COMP1100: Lab 9

In this lab, we cover type classes and ad hoc polymorphism, and how we can use these concepts to gen eralise functions that require some assumptions about the input type. We also continue the topic of tree s from last lab, and introduce binary search trees, which are trees with a special ordering constraint that gives them a great advantage over binary trees in terms of computational efficiency.

> Don't forget to submit each exercise for marking to receive marks for attempting the exercises. You have until the start of your lab next week to do this.

Pre-Lab Checklist

- You have completed Lab 8, understand binary trees, and how to write recursive functions over them.
- You are comfortable with the concept of parametric polymorphism.
- You are comfortable with the recursive data type Nat from Lab 7.

Types

We've seen before in Lab 4 that some functions can be written to work on an arbitrary type, like leng th:

```
length :: [a] -> Integer
length list = case list of
  [] -> 0
  _:xs -> 1 + length xs
```

The actual values of the elements in the list do not concern us, we only want to know how many there are, which is why this function can be written in a way that's **parametrically polymorphic**, that is, it will work regardless of the type of the elements in the list.

Now we would like to generalise other functions. Consider the following function`sumInteger`:

```
sumInteger :: [Integer] -> Integer
sumInteger list = case list of
  [] -> 0
  x:xs -> x + sumInteger xs
```

It’ s possible to write the same function for types other than Integer, like Double.

```
sumDouble :: [Double] -> Double
sumDouble list = case list of
  [] -> 0
  x:xs -> x + sumDouble xs
```

Clearly this function cannot be made parametrically polymorphic, as the addition operator + isn’ t defi ned for all types. (We can’ t add together strings or booleans, for example.)
So we want a way to define the sum function to operate over all lists that contain types that “act like n umbers”, that is, types where addition is well defined. This is a weaker form of polymorphism, called **ad-hoc polymorphism**.
If we had just tried to make the type of sum as general as possible by changing the type signature to

```
sumEverything :: [a] -> a
```

then we get the error message

```
No instance for (Num a) arising from a use of `+'
```

This is a rather cryptic way of saying “The input type can be anything, but addition is only defined o n things that are numbers, that is, types that belong to the type class of Num.” To solve this problem, we need to introduce type classes.

Type Classes

Haskell handles **ad-hoc polymorphism** by using **type classes**, grouping types together into classes that all share similar properties, (they all act like numbers, they all have equality defined, etc.)
The main type classes you’ re likely to see in Haskell are as follows:

Type Class	(some of the) functions defined	Description	Depends on?
`Eq`	`(==)`, `(/=)`	Has equality defined	N/A
`Ord`	`<=`, `max`, `compare`	Has ordering defined	`Eq`
`Enum`	`succ`, `pred`, `toEnum`, `fromEnum`	Sequentially ordered types	N/A
`Bounded`	`minBound`, `maxBound`	Has minimal and maximal elements	N/A
`Num`	`(+)`, `(*)`, `(-)`, `abs`, `negate`, `signum`, `fromInteger`	Numbers, can perform arithmetic	N/A
`Show`	`show`	Can be printed as a string	N/A
`Fractional`	`(/)`	Fractional numbers, can be divided	`Num`
`Floating`	`sqrt`, `exp`, `log`, `sin`, `cos`, `pi`	Acts like real numbers, many mathematical functions defined	`Fractional`
`Real`	`toRational`	Another numeric types that resemble real numbers	`Num`, `Ord`
`Integral`	`div`, `mod`	Acts like integers	`Real`, `Enum`
`Foldable`	`foldl`, `foldr`	Type constructors like lists which can be folded	N/A

Some types depend on others, and the type classes form a hierarchy. If you ever want to know more ab out a type or typeclass, you can type in :info Name into GHCi, for example:

```
ghci> :info Eq
class Eq a where
  (==) :: a -> a -> Bool
  (/=) :: a -> a -> Bool
  {-# MINIMAL (==) | (/=) #-}
        -- Defined in `ghc-prim-0.8.0:GHC.Classes'
  instance Eq a => Eq [a] -- Defined in ‘GHC.Classes’
  instance Eq Word -- Defined in ‘GHC.Classes’
  instance Eq Ordering -- Defined in ‘GHC.Classes’
  instance Eq Int -- Defined in ‘GHC.Classes’
  instance Eq Float -- Defined in ‘GHC.Classes’
  instance Eq Double -- Defined in ‘GHC.Classes’
  instance Eq Char -- Defined in ‘GHC.Classes’
  instance Eq Bool -- Defined in ‘GHC.Classes’
```

...

