LULEÅ UNIVERSITY OF TECHNOLOGY

Final exam in **Declarative languages**
Number of problems: 7
Teacher: Håkan Jonsson, 491000
The result will be available: ASAP.

Course code	D7012E
Date	2018-04-25 and 2018-05-28
Total time	2h+4h (should have been 4h)

General information

I. Predefined functions and operators Note that Appendices A and B – roughly half the exam – lists predefined functions and operators you may use freely, if not explicitly stated otherwise.

II. The Prolog database If not explicitly stated otherwise, solutions may *not* be based on the direct manipulation of the database with built-in procedures like `asserta`, `assertz`, `retract`, etc.

III. Helper functions It is allowed to add helper functions, if not explicitly stated otherwise. (Maybe needless to write but, of course, all added helpers must also be written in accordance with the limitations and requirements given in the problem description.)

IV. Explanations You must give **short** explanations for all declarations. Haskell declarations must include types. For a function/procedure, you must explain what it does and what the purpose of each argument is.

Solutions that are poorly explained might get only few, or even zero, points. This is the case regardless of how correct they otherwise might be.

Explain with at most a few short and clear (readable) sentences (not comments). Place them next to, but clearly separate from, the code. Use arrows to point out what is explained. Below is one example, of many possible, with both Haskell and Prolog code explained.

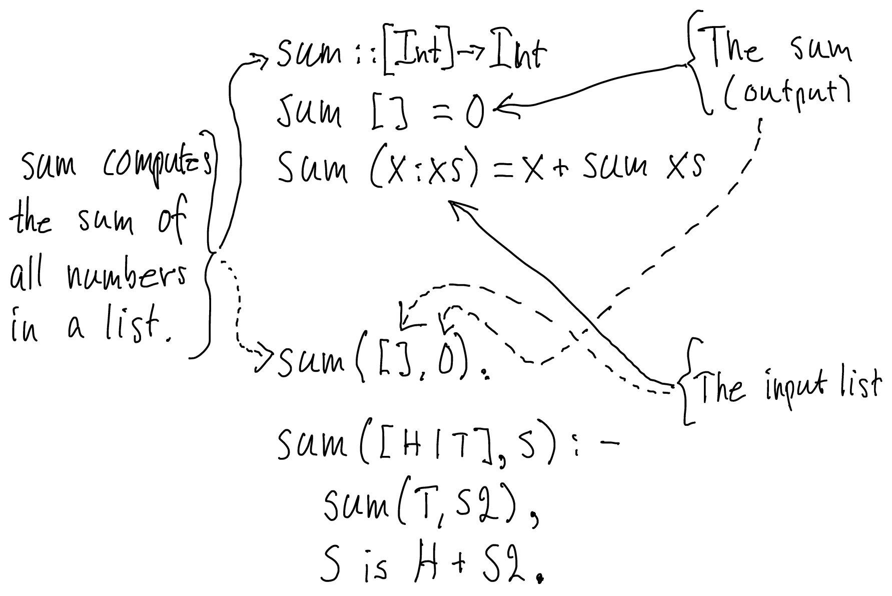

Figure 1: Example showing how to explain code.

1 Types and pattern matching

(a) Here are five expressions. State, for each valid expression, its type. If an expression is invalid, state instead this fact and explain what is wrong. (2,5p)

1) `head "8"` 2) `[]:[]` 3) `length.map (+1)` 4) `\f -> "f" 0` 5) `map filter`

(b) The *Ackermann function* is defined like this:

```
ack 0 n = n+1
ack m 0 = ack (m-1) 1
ack m n = ack (m-1) (ack m (n-1))
```

Now, what does each of the following expressions evaluate to? (1,5p)

1) `ack 2 0` 2) `ack 0 2` 3) `ack 1 1`

2 Lists

(a) Write a function `isOrdered` that checks whether or not orderable elements of a list occur in non-decreasing order. Clearly state the type of your function. NB! Your function should work for *all* types of lists whose elements can be ordered. (2p)

`isOrdered [-3,-1,1,3,7,5,9]` returns `False` while `isOrdered []`, `isOrdered [3.5]`, `isOrdered [-5.0,-5.0]`, `isOrdered [-2,-1,0,1,2]`, and `isOrdered [-2,2,2,2,22]` all return `True`.

