Chapter 3

Propositional logic

When we study logic we usually start with propositional logic, which introduces the basic logical connectives:

conjunction The logical operation **and** which is usually denoted as _∧_ or as _&_ as an operation on Bool.

disjunction The logical operation **and** which is denoted as _∨_ or as _|_ for Bool.

negation The logical operation **not** which is denoted as ¬ or as ! for Bool.

implication The logical operation **if** ... **then** ... which is written as _⇒_ but has not standard notation as an operation on booleans.

logical equivalence We often say **if, and only if** which is abbreviated as **iff**, this is written as ⇔. For Bool this just corresponds to equality on booleans which we will write _==_.

Propositional logic is usually the first step which is later followed by predicate logic which introduces the quantifiers **for all** ∀ and **exists** ∃. While propositional logic is simpler it is somehow abstract because we cannot say anything interesting because we cannot form interesting propositions yet. Hence we use logical variables to form propositions and study logical tautologies, that is propositions that are true for every assignment of truth values.

Before we explain the type theoretic approach using propositions as types, we review the classical semantics using Bool.

3.1 Boolean logic

What is a truth value or a proposition? The classical approach is to equate propositions with Bool which is a type with exactly two values true and false. We could use the operations on types we have introduced in the last section to

define Bool as $\top \uplus \top$ but we will just use the mechanism of datatypes and define:

```
data Bool : Set where
  true : Bool
  false : Bool
```

Now to define _&_ we use pattern matching:

```
_&_ : Bool → Bool → Bool
true & y = y
false & y = false
```

The idea is that if the first parameter is true then the result of true & y is just y, while if the first parameter is false then false & y == false for all y. We could have also used the 2nd parameter since the operation is commutative. Actually this is a tautology: x & y == y & x that is this expression is true for all possible values of x y : Bool. We can show this by drawing a truth table:

x	y	x & y	y & x	x & y == y & x
false	false	false	false	true
false	true	false	false	true
true	false	false	false	true
true	true	true	true	true

Actually I haven't yet defined the function _==_ but I leave this as an exercise.

Next we define the operation boolean or:

```
_|_ : Bool → Bool → Bool
true | y = true
false | y = y
```

The idea is similar as for &: if the first parameter is true we already know that true | y ≡ true but of the first parameter is false then false | y ≡ y. We leave the verification that _|_ is commutative as an exercise.

Already with _&_ and _|_ we can explore a number of laws, which are the laws of *boolean algebra*. As in arithmetic we have the law of distributivity x* (y+z) ≡ x*y + x*z which now becomes x & (y | z) ≡ x & y | x & z. When writing the law we already exploited the convention that _&_ binds stronger that _|_.

As above we can verify the law by drawing the truth table:

x	y	z	x & (y | z)	x & y | x & z	x & (y | z) == x & y | x & z
false	false	false	false	false	true
false	false	true	false	false	true
false	true	false	false	false	true
false	true	true	false	false	true
true	false	false	false	false	true
true	false	true	true	true	true
true	true	false	true	true	true
true	true	true	true	true	true

Actually we don't have to look at the whole truth table but we can show directly that x & (y | z) and x & y | x & z are equal. We only look at x we have two case x = true and x = false. In the case x = true we can reason for the left hand side:

$$\begin{aligned} \mathsf{x}\ \&\ (\mathsf{y} \mid \mathsf{z}) &= \mathsf{true}\ \&\ (\mathsf{y} \mid \mathsf{z}) && \text{using the definition of } \_\&\_ \\ &= \mathsf{y} \mid \mathsf{z} \end{aligned}$$

and for the right hand side:

$$\begin{aligned} (\mathsf{x} \mid \mathsf{y})\ \&\ (\mathsf{x} \mid \mathsf{z}) &= (\mathsf{true}\ \&\ \mathsf{y}) \mid (\mathsf{true}\ \&\ \mathsf{z}) && \text{using the definition of } \_\&\_ \text{ twice} \\ &= \mathsf{y}\ \&\ \mathsf{z} \end{aligned}$$

