UNIVERSITY OF EDINBURGH

COLLEGE OF SCIENCE AND ENGINEERING

SCHOOL OF INFORMATICS

# INFORMATICS 1A - FUNCTIONAL PROGRAMMING

**Friday 14 December 2007**

**09:30 to 11:30**

Convener: M O'Boyle
External Examiner: R Irving

## INSTRUCTIONS TO CANDIDATES

1. **Note that ALL QUESTIONS ARE COMPULSORY.**
2. **DIFFERENT QUESTIONS MAY HAVE DIFFERENT NUMBERS OF TOTAL MARKS. Take note of this in allocating time to questions.**

## THIS EXAMINATION WILL BE MARKED ANONYMOUSLY

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.

```
div, mod :: Integral a => a -> a -> a
(+), (*), (-), (/) :: Num a => a -> a -> a
(<), (<=), (>), (>=) :: Ord => a -> a -> Bool
(==), (/=) :: Eq a => a -> a -> Bool
(&&), (||) :: Bool -> Bool -> Bool
not :: Bool -> Bool
```

Figure 1: Arithmetic, comparison, and logic

```
isAlpha :: Char -> Bool
isAlpha 'a' = True    isAlpha '1' = False
isAlpha 'A' = True    isAlpha '.' = False

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 product of one greater than each number between `3` and `7` (inclusive) in the list. For example, `f [0,3,8,-42,7,1,4]` returns `160`, which is `(3+1)*(7+1)*(4+1)`. Your definition may use *list comprehension*, arithmetic, comparison, logic, and *library functions*, but not recursion. [*12 marks* ]

   (b) Write a second function `g :: [Int] -> Int` that behaves like `f`, this time using *recursion*, arithmetic, comparison, logic, and list constructors `(:)` and `[]`, but not comprehensions or other library functions. [*12 marks* ]

   (c) Write a third function `h :: [Int] -> Int` that also behaves like `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, comparison, and logic, but do not use list comprehension, recursion, or any other library functions. [*12 marks* ]

2. We want to count the total number of words and non-words in a non-empty string. A word consists of letters, and a non-word consists of characters that are not letters. For example, `"Hello, world!"` contains two words (`"Hello"` and `"world"`) and two non-words (`", "` and `"!"`), so has a total of four; while `"Hello, world"` contains the same two words but only one non-word (no `"!"` at end), so has a total of three. By these rules, `"Phil's ?#*! class"` contains three words (`"Phil"` and `"s"` and `"class"`) and two non-words (`"'"` and `" ?#*! "`), so has a total of five.

   We can count the total number of words and non-words by counting transitions from letter to non-letter or non-letter to letter. For instance, there are a total of four words and non-words in the first string because there are three transitions from letter to non-letter or non-letter to letter (namely, `"o,"` and `" w"` and `"d!"`), and a total of three in the second string because there are two transitions (namely, `"o,"` and `" w"`), and a total of five in the third string because there are four transitions (namely, `"l'"` and `"'s"` and `"s "` and `" c"`).

   (a) Write a function `p :: Char -> Char -> Bool` that returns true if the first character is a letter and the second character is a non-letter, or if the first character is a non-letter and the second is a letter, but returns false if both characters are letters or both characters are non-letters. [*12 marks*]

   (b) Using `p`, write a function `q :: String -> Int` that counts the total number of words and non-words in the string, which you may assume is not empty. Your definition may use *list comprehension*, arithmetic, comparison, logic, and *library functions*, but not recursion. [*12 marks*]

   (c) Again using `p`, write a second function `r :: String -> Int` that behaves like `q`, this time using *recursion*, arithmetic, comparison, logic, and list constructors `(:)` and `[]`, but not comprehensions or other library functions. [*12 marks*]

3. (a) Write a function `t :: Int -> [a] -> [a]` that takes a non-negative integer and a non-empty list, and returns a list with length equal to the given integer, consisting of the given list repeated as many times as will fit, truncating at the end if necessary. For example,

| | | | |
|---|---|---|---|
| `t 0 "abcd"` | returns | `""`, | and |
| `t 2 "abcd"` | returns | `"ab"`, | and |
| `t 4 "abcd"` | returns | `"abcd"`, | and |
| `t 6 "abcd"` | returns | `"abcdab"`, | and |
| `t 8 "abcd"` | returns | `"abcdabcd"`, | and |
| `t 17 "abcd"` | returns | `"abcdabcdabcdabcda"`. | |

Your definition may use *list comprehension*, arithmetic, comparison, logic, and *library functions*, but not recursion. [*14 marks* ]

(b) Write a second function `u :: Int -> [a] -> [a]` that behaves like `t`, this time using *recursion*, arithmetic, comparison, logic, and list constructors `(:)` and `[]`, but not comprehensions or other library functions. [*14 marks* ]