(b) Write a function *compress* that, given a list of integers given in non-decreasing order, returns a list of pairs: (3p)

`compress []` returns `[]`.
`compress [3]` and `compress [3,3,3,3,3,3,3]` both return `[(3,3)]`.
`compress [2,3]` and `compress [2,2,2,2,3,3,3]` both return `[(2,3)]`.
`compress [1,3]` and `compress [1,1,1,3,3,3,3]` both return `[(1,1),(3,3)]`.
`compress [2,2,2,3,3,3,4,4,4,4,5,5,5,5,6,7,7]` returns `[(2,7)]`.
`compress [2,2,2,4,4,4,4,5,5,5,5,6,7,7]` returns `[(2,2),(4,7)]`.
`compress [1,2,2,3,5,6,9,11,12]` returns `[(1,3),(5,6),(9,9),(11,12)]`.

Each pair in the result corresponds to a maximum lenght subsequence, a "range", in which the difference between neighboring elements is at most one.

(c) Write a function `mkList :: Int -> Int -> [Int]` that, given two integers `f` and `s`, creates the same infinite list as the built-in `[f,s..]` would produce. Do so without using `[f,s..]`. (2p)

`mkList 1 (-4)` returns `[1,-4,-9,-14,-19,-24,-29,-34,-39,-44,...]`
`mkList 1 1` returns `[1,1,1,1,1,1,1,1,1,1,...]`
`mkList 1 4` returns `[1,4,7,10,13,16,19,22,25,28,...]`

3 Algebraic data types

(a) Declare a recursive algebraic data type `Tree a` for trees in which each leaf is empty and each inne node has three subtrees of type `Tree a` and a pair of type `(a,Int)`. Figure 2 shows an example of such a tree. (2p)

(b) Write a function `extract :: Int -> Tree a -> [a]` that, given an integer `i` and a tree `t`, returns a list with the first components of all pairs in `t` that has `i` as their second component. If we apply `extract 2` on the tree in Figure 2, we get a list with the two characters `'H'` and `'J'` (the order does not matter). (3p)

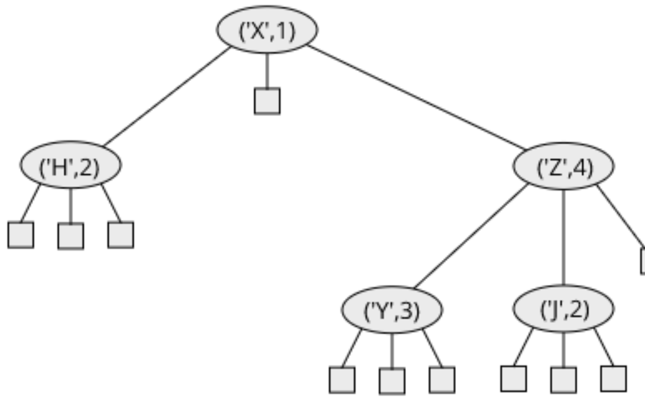

Figure 2: A `Tree Char` containing the pairs `('Z',4)`, `('X',1)`, `('J',2)`, `('Y',3)`, and `('H',2)`.