Hence both sides are identical. In the case x = false we can also show that both sides are identical:

$$\begin{aligned} \mathsf{x}\ \&\ (\mathsf{y} \mid \mathsf{z}) &= \mathsf{false}\ \&\ (\mathsf{y} \mid \mathsf{z}) && \text{using the definition of } \_\&\_ \\ &= \mathsf{false} \end{aligned}$$

and for the right hand side:

$$\begin{aligned} (\mathsf{x} \mid \mathsf{y})\ \&\ (\mathsf{x} \mid \mathsf{z}) &= (\mathsf{false}\ \&\ \mathsf{y}) \mid (\mathsf{false}\ \&\ \mathsf{z}) && \text{using the definition of \& twice} \\ &= \mathsf{false} \mid \mathsf{false} && \text{using the definition of } \mid \\ &= \mathsf{false} \end{aligned}$$

Other important equations are the de Morgan laws, which are basically the observation that the truth tables of _&_ and _|_ can be obtained from each other by switching true and false. To express this we need negation:

```
!_ : Bool → Bool
! true  = false
! false = true
```

The de Morgan laws now are:

x	y	! (x | y)	! x & ! y	! (x | y) == ! x & ! y
false	false	true	true	true
false	true	true	true	true
true	false	true	true	true
true	true	false	false	true

x	y	! (x & y)	! x | ! y	! (x & y) == ! x | ! y
false	false	true	true	true
false	true	false	false	true
true	false	false	false	true
true	true	false	false	true

Let us verify the 2nd one again by equational reasoning, again we only need to consider x.

x=true

$$\begin{aligned} \texttt{! (x \& y)} &= \texttt{! (true \& y)} \\ &= \texttt{! y} \end{aligned}$$

$$\begin{aligned} \texttt{! x \| ! y} &= \texttt{! true \| ! y} \\ &= \texttt{false \| ! y} \\ &= \texttt{!y} \end{aligned}$$

x=false

$$\begin{aligned} \texttt{! (x \& y)} &= \texttt{! (false \& y)} \\ &= \texttt{! false} \\ &= \texttt{true} \end{aligned}$$

$$\begin{aligned} \texttt{! x \| ! y} &= \texttt{! false \| ! y} \\ &= \texttt{true \| ! y} \\ &= \texttt{true} \end{aligned}$$

We will soon see how paper proofs like this can be done in Agda itself (see section 6.7 exercise 3).

3.2 Propositions as types

While the boolean semantics is quite standard it has some serious shortcomings. Can we really identify truth values with Bool? There are many propositions of which we don't know wether they are true or false hence we cannot assign a boolean value to them. The problem becomes obvious once we move on to predicate logic: a predicate over the natural number would be just a function P : ℕ → Bool but what is the truth value of the statement that P is true for all natural numbers ∀ x : ℕ → P x : Bool? We would have to check P for infinitely many inputs which is clearly not possible.

In Type Theory we use another definition of truth values, instead of saying that a proposition is either true or false, we say that a proposition is something

we can have evidence for. A proposition holds if we can find evidence. It isn't always clear wether we can find evidence for a proposition hence this is different from the boolean explanation. Indeed this interpretation is associated with *intuitionistic logic*, which has different laws than boolean algebra (it is called Heyting algebra).