Here, we can see that members of the type class Eq have the functions (==) and (/=)defined, but a mini mal definition requires only one of (==) or (\=). We can also see that a few familiar types, Double, Char, Bool, Int all have equality defined on them. Digging deeper, we find the interesting lines

```
instance Eq a => Eq [a] -- Defined in ‘GHC.Classes’
instance (Eq a, Eq b) => Eq (a, b) -- Defined in ‘GHC.Classes’
```

which tell us that if a type a has equality defined, then so too does the type [a]. This is **very** powerful, a s it saves on extra code that we don’ t need to write. As soon as you’ ve told haskell how to compare integers, it can automatically work out that [1,2] == [1,3]should evaluate to False.
The same goes for tuples: since Integer and Char both have equality defined, so does (Integer, Char) witho ut any further work.

Applications of Type Classes

Consider the following function:

```
sum :: Num a => [a] -> a
sum list = case list of
   [] -> 0
   x:xs -> x + sum xs
```

The notation

```
Num a => [a] -> a
```

means that the function has type [a] as input, and returns an element of type a as output, where a is any type from the type class Num a. So [Integer] -> Integer, or [Double] -> Double are special cases of Num a => [a] -> a, but [Double] -> Integer or [Bool] -> Bool is not.

If we want a type to be a member of two or more type classes, we can enforce this.

```
areTheyEqual :: (Eq a, Show a) => a -> a -> String
areTheyEqual x y
  |x == y = (show x) ++ " is equal to " ++ (show y)
  |otherwise = (show x) ++ " is not equal to " ++ (show y)
```

Here, we are enforcing that the input type has equality defined, and can be turned into a string. Sometimes specifying two type classes is redundant, e.g. (Ord a, Eq a) => ...is the same as Ord a => ... as Ord is a subclass of Eq.

Type exercises

Exercise 1 (Optional)

This exercise is optional, and is here just to help you further get your head around type classes. If you feel confident, feel free to skip ahead to Exercise 2.

See if you can work out what type classes the following types belong to. You might like to experiment and try out some operations in GHCi (like trying to compare two elements with <=, or trying to add two elements together) to see what is defined.

```
Integer
Double
Char
Bool
(Integer, Integer)
(Integer, Double)
String
[Integer]
Integer -> Integer

Maybe Integer
```

Exercise 2

Try and work out the type of the following functions. Try to make the type as general as possible.

You can check your answers by asking GHCi what it had inferred the type to be, by entering :type functi

onName.

```
-- cow :: ???
cow x y z = x && (y `elem` z)

-- foo :: ???
foo x y z = x `elem` y && y `elem` z

-- bar :: ???
bar x y = case y of
   Nothing -> x
   Just z  -> x + z

-- snap :: ???
snap x y
   |x > y = show x

   |otherwise = show y
```

Instances

When we create a new type, we would like to tell Haskell what type class our new type shall belong to.

We use the following code,

```
instance TypeClass MyNewType where
   ...
```

which breaks down as:

- instance is a reserved keyword indicating we are adding a new type to a type class
- TypeClass is the type class we want our new type to belong to
- MyNewType is the name of our new type

We then have to define all the functions that a member of that type class should have defined.

- For the Eq type class, we need only (==), (and we get (/=) for free.)
- For Ord, we need to already be in Eq, and then also define either (<=) or compare(and Haskell ca n work out all the other operators like (<), (>), (>=) for free.)

If you're interested, you can look at the documentation for the Prelude, and under each type class, the **minimal complete definition** tells you the fewest functions that you need to define, and then the rest are automatically derived for you.

