# Written Examination
# DM22 Programming Languages

Department of Mathematics and Computer Science
University of Southern Denmark

Monday, June 18, 2007, 09.00–13.00

The exam set consists of 6 pages (including this front page), and contains 4 questions. The weight of each question is as follows:

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

The parts of a question do not necessarily have equal weight.

Note that often a part can be answered independently from the other parts, and that your solutions to one part may exploit the results of other parts, even if these parts have not been answered.

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 (30%)

**Part a:** Make in Haskell a definition of

```
limit :: Eq a => (a -> a) -> a -> a
```

such that `limit f x` is the first value that appears two times in a row in the sequence `x` , `f x` , `f(f x)` , `f(f(f x)))` , .... □

The remaining part of this exercise is inspired by issues from databases. However, it is *not* necessary to have any knowledge of databases in order to answer the questions.

Let a *relation* be a list of letters (i.e., a string). For a given relation, let a *functional dependency* be a pair of sublists of the relation. All lists in this question are dublicate free (no element appears twice). Below are some examples of relations and functional dependencies.

```
rel :: String
rel = "ABCDEFGK"

fd1,fd2 :: (String,String)
fd1 = ("AB","E")
fd2 = ("BCDFK","BG")
```

For a given relation $R$, let $S$ be a sublist of $R$, and let $f = (X, Y)$ be a functional dependency on $R$. The following pseudo-code defines what it means to *expand* $S$ using $f$:

**If** $X \subseteq S$ **then** $S = S \cup Y$

Here, the set operators have their usual meaning when thinking of duplicate free lists as sets.

**Part b:** Make in Haskell a definition of

```
expand :: String -> (String,String) -> String
```

such that `expand s fd` is the result of expanding `s` using `fd`. □

**Part c:** Make in Haskell a definition of

```
expandList :: String -> [(String,String)] -> String
```

such that `expandList s fdList` is the result of expanding `s` using each functional dependency in `fdList` in turn (i.e., it is the result of expanding `s` using the first functional dependency of `fdList`, then expanding the result of that using the next functional dependency of `fdList`, etc.). □

For a given relation $R$, let $T$ be a sublist of $R$ and let $F$ be a list of functional dependencies on $R$. The following pseudo-code describes an algorithm which calculates what is known as the *closure* of $T$.

$S = T$
**Repeat**
  **For** each functional dependency $f$ in $F$
    expand $S$ using $f$
**Until** no change in $S$ happened during the last iteration of repeat
**Return** $S$

**Part d:** Implement the algorithm above in Haskell—that is, make in Haskell a definition of

```
closure :: String -> [(String,String)] -> String
```

such that `closure t fdList` is the closure of `t`, where `fdList` is the list of functional dependencies used by the algorithm. □

## Question 2 (20%)

**Part a:** Implement a Prolog predicate `pairsums(L1,L2)` which for `L1` a list of numbers is true iff `L2` is the list of sums of each neighboring pair of elements in `L1`.

As an example, if `L1` is `[1,3,6,10]`, then `L2` should be instantiated to `[4,9,16]` by the predicate. If `L1` contains less than two elements, `L2` should be instantiated to the empty list. □

**Part b:** The aim here is to implement in Prolog a predicate `zip(L1,L2,L3)` with functionality corresponding to the library function of the same name in Haskell.

More precisely, implement a Prolog predicate `zip(L1,L2,L3)` which is true iff `L1` and `L2` are lists and `L3` is a list of tuples combining elements in equal positions in `L1` and `L2`.

As an example, if `L1` is `[a,b,c]` and `L2` is `[4,5,6,7]`, then `L3` should be instantiated to `[(a,4),(b,5),(c,6)]` by the predicate. □

**Part c:** The aim here is to implement in Prolog a predicate `zipWith(F,L1,L2,L3)` with functionality corresponding to the library function of the same name in Haskell.

More precisely, implement a Prolog predicate `zipWith(F,L1,L2,L3)` which is true iff `F` is the name (i.e., functor) of a ternary predicate, `L1` and `L2` are lists, and `L3` is a list of values occuring as third argument `Z` in `f(X,Y,Z)` when `f` is the value of `F` and `X` and `Y` are elements in equal positions in `L1` and `L2`.

As an example, if `L1` is `[4,5,7]`, `L2` is `[10,11,12,25]`, and `F` is `add` which is the name of a ternary predicate defined by the single clause

```
add(X,Y,Z) :- Z is X+Y. ,
```

then `L3` should be instantiated to `[14,16,19]` by the predicate.

Hint: use the Prolog operator `=..` and predicate `call`. □

## Question 3 (25%)

**Part a:** For the Prolog program below, state for each of the two goals `a(X,Y)` and `b(X,Y)` all results (i.e. all instantiations of `X` and `Y`) which will be produced by repeated satisfaction (i.e. by repeated use of `';'`) of the goal.

```
a(X,Y) :- c(X),c(Y).
b(X,Y) :- !,c(X),!,c(Y),!.
c(1).
c(2) :- !.
c(3).
```

□

**Part b:** 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) `a(b(b(Y)),c(c(Z)),d(d(X)))` and `a(X,Y,Z)`

ii) `x(y(y(T)),y(y(Y)),y(t))` and `x(y(Z),Z,y(Y))`

iii) `test(Z,2,4,6)` and `test(X+Y,X,Y,Z)`

iv) `abc` and `[a,b,c]`

□

Consider the following Haskell definition:

```
f a []     = []
f a (x:xs) = sum : f sum xs
  where sum = a+x
```

**Part c:** Find the most general type for `f`. Explain each step of your derivation.

□

**Part d:** Make in Haskell an interactive function

```
findLongestLine :: IO ()
```

which reads two lines from the user and then prints on the screen the longest of these. □

## Question 4 (25%)

For a list of numbers, we would like to find the sums of all prefixes. The empty prefix has sum 0 by definition. As an example, the prefix sums of the list `[1,2,5,10]` is the list `[0,1,3,8,18]`.

In this question, we will consider generalized prefix sums, where a fixed value *shift* is added to each prefix. As an example, the generalized prefix sums with shift 10 of the list `[1,2,5,10]` is the list `[10,11,13,18,28]`.

The following function finds generalized prefix sums with shift `s`, where `f` is the function from part c of the previous question:

```
gpfs1 s xs = s : f s xs
```

Consider the following alternative definition:

```
gpfs2 = scanl (+)

scanl f e []     = [e]
scanl f e (x:xs) = e : scanl f (f e x) xs
```

**Part a:** Prove that for all finite lists `xs` and for all `s`, the following holds:

```
gpfs1 s xs = gpfs2 s xs
```

Hint: use induction on `xs` □

**Part b:** Prove that `gpfs1 s` is different from `gpfs2 s` (for all `s`). □

**Part c:** It is also possible to define `gpfs` using a cyclic structure. Do this by completing the `...` part in the following partial definition.

```
gpfs3 s xs = ys
  where ys = ...
```

No proof of equality with the previous definitions needs to be given. □