To be precise we identify propositions with Set that is we assign to any proposition the type of evidence for it. [1]

```
prop = Set
```

We will use P Q R to stand for arbitrary propositions (i.e. elements of prop). What is evidence for P **and** Q that is P ∧ Q? Clearly it is a pair p , q of evidence p : P and evidence q : Q. That is:

```
_∧_ : prop → prop → prop
P ∧ Q = P × Q
```

Similarly, evidence for P **or** Q is the sum of evidence for P and Q:

```
_∨_ : prop → prop → prop
P ∨ Q = P ⊎ Q
```

Implication which doesn't play a central role in boolean logic is central in intuitionistic logic. What is evidence for P ⇒ Q? It is a function that transforms evidence for P to evidence for Q. Hence:

```
_⇒_ : prop → prop → prop
P ⇒ Q = P → Q
```

We also need to define the logical constants: True has a trivial proof, while False is the empty type, there is no evidence for it.

```
True : prop
True = ⊤
False : prop
False = ⊥
```

Having all these we can define negation by saying that ¬ P holds if P entails impossible:

```
¬ : prop → prop
¬ P = P ⇒ False
```

We also define logical equivalence as implications going in both direction, which is somehow obvious from the symbol used:

[1] Don't confuse this with Agda's proof-irrelevant Prop, which identifies all elements of a proposition. We will discuss proof-irrelevance later.

```
_⇔_ : prop → prop → prop
P ⇔ Q = (P ⇒ Q) ∧ (Q ⇒ P)
```

In the boolean setting we were able to use equality _≡_ for logical equivalence. This is not possible in the propositions as types explanation presented here. [2]

We use the usual assumptions how to read logical formulas:

- _∧_ binds stronger than _∨_.
- _∨_ binds stronger than _⇒_.
- _⇒_ binds stronger than _⇔_.
- _⇒_ is right associative (like _→_).

The following table summarizes our definitions of logical connectives:

latin	*english*	*logic*	*types / definitions*
Conjunction	And	P ∧ Q	P × Q
Disjunction	Or	P ∨ Q	P ⊎ Q
Implication	if-then	P ⇒ Q	P → Q
Verum	true	True	⊤
Falsum	false	False	⊥
Negation	not	¬ P	P ⇒ False
Equivalence	if-and-only-if	P ⇔ Q	(P ⇒ Q) ∧ (Q ⇒ P)

Let's revisit the tautologies we have shown using the boolean interpretation earlier. Certainly _∧_ is commutative, because _×_ is. We only need to show one direction because the other one follows by symmetry. We can show it easily by copatternmatching:

```
∧-comm→ : P ∧ Q ⇒ Q ∧ P
proj₁ (∧-comm→ x) = proj₂ x
proj₂ (∧-comm→ x) = proj₁ x
∧-comm : P ∧ Q ⇔ Q ∧ P
proj₁ ∧-comm = ∧-comm→
proj₂ ∧-comm = ∧-comm→
```

Since we have declared a constructor for _×_, namely _,_ we could have proven ∧-comm→ also by pattern matching:

```
∧-comm→ : P ∧ Q ⇒ Q ∧ P
∧-comm→ (p , q) = q , p
```

which is sometimes more convenient as the next example shows.

Let's prove that _∧_ distributes over _∨_, that is:

[2] However, we will recover this identification when we introduce Homotopy Type Theory later, which also uses a small modification of the translation presented here.

```
∧∨-distr : P ∧ (Q ∨ R) ⇔ (P ∧ Q) ∨ (P ∧ R)
```

Let's use the Agda system to construct the proof / program interactively. We start with

```
∧∨-distr = ?
```

Our first step is to use copattern matching to split _⇔_ which is defined as a conjunction in its two parts:

```
proj₁ ∧∨-distr = ?
proj₂ ∧∨-distr = ?
```

We first focus on the _⇒_ direction. This is a function hence let us introduce an assumption / parameter:

```
proj₁ ∧∨-distr x = ?
```

Our goal is to prove P ∧ Q ∨ P ∧ R and we can use x : P ∧ (Q ∨ R). Our next step is to analyze our assumption using pattern matching:

```
proj₁ ∧∨-distr (p , qr) = ?
```

We now have p : P and qr : Q ∨ R. We analyze qr further which means we get two cases:

```
proj₁ ∧∨-distr (p , inj₁ q) = ?
proj₁ ∧∨-distr (p , inj₂ r) = ?
```