4 I/O and some monadic programming

The output below is from a simple game in which a player is to guess a secret random number known only to the computer and gets hints after every false guess. The player has entered `game`, to start, and then `50`, `25`, `12`, `18`, `21`, and `20`. The rest of the output has been printed by the game (and the Haskell system).
Write a Haskell program that plays the game. Use the following extra functions: (4p)

- `getRnd :: IO Int`, that returns a pseudo-random number in $[1, 2, \ldots 100]$.
- `getInt :: IO Int`, that reads an integer from the keyboard.
- `putStr :: IO ()`, that prints to the screen.

```
*Main> game
 Welcome to the Game!
 What secret number between 1 and 100 am I thinking of?!
 Enter guess number 1: 50
 Too high!
 Enter guess number 2: 25
 Too high!
 Enter guess number 3: 12
 Too low!
 Enter guess number 4: 18
 Too low!
 Enter guess number 5: 21
 Too high!
 Enter guess number 6: 20
 Correct after 6 guesses!
 Thank's for playing!
*Main>
```

5 Fibonacci numbers [To be solved using Prolog]

a) The *Fibonacci numbers* are defined as follows:

$$F_n = \begin{cases} 0 & \text{if } n = 0, \\ 1 & \text{if } n = 1, \\ F_{n-1} + F_{n-2} & \text{if } n > 1. \end{cases}$$

Write a procedure `fib(N,FN)` that returns Fibonacci number $F_{\mathtt{N}}$ as `FN`. Write `fib` so it does not backtrack in attempts to find more than one solution. (4p)

b) Use `fib` and write a procedure `fibs(N,L)` that returns `L`= $[F_0, F_1, \ldots, F_{N-1}, F_N]$. (3p)

Example:

```
?- fibs(8,L).
L = [0, 1, 1, 2, 3, 5, 8, 13, 21].
```

In this subproblem you are allowed, if you like, to use the built-in procedure `sort(0, @=<, In, Out)` that sorts the list `In` in ascending order, while keeping duplicate elements, and returns it as `Out`.

6 Logical equivalence [To be solved using Prolog]

What is the logical equivalent of the following programs? State a logical (boolean) expression equivalent to `p` in each of the four cases. (4p)

a)
```
P :- !, A, B, C.
P :- A.
```

b)
```
P :- A, !, B, C.
P :- B.
```

c)
```
P :- A, B, !, C.
P :- C.
```

d)
```
P :- A, B, C, !.
P :- B.
```

7 Connection problems [To be solved using Prolog]

In this problem, `arc(N1,N2)` represent a directed arc going from node `N1` to `N2`. Arcs may not start and end at the same node, so `N1`$\neq$`N2`. Moreover, the Prolog database contains

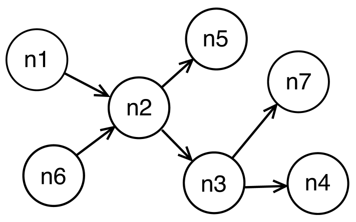

Figure 3: An acyclic graph.

`arc`-terms that form an acyclic graph. Fig. 3 shows such a graph with 7 nodes and 6 arcs between the nodes. In this case, the database contains `arc(n1,n2)`, `arc(n2,n5)`, `arc(n2,n3)`, `arc(n3,n7)`, `arc(n3,n4)`, and `arc(n6,n2)`.

a) A path from `A` to `B` is either a single arc `arc(A,B)` or a sequence `arc(A,N1)`, `arc(N1,N2)`, `arc(N2,N3)`, ..., `arc(NK,B)` of arcs. The graph in Fig. 3 contains several paths, one of which being `[arc(n1,n2),arc(n2,n3),arc(n3,n4)]` that goes from `n1` to `n4`.

Write a general procedure `path(A,B,P)` that returns, in `P`, a path from `A` to `B`[1]. (3p)

b) A new term `T` can be added to the database with `asserta(T)`. For instance, writing `asserta(arc(n6,n3))` adds the term `arc(n6,n3)`.

