1 Scala

1.1 Functions

1. Write a function called `sum-numbers` that takes in two integers a and b as input, and outputs the sum of all the numbers $a, a+1, \ldots, b$.

2. Write a function called `fibonacci` that takes a number and outputs the corresponding Fibonacci number. How many times is your function run when evaluating `fibonacci(4)`? Write a version that runs in linear time.

3. Write a function that reverses a list.

4. Write a function `counts` that takes a list and outputs a map which says how many of each element is in the list. For example, `counts(List(1,2,1,3,4,4,4))` should be

```
Map(1 -> 2, 2 -> 1, 3 -> 1, 4 -> 3).
```

5. Write a function that uses the `/:` operator to find the maximal element of a list.

6. Implement the merge sort algorithm on lists. (If you don't know how merge sort works, look it up.)

7. Write a function called `choose2` which takes in a list l and outputs a list of lists of all possible combinations of two distinct elements chosen from l. Now write `choosek` which takes in a list l and an integer k, and outputs a list of all possible combinations of k distinct elements chosen from l.

1.2 Classes

1. Look up how inheritance works in Scala. Write a class `SyntaxTree` to represent mathematical expressions of the form $3 \cdot (5 + x)$ as a tree. (For example, that one corresponds to a tree with one node for the multiplication with two children, one of which is a leaf labeled 3 and one of which is a node for the addition, which in turn has 5 and x as children. Come ask if you want more clarification on how this is supposed to work.) There should be nodes for addition and multiplication, and leaves for numeric constants and for variables. You can assume all the numbers are integers.

   Hint: The best way to do this is probably with an abstract base class called `SyntaxTree` and different subclasses for the different types of nodes and leaves.

   (a) There should be methods `+` and `*` that produce new `SyntaxTree`s representing the sum or product of other syntax trees.
   (b) There should be a method `toString` that outputs a reasonable-looking string representation of the tree.
   (c) There should be a method `subst` that takes a variable name (as a string) and a `SyntaxTree`, and returns a new tree in which the supplied tree is substituted for the variable.

(d) There should be a method `eval` that outputs the number you get when you evaluate the tree. For example, the tree for $3 \cdot (5+4)$ should evaluate to 27. If there are variables, you should reduce as far as you can. For example, the tree for $(2+2)\cdot((4\cdot x)\cdot 2)$ should evaluate to $32\cdot x$. [Hint: Consider implementing another class to represent polynomials with rules for adding and multiplying them.]

2. Write a class `Graph` to represent a graph (the vertices-and-edges kind) where the vertices are labeled with arbitrary integers. These graphs are not directed, and have no self-loops. (Ask if you want to know what these things mean.) You can choose whatever private implementation you want, so long as the public interface does everything in this problem.

   (a) The constructor should have some way to supply the vertices and edges.

   (b) There should be global functions `completeGraph` and `emptyGraph` that take an integer and return a complete graph or an empty graph on that many vertices. (Note: In actual Scala style, this would be done using *companion objects* instead of global functions. If you'd like, look up how these work and do it that way instead.)

   (c) There should be a method `connect` that takes a list of vertex pairs and returns a new graph which is like `this` but with those pairs of vertices connected. Similarly, there should be a `disconnect`.

   (d) There should be a method + that takes another `Graph` and returns their disjoint union.

   (e) There should be a method == that returns `true` if and only if the graphs are the same (including vertex labels).

   (f) There should be a method `isConnected` that returns `true` if and only if the graph is connected.

   (g) There should be a method `hasCycles` that returns `true` if and only if the graph has at least one cycle.

   (h) There should be a method ~= that returns `true` if and only if the graphs are isomorphic (neglecting vertex labels).

   (i) Look up how partial functions work in Scala. Write a method `spanningTree` which is only defined if the graph is connected, and returns a spanning tree on the graph.

2 Clojure

2.1 Arithmetic, recursion, etc

1. Write a function called `disc` that takes in three numbers a, b, c as input, and computes the larger root of the quadratic polynomial $ax^2 + bx + c$, if the roots are real. It is up to you what to do if the roots are complex.

2. Write a function called `sum-numbers` that takes in two integers a and b as input, and outputs the sum of all the numbers $a, a+1, \ldots, b$.