Now we are cooking with gas. In the first case we have q : Q hence we can prove P ∧ Q ∨ P ∧ R. But first of all we have to decide wether we use inj_1 or inj_2. No proces for guessung which one:

```
proj₁ ∧∨-distr (p , inj₁ q) = inj₁ ?
```

Now we are left to prove P ∧ Q which is no problem with the assumptions lying around:

```
proj₁ ∧∨-distr (p , inj₁ q) = inj₁ (p , q)
```

The next line works the same way:

```
proj₁ ∧∨-distr (p , inj₂ r) = inj₂ (p , r)
```

At this point I am getting bored and just complete the program:

```
proj₁ ∧∨-distr (p , inj₁ q)    = inj₁ (p , q)
proj₁ ∧∨-distr (p , inj₂ r)    = inj₂ (p , r)
proj₂ ∧∨-distr (inj₁ (p , q)) = p , inj₁ q
proj₂ ∧∨-distr (inj₂ (p , r)) = p , inj₂ r
```

I hope you have been getting bored too. Indeed it shouldn't be difficult to write this little program without much interaction but I wanted to demonstrate how to do it step by step because in some cases this may be necessary.

In this instance the two functions we define to show a logical equivalence are actually inverse to each other, but this is not always the case. So for example:

$$\begin{aligned}&\wedge\text{-dbl} : P \Leftrightarrow P \wedge P\\ &\text{proj}_1\ \wedge\text{-dbl}\ p = p , p\\ &\text{proj}_2\ \wedge\text{-dbl}\ (p , \_) = p\end{aligned}$$

These two functions are not inverse, e.g. if we start with $2 , 3$ and go back and forth we end up with $2 , 2$.

Next we revisit the de Morgan laws, the first one is no problem:

$$\begin{aligned}&\text{dm}_1 : \neg (P \vee Q) \Leftrightarrow \neg P \wedge \neg Q\\ &\text{proj}_1\ \text{dm}_1\ f = (\lambda\, p \to f\,(\text{inj}_1\ p)) , \lambda\, p \to f\,(\text{inj}_2\ p)\\ &\text{proj}_2\ \text{dm}_1\ (g , h)\ (\text{inj}_1\ p) = g\ p\\ &\text{proj}_2\ \text{dm}_1\ (g , h)\ (\text{inj}_2\ q) = h\ q\end{aligned}$$

But for the 2nd one we run into a problem in one direction:

$$\begin{aligned}&\text{dm}_2 : \neg (P \wedge Q) \Leftrightarrow \neg P \vee \neg Q\\ &\text{proj}_1\ \text{dm}_2\ f = ?\\ &\text{proj}_2\ \text{dm}_2\ (\text{inj}_1\ f)\ (p , q) = f\ p\\ &\text{proj}_2\ \text{dm}_2\ (\text{inj}_2\ g)\ (p , q) = g\ q\end{aligned}$$

We cannot show $\neg (P \wedge Q) \Rightarrow \neg P \vee \neg Q$. When we attempt to show this we assume $f : \neg (P \wedge Q)$, and now we need to establish $\neg P \vee \neg Q$. The problem is that we have no information which case we should choose. E.g. if we choose to show $\neg P$ we assume $p : P$ and we need to derive False. The only possibility is to use f but this requires $P \wedge Q$ as the input and we only have P.

After all this should not be surprising: We may know that not both propositions can be false but we don't know which one it is. E.g. if somebody tells us that it is not true that both it rains and we go to the zoo, then we still don't know wether it doesn't rain or we don't go to the zoo or both.

So indeed while the 2nd de Morgan law holds in the boolean semantics it doesn't hold intuitionistically (it is not a law of Heyting algebra).

3.3 Classical principles

So what is the exact difference between the boolean logic which is also called *classical logic*, and the evidence based logic which is called intuitionistic logic. It seems that certain propositions, or actually propositional schemes since they involved propositional variables, which are provable classically are not provably intuitionistically. But what exactly is the difference?

