Written Examination
DM22 Programming Languages

Department of Mathematics and Computer Science
University of Southern Denmark

Monday, June 27, 2005, 09.00–13.00

The exam set consists of four pages (including this front page), and contains four questions. The weight of each question is as follows:

Question 1: 25%
Question 2: 25%
Question 3: 30%
Question 4: 20%

The parts of a question do not necessarily have equal weight. Note that often a part can be answered independently from the other parts.

All written aids are allowed. Unless otherwise stated in a question, use of results from the course textbooks, and of the standard libraries of the programming languages used, is allowed.

Question 1 (25%)

SelectionSort is an $O(n^2)$ time sorting algorithm which works by repeatedly finding and removing the smallest element in a list.

Part a: Define in Haskell a function `selsort` which implements SelectionSort. □

Binary strings can be represented in Haskell by strings containing the characters 0 and 1. One natural ordering of binary strings is as follows: strings appear by increasing lengths, and for each length, the strings appear in lexicographical order. A list of the first ten strings in this ordering looks as follows in Haskell:

```
["","0","1","00","01","10","11","000","001","010"]
```

Part b: Define in Haskell an infinite list `binstrings` containing all binary strings in the above ordering. In particular, `take 10 binstrings` should be the list above. □

Question 2 (25%)

In this question, we consider sequences of elements from a set of size three. For concreteness, let the set be $S = \{1, 2, 3\}$, and the sequences be strings over S. In such a string, two identical neighboring substrings (non-empty, of course) are said to form a *repetition*. As an example, the following string contains the two underlined repetitions:

$$3\underline{11}32\underline{123123}2$$

A string having no repetitions is said to be *repetition-free*. The task of this question is to develop a Prolog predicate which generates all repetition-free strings over S of a given length. Strings will be represented as lists of integers from S.

Part a: Implement a Prolog predicate `frontRep(L)` which is true iff there is repetition starting at the front of the list `L`. [Hint: standard predicates (from textbook or standard library) on lists may come in handy.] □

Part b: Implement a Prolog predicate `repFree(X,N)` which is true iff `X` is a repetition-free list of elements in S and has length `N`. The predicate must be able to generate (as instantiations of `X`) all repetition-free lists of length some supplied `N`, by repeated use of `';'`. □

Part c: Implement a Prolog predicate `countLessThanEq(N,R)` which is true iff `R` is the number of repetition-free lists of elements in S of length less than or equal to `N`. The number of repetition-free lists of length zero is defined as one. □

Question 3 (30%)

Part a: For the Prolog program below, state all results (i.e. all instantiations of X and Y) which will be produced by repeated satisfaction of the goal t(X,Y) (i.e. by repeated use of ';').

```
t(X,Y):-s(X),!,v(Y),u(Y).
v(a).
v(b).
v(c).
u(b).
u(c).
s(1).
s(2).
```

□

Part b: Convert the following predicate logic expression to clausal form:

$$\forall X(\exists Y((a(X,Y) \vee b(Y)) \Rightarrow c(X)))$$

Document the steps of your conversion. □

Part c: For each of the following pairs of Prolog predicates, find a most general unifier (with occur-check), or argue that none exists. Explain each step of your derivations.

i) `f(Y,X,Y)` and `f(g(X),t,g(Z))`

ii) `add(X,g(Y),g(g(Z)))` and `add(g(g(Y)),g(T),T)`

iii) `length(X+Y,[Y|Z])` and `length(X,[0,1,2])`

(Recall that `[0,1,2]` is the same as `[0|[1,2]]`.) □

Part d: Consider the Haskell functions

```
map :: (a -> b) -> [a] -> [b]
zip :: [a] -> [b] -> [(a,b)]
```

For each of the following expressions, find its most general type. Explain each step of your derivations.

i) `map zip`

ii) `map . zip`

□

Question 4 (20%)

In the textbook, the following two functions appear (pages 197 and 199):

```
reverse :: [a] -> [a]
reverse []     = []
reverse (z:zs) = reverse zs ++ [z]

filter :: (a -> Bool) -> [a] -> [a]
filter p [] = []
filter p (x:xs)
  | p x       = x : filter p xs
  | otherwise = filter p xs
```

In part **a** and **b** below, we assume that `p :: a -> Bool` never returns the value undefined.

Part a:

Prove that for all `p` and for all finite lists `xs`, the following holds:

```
reverse (filter p xs) = filter p (reverse xs)
```

You may without proof use that

```
filter p (xs ++ ys) = (filter p xs) ++ (filter p ys)
```

for all `p` and all lists `xs` and `ys`. □

Part b: Extend the argumentation from the previous question to prove that for all `p`, we have

```
reverse . filter p = filter p . reverse
```

□

Part c: Argue that the equations in **a** and **b** do not hold without the assumption on `p` stated in the beginning. □