Write a procedure `add(arc(A,B))` that adds `arc(A,B)` to the database *but only* if all the arcs in the database still form an acyclic graph after the addition. (3p)

[1]In the figure, there is only one path between any pair of nodes but in general there could be several.

c) Finally, write a procedure `allPaths(N,L)` that gives back a list `L` with all paths in the database that contain `N` nodes. (3p)

Examples when the database contains the arcs of the graph in Fig. 3.

```
?- allPaths(4,List).
List = [[arc(n1, n2), arc(n2, n3), arc(n3, n4)],
[arc(n1, n2), arc(n2, n3), arc(n3, n7)],
[arc(n6, n2), arc(n2, n3), arc(n3, n4)],
[arc(n6, n2), arc(n2, n3), arc(n3, n7)]].

?- allPaths(3,List).
List = [[arc(n1, n2), arc(n2, n3)], [arc(n1, n2), arc(n2, n5)],
[arc(n2, n3), arc(n3, n4)], [arc(n2, n3), arc(n3, n7)],
[arc(n6, n2), arc(n2, n3)], [arc(n6, n2), arc(n2, n5)]].

?- allPaths(2,List).
List = [[arc(n1, n2)], [arc(n2, n3)], [arc(n2, n5)], [arc(n3, n4)],
[arc(n3, n7)], [arc(n6, n2)]].
```

Hint: Use `path` from a). `allPaths` should, for all possible pairs of start and end nodes, collect the set of paths that have the requested number of nodes, which is always one more than the number of arcs. Note that the start and end nodes should not be part of the result.

A List of predefined Haskell functions and operators

NB! If `$` is a binary operator, `($)` is the corresponding two-argument function. If `f` is a two-argument function, `‘f‘` is the corresponding binary infix operator. Examples:

```
          [1,2,3] ++ [4,5,6] ⟺ (++) [1,2,3] [4,5,6]
map (\x -> x+1) [1,2,3] ⟺ (\x -> x+1) ‘map‘ [1,2,3]
```

A.1 Arithmetics and mathematics in general

```
For integers: +  -  *  div  mod  ^
              abs, negate

For floats:   +  -  *  /  **
              cos, acos, sin, asin, tan, atan, abs, negate,
              exp, log, ceiling, floor, round, fromInt, sqrt
```

A.2 Relational and logical

```
(==), (!=)           :: Eq t => t -> t -> Bool
(<), (<=), (>), (>)  :: Ord t => t -> t -> Bool
(&&), (//)           :: Bool -> Bool -> Bool
not                  :: Bool -> Bool
```

A.3 List processing (from the course book)

```
(:)         :: a -> [a] -> [a]         1 : [2,3] = [1,2,3]
(++)        :: [a] -> [a] -> [a]       [2,4] ++ [3,5] = [2,4,3,5]
(!!)        :: [a] -> Int -> a         (!!) 2 (7:4:9:[]) = 9
concat      :: [[a]] -> [a]            concat [[1],[2,3],[],[4]] = [1,2,3,4]
length      :: [a] -> Int              length [0,-1,1,0] = 4
head, last  :: [a] -> a                head [1.4, 2.5, 3.6] = 1.4
                                       last [1.4, 2.5, 3.6] = 3.6
tail, init  :: [a] -> [a]              tail (7:8:9:[]) = [8,9]
                                       init [1,2,3] = [1,2]
reverse     :: [a] -> [a]              reverse [1,2,3] = 3:2:1:[]
replicate   :: Int -> a -> [a]         replicate 3 'a' = "aaa"
take, drop  :: Int -> [a] -> [a]       take 2 [1,2,3] = [1,2]
                                       drop 2 [1,2,3] = [3]
zip         :: [a] -> [b] -> [(a,b)]   zip [1,2] [3,4] = [(1,3),(2,4)]
unzip       :: [(a,b)] -> ([a],[b])    unzip [(1,3),(2,4)] = ([1,2],[3,4])
and, or     :: [Bool] -> Bool          and [True,True,False] = False
                                       or [True,True,False] = True
```

A.4 General higher-order functions, operators, etc

```
(.) :: (b -> c) -> (a -> b) -> (a -> c)          (Function composition)
map :: (a -> b) -> [a] -> [b]
filter :: (a -> Bool) -> [a] -> [a]
foldr :: (a -> b -> b) -> b -> [a] -> b
curry :: ((a,b) -> c) -> a -> b -> c
uncurry :: (a -> b -> c) -> ((a,b) -> c)
fst :: (a,b) -> a
snd :: (a,b) -> b
```

B List of predefined Prolog functions and operators

B.1 Mathematical operators

Parentheses and common arithmetic operators like `+`, `-`, `*`, and `/`.

