2 *Basics*

Our laboratory for this course is the Coq proof assistant. Coq can be seen as a combination of two things:

1. a simple and slightly idiosyncratic (but, in its way, extremely expressive) *programming language*, and
2. a set of tools for stating *logical assertions* (including assertions about the behavior of programs) and assembling evidence of their truth.

We will be investigating both aspects together.

2.1 Enumerated Types

In Coq's programming language, almost nothing is built in—not even booleans or numbers! Instead, it provides powerful tools for defining new types of data and functions that process and transform them.

Let's start with a very simple example. The following definition tells Coq that we are defining a new set of data values. The set is called `day` and its members are `monday`, `tuesday`, etc. The lines of the definition can be read "`monday` is a `day`, `tuesday` is a `day`, etc."

```
Inductive day : Set :=
  | monday : day
  | tuesday : day
  | wednesday : day
  | thursday : day
  | friday : day
  | saturday : day
  | sunday : day.
```

Having defined this set, we can write functions that operate on its members.

```
Definition next_weekday (d:day) : day :=
  match d with
  | monday => tuesday
  | tuesday => wednesday
  | wednesday => thursday
  | thursday => friday
  | friday => monday
  | saturday => monday
  | sunday => monday
  end.
```

One thing to note is that the argument and return types of this function are explicitly declared. Like most functional programming languages, Coq can often work out these types even if they are not given explicitly, but we'll always include them to make reading easier.

Having defined a function, we might like to check how it works on some examples. There are actually three different ways to do this in Coq.

1. We can use the command `Eval simpl` to evaluate a compound expression involving `next_weekday`. For example, if we give Coq the following input

   ```
   Eval simpl in (next_weekday (next_weekday saturday)).
   ```

   it will print the simplified result and its type:

   ```
   ▶    = tuesday
        : day
   ```

   The keyword `simpl` (for "simplify") tells Coq precisely how to evaluate the expression we give it. For the moment, `simpl` is the only one we'll need; later on we'll see some alternatives that are sometimes useful.

   If you have a computer handy, now would be an excellent moment to fire up the Coq interpreter under your favorite IDE—either CoqIde or Proof General—and try this for yourself. Load the file `Basics.v` from the book's accompanying Coq sources, find the above example a little ways from the top, send it to Coq, and observe the result.

2. We can record what we *expect* the result to be in the form of a Coq `Example`:

   ```
   Example test_next_weekday:
     (next_weekday (next_weekday saturday)) = tuesday.
   ```

This declaration does two things: It makes an assertion (that the second weekday after `saturday` is `tuesday`), and it gives the assertion a name that can be used to refer to it later.

Having made the assertion, we can also ask Coq to verify it, like this:

```
Proof. reflexivity.  □
```

The details are not important for now (we'll come back to them in a few chapters), but essentially this can be read "The assertion we've just made can be proved by observing that both sides of the equality are the same after simplification." The symbol □ is written `Qed.` in ascii `.v` files

3. Third, we can ask Coq to "extract," from a `Definition`, a program in some other, more conventional, programming language (OCaml, Scheme, or Haskell) with a high-performance compiler.

   This facility is very interesting, since it gives us a way to construct *fully certified* programs in mainstream languages. This is actually one of the main uses for which Coq was developed. We'll have more to say about this a little later on.

2.2 Booleans

Similarly, we can define the type `bool` of booleans, with constants `true` and `false`.

```
Inductive bool : Set :=
  | true : bool
  | false : bool.
```

Although we are rolling our own booleans here for the sake of building up everything from scratch, Coq does, of course, provide a default implementation of the booleans in its standard library, together with a multitude of useful functions and lemmas. (Take a look at `Coq.Init.Datatypes` in the Coq library documentation if you're interested.) Whenever possible, we'll name our own definitions and theorems so that they exactly coincide with the ones in the standard library.

Functions over booleans can be defined in the same way as above:

```
Definition negb (b:bool) :=
  match b with
  | true => false
  | false => true
```