Written Examination
DM 509 Programming Languages
– Solution –

Department of Mathematics and Computer Science
University of Southern Denmark

Tuesday, January 12, 2010, 09:00 – 13:00

This exam set consists of 7 pages (including this front page) and contains a total of 5 problems. Each problem is weighted by the given percentage. The individual questions of a problem are not necessarily weighted equally.

Most questions in a problem can be answered independently from the other questions of the same problem.

All written aids are allowed. Answering questions by reference to material not listed in the course curriculum is not acceptable.

You may answer the exam in English or in Danish.

This document contains the essential parts of the solutions to the exam set identified above. Note that many times, there are several possible solutions, and this document just lists one. Also, perfect answers to some of the exam questions should contain explanations which are generally omitted in this document. Finally, this document has not been scrutinized in the same meticulous manner as an exam set and may contain typos, etc.

Problem 1 (20%)

Question a: Implement a Prolog predicate `take/3` such that `take(N,L,M)` is true if, and only if, `M` is the longest prefix of length at most `N` of the list `L`.

For example, the query

```
?- take(2, [6,3,4], M).
```

should yield the answer `M / [6,3]`. Likewise, the query

```
?- take(4, [6,3,4], M).
```

should yield the answer `M / [6,3,4]`.

Possible Solution:

```
take(N,L,M) :- length(L,O), P is min(O,N), append(M,Q,L), length(M,P).
```

Alternative Solution:

```
take(0,_,[]).
take(_,[],[]).
take(N,[X|Xs],[X|Ys]) :- N > 0, O is N-1, take(O,Xs,Ys).
```

Question b: Implement a Prolog predicate `firstHalf/2` such that `firstHalf(L, M)` is true if, and only if, `M` is the list that contains exactly the first half of the elements of the list `L`.

For example, the query

```
?- firstHalf([6,3,4,5], M).
```

should yield the answer `M / [6,3]`. Likewise, the query

```
?- firstHalf([6,3,4], M).
```

should yield the answer `M / [6]`.

Possible Solution:

```
firstHalf(L,M) :- length(L,N), O is N//2, take(O,L,M).
```

Question c: Implement a Prolog predicate `fib/4` such that `fib(A,B,N,M)` is true if, and only if, `M` is the `N`-th number of the Fibonacci sequence starting with the numbers `A` and `B`.

For example, the query

```
?- fib(0,1,3,M).
```

should yield the answer `M / 2` and the query

```
?- fib(0,1,7,M).
```

should yield the answer `M / 13`.

Your implementation should return the answer in time **linear** in **N**. You may assume that built-in addition has constant time complexity.

Possible Solution:

```
fib(A,B,0,A).
fib(A,B,1,B).
fib(A,B,N,M) :- N > 1, 0 is N-1, C is A+B, fib(B,C,0,M).
```

Question d: A magic square is a matrix of dimension $n \times n$ containing all numbers from 1 to n^2 such that the sum of each row and of each column is exactly $\frac{n(n^2+1)}{2}$.

The following is an example of a square of dimension 3×3.

4	9	2
3	5	7
8	1	6

We represent such a square as a list of the concatenated rows. I.e., the above square would be represented as follows.

```
[4,9,2,3,5,7,8,1,6]
```

Implement a Prolog predicate `magic/1` such that the query `?- magic(L).` has exactly those lists `L` as answers that represent a magic square of dimension 3×3.

You may (but do not have to) use constraint logic programming for your implementation.

Possible Solution:

```
magic(L) :- L = [A,B,C,D,E,F,G,H,I],
            fd_domain(L,1,9),
            fd_all_different(L),
            A+B+C #= 15, D+E+F #= 15, G+H+I #= 15,
            A+D+G #= 15, B+E+H #= 15, C+F+I #= 15,
            fd_labeling(L).
```