B.2 List processing functions (with implementations)

`length(L,N)`	returns the length of `L` as the integer `N` `length([],0).` `length([H|T],N) :- length(T,N1), N is 1 + N1.`
`member(X,L)`	checks if `X` is a member of `L` `member(X,[X|_]).` `member(X,[_|Rest]):- member(X,Rest).`
`conc(L1,L2,L)`	concatenates `L1` and `L2` yielding `L` ("if") `conc([],L,L).` `conc([X|L1],L2,[X|L3]):- conc(L1,L2,L3).`
`del(X,L1,L)`	deletes `X` from `L1` yielding `L` `del(X,[X|L],L).` `del(X,[A|L],[A|L1]):- del(X,L,L1).`
`insert(X,L1,L)`	inserts `X` into `L1` yielding `L` `insert(X,List,BL):- del(X,BL,List).`

B.3 Procedures to collect all solutions

`findall(Template,Goal,Result)`	finds and always returns solutions as a list
`bagof(Template,Goal,Result)`	finds and returns all solutions as a list, and fails if there are no solutions
`setof(Template,Goal,Result)`	finds and returns *unique* solutions as a list, and fails if there are no solutions

B.4 Relational and logic operators

`<`, `>`, `>=`, `=<`	relational operations
`=`	unification (doesn't evaluate)
`\=`	true if unification fails
`==`	identity
`\==`	identity predicate negation
`=:=`	arithmetic equality predicate
`=\=`	arithmetic equality negation
`is`	variable on left is unbound, variables on right have been instantiated.

B.5 Other operators

`!`	cut
`\+`	negation
`->`	conditional ("if")
`;`	"or" between subgoals
`,`	"and" between subgoals

Existing suggested solutions (D7012E, 2018-04-25 and 2018-05-28)