3. (a) Write a recursive function called `fib`, which on input n computes the nth Fibonacci number. How many times is the function called when `(fib 4)` is evaluated?
   (b) Now write a function to compute the nth Fibonacci number in linear time.
   (c) Can you write a new version of `fib` using `lazy-seq`?

4. Write a function called `prime-test` that takes in an integer m and outputs true if and only if m is prime.

5. Write a function called `divisible?` that takes in two integers m and n and outputs true if and only if $n \mid m$.

6. Write a function called `num-digits` that takes in an integer n and outputs the number of digits in the decimal expansion of n.

7. Write a function called `reciprocals` that takes in a list of integers and outputs a list of their reciprocals. Since division by zero is a violation of Rule 4, the "reciprocal" of zero should be set to 1.

8. Write a function called `pythagorean-test` that takes in an integer n and outputs true if and only if n is part of a Pythagorean triple.

2.2 Data structure processing

1. Write a function called `reverse-list` that takes in a list l and outputs the reversed list.

2. Write a function called `uniquify-list` that takes in a list l and outputs another list which contains the distinct elements of l.

3. Write a function called `choose-2` which takes in a list l and outputs a list of lists of all possible combinations of two distinct elements chosen from l. Now try to write `choose-k` which takes in a list l and an integer k, and outputs a list of all possible combinations of k distinct elements chosen from l.

2.3 Creating data structures

In these exercises, we will learn how to create a data structure in Clojure. Since Clojure does not have type constructors, we will do this using the existing data structures by writing our own abstraction functions.

2.3.1 Binary trees

In this part we will create binary trees. (If you don't know what a binary tree is, look it up.) In each of the exercises, remember to use the abstraction functions that you created.

1. Create a data structure called a binary tree using either lists or vectors. A binary tree should be in some prespecified format. (Hint: For example, a binary tree can be represented as a list of three things, where the first element is the top node, the second element is the left subtree and the third element is the right subtree. Choose carefully what you want to do at a leaf.)
2. Write an abstraction function called `test-binary-tree` that takes in an object and outputs true if and only if the object fits the format of a binary tree.
3. Write abstraction functions `top-node`, `left-subtree` and `right-subtree` that take in a binary tree and output the top node, the left subtree and the right subtree respectively.
4. Write a function that outputs a list of the contents of a binary tree in depth-first order.
5. Write a function that outputs a list of the contents of a binary tree in breadth-first order.
6. Write a function that takes in a binary tree of numbers and outputs the maximum element.
7. Write a function that takes in a binary tree and determines whether it is symmetric (i.e., if flipping the left side about the middle gives the right side).
8. Write a function that takes in a binary tree and an integer n, and outputs a list of all the nodes at the nth level from the top (where the top level is level 0).

3 Haskell

3.1 Arithmetic

1. Define a function `leastTwo :: Int -> Int -> Int -> Int` that outputs the sum of the smallesttwo of its inputs.

2. Define a function `largestDivisor :: Int -> Int` that takes a positive integer n as input and outputs the largest divisor of n that is strictly smaller than n.

3. Write a fast (order $n \log n$) function called `power` that takes in a float x and an integer n and outputs x^n.

4. Write a function called `factors` which takes in a positive integer n and outputs a list of all the factors of n.

3.2 Lists

1. Define a function called `halve` that takes in a list and outputs

2. Define a function `descending :: [Int] -> Bool` that outputs `True` if each element in its input list is at least as big as the one before it.

3. Define a function `alternating :: [Int] -> Bool` that outputs `True` if and only if elements of the input list alternatingly go up and down.

4. Define a function `findUnique` that takes in two lists of the same type as input, and outputs all values that occur in either one list or the other, but not both. Also write a type signature for this function.

5. Create a (lazy) list of all prime numbers.

6. Create a function `longestWords` which when given a piece of text as a string, outputs a list of the longest words in that piece of text. (Hint: the built-in function `words` may be useful.)

7. Implement the insertion-sort sorting algorithm in Haskell. If you don't know how insertion-sort works, look it up.

8. Implement the merge-sort sorting algorithm in Haskell. If you don't know how merge-sort works, look it up.

9. Create a (lazy) list of all the Fibonacci numbers.

3.3 Data types

1. Define a data type called `Day`, which consists of the exhaustive set of all the days of the week (`Sunday`, `Monday`, etc). Write functions called `nextDay` and `previousDay` that take a `Day` as an input and output the next and previous day respectively. Write type signatures for these functions.

