Exam in
Declarative Languages

Course code:	D7012E
Time:	4 hours, 09:00-13:00
Number of assignments:	6
Total number of points:	32.5
Date of exam:	2015-05-30
Teacher:	Fredrik Bengtsson, tel. 0920492431, 0738166670
Allowed aiding equipment:	Dictionary

Don't forget the course evaluation – hand in in a separate bin!

Good luck!

Assignment 1 (7p)

a (2p): Declare a data type `ETree`, in Haskell that are able to represent a special type of expressions. Our expressions are much like arithmetic expressions, but contains additions, unary minus (a single minus sign in front of an expression) and numbers. The data type should represent the expression in a tree-like fashion (sometimes called parse-tree or abstract syntax tree). The operations should be called `Add`, `Minus`, `Number` in the tree. Observe that `Add` is a binary operation and `Minus`, as declared here is a unary operation.

b (2p): Declare a function, `eval`, that takes an `ETree` as argument and evaluates the tree to an integer, according to the operations as stated above.

c (3p): Declare a function, transform, that takes an `ETree` as argument and returns another `Etree` as result. The function should transforms the tree by replacing all instances of `Number` representing numbers less than 0 by a unary minus and the same number, but with positive sign. The new tree hereby represents the same numerical value when evaluated, but contains only positive numbers.

Assignment 2 (6p)

a (3p): Write a predicate in prolog, `subset(S, Sub)`, that binds a subset `Sub`, to a set `S`, both represented by lists (order is not important, since we consider sets). All subsets should be considered via backtracking.

```
2 ?- subset([5,2],Sub).
Sub = [5, 2] ;
Sub = [5] ;
Sub = [2] ;
Sub = [].
```

b (3p): Write a predicate, `numSubsetSum(S, Sum, Num)`, in prolog that computes the number, `Num`, of subsets whose numerical sum of the element equals `Sum`. Use `subset` from (a). You may use the built-in `sum(+L,-S)` and `length(L,N)` without declaring them (as well as other built-in predicates that does not solve the problem completely).

Assignment 3 (6p)

Consider the following standard Haskell-functions:

```
foldr :: (a -> b -> b) -> b -> [a] -> b
map :: (a -> b) -> [a] -> [b]
replicate :: Int -> a -> [a]
id :: a -> a
(.) :: (a -> b) -> (c -> a) -> c -> b
(*) :: Num a => a -> a -> a
```

a (3p), Implement the function `map` with the help of `foldr`

b (3p), Implement a function

```
pow :: Num a => a -> Int -> a
```

that computes `a` to the power of `b`, where `a` is the first argument and `b` is the second argument. You may use the functions and operators declared above. You may not use direct recursion in your solution (use a higher-order function).

Assignment 4 (5p)

In haskell, write a simple calculator program with history function. The program should ask for the first operand (integer), the operation (addition or subtraction, denoted by "+" and "-", respectively) and the second operand (integer). The result of the operation should be presented and a question to exit the program (program exists if answer is "e"). If not exiting, the program should start over and ask again. Before asking for the first operand, the history of results should be presented as a list with the latest result first.

Example:

```
Main> calc
History: []
Ange operand 1: 5
Ange operation: +
Ange operand 2: 3
Result is:      8
Exit (e)?
History: [8]
Ange operand 1: 1
Ange operation: -
Ange operand 2: 4
Result is:      -3
Exit (e)?e

Main>
```

Assignment 5 (5.5p)

Consider predicate `close(L1, L2, N)`, for computing the number, `N`, of differences between two lists, `L1` and `L2`:

```
close([],[],0).
close([H|T1], [H|T2], N):-
  close(T1, T2, N).
close([_|T1], [_|T2], N):-
  close(T1, T2, N1),
  N is N1+1.
```

a (2p): When backtracking the predicate `close`, `N` will increase, for each new solution. Only the first solution will be correct. Modify the code by placing a cut somewhere, so that only the correct solution is retained.

b (2p): Modify the code as in (a), but without using cut, in order to achieve the same result.

c (1.5p): The following predicate, `gendifflist(-L1, -L2, +N, +K)`, will generate two lists, `L1` and `L2`, of length `N`, with `K` differences between the lists. All lists will be generated through backtracking (`length(L,N)` binds `L` to a list of length `N`):

```
gendifflist(L1, L2, N, K):-
  length(L1,N),
  length(L2,N),
  close(L1,L2,K).
```

For each of the three implementations of `close`, state if `gendifflist` will work correctly using that implementation. Answer yes/no for 1-3 below:

1: Original implementation of `close` (as declared above)
2: Implementation of `close` from (a)
3: Implementation of `close` from (b)

Assignment 6 (3p)

What is the logical equivalent of the following programs:

a (1,5p)

```
p :- a,b ; c,d.
p :- e,f.
```

b (1,5p)

```
p :- a,!,b ; c,d.
p :- e,f.
```

State a logical (boolean) expression equivalent to `p`.