Håkan's solutions etc to the functional programming problems

```
----------------------------------------------------------------------

-- Solutions (suggestions) D7012E Deklarative languages
-- Date: 180425+180528
-- Part 1: Functional programming
--
-- Håkan Jonsson


-- 2018-04-25:1a

-- *Main> :type head "8"
-- head "8" :: Char
-- *Main> :type []:[]
-- []:[] :: [[a]]
-- *Main> :type length.map (+1)
-- length.map (+1) :: Num a => [a] -> Int
-- *Main> :type \f -> "f" 0

-- <interactive>:1:7: error:
--     • Couldn't match expected type ‘Integer -> t’
--                   with actual type ‘[Char]’
--     • The function ‘"f"’ is applied to one argument,
--       but its type ‘[Char]’ has none
--       In the expression: "f" 0
--       In the expression: \ f -> "f" 0
-- *Main> :type map filter
-- map filter :: [a -> Bool] -> [[a] -> [a]]
-- *Main>


-- 2018-04-25:1b

ack 0 n = n+1
ack m 0 = ack (m-1) 1
ack m n = ack (m-1) (ack m (n-1))

-- *Main> ack 2 0
-- 3
-- *Main> ack 0 2
-- 3
-- *Main> ack 1 1
-- 3
-- *Main>


-- 2018-04-25:2a

isOrdered :: Ord t => [t] -> Bool
isOrdered [] = True
isOrdered [_] = True
isOrdered (x:y:ls) = x <= y && isOrdered (y:ls)

-- *Main> isOrdered [1,2,3,4,6,6,6,7,7,8,8,9]
-- True
-- *Main> isOrdered [1,2,3,100,4,6,6,6,400,7,7,8,8,9]
-- False
-- *Main> isOrdered []
-- True
-- *Main> isOrdered [1.2,3.4,5.6]
-- True
-- *Main> isOrdered [20000000]
-- True
-- *Main>


-- 2018-04-25:2b

compress :: [Int] -> [(Int,Int)]
compress [] = []
compress [x] = [(x,x)]
compress (x:xs) = if fst n-x<=1 then (x,snd n) : ns else (x,x):n:ns
   where (n:ns) = compress xs

compress2 :: [Int] -> [(Int,Int)]
compress2  = foldr (\x ra -> case ra of [] -> (x,x):[] ; (n:ns) ->if fst n-x<=1 then (

-- *Main> compress [1,2,2,3,5,6,9,1,12]
-- [(1,3),(5,6),(9,1),(12,12)]
-- *Main> compress2 [1,2,2,3,5,6,9,1,12]
-- [(1,3),(5,6),(9,1),(12,12)]
-- *Main>


-- 2018-04-25:2c

-- [f, s ..] = [f, s, f + 2*(s-f), f + 3*(s-f) ,f + 4*(s-f),...]

mkList :: Num t => t -> t -> [t]
mkList f s = f : mkList (f+d) (s+d)
  where d = s-f

-- *Main> take 10 (mkList 1 1)
-- [1,-4,-9,-14,-19,-24,-29,-34,-39,-44]
-- *Main> take 10 (mkList 1 4)
-- [1,1,1,1,1,1,1,1,1,1]
-- *Main> take 10 (mkList 1 4)
-- [1,4,7,10,13,16,19,22,25,28]
-- *Main>


-- 2018-04-25:3a

data Tree a =
    Leaf
  | Node (a,Int) (Tree a)  (Tree a) (Tree a)
  deriving (Eq, Show)

t = Node ('X',1)
          (Node ('H',2)
                Leaf
                Leaf
                Leaf
          )
          Leaf
          (Node ('Z',4)
                (Node ('Y',3)
                      Leaf
                      Leaf
                      Leaf
                     )
                (Node ('J',2)
                      Leaf
                      Leaf
                      Leaf
                     )
                Leaf
          )



-- 2018-04-25:3b

extract :: Int -> Tree a -> [a]
extract _ Leaf = []
extract i (Node p t1 t2 t3)
  | i == snd p = fst p : ra
  | otherwise  = ra
  where ra = extract i t1 ++ extract i t2 ++ extract i t3

-- *Main> extract 2 t
-- "HJ"
-- *Main>


-- 2018-04-25:4

game :: IO ()
game = do putStr "Welcome to the Game!\n"
          putStr "What secret number between 1 and 100 am I thinking of?!\n"
          -- should do the following, but it is in practice complicated
          -- to make this work in the Haskell platform
          --
          -- secret <- getRnd
          -- game' secret 1
          game' 20 1
   where
         getInt :: IO Int
         getInt =    do line <- getLine
                        return (read line)
         game' s n = do putStr ("Enter guess number " ++ show n ++ ": ")
```

```
                    guess <- getInt
                    if guess > s
                    then do putStr "Too high!\n"
                            game' s (n+1)
                    else if guess < s
                    then do putStr "Too low!\n"
                            game' s (n+1)
                    else do putStr ("Correct after " ++ show n ++ " guesses!\n")
                            putStr "Thank's for playing!\n"
                            return ()

-- *Main> game
-- Welcome to the Game!
-- What secret number between 1 and 100 am I thinking of?!
-- Enter guess number 1: 50
-- Too high!
-- Enter guess number 2: 25
-- Too high!
-- Enter guess number 3: 12
-- Too low!
-- Enter guess number 4: 18
-- Too low!
-- Enter guess number 5: 21
-- Too high!
-- Enter guess number 6: 20
-- Correct after 6 guesses!
-- Thank's for playing!
-- *Main>
```

Some student solutions etc

```
---------------------------------------------------------------------

-- Solutions (suggestions) D7012E Deklarative languages
-- Date: 180425+180528
-- Part 1: Functional programming
--
-- Some student solutions and tests done during the grading

c [] = []
c (h:t) = (h,head end) : c (tail end)
 where
  end = fe (h:t)

fe (h:[]) = [h]
fe (h:s:t)
 | s-h <= 1 = fe (s:t)
 | s-h > 1 = h:s:t
```