It turns out that the extra power of classical logic is exactly the assumption that every proposition is either true or false. This is expressed as the following

scheme which is called the *law of the excluded middle* or in latin *tertium non datur* (the third is not given). We define

```
TND : prop → prop
TND P = P ∨ ¬ P
```

This means that we write TND P to express that excluded middle holds for P. Classical logic corresponds to assuming that TND holds for all propositions.

Let's go back to the instance of the de Morgan law which we couldn't prove. It turns out that we can prove it using TND:

```
dm₂₁tnd : TND P → ¬ (P ∧ Q) ⇒ ¬ P ∨ ¬ Q
dm₂₁tnd (inj₁ p) f = inj₂ (λ q → f (p , q))
dm₂₁tnd (inj₂ np) f = inj₁ np
```

The idea is that If P holds then we can prove ¬ P because from assuming Q we can derive a contradiction using the premise ¬ (P ∧ Q). On the other hand if ¬ P holds we are done using just this.

We can see that using TND we can simulate truth tables: this proof corresponds to the algebraic proof for booleans we have done in section 3.1.

There is an alternative principle we can use, which we will see is equivalent to TND. This is the *principle of indirect proof*, also called *reductio ad absurdo* (reduction to the absurd) in latin. This you see in many mathematical proofs: to prove a proposition P we assume ¬ P and derive a contradiction. Since deriving a contradiction is the same as negation this just means that ¬ (¬ P) implies P:

```
RAA : prop → prop
RAA P = ¬ (¬ P) → P
```

We can show that we can derive RAA from TND using the same technqiue we have used for the de Morgan law above:

```
tnd→raa : TND P → RAA P
tnd→raa (inj₁ p) nnp = p
tnd→raa (inj₂ np) nnp = case⊥ (nnp np)
```

We assume nnp : ¬ (¬ P). Now we analyze P using tnd: if it is p : P we are done. On the other hand if we have np : ¬ P we can derive a contradiction nnp np : False and then use case⊥ to conclude.

The other direction is somehow more interesting: while we cannot prove P ∨ ¬ P in general, we can show that it cannot be false. That is we can prove ¬ (¬ (P ∨ ¬ P)) [3]

```
nntnd : ¬ (¬ (P ∨ ¬ P))
nntnd npnp = npnp (inj₂ (λ p → npnp (inj₁ p)))
```

[3] This should be called the principle of the excluded middle: it states that it is impossible that a proposition is neither true nor false.

It is worthwhile to step through this proof. Ok we assume npnp : ¬ (P ∨ ¬ P) to derive False. The only option now is just to use npnp which means we need to show P ∨ ¬ P. There is no chance that we could prove P hence lets try ¬ P. Ok we assume p : P and have to derive False again and again there is no choice but to use npnp and what dejavu we are back proving P ∨ ¬ P. But this time we actually know p : P hence a we have to do is to use this assumption.

Now we can just use RAA (P ∨ ¬ P) to derive TND P from nntnd:

```
raa→tnd : RAA (P ∨ ¬ P) → TND P
raa→tnd raa = raa nntnd
```

So if we assume that RAA P for all propositions P then we can also show TND Q for all propositions, even though it is not the same one. In this sense the two principles are equivalent.

3.4 The negative translation

We may wonder when exactly is a classical tautology provable intuitionistically, i.e. with the propositions as types interpretation, and when not. We have seen that one of the de Morgan rules ¬ (P ∨ Q) ⇔ ¬ P ∧ ¬ Q is provable intuitionistically but for the other one only one direction ¬ P ∨ ¬ Q ⇒ ¬ (P ∧ Q) is provable but the other direction ¬ (P ∧ Q) ⇒ ¬ P ∨ ¬ Q required classical reasoning.