For example, (see Fruit.hs) suppose we define a new type for fruit,

```
data Fruit = Apple | Banana | Orange
```

and we wanted to define equality on our fruit. We need to tell Haskell what (==) means for fruit.

```
instance Eq Fruit where
  Apple == Apple = True
  Banana == Banana = True
  Orange == Orange = True
  _ == _         = False
```

So we define equality in the obvious way. All fruits are equal to themselves, and any fruit is not equal to a different fruit.

> Note that there was nothing forcing me to define equality in this way, I could have defined Apple == Banana to be True but Banana == Apple to be False. This doesn't mean that we should, though. To do so would be **very** poor style, and go against the Haskell standard.

I can also define what it means to show a Fruit.

```
instance Show Fruit where
  show fruit = case fruit of
    Apple -> "An apple."
    Banana -> "A banana! Yum!"
    Orange -> "Yuck, an orange."
```

So when I type Banana into GHCi, a Banana is printed to the terminal using my custom definition for show.

```
> Banana
A banana! Yum!
```

I can also define ordering on fruit. I'm going to order fruit by how much I enjoy eating them: Banana being the best, Apple being okay, and Orange is the worst. (Deepest apologies to those that love oranges.)

If we give a full definition of the (<=) function:

```
instance Ord Fruit where
  (<=) Orange Apple = True
  (<=) Apple Banana = True
  (<=) Orange Banana = True

  (<=) Banana Apple = False
  (<=) Apple Orange = False
  (<=) Banana Orange = False

  (<=) Banana Banana = True
  (<=) Apple Apple = True
  (<=) Orange Orange = True
```

then Haskell can use that to define all the usual ordering operators:

```
> (Banana > Orange)
True
```

You'll note how tedious it is to write out all the cases. Surely if Orange <= Apple and Apple <= Banana, then Orange <= Banana, right?

Sadly, Haskell doesn't place any sensible constraints on the <= function we define. It's up to the user to define <= in such a way that the usual ordering properties you would expect are satisfied.

Deriving Type Classes

Rather than doing the hard work ourselves, we can ask Haskell to define a lot of these properties for us, using the deriving keyword.

```
data Fruit = Orange | Apple | Banana

  deriving (Ord, Show)
```

Haskell will assume the equality you wanted is the "obvious" implementation, that every fruit is equal to itself, and that no two different fruits are equal.

It will also place a ordering on fruit, on the order they appear in the definition of Fruit, so Orange is less than Apple, which is in turn less than Banana. The default definitions of equality and ordering Haskell pr

ovides satisfy all the nice properties you would expect them to. The default implementation of show wil l also print each fruit exactly as they are defined.

```
> show Banana
"Banana"
```

Type classes where you can ask Haskell to do the hard work (defining all the functions automatically) for you are called **derivable** classes. Not all type classes are derivable. If we tried to add arithmetic to our fruit by making it a member of the type class Num,

```
data Fruit = Orange | Apple | Banana
  deriving (Ord, Show, Num)
```

then Haskell will throw an error

```
Fruit.hs:4:14:
  Can't make a derived instance of `Num Fruit':
    `Num' is not a derivable class
```

and inform you that it doesn't know how to define arithmetic on Fruit. What should Apple + Banana b e? What about Banana * Orange? There's no obvious way to define it.

Exercise 3A

Recall our definition of Nat from Lab 7.

```
data Nat = Z | S Nat
  deriving Show
```

We'd like to define equality on natural numbers: Two arbitrary natural numbers are equal if they hav e the successor S applied the same number of times.

Using the template given, define the function (==) for the type Nat.

> Do **not** use deriving Eq here. You should define equality yourself.

```
> (S Z) == (S Z)
True
> (S Z) == (S (S Z))

False
```

Exercise 3B

It’ s possible to place an ordering on natural numbers as follows:

Z < (S Z) < S (S Z) < ...

So if we want to ensure Nat belongs to the type class Ord, we need only to define a definition of (<=), a nd Haskell can work out corresponding definitions for all the other operators defined on Ord: (<), (>), m ax, min, compare.

Using the template given, define the function (<=) for the type Nat.