Håkan's solutions etc to the logic programming problems

```
%--------------------------------------------------------------------

%- Solutions (suggestions) D7012E Deklarative languages
%- Date: 180425+180528
%- Part 2: Logic programming
%-
%- Håkan Jonsson

%% the graph in the figure

arc(n1,n2).
arc(n6,n2).
arc(n2,n5).
arc(n2,n3).
arc(n3,n7).
arc(n3,n4).

% arc(n6,n3).


%% 180528: 1

% a)

fib(0,0) :- !.
fib(1,1) :- !.
fib(N,R) :-
    N3 is N-1,
    N4 is N-2,
    fib(N3,R3),
    fib(N4,R4),
    R is R3+R4.

% b)

fibs(N,L) :-
    N >= 0,
    fib(N,FN),
    N2 is N-1,
    fibs(N2,List),
    L2 = [FN | List],
    sort(0, @=<, L2, L), !.
fibs(_,[]) :- !.

% second version with findall
list(0,[0]) :- !.
list(N,R2) :-
    N2 is N-1,
    list(N2,R),
    append(R,[N],R2).

fibs2(N,List) :- findall(X,(list(N,NumL),member(Y,NumL),fib(Y,X)),List).


%% 180528: 2

a) A and B and C

b) A and B and C   or   not A and B
```

```
c) A and B and C   or   not(A and B) and C

d) A and B and C   or   not A and B   or   not C and B (ignore not B and B, that is fal


%% 180528: 3

% a)

path(A,B,[arc(A,B)]) :- arc(A,B).%,!.
path(A,B,[arc(A,X) | R]) :-
    arc(A,X),
    path(X,B,R).%,!.

% b)

add(arc(A,A)) :- !,fail. % arcs that start and end at the same node are not allowed
add(arc(A,B)) :- path(B,A,_), !,fail. % arc would cause a cycle
add(arc(A,B)) :- asserta(arc(A,B)).

add2(arc(A,B)) :-
    asserta(arc(A
% c)

allPaths(N,L) :-
    N2 is N-1,
    setof(X, A^B^(path(A,B,X),length(X,N2)), L).
```

