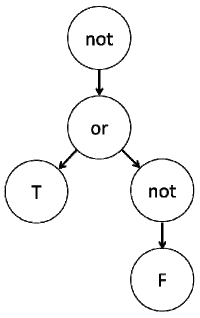
Exam in
Declarative Languages

Course code:	D7012E
Time:	4 hours, 9:00-13:00
Number of assignments:	6
Total number of points:	35
Date of exam:	2013-08-21
Teacher:	Fredrik Bengtsson, tel. 0920492431, 0738166670
Allowed aiding equipment:	Dictionary

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 `btransand` that takes a `BTree` as argument, transforms the formula to only contain *and*, *not*, *F* and *T* (removes *or*). This is accomplished by using de morgans law *a or b = not ((not a) and (not b))*. The transformed parse tree should be returned.

Assignment 2 (6p)

a, Assume that you have access to a simple database in the form of a Haskell function `getDBValueForKey :: String -> IO (Maybe Float)`. The behavior of this function is to search for a database entry with a key corresponding to the given argument string, and if such an entry exists, return it as a floating-point value. If the entry doesn't exist, `Nothing` is returned. The following code attempts to use the database to compute the difference between two specific entries, if they *both* exist:

```
diff = case (getDBValueForKey "AAA", getDBValueForKey "BBB") of
         (Just a, Just b) -> Just (a-b)
         _                -> Nothing
```

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 difference.

b, Now suppose you'd like to "improve the looks" of your code by introducing short names for the somewhat unwieldy database accesses. Here's an attempt:

```
diff = …
  where
    getA = getDBValueForKey "AAA"
    getB = getDBValueForKey "BBB"
```

Now the question is: would these definitions be usable at all in your corrected solution to a) above? If you answer no, explain why. If you answer yes, show what your solution would look like if rewritten to take advantage of the defined short names.

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 (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 x 'y' = 0
  compute _ _   = 1
  unit = 'y'
```

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

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

We are about to load a passenger ferry with cars. Each lane in the ferry has a specific lengt. We want to fill up each lane with cars, such that the total length of the cars is as close as possible to the length of the lane (but, of course, not longer).
You should implement, in prolog, a predicate

`loadcars(CarList, LaneLength, ResultList)`

where `CarList` is a list of values of the lengths of each car waiting (one value for each car) and `LaneLength` is a value bound to the length of the lane for cars on the ferry. The predicate should compute the best possible combination of cars such that the lane is filled to the largest extent possible. The length of the cars chosen should be in `ResultList`. The predicate should fail if it is not possible to load any cars on the ferry.

Assignment 6 (5p)

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 all powers of 2, starting from one.

```
pow2 = powersof 1 2
  where powersof n m = n : powersof (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 pow2)`
b, `head pow2 == 1 or 2 < length pow2`
c, `length [pow2@pow2, pow2, [5]]`
d, `take 4 (zip pow2 pow2)`
e, `pow2 == pow2`

Note: functions `head`, `tail`, `length`,`take` and `zip` are the standard list operations from the Haskell Prelude. The expression `take k l` returns the `k` first elements if list `l`. The definition of zip is also given below, for reference.

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