> Do **not** use deriving Ord here. You should define equality yourself.

```
> Z >= (S Z)
False
> max (S Z) (S (S Z))

S (S Z)
```

Binary Search Trees

We saw in Lab 8 how we can write functions over binary trees, and over rose trees, but we haven’ t y et seen how efficient the tree data structure can be. You might have noticed that for the exercises in las t lab, you were forced to look through all the elements in a tree to check if something was there, or to f ind the maximum/minimum element, as you would have to do with a list. So far, trees appear to just be lists, but more annoying to deal with.

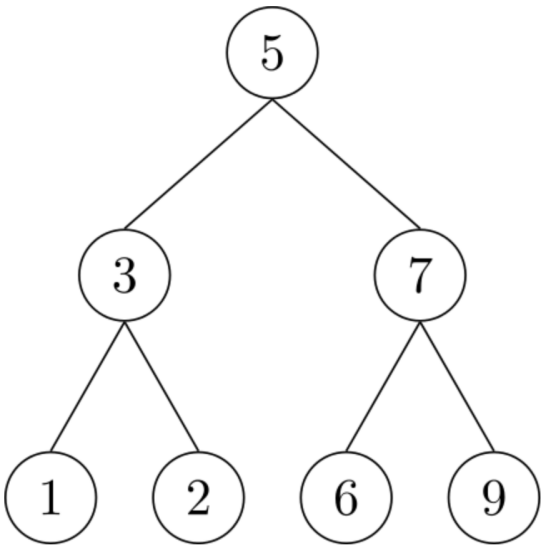
We need to add one extra property to provide trees with much more power than they currently have, a property we call the **binary search ordering constraint**.

A binary tree satisfies the **binary search ordering constraint**, if either:

- It is a Null tree
- Every element in the left hand subtree is strictly less than the root, and every element in the rig ht hand subtree is strictly more than the root. Furthermore, both subtrees also satisfy the binary search ordering constraint.

Binary trees that meet this constraint are called **binary search trees**.

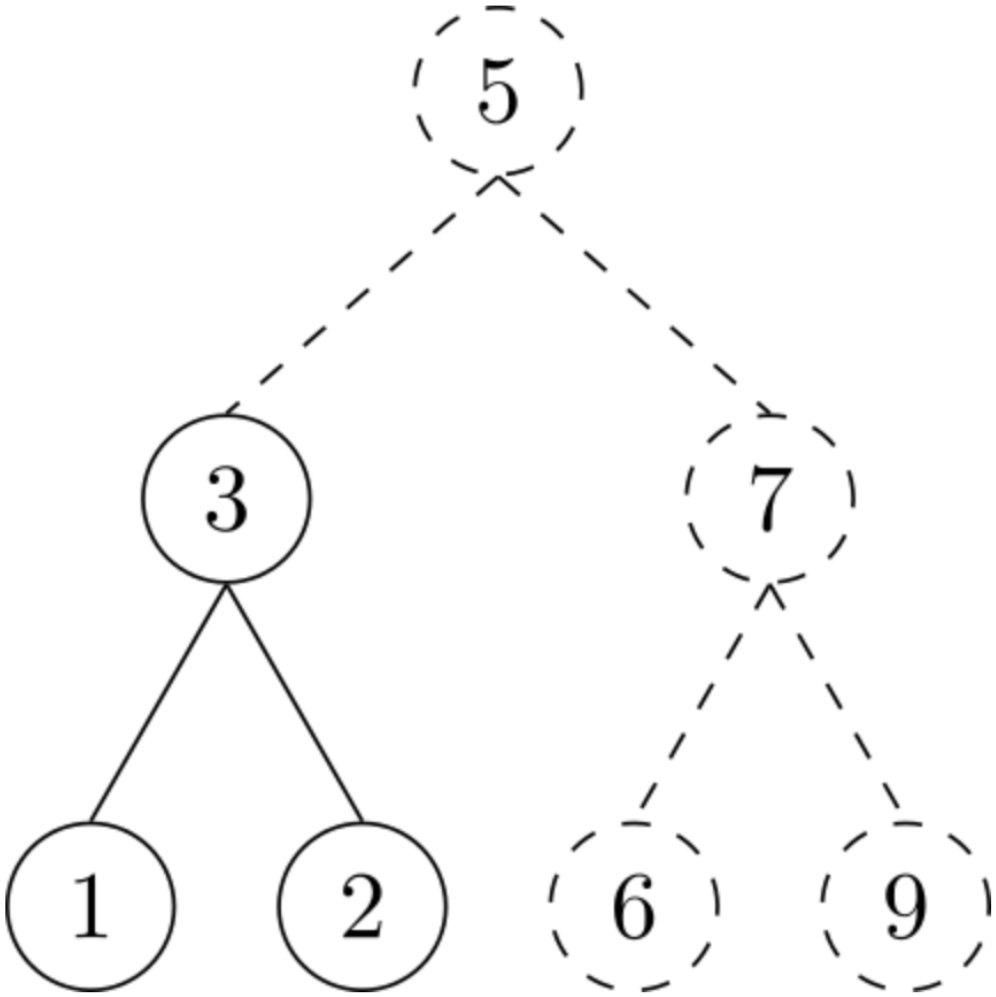
Note that it is very important for the subtrees to also pass the ordering constraint, given the following e xample tree. (This tree is included for you as notBStree.)