The difference is evidence for an intuitionistic disjunction P ∨ Q contains information, namely which of the P and Q actually holds. However, in the 2nd de Morgan rule ¬ (P ∧ Q) merely states that it cannot be that both P and Q hold while ¬ P ∨ ¬ Q tells us which of the two actually doesn't hold and we cannot create this information from nothing.

We call propositions that contain information *positive* and the ones that do not *negative*. We will see that we can understand classical logic as the logic of negative propositions and that negative propositions are the one for which RAA holds.

First of all we show that any negated proposition is negative

```
neg-is-neg : RAA (¬ P)
```

which expands to showing that ¬ (¬ (¬ P)) → ¬ P which may be slightly surprising because in general we don't have ¬ (¬ P) → P in general. But it makes now sense because both sides are negative, while in the later case we can replace P by a positive proposition.

We can show this directly, but maybe it is more instructive to do this a bit more slowly. We can show that every proposition implies its double negation:

```
nn : P → ¬ (¬ P)
nn p np = np p
```

Indeed this is just simple application. We can also show the following principle:

$$\mathsf{neg\text{-}impl} : (\mathsf{P} \to \mathsf{Q}) \to \neg\,\mathsf{Q} \to \neg\,\mathsf{P}$$

It is always good to do a little intuitive check: If we know *If the sun shines we go to the zoo* then *If we don't go to the zoo the sun doesn't shine.* Here is the proof:

$$\mathsf{neg\text{-}impl}\ \mathsf{f}\ \mathsf{nq}\ \mathsf{p} = \mathsf{nq}\ (\mathsf{f}\ \mathsf{p})$$

Now we can show neg-is-neg by just combining the two results:

$$\mathsf{neg\text{-}is\text{-}neg} = \mathsf{neg\text{-}impl}\ \mathsf{nn}$$

We can also show that conjunction and implication preserve negativity (for implication we only need that the conclusion is negative):

$$\mathsf{and\text{-}neg} : \mathsf{RAA}\ \mathsf{P} \to \mathsf{RAA}\ \mathsf{Q} \to \mathsf{RAA}\ (\mathsf{P} \wedge \mathsf{Q})$$
$$\mathsf{impl\text{-}neg} : \mathsf{RAA}\ \mathsf{Q} \to \mathsf{RAA}\ (\mathsf{P} \to \mathsf{Q})$$

I leave this as an exercise.

The only connective that is not negative is disjunction. Classically we can show that $\mathsf{P} \vee \mathsf{Q}$ is equivalent to the negative $\neg\ (\neg\ \mathsf{P} \wedge \neg\ \mathsf{Q})$ because $\mathsf{P} \vee \mathsf{Q} \Leftrightarrow \neg\ (\neg\ (\mathsf{P} \vee \mathsf{Q}))$ and using de Morgan $\neg\ (\mathsf{P} \vee \mathsf{Q}) \Leftrightarrow \neg\ \mathsf{P} \wedge \neg\ \mathsf{Q}$. Hence we can define a classical disjunction:

$$\_ \vee^c \_ : \mathsf{prop} \to \mathsf{prop} \to \mathsf{prop}$$
$$\mathsf{P} \vee^c \mathsf{Q} = \neg\ (\neg\ \mathsf{P} \wedge \neg\ \mathsf{Q})$$

What can we do with $\_ \vee^c \_$? We can actually prove the injections easily:

$$\mathrm{inj}_1^c : \mathsf{P} \to \mathsf{P} \vee^c \mathsf{Q}$$
$$\mathrm{inj}_2^c : \mathsf{Q} \to \mathsf{P} \vee^c \mathsf{Q}$$

But what about $\mathrm{case}^c : (\mathsf{P} \to \mathsf{R}) \to (\mathsf{Q} \to \mathsf{R}) \to \mathsf{P} \vee^c \mathsf{Q} \to \mathsf{R}$? It turns out that we can derive it, if we assume that R is negative:

$$\mathrm{case}^c : \mathsf{RAA}\ \mathsf{R} \to (\mathsf{P} \to \mathsf{R}) \to (\mathsf{Q} \to \mathsf{R}) \to \mathsf{P} \vee^c \mathsf{Q} \to \mathsf{R}$$

