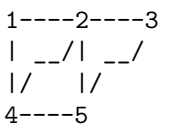
Exercise 1: Graph coloring (10 points)

Graph coloring is the problem of assigning *colors* to vertices in a non-directed graph, so that no two adjacent nodes have the same color. In other words, if two vertices are adjacent, they must have different colors. Each vertex must be assigned exactly one color. Of course, the set of colors is constrained (otherwise we could simply assign a different color to each vertex). In this assignment, we will assume there are exactly 3 colors available: Red, Green and Blue.

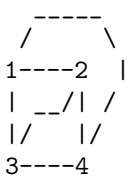
Given the following graph:



We can validly color it with RGB as:

1. Red
2. Green
3. Blue
4. Blue
5. Red

The following graph, however, cannot be colored with 3 colors:



Indeed, the constraints in that graph impose that all four vertices need to have different colors, but we only have 3 colors.

We represent a graph with the following data structure:

```
final case class Vertex(id: Int, neighbors: List[Int])
final case class Graph(vertices: List[Vertex])
```

- Vertices in the graph all have distinct, strictly positive `id`s.
- It is *not* guaranteed that `id`s are consecutive integers from 1 to N.
- It is guaranteed that the `neighbors` of a `Vertex` actually exist in the graph (i.e., if `neighbors` contains an i, then `vertices` contains a `Vertex` with id i).
- Since the graph is non-directed, it is guaranteed that, if the `neighbors` of a `Vertex` a contains b, then the `neighbors` of b contains a.

- It is *not* guaranteed that a `Vertex` is not its own neighbor (note that if that happens, obviously there cannot be any valid coloring for the graph).

We want to solve the coloring of a `graph: Graph` using *constraint programming*. To do this, we want to generate a set of constraints that encode the properties of a valid coloring for the graph. The complete set of constraints is modeled as a `Formula`, a data structure defined as follows:

```
sealed abstract class Formula
final case class Var(name: String) extends Formula
final case class Not(p: Formula) extends Formula
final case class And(p: Formula, q: Formula) extends Formula
final case class Or(p: Formula, q: Formula) extends Formula
final case class Implies(p: Formula, q: Formula) extends Formula
```

A `Var` is a boolean variable in the constraint. Since we have 3 colors, we cannot simply use `Var("1")` as "the color" of vertex 1. Instead, we use one `Var` per color per vertex: `Var("1R")` is `true` if and only if vertex 1 is Red. Similarly, we can have `Var("3G")` or `Var("151B")`.

Your task: implement the method

```
def graphColoring(graph: Graph): Formula = ...
```

which, given an arbitrary non-directed graph `Graph`, returns a complete `Formula` encoding all the constraints of the graph coloring problem for that graph.

Hint: decompose the problem in smaller functions that generate specific `Formula`s for some specific parts of the problem constraints.

Exercise 2: Streams (10 points)

In this task you are going to work on Streams and implement several types of averages on them. For the duration of this task, you can assume that arithmetic operations on `Double`s do not suffer from loss of precision.

Average of last two elements (2 points):

Implement a function that given a stream of doubles returns a stream of averages of last two elements.

Example: given a stream `(1, 1, 4, 5, 7, 0)` you should return `(1, 2.5, 4.5, 6, 3.5)`.

```
def pairAverages(data: Stream[Double]): Stream[Double] = ???
```

Average in a arbitrary window (4 points)

Implement a function that given a number `n` and a stream of doubles returns a stream of averages of the last `n` elements of that stream.

Example: given a number `2` and a stream `(1, 1, 4, 5, 7)` you should return `(1, 2.5, 4.5, 6)`. Example: given a number `3` and a stream `(1, 2, 5, 8, 15)` you should return `(2.67, 5, 9.33)` (rounding is only for display here, you should not round yourself).

```
def windowAverage(windowSize: Double, data: Stream[Double]): Stream[Double] = ???
```

Rolling average (4 points)

Implement a function that given a stream of doubles returns a stream of averages of all data streamed so-far.

Example: given a stream `(1, 2, 3, 4, 5)` you should return `(1, 1.5, 2, 2.5, 3)`.

```
def rollingAverage(data: Stream[Double]): Stream[Double] = ???
```

Exercise 3: Variable Substitution in Lisp (10 points)

After having written your Lisp interpreter, you decide that you want to implement some transformations over your Lisp programs. The first transformation you consider is inlining value definitions. Namely, you want terms of the shape `(val name expr rest)` to be transformed to some new term `rest2` that corresponds to `rest` where all occurences of `name` have been replaced by `expr`.

For example, the term `(val a 1 (+ a (+ 2 a)))` should be rewritten to `(+ 1 (+ 2 1))`.

Scala implementation (5 points)

In order to achieve the above task, complete the following function:

```
def substitute(term: Any, symbol: Symbol, replaceBy: Any): Any = ???
```

Note that in the example above, you would call

```
substitute(List('+, 'a, List('+, 2, 'a)), 'a, 1)
```

and the expected result is

```
List('+, 1, List('+, 2, 1))
```

Remember that Lisp terms are of type `Any`. Composite terms have type `List[Any]`, and variable names are instances of `Symbol` (you can use `==` between symbols). You may use any method in the Scala standard library (see appendix for most typical ones).