Some student solutions etc

```
%------------------------------------------------------------------

%- Solutions (suggestions) D7012E Deklarative languages
%- Date: 180425+180528
%- Part 2: Logic programming
%-
%- Some student solutions and tests done during the grading

%% the graph in the figure

arc(n1,n2).
arc(n6,n2).
arc(n2,n5).
arc(n2,n3).
arc(n3,n7).
arc(n3,n4).

% arc(n6,n3).


%% 180528: 1

% a)
```

```
fib(0,0) :- !.
fib(1,1) :- !.
fib(N,R) :-
    N3 is N-1,
    N4 is N-2,
    fib(N3,R3),
    fib(N4,R4),
    R is R3+R4.


fib2(0,0).
fib2(1,1).
fib2(N,FN) :-
    N>1, fib(N-1,FN1), fib(N-2,FN2),FN is FN1+FN2.

% b)

fibs(N,L) :-
    N >= 0,
    fib(N,FN),
    N2 is N-1,
    fibs(N2,List),
    L2 = [FN | List],
    sort(0, @=<, L2, L), !.
fibs(_,[]) :- !.


fibs2(N,L) :-
    N1 =< N, N1 >= 0,
    findall(FN, fib(N1,FN),L2),
    sort(0, @=<, L2, L), !.

fibs3H(0,[0]).
fibs3H(N,[FN|L2]) :-
    N1 is N-1,
    fibs(N1,L2),
    fib(N,FN).
fibs3(N,L) :-
    fibs3H(N,FL),
    sort(0, @=<, FL, L).

fibs4(0,[R]) :- fib(0,R).
fibs4(N,L) :-
    fib(N,F), N1 is N-1, fibs(N1,R),append(R,[F],L),!.

% fibs5 is wrong, stack overflow
%% fibs5(0,[0|L]).
%% fibs5(N,L) :-
%%     fib(N,FN), N1 is N-1, fibs5(N1,[FN|L]).

% second version with findall
list(0,[0]) :- !.
list(N,R2) :-
    N2 is N-1,
    list(N2,R),
    append(R,[N],R2).

fibsH2(N,List) :- findall(X,(list(N,NumL),member(Y,NumL),fib(Y,X)),List).

%% 180528: 2

%% a) A and B and C


%% b) A and B and C   or   not A and B


%% c) A and B and C   or   not(A and B) and C
%%    == C

%% d) A and B and C   or   not A and B   or   not C and B (ignore not B and B, that is
%%    == B

%% 180528: 3

% a)

path(A,B,[arc(A,B)]) :- arc(A,B).%,!.
path(A,B,[arc(A,X) | R]) :-
    arc(A,X),
    path(X,B,R).%,!.

path2(A,A,[]).
path2(A,B,[arc(A,N)|T]) :-
    arc(A,N),path2(N,B,T).


path3(N,N,[]).
path3(A,B,[X|L]) :-
    arc(A,N),
    X = arc(A,N),
    path3(N,B,L).

% b)

add(arc(A,A)) :- !,fail. % arcs that start and end at the same node are not allowed
add(arc(A,B)) :- path(B,A,_), !,fail. % arc would cause a cycle
add(arc(A,B)) :- asserta(arc(A,B)).

% c)

allPaths(N,L) :-
    N2 is N-1,
    setof(X, A^B^(path(A,B,X),length(X,N2)), L).

% a3, reports every path twice

a3(N,L) :-
    setof(P,(arc(S,_),arc(_,E),path(S,E,P),N1 is N-1,length(P,N1)),L).
```

```
% a2, working alternative below

all2(A,L,L) :- member(A,L), !.
all2(A,L,[A|L]) :- not(member(A,L)).

nodes2(P,N) :- nodes2(P,[],N).

nodes2([],N,N).
nodes2([arc(A,B)|Rest],Used,Res) :-
    all2(A,Used,L1),
    all2(B,L1,L2),
    nodes2(Rest,L2,Res).

a2(N,L) :- findall(P,(path(_,_,P),nodes2(P,Nodes),length(Nodes,N)),L).


% a4, wrong
%% a4(N,L) :- findall(Path,getPath4(N,Path),L).
%% getPath4(0,[]).
%% getPath4(1,[arc(X,Y)]).
%% getPath4(N,P) :-
%%     N is N1-1, getPath4(N1, [arc(X,Y)|Pprev]),P=[arc(S,X),arc(X,Y)|Pprev].

% a5,

a5(0,[]).
a5(N,Out) :- findall(L,path(A,B,L),S),!,
     N1 is N-1,
     a5help(S,N1,Out).
a5(_,_) :- fail.

a5help([],_,[]).
a5help([X|Xs],N,[X|Ys]) :-
    length(X,R), R =:= N,
    a5help(Xs,N,Ys).
a5help([_|Xs],N,Ys) :-
    a5help(Xs,N,Ys).

%a6

a6(N,L) :- N1 is N-1, setof(P,a6help(N1,P),L),!.

a6help(N,P) :- length(P,N), path(_,_,P).

% a7
a7(N,L) :-
    N > 1,
    findall(P, path(A,B,P), Paths), a7help(N-1,Paths,L).
a7help(_,[],[]).
a7help(N, [P|T],L) :-
    a7help(N,T,L1), length(P,Len),
    (N =:= Len -> append(L1,[P],L);L=L1).

% a8
```

```
a8(N,L) :- N1 is N-1, findall(P,a8help(_,_,P,N1),L),!.

a8help(A,B,P,L) :-  path(A,B,P), length(P,L).

% a9

a9(N,L) :- A is N-1, length(P,A), findall(P,path(_,_,P),L).

% a10

a10(N,L) :- A is N-1, findall(P,(path(_,_,P),length(P,A)),L).

% helper function used at the grading
pr([]).
pr([H|T]) :- write(H),write('  '),length(H,N),write(N),nl,pr(T).
```