2. Define a binary tree data type as follows.

```
    data BinaryTree a = Empty | Branch a (BinaryTree a) (
        BinaryTree a)
                        deriving (Show, Eq)
```

   (a) Write a function that outputs a list of the contents of a binary tree in depth-first order.
   (b) Write a function that outputs a list of the contents of a binary tree in breadth-first order.
   (c) Write a function that takes in a binary tree of numbers and outputs the maximum element.
   (d) Write a function that takes in a binary tree and determines whether it is symmetric (i.e., if flipping the left side about the middle gives the right side).
   (e) Write a function that takes in a binary tree and an integer n, and outputs a list of all the nodes at the nth level from the top (where the top level is level 0).

3. Define a data type called `Tree` where each node can have multiple branches. Try to solve the previous exercises for this data type.

3.4 Monads

1. There is another common example of a monad that you may not have realized is a monad. It is called the `List` monad. By this I mean that the `List` datatype is secretly a monad. You may need to type `import List` in the interpreter to run this set of exercises.

   (a) What should the `return` function do for the `List` monad? Check that `return :: a -> [a]` does what you think it should do.
   (b) How should >>= work for lists? Try to write a function `map'` that works like `map`, but using >>= instead.
   (c) Write a function `filter'` that works like `filter`, but using >>= instead.
   (d) Can you write a function `foldl'` that works like `foldl`, by just using monadic syntax (i.e. >>= and `return`)? Why or why not?

2. Write a function that waits for input from the user. Once the user enters an input, it should ask for an input again unless the input is "exit". In this case, it should output a concatenated list of all the inputs that the user typed, one after the other. You will have to use the IO monad. The following functions may be useful: `read`, `putStr`, `putStrLn`, `getLine`. You may also want to use the `do` notation.

4 Erlang

In this problem set, we will mostly explore the concurrency features of Erlang (in baby steps). To become familiar with other Erlang syntax, you may want to re-write some of the functions from previous homework assignments in Erlang so as to test the waters.

1. Explore the functions `self/0`, `spawn/3`, `register/2`. Also explore the syntax `?MODULE`.
2. You can send a *message* to the PID (process ID number) of any process inside Erlang. This is done using the syntax `<Pid> ! <message>`, where the message can be any object. You can 'receive' the messages passed to a process by using `receive`. Explore the syntax of `receive`. What happens if you try to `receive` when there are no more messages?
3.
   (a) Write a module that will start a new Erlang process, which does the following. If we send a formatted message to it, it sends the current process the same message back. (Hint: You will need a loop for the created process to listen for messages. You will also need to make sure to pass the Pid of the current process in any message that you send to the new process.)

   (b) Write a module that will start a new Erlang process, which does the following. If we send a formatted message to it containing a string, it sends the current process the reverse of the string back.
4. Create a module that starts a new Erlang process, which does the following. Once a message is sent to it, it should print out the message to the interpreter as it arrives.
5. Create a module that starts a new Erlang process, which does the following. Once a message is sent to it containing a formatted list of numbers, it prints out the sum of all the numbers in the list.

5 Prolog

1. Write a rule that checks whether a list is empty.
2. Write your own versions of the rules `append`, `member`, `takeout`.
3. Write a rule that will add up all elements of a list.
4. Write a rule to compute the Fibonacci numbers.
5. Write a rule that will reverse the elements of a list.
6. Try to do problems from all the previous exercise sheets in Prolog. This is to attempt to think in a logic programming paradigm as opposed to just a declarative programming paradigm.
7. Write some Prolog code to output a 3×3 magic square. You could also try to do larger magic squares. Your code should find a magic square and then pretty-print it.
8. Write a Sudoku solver. For simplicity, you may use a 4×4 grid.
9. Solve the following puzzle in Prolog.

Four couples in all
    Attended a costume ball.
The lady dressed as a cat
    Arrived with her husband Matt.
Two couples were already there,
    One man dressed like a bear.
First to arrive wasn't Vince,
    But he got there before the Prince.
The witch (not Sue) is married to Chuck,
    Who was dressed as Donald Duck.
Mary came in after Lou,
    Both were there before Sue.
The Gipsy arrived before Ann,
    Neither is wed to Batman.
If Snow White arrived after Tess,
    Then how was each couple dressed?