See how on the left subtree, it is true that all elements

{1,3,2}

{1,3,2} are less than the root node of 5. But if we consider the left subtree in isolation:



We can see that the left subtree does not satisfy the ordering constraint, as the root node 3, is bigger th an the biggest element in the right subtree, 2.

This means the original tree is not a binary search tree.

In Haskell, binary search trees are no different structurally from binary trees, and so we define them ap propriately

```
type BSTree a = BinaryTree a
```

while keeping in mind that any instance of a BSTree had better satisfy the ordering constraint, as ther e’ s no easy way to enforce that property in the type itself.

This also means the type a that a BSTree contains must be a member of the type class Ord.

This ordering property is what gives binary search trees a huge improvement in performance on lists. We can search through the tree, moving left if the root node is too large, and moving right if it' s too small.

Binary Search Tree exercises

Exercise 4

Rewrite the elemTree function from Lab 8 but this time you may assume that the input tree satisfies the binary search ordering constraint.

```
elemBSTree :: (Ord a) => a -> (BSTree a) -> Bool
```

Do not use the same function as before, you should be able to search more efficiently. As a reminder, t his function should take an element, and a binary search tree, and check if that element is present in th e tree.

```
> elemBSTree 5 goodTree
True
> elemBSTree 30 goodTree
```

False

Exercise 5

Rewrite the treeMaximum and treeMinimum functions, again assuming the input tree is a binary search tre e. Be efficient!

```
treeBSMax :: (Ord a) => BSTree a -> a
treeBSMin :: (Ord a) => BSTree a -> a
```

```
> treeBSMax goodTree
31
> treeBSMin goodTree
```

1

Exercise 6

Write a function isBSTree that takes a binary tree as input, and checks if the binary search constraint hol ds.

```
isBSTree :: (Ord a) => BinaryTree a -> Bool
```

```
> isBSTree goodTree
True
> isBSTree unbalancedTree
True
> isBSTree notBSTree

False
```

Exercise 7

Write a function treeInsert that takes a binary search tree, and an element, and inserts that element into t he tree, ensuring the binary search ordering constraint still holds.

```
treeInsert :: (Ord a) => BSTree a -> a -> BSTree a
```

(If the element is already in the tree, leave the tree unchanged.)

You may find the printTree function useful for this exercise, so that you can verify that the function inse rts a new element into the correct place.

```
> treeInsert smallTree 3
Node (Node Null 1 (Node Null 3 Null)) 5 (Node Null 10 Null)
> treeInsert smallTree 20
Node (Node Null 1 Null) 5 (Node Null 10 (Node Null 20 Null))
> treeInsert smallTree 1

Node (Node Null 1 Null) 5 (Node Null 10 Null)
```

Exercise 8

Write a function flattenTreeOrd that flattens a binary tree, but preserves the ordering.