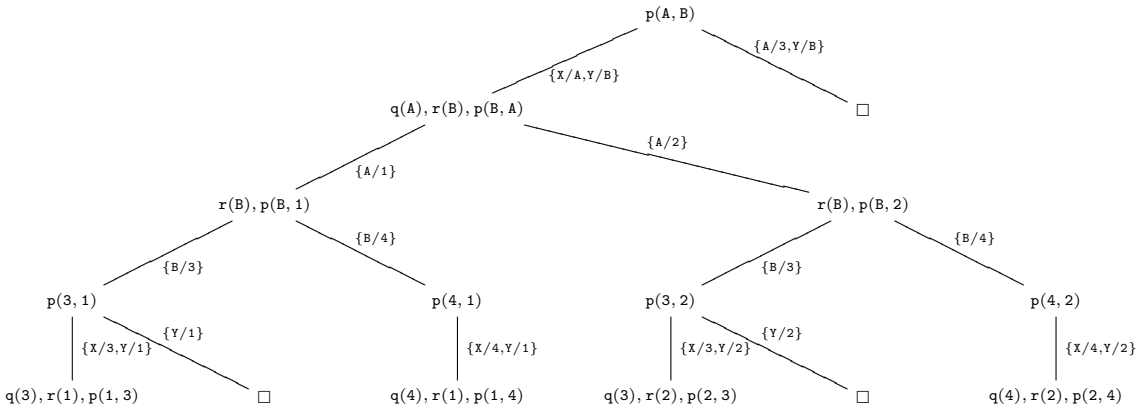
Problem 2 (25%)

Question a: Consider the following Prolog program.

```
p(X,Y) :- q(X), r(Y), p(Y,X).
p(3,Y).
q(1).
q(2).
r(3).
r(4).
```

Draw the SLD tree for the query `?- p(A,B).` and list all answers with the instantiations of `A` and `B`.

Possible Solution:



The answers returned by Prolog are:

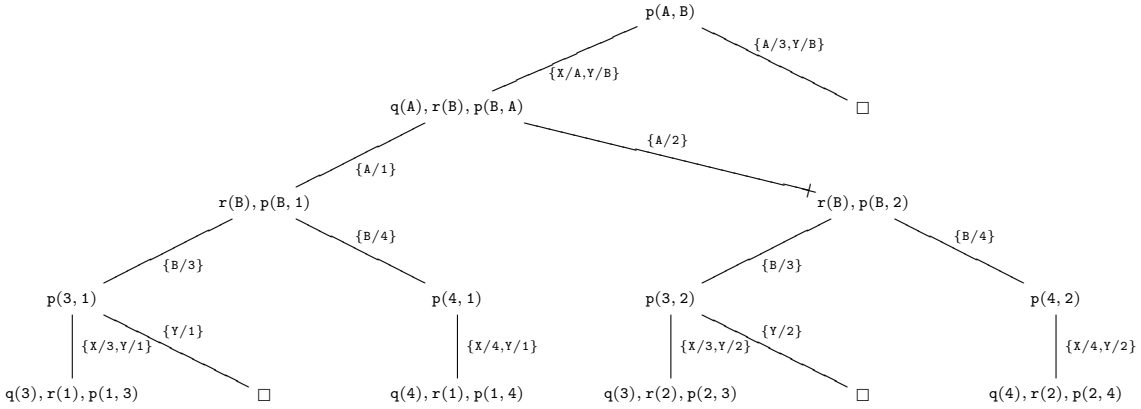
```
A = 1, B = 3
A = 2, B = 3
A = 3
```

Question b: We now introduce a cut into the body of the third clause from Question a, i.e., we now have the following Prolog clauses for `q/1`.

```
q(1) :- !.
q(2).
```

Indicate in the SLD tree of Question a which branches are cut and list all remaining answers with the instantiations of `A` and `B`.

Possible Solution:



The answers returned by Prolog are:

```
A = 1, B = 3
A = 3
```

Question c: For the following pairs of Prolog terms, find a most general unifier or argue that none exists. Show the steps of the algorithm. In case of success, give the resulting substitution. In case of failure, state if it is an occur failure or a clash failure.

1. `p(f(X),a,Y)` and `p(f(Y),X,b)`
2. `q(g(X),g(Y),Y)` and `q(g(A),A,g(X))`
3. `r(a,0,[X,Y])` and `r(X,Y,[X|Xs])`

Possible Solution:

1. CLASH FAILURE
$$\begin{aligned} & \{\mathtt{p(f(X),a,Y)} \stackrel{?}{=} \mathtt{p(f(Y),X,b)}\} \\ \Rightarrow (DECOMPOSE) \quad & \{\mathtt{f(X)} \stackrel{?}{=} \mathtt{f(Y)}, \mathtt{a} \stackrel{?}{=} \mathtt{X}, \mathtt{Y} \stackrel{?}{=} \mathtt{b}\} \\ \Rightarrow (DECOMPOSE) \quad & \{\mathtt{X} \stackrel{?}{=} \mathtt{Y}, \mathtt{a} \stackrel{?}{=} \mathtt{X}, \mathtt{Y} \stackrel{?}{=} \mathtt{b}\} \\ \Rightarrow (ELIMINATE) \quad & \{\mathtt{X} \stackrel{?}{=} \mathtt{b}, \mathtt{a} \stackrel{?}{=} \mathtt{X}, \mathtt{Y} \stackrel{?}{=} \mathtt{b}\} \\ \Rightarrow (ELIMINATE) \quad & \{\mathtt{X} \stackrel{?}{=} \mathtt{b}, \mathtt{a} \stackrel{?}{=} \mathtt{b}, \mathtt{Y} \stackrel{?}{=} \mathtt{b}\} \end{aligned}$$
2. OCCUR FAILURE
$$\begin{aligned} & \{\mathtt{q(g(X),g(Y),Y)} \stackrel{?}{=} \mathtt{q(g(A),A,g(X))}\} \\ \Rightarrow (DECOMPOSE) \quad & \{\mathtt{g(X)} \stackrel{?}{=} \mathtt{g(A)}, \mathtt{g(Y)} \stackrel{?}{=} \mathtt{A}, \mathtt{Y} \stackrel{?}{=} \mathtt{g(X)}\} \\ \Rightarrow (DECOMPOSE) \quad & \{\mathtt{X} \stackrel{?}{=} \mathtt{A}, \mathtt{g(Y)} \stackrel{?}{=} \mathtt{A}, \mathtt{Y} \stackrel{?}{=} \mathtt{g(X)}\} \\ \Rightarrow (ELIMINATE) \quad & \{\mathtt{X} \stackrel{?}{=} \mathtt{A}, \mathtt{g(Y)} \stackrel{?}{=} \mathtt{A}, \mathtt{Y} \stackrel{?}{=} \mathtt{g(A)}\} \\ \Rightarrow (ORIENT) \quad & \{\mathtt{X} \stackrel{?}{=} \mathtt{A}, \mathtt{A} \stackrel{?}{=} \mathtt{g(Y)}, \mathtt{Y} \stackrel{?}{=} \mathtt{g(A)}\} \\ \Rightarrow (ELIMINATE) \quad & \{\mathtt{X} \stackrel{?}{=} \mathtt{A}, \mathtt{A} \stackrel{?}{=} \mathtt{g(Y)}, \mathtt{Y} \stackrel{?}{=} \mathtt{g(g(Y))}\} \end{aligned}$$
3. SUBSITUTION {X/a, Y/0, Xs/[0]}
$$\begin{aligned} & \{\mathtt{r(a,0,[X,Y])} \stackrel{?}{=} \mathtt{r(X,Y,[X|Xs])}\} \\ = \quad & \{\mathtt{r(a,0,.(X,.(Y,[])))} =?\mathtt{r(X,Y,.(X,Xs))}\} \\ \Rightarrow (DECOMPOSE) \quad & \{\mathtt{a} \stackrel{?}{=} \mathtt{X}, \mathtt{0} \stackrel{?}{=} \mathtt{Y}, \mathtt{.(X,.(Y,[]))} \stackrel{?}{=} \mathtt{.(X,Xs)}\} \\ \Rightarrow (DECOMPOSE) \quad & \{\mathtt{a} \stackrel{?}{=} \mathtt{X}, \mathtt{0} \stackrel{?}{=} \mathtt{Y}, \mathtt{X} \stackrel{?}{=} \mathtt{X}, \mathtt{.(Y,[])} \stackrel{?}{=} \mathtt{Xs}\} \\ \Rightarrow (DELETE) \quad & \{\mathtt{a} \stackrel{?}{=} \mathtt{X}, \mathtt{0} \stackrel{?}{=} \mathtt{Y}, \mathtt{.(Y,[])} \stackrel{?}{=} \mathtt{Xs}\} \\ \Rightarrow (ORIENT) \quad & \{\mathtt{X} \stackrel{?}{=} \mathtt{a}, \mathtt{0} \stackrel{?}{=} \mathtt{Y}, \mathtt{.(Y,[])} \stackrel{?}{=} \mathtt{Xs}\} \\ \Rightarrow (ORIENT) \quad & \{\mathtt{X} \stackrel{?}{=} \mathtt{a}, \mathtt{Y} \stackrel{?}{=} \mathtt{0}, \mathtt{.(Y,[])} \stackrel{?}{=} \mathtt{Xs}\} \\ \Rightarrow (ORIENT) \quad & \{\mathtt{X} \stackrel{?}{=} \mathtt{a}, \mathtt{Y} \stackrel{?}{=} \mathtt{0}, \mathtt{Xs} \stackrel{?}{=} \mathtt{.(Y,[])}\} \\ \Rightarrow (ELIMINATE) \quad & \{\mathtt{X} \stackrel{?}{=} \mathtt{a}, \mathtt{Y} \stackrel{?}{=} \mathtt{0}, \mathtt{Xs} \stackrel{?}{=} \mathtt{.(0,[])}\} \\ = \quad & \{\mathtt{X} \stackrel{?}{=} \mathtt{a}, \mathtt{Y} \stackrel{?}{=} \mathtt{0}, \mathtt{Xs} \stackrel{?}{=} \mathtt{[0]}\} \end{aligned}$$