Now using classical disjunction we can actually prove excluded middle:

$$\mathrm{tnd}^c : \mathsf{P} \vee^c \neg\,\mathsf{P}$$

Having inj_1 , inj_2, case^c and tnd^c together with all the standard principles for the remaining negative connectives we can simulate any classical proof. Any propositional variable quantifies only over negative propositions, that is when we assume $\mathsf{P} : \mathsf{prop}$ we need to add $\mathsf{raap} : \mathsf{RAA}\ \mathsf{P}$. So basically the negative translation provides an alternative to the boolean explanation of classical logic. However, it has a big advantage, because as we will see it also works for predicate logic where the boolean explanation fails.

Hence classical logic can be understood as the logic of negative propositions. Hence we can summarise that *a classical logician is somebody who just can't say anything positive.*

3.5 History

Boolean algebra was invented by George Boole in the 19th century who also gave his name to the type of booleans or short Bool. However, the laws of boolean algebra don't play a big role in our presentation. Truth tables were already used in the 19th century first by Peirce and they were popularized by Wittgenstein and Post in the early 20th century.

Intuitionistic logic was introduced by Brouwer in the 1920ies who also had a big fight with Hilbert about the use of the excluded middle. This was called the *Grundlagenstreit* (Argument about foundations).[4] Hilbert complained:

> Taking the principle of excluded middle from the mathematician would be the same, say, as proscribing the telescope to the astronomer or to the boxer the use of his fists. To prohibit existence statements and the principle of excluded middle is tantamount to relinquishing the science of mathematics altogether.

It is fair to say that Hilbert won, at least as far as mainstream Mathematics is concerned. But don't forget this was before the invention of computers.

It was Brouwer's student Heyting who made the formal rules of intuitionistic logic precise and who also came up with a constructive semantics for it, the Brouwer-Heyting-Kolmogorov interpretation which is closely related to the propositions as types explanation but it uses untyped *realizers*.

The propositions as types explanation is also called the *Curry-Howard Isomorphism*: [5] they observed an analogy between the rules of logic and the rules of typed λ-calculus. I prefer calling it the propositions as types explanation because where does logic come form in the first place? I think we should use the idea that propositions can be identified with the type of their evidence to justify the rules of logic. In this way the semantics comes first and the formal rules are just a reflection of the semantics. While this is not historically correct, I think this is a better way to look at this.

The negative translation which is also called the double negation translation goes back to Gödel and Gentzen. In the context of functional programming it is also known as the continuation passing style transformation (CPS transform) which provides a translation for control operators like call-cc (*call with current continuation*).

3.6 Exercises

1. Define the operations

```
implb : Bool → Bool → Bool
_==_ : Bool → Bool → Bool
```

[4] I recommend van Dalen's biography of Brouwer which also includes lots of details about this famous disagreement. [VD99]

[5] It was first observed by Curry in a letter in 1934 and later refined by Howard in 1969 but this was only published in 1980 [How80].

Here implb is the boolean version of implication. There are given by the following truth tables:

P	Q	implb P Q	P == Q
false	false	true	true
false	true	true	false
true	false	false	false
true	true	true	true

2. if_then_else_ is a basic combinator on Bool. It is defined as follows

```
if_then_else : Bool → A → A → A
if true then y else z = y
if false then y else z = z
```

Define all the logical operations

```
_&_ : Bool → Bool → Bool
_|_ : Bool → Bool → Bool
!_ : Bool → Bool
implb : Bool → Bool → Bool
_==_ : Bool → Bool → Bool
```

in agda using only if_then_else (i..e without using pattern matching).

3. Prove the following equivalences. It is helpful to use some auxilliary statements to be able to use pattern matching more easily.

