

# Programming Languages — CPSC 604, spring 2009

## Assignment 2

February 3, 2009

### General rules

Please return your answers via *CSNet*. Please submit five files named `Sets.hs`, `Relations.hs`, `TriTree.hs`, `NB.hs`, and `Main.hs`, as explained in the different exercises below. Late penalty is calculated in the same manner than for Assignment 1; deadline is announced in class and/or on the class web pages.

Enjoy!

### Assignment

1. Install GHC (or Hugs if you insist) and a suitable editing mode for your editor, or use the GHC installed in `linux.cs.tamu.edu`. Whatever your choice, get comfortable with writing, compiling, and interpreting Haskell programs.
2. Provide the implementations of the functions in the module below:

```
module Sets where

type Set a = [a]

isSubset :: Eq a => Set a -> Set a -> Bool
sameSet  :: Eq a => Set a -> Set a -> Bool
powerSet :: Set a -> Set (Set a)
union   :: Eq a => Set a -> Set a -> Set a
intersection :: Eq a => Set a -> Set a -> Set a
setProduct :: Set a -> Set b -> Set (a,b)
disjointUnion :: (Eq a, Eq b) => Set a -> Set b -> Set (Either a b)
setDifference :: Eq a => Set a -> Set a -> Set a
```

The data type `Either` is defined as:

```
data Either a b = Left a | Right b deriving (Eq, Ord, Read, Show)
```

and thus `disjointUnion` should work in the following way:

```
*Sets> disjointUnion [1, 2, 3] [1, 2]
[Left 1,Left 2,Left 3,Right 1,Right 2]
```

Note that here we simply use lists to represent sets. The list type allows duplicate elements, which sets should not allow. Your functions can, as their precondition, assume that their inputs are sets (no duplicate elements). Those functions that return sets must, as their postcondition, guarantee that the return value is a set (no duplicate elements).

3. Extend the following module to include the implementations of the functions:

```
module Relations where
import Sets

type Relation a b = Set (a,b)

composeRelations :: (Eq a, Eq b, Eq c) =>
    Relation a b -> Relation b c -> Relation a c
isReflexive :: Eq a => Set a -> Relation a a -> Bool
isSymmetric :: Eq a => Relation a a -> Bool
isTransitive :: Eq a => Relation a a -> Bool
isEquivalence :: Eq a => Set a -> Relation a a -> Bool
```

The following example definitions, and example invocations give guidance on how the functions should behave:

```
r1 = [(a,a) | a <- [0..10]]
r2 = [(a,b) | a <- [0..10], b <- [0..10], b > a]
r3 = [(a,a+1) | a <- [0..10]]
```

```
*Relations> isReflexive [1..10] r1
True
*Relations> isSymmetric r2
False
*Relations> isEquivalence [1..10] r1
True
*Relations> isSymmetric r3
False
```

Note that the first parameter of the `isReflexive` function is a `Set`. The function should examine whether the relation is reflexive for the set of elements specified with the first parameter.

4. Define an algebraic data type `TriTree a b` that represents a tree where each node can have 0, 1, 2, or 3 children. Additionally, each node holds a value. The values of the leaf nodes (nodes with zero children) have type `a`, the values in tree nodes (nodes with one or more children) have type `b`.

Make `TriTree a b` an instance of the `Show` type class (do not use `deriving`, rather write a separate instance declaration). Your instance declaration must be conditional, that is, it must require that the value types `a` and `b` are instances of `Show`. The formatting you provide does not have to be very fancy, but do something a bit more than what `deriving (Show)` would give.

Define a function `createExampleTree` that returns a value of type `TriTree Integer String`, that has at least one of each kind of nodes.

5. Define a data type `NBTerm` that can represent terms of our  $\mathcal{NB}$  language. `NBTerm` should be an instance of `Show`, and output of `show` should be something comprehensible. Define at least three terms `t1`, `t2`, and `t3` for testing. Define the functions `bigEval` and `smallEval` that implement the big-step and, respectively, small-step semantics we defined for  $\mathcal{NB}$ . Place your definitions into module `NB`, and in a file `NB.hs`.
6. Write a main function (in module `Main` that imports `NB`) that for each of your terms `t1-t3` (see exercise 5) prints out the original term and the terms evaluated with both evaluators. Make sure that the program can be compiled into a stand-alone program. E.g., you can use the command `ghc --make Main.hs -o main`