Problem 3 (15%)

Question a: Define a HASKELL function `divisibleByTwo` which takes a positive integer and determines if it is divisible by `2`.

For example, `divisibleByTwo 3 = False` and `divisibleByTwo 2 = true`.

Here, you may not use any pre-defined functions except for `(+)` and `(-)`.

Possible Solution:

```
divisibleByTwo 0 = True
divisibleByTwo 1 = False
divisibleByTwo n = divisibleByTwo (n-2)
```

Question b: Define a HASKELL function `divisibleByTwoList` which takes a list of positive integers and determines if at least one of the elements is divisible by `2`.

For example, `divisibleByTwoList [1,2,3] = True`.

You should use the function `divisibleByTwo` from Part a.

Possible Solution:

```
divisibleByTwoList xs = any divisibleByTwo xs
```

Alternative Solution:

```
divisibleByTwoList [] = False
divisibleByTwoList (x:xs) = divisibleByTwo x || divisibleByTwoList xs
```

Question c: Define a HASKELL function `productPairs` which takes a positive integer `n` and computes the list of all pairs `(x,y)` of natural numbers such that their product is exactly `n`.

For example, `productPairs 4` should return the following list.

```
[(1,4), (2,2), (4,1)]
```

Possible Solution:

```
productPairs n = filter (\(x,y) -> x * y == n) (map (\x -> (x, div n x)) [1..n])
```

Alternative Solution:

```
productPairs n = p n n where
  p 0 n = []
  p i n | mod n i == 0 = (div n i, i) : p (i-1) n
        | otherwise = p (i-1) n
```

Question d: The sequence of Fibonacci words is defined as the $w_0, w_1, w_2, \ldots$ such that $w_0 =$ `"a"`, $w_1 =$ `"ab"`, and $w_{n+2} = w_{n+1}w_n$. Thus, $w_2 =$ `"aba"`, $w_3 =$ `"abaab"` etc.

Give a HASKELL declaration for the infinite list `fibWord` of all Fibonacci words as defined above.

For example, `take 4 fibWord` should return the following list.

```
["a", "ab", "aba", "abaab"]
```

Possible Solution:

```
fibWord = "a" : "ab" : zipWith (++) (tail fibWord) fibWord
```

Alternative Solution:

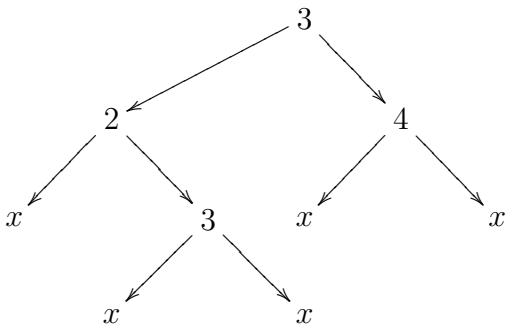
```
fibWord = f "a" "ab" where
  f n m = n : f m (m ++ n)
```

Problem 4 (20%)

Question a: Consider the following data type for binary trees.

```
data Tree a = Leaf | Node (Tree a) a (Tree a)
```

Thus, the expression `ex = Node (Node Leaf 2 (Node Leaf 3 Leaf)) 3 (Node Leaf 4 Leaf)` corresponds to the following tree (with leaves marked by x).