   All propositions in this part are provable.

```
e1 : (P → Q ∧ R) ⇔ ((P → Q) ∧ (P → R))
e2 : (P ∨ Q → R) ⇔ ((P → R) ∧ (Q → R))
```

4. We are playing *logic poker* today.

   Try to prove the following propositions using Agda.

```
P1  = {P Q : prop} → (P ⇒ Q) ⇒ ¬ P ⇒ ¬ Q
P2  = {P Q : prop} → (P ⇒ Q) ⇒ ¬ Q ⇒ ¬ P
P3  = {P Q : prop} → (P ⇒ Q) ⇒ ¬ P ∨ Q
P4  = {P Q : prop} → ¬ P ∨ Q ⇒ (P ⇒ Q)
P5  = {P : prop} → ¬ (P ⇔ ¬ P)
P6  = {P : prop} → ¬ (P ∨ ¬ P)
P7  = {P : prop} → ¬ (¬ (¬ P)) ⇒ ¬ P
P8  = {P Q : prop} → ((P ⇒ Q) ⇒ P) ⇒ P
P9  = {P : prop} → (¬ (¬ P) ⇒ P) ⇒ P ∨ ¬ P
P10 = {P : prop} → P ∨ ¬ P ⇒ (¬ (¬ P) ⇒ P)
```

   Consider the following cases:

(a) You can prove it directly (intuitionistically). E.g. you have

```
Pa = {P : prop} → P → P
```

in this case you just provide a proof:

```
pa : Pa
pa x = x
```

(b) You cannot prove it with intuitionistic logic but you can prove it with classical logic.
E.g. if I ask you to prove

```
Pb = {P Q : prop} → ¬ (P ∧ Q) → ¬ P ∨ ¬ Q
```

and you realize that you cannot prove this intutionistically but you can prove it with classical logic that is using RAA P for some proposition P : prop. I define

```
CLASS = {P : prop} → RAA P
```

and you prove

```
pb : CLASS → Pb
```

When you prove pa you get an extra parameter raa which you can use in the proof:

```
pb raa h =
  raa (λ x → x (inj₁ (λ p → x (inj₂ (λ q → h (p , q))))))
```

If you need TND P you can use raa→tnd.

(c) It isn't even true in classical logic. In this case you should find a counterexample. That is you should be able to instantiate the propositional variables with ⊤ and ⊥ and prove the negation.

```
Pc = {P Q : prop} → P → Q
```

but you realize that this is false by instantiating P = ⊤ and Q = ⊥. Ok, in this case you prove

```
pc : Pc → ⊥
pc h = h {⊤} {⊥} tt
```

5. Complete the missing proofs from the negative translation (section 3.4):

```
and-neg : RAA P → RAA Q → RAA (P ∧ Q)
impl-neg : RAA Q → RAA (P → Q)
```

$$\begin{array}{l}
\mathrm{inj}_1^c \;:\; \mathsf{P} \;\to\; \mathsf{P} \;\vee^c\; \mathsf{Q} \\
\mathrm{inj}_2^c \;:\; \mathsf{Q} \;\to\; \mathsf{P} \;\vee^c\; \mathsf{Q} \\
\mathrm{case}^c \;:\; \mathsf{RAA}\,\mathsf{R} \;\to\; (\mathsf{P} \;\to\; \mathsf{R}) \;\to\; (\mathsf{Q} \;\to\; \mathsf{R}) \;\to\; \mathsf{P} \;\vee^c\; \mathsf{Q} \;\to\; \mathsf{R} \\
\mathrm{tnd}^c \;:\; \mathsf{P} \;\vee^c\; \neg\,\mathsf{P}
\end{array}$$

Hint: It is useful to prove that double negation is monotone:

$$(\mathsf{P} \;\to\; \mathsf{Q}) \;\to\; \neg\,(\neg\,\mathsf{P}) \;\to\; \neg\,(\neg\,\mathsf{Q})$$