You can download the code that accompanies the lecture slides from here. The code is divided into nine modules. Any one of these can be loaded in `ghci` to explore the state of the prover at a particular point in its development. You'll need GHC 7.10 or later.

```
> :l Proposition
[1 of 1] Compiling Proposition      ( Proposition.hs, interpreted )
Ok, modules loaded: Proposition.
```

This brings in the kernel of the prover. Note that the axioms defined by this module differ very slightly from those in the slides. In the slides, we use terms of type `Term Char`, but this is just to simplify the presentation.

To be slightly more robust, we use more refined alphabets from which to pick our variables: axioms 1 and 3 only use two distinct propositional variables, and so have type `Theorem Two`, while axiom 2 uses three variables and so has type `Theorem Three`. These are defined by

```
data Two = X | Y
data Three = P | Q | R
```

In order to move between alphabets, you can use `instTerm` or `inst`.

```
> :l Instances
[1 of 3] Compiling Proposition      ( Proposition.hs, interpreted )
[2 of 3] Compiling Instances        ( Instances.hs, interpreted )
Ok, modules loaded: Proposition, Instances.
```

Terms and theorems can now be pretty-printed and compared. And if you want to change the alphabet used by a term or a theorem, you can just use the function `fmap`:

```
> fmap (splitTwo "forest" "trees") (Var X :=>: Var Y)
"forest" ==> "trees"
> fmap length it
6 ==> 5
> fmap (`lookup` [(6,"trees"), (5,"roots")]) it
(Just "trees") ==> (Just "roots")
```

```
> :l Utils
[1 of 3] Compiling Proposition      ( Proposition.hs, interpreted )
[2 of 3] Compiling Instances        ( Instances.hs, interpreted )
[3 of 3] Compiling Utils            ( Utils.hs, interpreted )
Ok, modules loaded: Utils, Proposition, Instances.
```

We have now added some derived syntax, and some functions to help with instantiation and term-matching.

```
> :l BootstrapDIY
[1 of 4] Compiling Proposition      ( Proposition.hs, interpreted )
[2 of 4] Compiling Instances        ( Instances.hs, interpreted )
[3 of 4] Compiling Utils            ( Utils.hs, interpreted )
[4 of 4] Compiling BootstrapDIY     ( BootstrapDIY.hs, interpreted )
Ok, modules loaded: Utils, Proposition, Instances, BootstrapDIY.
```

This loaded a skeleton file for a bunch of theorems we'll use later. They are many of the same theorems you would prove if developing propositional logic in a mathematical logic class. Can you fill in the proofs yourself? Answers can be found in the next module:

```
> :l Bootstrap
[1 of 4] Compiling Proposition      ( Proposition.hs, interpreted )
[2 of 4] Compiling Instances        ( Instances.hs, interpreted )
[3 of 4] Compiling Utils            ( Utils.hs, interpreted )
[4 of 4] Compiling Bootstrap        ( Bootstrap.hs, interpreted )
Ok, modules loaded: Bootstrap, Utils, Proposition, Instances.
```

We now have most of the basic theorems we'll need to develop some proof tools.

```
> :l BootstrapConversions
[1 of 5] Compiling Proposition      ( Proposition.hs, interpreted )
[2 of 5] Compiling Instances        ( Instances.hs, interpreted )
[3 of 5] Compiling Utils            ( Utils.hs, interpreted )
[4 of 5] Compiling Bootstrap        ( Bootstrap.hs, interpreted )
[5 of 5] Compiling BootstrapConversions ( BootstrapConversions.hs, interpreted )
Ok, modules loaded: Bootstrap, Utils, BootstrapConversions, Proposition, Instances.
```

We now have some basic conversions which you can use to rewrite theorems. The selection of conversions is far from complete. We just provide enough for the next module.

```
> :l Sequent
[1 of 6] Compiling Proposition      ( Proposition.hs, interpreted )
[2 of 6] Compiling Instances        ( Instances.hs, interpreted )
[3 of 6] Compiling Utils            ( Utils.hs, interpreted )
[4 of 6] Compiling Bootstrap        ( Bootstrap.hs, interpreted )
[5 of 6] Compiling BootstrapConversions ( BootstrapConversions.hs, interpreted )
[6 of 6] Compiling Sequent          ( Sequent.hs, interpreted )
Ok, modules loaded: Bootstrap, Sequent, Utils, BootstrapConversions, Proposition, Instances.
```

Theorems can now be interpreted as sequents, meaning that our proofs do not rely on the mini-language provided by `Bootstrap.hs`.

```
> :l Conversions
[1 of 7] Compiling Proposition      ( Proposition.hs, interpreted )
[2 of 7] Compiling Instances        ( Instances.hs, interpreted )
[3 of 7] Compiling Utils            ( Utils.hs, interpreted )
[4 of 7] Compiling Bootstrap        ( Bootstrap.hs, interpreted )
[5 of 7] Compiling BootstrapConversions ( BootstrapConversions.hs, interpreted )
[6 of 7] Compiling Sequent          ( Sequent.hs, interpreted )
[7 of 7] Compiling Conversions      ( Conversions.hs, interpreted )
Ok, modules loaded: Bootstrap, Sequent, Conversions, Utils, BootstrapConversions, Proposition, Instances.
```

A more meaty selection of conversions is now available, ones that work on sequents rather than plain theorems.

```
> :l Tautology
[1 of 8] Compiling Proposition      ( Proposition.hs, interpreted )
[2 of 8] Compiling Instances        ( Instances.hs, interpreted )
[3 of 8] Compiling Utils            ( Utils.hs, interpreted )
[4 of 8] Compiling Bootstrap        ( Bootstrap.hs, interpreted )
[5 of 8] Compiling BootstrapConversions ( BootstrapConversions.hs, interpreted )
[6 of 8] Compiling Sequent          ( Sequent.hs, interpreted )
[7 of 8] Compiling Conversions      ( Conversions.hs, interpreted )
[8 of 8] Compiling Tautology        ( Tautology.hs, interpreted )
Ok, modules loaded: Tautology, Bootstrap, Sequent, Conversions, Utils, BootstrapConversions, Proposition, Instances.
```

And with all the other tools in place, we have a tautology checker. Behold!

```
> tautology (Not (x /\ y) <=> Not x \/ Not y)
Just |- ~~((~~(X ==> ~Y) ==> ~X) ==> ~(~X ==> ~~(X ==> ~Y))) ==> ~Y
```

Does that look right? Let's double check:

```
> (Not (x /\ y) <=> Not x \/ Not y)
~~((~~(X ==> ~Y) ==> ~X) ==> ~(~X ==> ~~(X ==> ~Y))) ==> ~Y
```