Define a HASKELL function `paths` which takes a `Tree a` and produces the list of paths from the root to a `Node` that has two `Leaf` children.

For example, `paths ex` should return `[[3,2,3], [3,4]]`.

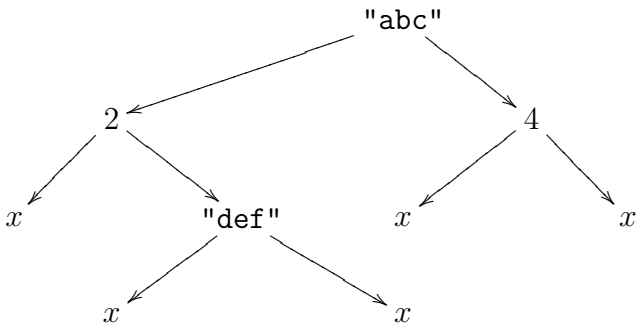
Possible Solution:

```
paths = p [] where
  p path Leaf = []
  p path (Node Leaf x Leaf) = [path++[x]]
  p path (Node left x right) = p (path++[x]) left ++ p (path++[x]) right
```

Alternative Solution:

```
paths t = map reverse (p [] t) where
  p path Leaf = []
  p path (Node Leaf x Leaf) = [x:path]
  p path (Node left x right) = p (x:path) left ++ p (x:path) right
```

Question b: Consider the following tree that contains elements of type `String` and of type `Int`.



Declare a HASKELL data type `LevelTree a b` using a `data` declaration that on the first level of the tree contains elements of type `a`, on the second level elements of type `b`, on the third level again elements of type `a` etc.

The tree above would be of type `LevelTree String Int`.

Possible Solution:

```
data LevelTree a b = Leaf | Node (LevelTree b a) a (LevelTree b a)
```

Problem 5 (20%)

Question a: Find the most general type for each of the following two HASKELL functions.

- `f x y z = f y z x`
- `g (x:xs) = \x -> [xs]`

Explain your reasoning.

Possible Solution:

- Assume `f::a -> c -> d -> b`, `x::e`, `y::g`, and `z::h`. As `f` is applied to `x` (on the left-hand side) and `y` (on the right-hand side), `a`, `e`, and `g` must unify. Similarly, we see that `c`, `g`, and `h` must unify. The most general unifier is {`e/a, g/a, h/a`. Thus, the resulting type is the following.

  ```
  f::a -> a -> a -> b
  ```

- Assume `g::c -> d` and `(:)::a -> [a] -> [a]`. Then due to `g` being applied to `x:xs` we need to unify `c` and `[a]`. Furthermore, we have to unify `d` with the type of `\x -> [xs]`. From `x:xs` we get `x::a` and `xs::[a]` and thus need to unify `d` with `b -> [[a]]` where `b` is the type of the `x` bound by lambda. Thus, the resulting type is the following.

  ```
  g::[a] -> b -> [[a]]
  ```

Question b: Consider the two following ways of defining HASKELL functions for computing the length of a list.

```
length1 [] = 0
length1 (x:xs) = 1 + length1 xs

length2 = len 0 where
  len n [] = n
  len n (x:xs) = len (n+1) xs
```

Prove by induction that for all `ys` of type `[a]`, these two definitions yield the same result, i.e., `length1 ys = length2 ys`.

You may assume the following lemma about `len` for all `n` and `m` of type `Int` and all `zs` of type `[a]`.

```
n + len m zs = len (n+m) zs
```

Possible Solution:

- base case (`ys = []`)

  `length1 [] = 0 = len 0 [] = length2 []`

- step case (assume theorem holds for `ys`, show it holds for `y:ys`)
  Using the definition of `length1` and the induction hypothesis we obtain:

  `length1 (y:ys) = 1 + length1 ys = 1 + length2 ys`

  Next, we use the definition of `length 2` and the given lemma:

  `1 + length2 ys = 1 + len 0 ys = len (1+0) ys`

  Now, we use commutativity of `(+)` and the definition of `len` backwards.

  `len (1+0) ys = en (0+1) ys = len 0 (y:ys)`

  Finally, by applying the definition of `length2` backwards, we obtain:

  `len 0 (y:ys) = length2 (y:ys)`