHOOL OF INFORMATICS

# INFORMATICS 1A

**Tuesday 21 August 2007**

**14:30 to 16:30**

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 numbers, and returns the product of one greater than every number in the input that is greater than or equal to two. For example, `f [2,0,1,3,4,-25]` returns `(2+1)*(3+1)*(4+1)`, that is, `60`. 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. We indicate missing characters in a string with asterisks, and combine two strings of equal length by picking whichever of the corresponding characters is not an asterisk, for example, combining `"ab**"` and `"**cd"` gives `"abcd"`. If both corresponding characters are asterisks, or if neither is, then an asterisk is put in that place, for example, combining `"*bc*"` and `"**cd"` gives `"*b*d"`.

   (a) Write a function `combine :: Char -> Char -> Bool`, that takes two characters and returns whichever of the two is not an asterisk, or returns an asterisk if both are asterisks or neither is. For example,

   ```
   combine 'a' '*'  returns  'a',  and
   combine '*' 'b'  returns  'b',  and
   combine '*' '*'  returns  '*',  and
   combine 'a' 'b'  returns  '*'.
   ```

   The function should indicate an error if either character is not an asterisk or a letter. You may use arithmetic and comparison and any library functions.

   [*6 marks* ]

   (b) Using `combine`, write a function `combines :: String -> String -> String` that combines the first string with the second. It should indicate an error if the two argument strings are not the same length. 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 `combine`, write a second definition of `combines`, 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 `hide :: Int -> Int -> String -> String` that takes two integers (say, `m` and `n`) and a string (say $s$), and returns a string of the same length as $s$ where the first `m` characters are asterisks, and every character after the first `n` characters are asterisks, and the other characters are copied from $s$. You may assume $m$ is less than or equal $n$ and $n$ is less than or equal the length of $s$. For example,

`hide 2 5 "abcdefgh"` returns `"**cde***"`.

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

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

## Part B COMPUTATION AND LOGIC

4. In Expression 1 and Expression 2 below, $A \rightarrow B$ means that $A$ implies $B$; $A$ *and* $B$ means that $A$ and $B$ both are true; $A$ *or* $B$ means that at least one of $A$ or $B$ is true; and $not(A)$ means that $A$ is false.

   **Expression 1:** $((a \rightarrow b)\ and\ not(b)) \rightarrow not(a)$

   **Expression 2:** $not(((not(a)\ or\ b)\ and\ not(b))\ and\ a)$

   (a) Prove that the expressions above are equivalent using truth tables. Show all the steps of calculation. [*10 marks* ]

   (b) Prove that the expressions above are equivalent by applying the following rules of equivalence between expressions:

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

   Show precisely how the equivalence rules are applied. [*5 marks* ]

5. You are given the following proof rules:

| Rule identifier | Sequent | Supporting proofs |
|---|---|---|
| 1 | $\mathcal{F} \vdash A$ | $A \in \mathcal{F}$ |
| 2 | $\mathcal{F} \vdash A\ and\ B$ | $\mathcal{F} \vdash A,\ \mathcal{F} \vdash B$ |
| 3 | $\mathcal{F} \vdash A\ or\ B$ | $\mathcal{F} \vdash A$ |
| 4 | $\mathcal{F} \vdash A\ or\ B$ | $\mathcal{F} \vdash B$ |
| 5 | $\mathcal{F} \vdash C$ | $\mathcal{F} \vdash (A\ or\ B),\ [A\|\mathcal{F}] \vdash C,\ [B\|\mathcal{F}] \vdash C$ |
| 6 | $\mathcal{F} \vdash B$ | $A \rightarrow B \in \mathcal{F},\ \mathcal{F} \vdash A$ |
| 7 | $\mathcal{F} \vdash A \rightarrow B$ | $[A\|\mathcal{F}] \vdash B$ |

   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 $\mathcal{F}$; $A \rightarrow B$ 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.

   Using the proof rules above, prove the following sequent:

$$[a \rightarrow b,\ (a\ and\ b) \rightarrow (c\ or\ d),\ c \rightarrow e,\ d \rightarrow e] \ \vdash\ a \rightarrow e$$

   Show precisely how the proof rules are applied (refer to appropriate rule identifiers and show the structure of the proof tree). [*10 marks* ]

6. You are given the following requirements description for the controller of an automated dispenser for cans of juice:

   The dispenser machine sells two types of juice: apple juice and orange juice. Each type costs 10 pence per can. The machine accepts only 5 pence and 10 pence coins. A customer inserts coins into the machine until the 10 pence cost is paid, at which point the customer selects apple or orange and an appropriate can of juice is dispensed. If the customer inserted 15 pence rather than 10 pence then the machine dispenses 5 pence change immediately after dispensing the can.

   (a) Define a finite state machine that models the behaviour of this dispenser. Assume that the machine always has internally a supply of coins, so that it always can dispense change. [*10 marks* ]

   (b) In reality a machine cannot be assumed always to have an inexhaustible internal supply of coins. Suppose that you are asked to extend your finite state machine so that it keeps track of the number of 5 pence coins inserted into the machine and if this has reached zero when it needs to dispense change it gives an "out of change" message to the customer. Either show how this can be defined in your finite state machine (stating any limiting assumptions you make) or explain precisely why it cannot be defined. [*5 marks* ]

7. Define a finite state machine that accepts the language defined by the regular expression:

$$\{c^*b\{a|b\}^*|\epsilon\}c$$

   where $\epsilon$ denotes the empty string. [*10 marks* ]