Tests

With the following module, we'd like to gain some confidence that we've done things correctly.

Again, this is literate haskell, so this is the source code.

```
> module P1Test where
```

Again, we import a bunch of stuff.

```
> import P1 (Node(..), distance, tour, tourDistance)
```

```
> import Data.Tuple (swap)
```

```
> import Test.QuickCheck ( Arbitrary(..), Args(..), choose, quickCheck
>                        , quickCheckWith, stdArgs
>                        )
```

Now we make an instance for generating arbitrary `Node`s.

```
> instance Arbitrary Node where
>     arbitrary = Node <$> arbitrary <*> choose (-90, 90) <*> choose (-180, 180)
```

And we'd like the following properties to be true:

1: distance x y == distance y x

```
> prop_SymmetricDistance :: Node -> Node -> Bool
> prop_SymmetricDistance x y = distance x y == distance y x
```

We'd also like for `distance` to be a metric. However, it does not satisfy reflexivity. We do have the following properties though:

Non-negativity

2: distance x y >= 0

```
> prop_NonNegativeDistance :: Node -> Node -> Bool
> prop_NonNegativeDistance x y = distance x y >= 0
```

Triangle inequality

3: distance x z <= distance x y + distance y z

```
> prop_Triangle :: Node -> Node -> Node -> Bool
> prop_Triangle x y z = distance x z <= distance x y + distance y z
```

4: tour xs++tour xs == tour (xs++xs)

```
> prop_ConcatTour :: [Node] -> Bool
> prop_ConcatTour xs = tour xs ++ tour xs == tour (xs ++ xs)
```

5: tourDistance (xs++ys) == tourDistance (ys++xs)

This property is a bit hard to understand, What it says is that if we have two parts of the same tour, then the distance of each part commutes.

```
> prop_CommutativeTourDistance :: [(Node, Node)] -> [(Node, Node)] -> Bool
> prop_CommutativeTourDistance xs ys =
>     tourDistance (xs ++ ys) == tourDistance (ys ++ xs)
```

6: tourDistance xs+tourDistance ys == tourDistance (xs++ys)

Here we have that `tourDistance` is a Semigroup Homomorphism to (N, +). Unfortunately, since `distance` is not a metric, it cannot be more powerful than that.

```
> prop_HomomorphismTourDistance :: [(Node, Node)] -> [(Node, Node)] -> Bool
> prop_HomomorphismTourDistance xs ys =
>     tourDistance xs + tourDistance ys == tourDistance (xs ++ ys)
```

For TSP we should be able to show that it is actually symmetric.

7: tourDistance (tour xs) == tourDistance (tour (reverse xs))

```
> prop_SymmetricTSP :: [Node] -> Bool
> prop_SymmetricTSP xs =
>     tourDistance (tour xs) == tourDistance (tour $ reverse xs)
```

And we should also be able to show that if we start the tour at a different point, the distance does not change. E.g. Instead of 1-2-...-22-1 we have 2-3-...-22-1-2.

```
> prop_ShiftTSP :: [Node] -> Int -> Bool
> prop_ShiftTSP xs n = let n' = n `mod` length xs in
>     tourDistance (tour xs) ==
>     tourDistance (tour . uncurry (++) . swap . splitAt n $ xs)
```

Here we begin running our properties.

```
> main :: IO ()
> main = do
>     quickCheck prop_SymmetricDistance
>     quickCheck prop_ConcatTour
>     quickCheck prop_CommutativeTourDistance
>     quickCheck prop_HomomorphismTourDistance
>     quickCheck prop_NonNegativeDistance
>     quickCheck prop_Triangle
>     quickCheck prop_SymmetricTSP
>     quickCheck prop_ShiftTSP
```

Finally, we have a few unit tests we'd like to have as well. For the sake of not making this module any larger, we simply run quickCheck once.

The tour constructed from the empty list of nodes, should be empty.

9: tour [] == []

```
>     quickCheckWith stdArgs {maxSuccess = 1} (tour [] == [])
```

The distance of no tour should be 0

10: tourDistance [] == 0

```
>     quickCheckWith stdArgs {maxSuccess = 1} (tourDistance [] == 0)
```

Half of the circumference of The Earth should be 20039 km.

11: distance (Node _ 0 0) (Node _ 0 180) == 20039

```
>     quickCheckWith stdArgs {maxSuccess = 1}
>         (distance (Node 1 0 0) (Node 2 0 180) == 20039)
```

A final sanity check is that the distance between the first two nodes should be 492 km.

```
>     quickCheckWith stdArgs {maxSuccess = 1}
>         (distance (Node 1 38.24 20.42) (Node 2 39.57 26.15) == 492)
```

Having all of these properties hold gives us some confidence that we've constructed a valid program.

We can see this in action by running on the terminal:

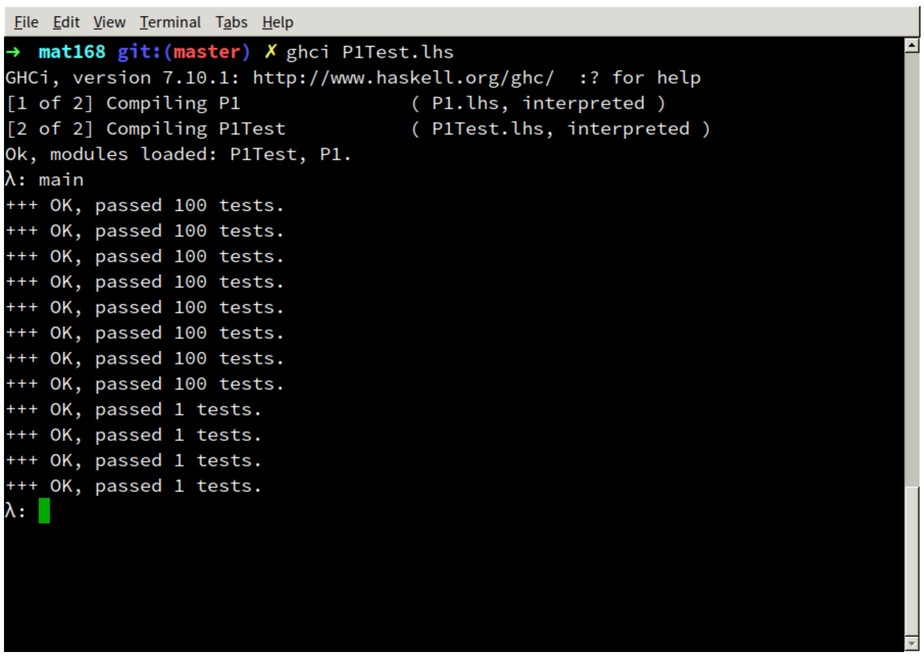

Figure 1: runit