(That is, when a binary search tree is flattened, the resulting list should be sorted.)

```
flattenTreeOrd :: BinaryTree a -> [a]
```

```
> flattenTreeOrd smallTree
[1,5,10]
> flattenTreeOrd notBSTree
```

[1,3,2,5,6,7,9]

Submission required: **Exercises 4 - 8**

Optional Extensions

Extension 1

Nat acts like a number, so we’ d also like it to be a member of the type class Num.

For those that are mathematically inclined, the operators (+) and (*) can actually be far more general than just addition and multiplication. The general term for any structure containing a set of values, a definition for

+

+ and for

×

× is called an **algebraic ring**, which you may see if you go on to study formal mathematics. In general, (+)and (*) can be any operator that satisfies a set of “nice proprieties”, such as the **law of distributivity**: x * (y + z) == (x * y) + (x * z). The full set of laws that(+) and (*) should satisfy is available here. As an example, if we choose (+) to be logical exclusive or, and (*) to be logical and, then we can define Bool as an instance of Num, and all the nice properties hold.

Using the template given, define the functions (+),(-),(*),abs,signum,fromIntegerfor the Nat type.

Note that for an instance a of Num:

- abs :: a -> a is the absolute value function
- signum :: a -> a returns the sign of a number,
- +1
- +1 if the number is positive,
- 0
- 0 if the number is zero, and
- `-1`
- `-1` if it is negative.
- fromInteger :: Integer -> a takes an Integer, and converts it to the new number type a.

You can get away without a definition for negate, as once (-) is defined, Haskell can work out negate for free. (Given that natural numbers cannot be negative anyway, the negatefunction is not very useful to u s.)

> Note that some of the functions required to make Nat an instance of Num may be sometimes undefined. There is no “mi nus one” in Nat, after all. Try to ensure as many functions are defined as possible, for as many inputs as possible. You s hould throw an error for attempting to perform any operation undefined in the natural numbers, like trying to compute 2 - 3.

```
> (S Z) + (S (S Z))
S (S (S Z))
> (S Z) - (S Z)
Z
> (S (S Z)) * (S (S (S Z)))
S (S (S (S (S Z))))
> abs (S Z)
S Z
> signum Z
Z
> (fromInteger 2) :: Nat
S (S Z)
> (S Z) - (S (S Z))
*** Exception: Can't have negative natural numbers
```

Extension 2

We’ d like to be able to define equality on binary trees based on the elements in the tree. If we just us ed deriving Eq, then two trees would be identical if they have precisely the same elements and structure. For the purposes of this question, we will define equality on binary trees to be the following weaker de finition: Two trees are defined to be equal if they share the same elements, with the same ordering (alt hough the structure need not be the same).

Add an instance of equality for binary trees that satisfies this weaker definition of equality.

Since we want this definition to work on any tree with elements who have equality defined on them, w e use the following notation.

```
instance (Eq a) => Eq (BinaryTree a) where
   -- (==) ...
```

which means that we are defining what == means for anything of type BinaryTree a, where a must be a member of the type class Eq.

> goodTree == unbalancedTree

True

> Submission is optional for **Extensions 1 and 2**. Make sure you have completed all other exercises first.

More Optional Extensions

Extension 3

Write a function

```
treeDelete :: (Ord a) => (BSTree a) -> a -> (BSTree a)
```

that takes a binary search tree, and an element, and removes it from the tree (if it is present).

> Note that this is much harder than insertion. Take the example tree above. Deleting 5 is obvious, as we just trim off that s ubtree and replace it will Null. But what should we do if I wanted to delete 4, or even worse, delete the root note, 3? The entire tree falls into two pieces, and must be restructured into a new tree that contains the remaining elements, while main taining the ordering property. I recommend drawing out a few examples by hand to get your head around it first.

Extension 4 (Tricky)

> Depending on how much you care about efficiency, this problem can be very difficult to implement efficiently, and may r equire some external research. Many different kinds of trees (Red-Black, AVL, ect.) are specifically defined to make cert ain operations like balancing easier to do.

Write a function

```
treeBalance :: (Ord a) => BSTree a -> BSTree a
```

that takes a binary search tree of integers, and rearranges the structure of the tree so it is now balanced. You may have to do some research as to how to implement this. (Really Tricky: Do it without rebuildi ng the entire tree from scratch.)