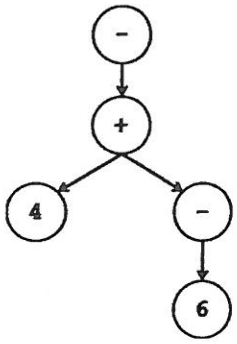
Exam in
Declarative Languages

Course code:	D7012E
Time:	4 hours, 9:00-13:00
Number of assignments:	5
Total number of points:	31
Date of exam:	2012-08-22
Teacher:	Fredrik Bengtsson, tel. 0920492431, 0738166670
Allowed aiding equipment:	Dictionary
Result announced:	2012-09-12

Good luck!

Assignment 1: Data structure (14p)

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`, and `Number` in the tree. An example of such an expression is "-(4+(-6)) and its corresponding tree would look like the tree below:



b (1p): State, using your data type from (a), the haskell expression that would correspond to the above tree.

c (2p): Declare a function that takes an `ETree` as argument, evaluates the expression represented by the `ETree` and returns the result of the evaluation.

d (3p): Declare a data type `CTree`, in Haskell, that are able to represent expressions containing general unary and binary operations. Unary operations should be called UnOp and be represented by a function taking one argument. The type of the argument should be a parameter of type `CTree`. Binary operations should, in the same way, be represented by a function taking two arguments. We assume the same argument and result types for both unary and binary operations.

e (4p): declare a function `conv`, that converts an expression represented by the data type from (a) into an expression represented by the data type from (d). Insert appropriate functions in the data tree in order to represent the operations from (a).

f (2p): declare a function `Ceval`, that performs the same operation as in (c), but uses the function from (e) to do so.

Assignment 2 (3p):

What is the logical equivalent of the following programs:

a:
```
p :- a,b.
p :- c.
```

b:
```
p :- a,!,b.
p :- c.
```

c:
```
p :- c.
p :- a,!,b.
```

State a logical (boolean) expression equivalent to p.

Assignment 3 (6p):

a: In prolog, declare a predicate `count(+E,+L,-N)`, that counts the number of occurrences of the element `E` in the list `L` and binds the result to `N`. Make sure the predicate always binds the correct number of elements to `N`, even when backtracking over the predicate. You are allowed to write a helper function that performs the actual work, but you are not allowed to use built-in predicates that performs the same or similar operations.

b: Using a correct implementation of count, what would happen if you call

```
count(A, [a,b,a,a,c,d,a], N).
```

Show what would be printed on screen, including attempts to perform backtracking until "false". (exact what characters are printed is not important, but you need to show what names are bound and to what value they are bound).

Assignment 4 (3p):

Implement a user interface for generating palindromes in haskell. The interface should prompt the user for a string, where after the string should be concatenated with a reversed version of itself and displayed (example: we enter "natur", the program displays "naturrutan"). After this, the user interface should prompt for a new string. The program exits when the user enters an empty string.

Assignment 5 (5p):

We are faced with the problem of constructing support for a building where one of four corner-pillars have disintegrated (rämnat!). We want to replace the disintegrated pillar with one of the same height. However; we only have a bunch of already-built blocks of different thickness which we are about to combine in order to get a combined thickness as close as possible to the height of the disintegrated pillar.

In prolog, define a predicate

```
combinepads(+PadList, +PillarHeight, -ResultList)
```

where `PadList` is a list of values of the thickness of each block (one value for each pad) and `PillarHeight` is a value bound to the height of the disintegrated pillar. The predicate should compute the best possible combination of blocks such that their combined thickness is as close as possible to the height of the pillar. The thickness of the blocks chosen should be in `ResultList`.