Exam in
Declarative Languages

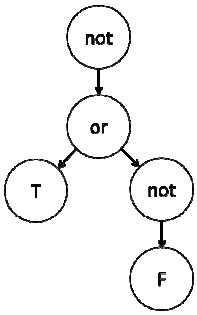
Course code:	D7012E
Time:	4 hours, 9:00-13:00
Number of assignments:	6
Total number of points:	36
Date of exam:	2013-06-01
Teacher:	Luleå: Fredrik Bengtsson, tel. 0920492431, 0738166670 Skellefteå: Johan Nordlander tel. 0706207461
Allowed aiding equipment:	Dictionary

Remember to hand in the course evaluations in a separate bin!

Good luck!

Assignment 1 (6p)

a (2p): Declare a data type `BTree` in Haskell that is able to represent logical formulae. The formulae should contain the binary operations "*and*" and "*or*", the unary operation "*not*", as well as the constant truth-values "*T*" and "*F*". The data type should represent the formulae in a tree-like fashion (sometimes called parse-tree or abstract syntax tree). The constructors corresponding to the formula alternatives should be called *And, Or, Not, T* and *F*. An example of such an expression is *"not (T or (not F))"*, and the corresponding syntax-tree would look like below:



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

c (3p): Declare a function `beval` that takes a `BTree` as argument, evaluates the logical formula represented by the `BTree`, and returns its truth-value.

Assignment 2 (6p)

a, Assume that you have access to a Haskell function `getFloat :: String -> IO Float`, whose behavior is to print the given string on the screen as a prompt, read user input, and convert the entered data into a floating-point value. The following code attempts to use `getFloat` to compute the diagonal of a user-specified rectangle.

```
diagonal = sqrt (xdiff*xdiff + ydiff*ydiff)
  where
    xdiff = getFloat "Enter x1: " - getFloat "Enter x2: "
    ydiff = getFloat "Enter y1: " - getFloat "Enter y2: "
```

The code isn't accepted by the Haskell type-checker, however. Explain why and rewrite the definition so that it both type-checks and computes the intended diagonal.

b, Now suppose you'd like to "improve the looks" of your code by removing some of the repeated occurrences of `getFloat`. One idea that might come up is to rewrite the code according to the following pattern (and accept slightly less informative prompts):

```
diagonal = …
  where
    getX = getFloat "Enter an x: "
    getY = getFloat "Enter a y: "
```

Now the question is: is this at all possible in Haskell? If you answer no, explain why. If you answer yes, show what your solution from a) above would look like if rewritten to this form. (Be aware that any mentioning of input facilities besides `getX` or `getY` in the missing part is not considered to be of the desired form!)

Assignment 3 (6p)

Consider the following predicate, written in prolog:

```
findMin([X|Xs],Res) :- findMin_([X|Xs], X, Res).

findMin_([], Result, Result).
findMin_([X|Xs], Acc, Result) :-
  X<Acc,
  findMin_(Xs, X, Result).
findMin_([_|Xs], Acc, Result) :-
  findMin_(Xs, Acc, Result).
```

The predicate finds the smallest element in a list and binds the result to `Res`. Now, consider the goal `findMin([2,5,7,1], N).`

a, What will happen when backtracking over this goal? Show what values will be bound to `N` upon repeated backtracking over the predicate until false. Explain what happens (how the backtracking affects which clauses of the predicate that are executed).
b, Modify the code, by placing cut(s) in the code, in such a way that the smallest element will always be returned, regardless of backtracking.
c, Can the code be modified to achieve the same behavior as in (b), but without placing a cut? If yes, show the modified code. If no, explain why.

Assignment 4 (6p)

The following Haskell program uses type classes to achieve overloading of the names compute and unit.

```
class MyClass a where
  compute :: a -> a -> Int
  unit :: a

instance MyClass Int where
  compute = (+)
  unit = 0

instance MyClass Char where
  compute _ 'y' = 100
  compute _ _ = 0
  unit = 'y'
```

a, Given these definitions, what would be the value of the following expression?

```
compute (length "x") (length "unit") + compute 'x' unit
```

b, Now suppose the program is extended with the following function:

```
fun x = compute (length "x") (length "unit") + compute x unit
```

What type will be inferred for fun?

Assignment 5 (6p)

Design a predicate, `cashExchange(ValueList, Sum, ResultList)`, that, in `ResultList`, provides a list of values such that the sum of all elements in `ResultList` is `Sum`. Legal values are present in `ValueList`. Each value from `ValueList` can be used as may times as appropriate. `ResultList` should have minimum length (as few values as possible). If no solution is possible `cashExchange` should fail. You may declare helper predicates as you like.

Assignment 6 (6p)

One feature of lazy languages like Haskell is the ability to define and compute with infinite data structures. Such structures can be dangerous, though, since careless use may lead to non-terminating programs. Below follows the definition of a Haskell list containing every third integer number, starting from zero.

```
every3 = enumBy 0 3
  where enumBy n m = n : enumBy (n+m) m
```

Now explain, for each of the following Haskell expressions, whether computing its result would be a terminating or non-terminating operation. Also show what the result is in the terminating cases.

a, `head (tail every3)`
b, `length every3 > length "abc"`
c, `length [every3, every3, every3]`
d, `zip "abc" every3`
e, `every3 == every3`

Note: functions `head`, `tail`, `length` and `zip` are the standard list operations from the Haskell Prelude. The definition of zip is also given below, for reference.

```
zip [] _ = []
zip _ [] = []
zip (x:xs) (y:ys) = (x,y) : zip xs ys
```