Hint: You can perform type checks in pattern matching by using the following syntax in the case statement:

```
case x: Type =>
```

Translation into Lisp (5 points)

You now want to implement the same functionality within the Lisp language itself. This means your input now consists of a Lisp list. Complete the following function definition (using the syntactic sugar proposed in the assignment):

```
(def (substitute term symbol replaceBy) ??? rest)
```

The example above would this time correspond to

```
(substitute '(+ a (+ 2 a)) 'a 1)
```

and your function should output

```
'(+ 1 (+ 2 1))
```

Note: The term `'(+ a (+ 2 a))` corresponds to the Lisp list

```
(cons '+ (cons 'a (cons (cons '+ (cons 2 (cons 'a nil))) nil)))
```

The available Lisp API can be found in the appendix. You may use the predicates `isCons?` and `isNil?` to determine whether a given term respectively corresponds to an instance of `cons` or `nil`. If you wish to use a function that was not defined in the appendix, please provide its definition as well.

Hint: Remember Lisp (non-)syntax of prefix notation, for example, comparisons are written as `(= a b)`.

Note: Indent your code properly when answering this question, syntax is very important here.

Appendix

A non-exhaustive (but useful) API

Here are some methods from the Scala standard library that you may find useful:

on List (containing elements of type A):

- `xs ++ (ys: List[A]): List[A]`: appends the list `ys` to the right of `xs`, returning a `List[A]`.
- `xs.apply(n: Int): A`, or `xs(n: Int): A`: returns the `n`-th element of `xs`. Throws an exception if there is no element at that index.
- `xs.drop(n: Int): List[A]`: returns a `List[A]` that contains all elements of `xs` except the first `n` ones. If there are less than `n` elements in `xs`, returns the empty list.
- `xs.filter(p: A => Boolean): List[A]`: returns all elements from `xs` that satisfy the predicate `p` as a `List[A]`.
- `xs.flatMap[B](f: A => List[B]): List[B]`: applies `f` to every element of the list `xs`, and flattens the result into a `List[B]`.
- `xs.foldLeft[B](z: B)(op: (B, A) => B): B`: applies the binary operator `op` to a start value and all elements of the list, going left to right.
- `xs.map[B](f: A => B): List[B]`: applies `f` to every element of the list `xs` and returns a new list of type `List[B]`.
- `xs.nonEmpty: Boolean`: returns `true` if the list has at least one element, `false` otherwise.
- `xs.reverse: List[A]`: reverses the elements of the list `xs`.
- `xs.take(n: Int): List[A]`: returns a `List[A]` containing the first `n` elements of `xs`. If there are less than `n` elements in `xs`, returns these elements.
- `xs.toMap: Map[K, V]`: provided `A` is a pair `(K, V)`, converts this list to a `Map[K, V]`.
- `xs.zip(ys: List[B]): List[(A, B)]`: zips elements of `xs` and `ys` in a pairwise fashion. If one list is longer than the other one, remaining elements are discarded. Returns a `List[(A, B)]`.
- `def sliding(size: Int, step: Int): Stream[Stream[A]]`: groups elements in fixed size blocks by passing a "sliding window" over them. Blocks all contain `size` elements. `step` is the distance between the first elements of successive groups.

You can use the same API for `Stream`, replacing `List` by `Stream`.

on Stream (containing elements of type A):

- `xs #:: (ys: => Stream[A]): Stream[A]`: Builds a new stream starting with the element `xs`, and whose future elements will be those of `ys`.

on Stream (the object):

- `Stream.from(i: Int): Stream[Int]`: Creates an infinite stream of integers starting at i.

on Map (containing keys of type K, values of type V):

- `mp.get(k: K): Option[V]`: For a given key `k`, returns `Some(v)` if a value exists in the map `mp`, `None` otherwise.

Lisp API

Here are a few Lisp functions you may find useful:

- `(isCons? term)`: returns `1` if the provided term is equal to `(cons x y)` for some `x` and `y`, and `0` otherwise.
- `(isNil? term)`: returns `1` if the provided term is equal to `nil`, and `0` otherwise.
- `(cons x y)`: constructs a list with head `x` and tail `y` (corresponds to `::` in Scala).
- `nil`: constructs an empty list (corresponds to `Nil` in Scala).
- `(car x)`: head of the list `x`.
- `(cdr x)`: tail of the list `x`.
- `(if cond then else)`: returns the term `then` if `cond` DOES NOT evaluate to `0`, and `else` otherwise.

You may also find the following conditional construct useful:

```
(cond (test1 expr1) ...
      (testN exprN)
      (else exprElse))
```

The `(test expr)` pairs are successfully considered until a `test` evaluates to something different from `0`, in which case the corresponding `expr` is returned, or until the `(else exprElse)` pair is reached in which case `exprElse` is returned.

Don't forget the syntax for

1. defining values:

   `(val name body rest)`

2. defining functions:

   `(def (name arg1 ... argN) body rest)`