Where does Coq define booleans and numbers?

In the standard library.

What's a "type" in Coq?

A set of data values.

As an example of an enumerated type, define the boolean type.

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

Define a boolean negation function.

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

end.

Define a function for boolean conjunction.

```
Definition andb (b1:bool) (b2:bool) : bool :=
  match b1 with
  | true => b2
  | false => false
```

end.

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

Example test_negation:
 (negb true) = false.

Proof. simpl. reflexivity. Qed.

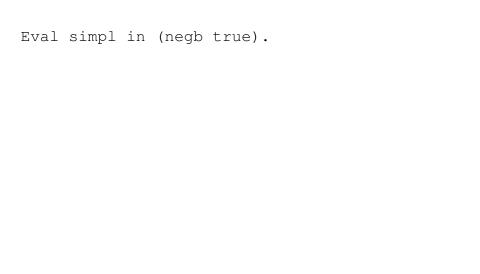
Name three ways to check that a function works.

- Use Eval on a test case and observe the result.

Haskell.

- Use Example/Theorem/whatever to record expected result, then as Cog to verify.
- "extract" function Definition to OCaml, Scheme, or

Apply negation to the boolean true.



What are Coq's names for boolean and/or/not?

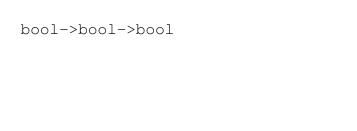
- andb
- -orb
- negb

How do you fill in a hole in a Definition? In an Example?

admit fills in holes in Definitions.

Admitted fills in holes in proofs.

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



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.

As an example of a type with a sum constructor, define nat.

Inductive nat : Type :=
 | 0 : nat
 | S : nat -> nat.

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

Functions come with *computation rules*. Data constructors have no behavior attached.

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 then in the scrutinee and between the two sides of each matching pattern.

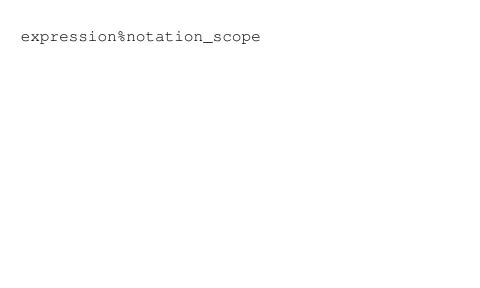
How is "language support" introduced for some definitions?

Name two kinds of language support available.

With ${\tt Notation}$ constructions which also define associativity and precedence.

- Numerals
- Operators
- Collections syntax

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



Which tactic is like simpl "on steroids"?



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 expand definitions. simpl never will.

What does the intros tactic do?

For a conditional it introduces the antecedent as an assumption. For a universally quantified statement it introduces an arbitrary element of the domain 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.

Describe the rewrite tactic.

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

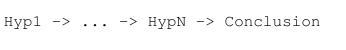
It rewrites the current goal using the provided rule and in the provided direction.

For example:

rewrite -> H

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

How are are propositions with multiple hypothesis written?



Why doesn't simplification be used to prove all theorems?

Unknown values may appear as arguments to functions, preventing simplification.

Give the syntax of 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 proves a theorem about an enumerated type for each possible ...

... constructor used to create that type.

Don't confused case with ...

... our hack Case and SCase.

What is the syntax of the induction tactic?

Just the same as the destruct tactic.

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.

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

Coq is choosing the wrong instance of a pattern to rewrite when you use the rewrite tactic.

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.