Where does Coq define booleans and numbers?

In the standard library.

What's a "type" in Coq?

A set of data values.

Make a named assertion that `~true` is `false`, then prove it.

```
Example test_negation:
  (negb true) = false.
Proof. simpl. reflexivity. Qed.
```

Name three ways to check that a function works.

- Use `Eval simpl in (expr)` on a test case and observe the result.
- Use `Example`/`Theorem`/whatever to record expected result, and then prove it. Coq will only accept your proof if it's correct.
- "extract" the function definition to OCaml, Scheme, or Haskell.

How might you create "unit tests"?

Create `Example`s that are nothing but an equation with the expected value on one side and a term built from function applications on the other.

Apply negation to the boolean `true` and evaluate.

```
Eval simpl in (negb true).
```

How do you fill in a hole in a \`Definition\`/\`Fixpoint'? In an \`Example\`?

`admit` fills in holes in a `Definitions`/`Fixpoint'.

`Admitted` fills in holes in proofs.

How does Coq write the type of a boolean conjunction function?

```
bool->bool->bool
```

What does the `Check` command do?

It causes Coq to print the type of an expression.

How will we use the module system?

If you put declarations between `Module X` and `End X` then after `End` the definitions are referred to as `X.foo`.

What is an enumerated type?

It's a type (a set of data values) whose members are fully enumerated in in the type's definition.

When we use `Inductive` to define a type, we should see it as what?

A set of *expressions*, inductively defined. The definition tells us exactly how members of the type can be constructed, and excludes all other expressions.

What "magic" does Coq provide for natural numbers?

The ability to use numerals instead of tediously constructing numbers with the `O` and `S` constructors.

What is the fundamental difference between a data constructor and functions?

Book: Functions come with *computation rules*. Data constructors have no behavior attached.
Me: The application of constructors are values **as written**. The application of functions are **never** values as written, they are terms which must be evaluated.

Name some keywords that can introduce a function.

- `Definition`
- `Fixpoint` in case of recursion

What kind of recursion does Coq allow?

Structural (or *primitive*) recursion. That means recursive calls must be on strictly smaller values, guaranteeing termination.

What notational convenience does Coq provide for multiple parameters of the same type?

The following are equivalent:

```
(n m : nat)
(n: nat) (m: nat)
```

How does one match on *multiple* expressions?

A comma is placed between them in the scrutinee and between the two sides of each matching pattern.

What is an underscore in the context of `match` expressions?

It is a *wildcard pattern*, matching any expression without giving that expression a name.

How is "language support" introduced for some definitions?

Name three kinds of language support available.

With `Notation` constructions which also define associativity and precedence.

- Numerals
- Operators
- Collections syntax

How can one choose between multiple notation interpretations for an expression.

```
expression%notation_scope
```

Name a few notation scopes.

- `nat`
- `Z`
- `type_scope`

What two tactics simplify the goal by performing computation?

- `simpl`
- `compute` - results in possibly larger terms

The `reflexivity` tactic implicitly does what?

What's the difference between the simplification of `simpl` and that of `reflexivity`?

Simplifies both sides before testing (including by using `simpl`).

Among other things, `reflexivity` may unfold definitions. `simpl` never will.

Why doesn't `reflexivity`'s implicit simplification unfold definitions?

`reflexivity` ends the current goal so it doesn't matter if the resulting term is horribly large and unwieldly.

What keywords behave identically to `Theorem`?

- Example
- Lemma
- Fact
- Remark

What is the "context"?

The list of current assumptions that can be used in proving the goal.

What does the `intros` tactic do?

- For a conditional it introduces the antecedent as an assumption into context.
- For a universally quantified statement it introduces an arbitrary element of the domain into context and discharges the quantifier.

What is the syntax of `intros`?

The keyword `intros` followed by a space-delimited list of names for the assumptions. These may be names of variables already in context, or they may be ones you're *introducing*. The names are interpreted in the order the relevant expressions appear in the current context.

Some of the simple `intros` examples don't actually require `intros`. Give two reasons you might actually need it in a proof.

- There may be conditions to the proposition being proved. Without first using `intros` to discharge the quantifiers you can't use `intros` to introduce the hypotheses into context.
- Without eliminating quantifiers you cannot use (by `rewrite`) theorems of the universal quantification form, since they operate on free variables.

Describe the `rewrite` tactic.

What does rewriting *left-to-write* mean?

It rewrites the current goal using the provided rule (in context or previously defined) and in the provided direction.

For example:

```
rewrite -> H
```

Left-to-write means rewriting the terms in the goal that matches the left-hand-side of the rule being used.

How are are propositions with multiple hypothesis written?

```
Hyp1 -> ... -> HypN -> Conclusion
```

Why can't simple calculation prove every theorem?

Unknown values may appear as arguments to functions, preventing simplification.
For example, given an arbitrary `n:nat` we can't simplify since we don't know which constructor applies.

Describe the `destruct` tactic.

```
destruct var as [pattern].
```

`as [pattern]` is optional.

The pattern consists of names for the data of the possible data constructors of `var` separated by `|`.
For a nullary constructor just put the pipe.

Why don't we say `destruct b as [true | false]`.?

Remember, the `as` pattern in a `destruct/induction` is for the **data** associated with a constructor. Nullary constructors (*values*) have none.

So you would write either of these two:

```
destruct b as [|].
destruct b.
```

`destruct` is used to prove a theorem about an enumerated type for each possible ...

... constructor used to create that type.

What do `Case`/`SCase`/etc do?

Don't confuse `Case` with ...

They add a string to the context of the current subgoal and is discharged when the current subgoal is proved.

... `case`.

What is the syntax of the `induction` tactic?

Just the same as the `destruct` tactic.

What do context items like `IHn'` stand for?

Inductive Hypothesis for `n'`

Name some fundamental facts concerning our definition of `plus` comes up over and over?

```
- S n' + m = S (n' + m)
- S (n + m) = n + (S m)
```

How can you create sub-theorems without creating a new top-level name?

Use the `assert` tactic.

```
assert (H: whatever).
  Case "Proof of assertion". whatever.
```

After the proof is done `H` will be added to context.

What is a common non-stylistic reason for using `assert`?

You want to use the `rewrite` tactic on an instance of the pattern that is not outermost.
In this case you can prove as a sub-theorem exactly the rewrite you want, and then use `rewrite` in terms of this sub-theorem.