Negation

Terminology

- The typical logical symbol for negation is ¬
  - This can be pronounced as “not”
- Other symbols can be used
- For example $\neg p$ can be represented as:
  - $\neg p$, $\sim p$, $!\,p$, $\bar{p}$, $-p$, or p'
- We will just use $\neg p$
- In Lean, you can use \not or \neg

Stating a negation

- Propositions can be true *or* false (or indeterminate!)
- One way to prove a proposition is *false*, is to prove its negation is true
- $\neg P$ is thus shorthand for $P \rightarrow$ false **— remember this!!**
- This means two things:
  1. A proof of $\neg P$ is a function that takes a proof of P and returns a proof of false
  2. The proposition $\neg P$ is an implication, with P as the antecedent (premise) and false as the consequent (conclusion)
     - Remember that we can use assume to assume the antecedent when this is a goal

What is a negation in constructive logic?

- As we've already said, ¬P is shorthand for P → false

```
theorem same{P: Prop}: (¬P) = (P → false) := rfl
```

- A proof that P → false necessarily means that there can be no (valid) proof for P.
- So negation of P means that we can prove that there is no proof of P.

Inequality (not equals)

- 0 ≠ 1 is just different notation for ¬ 0 = 1
  - ≠ can be written with \ne or \neq
    - On a Mac, ≠ can also be written with [option]+=, and ¬ can be written with [option]+l (that's a lower-case L)

```
theorem zneqo : 0 ≠ 1.
#check zneqo
```

- Woah, a period? This means that Lean just *knows* it is true.

```
theorem zneqoeqzneqo : (0 ≠ 1) = ¬(0 = 1) := rfl
```

Disjointness of constructors

- Zero is created by the “base” constructor for naturals
  - #reduce nat.zero
- One is created by using the successor of zero
  - #reduce nat.succ(0)
- Two is the successor of 1
  - #reduce nat.succ(nat.succ(0))

```
theorem zneqo' : 0 = 1 → false :=
    λ h : (0 = 1),
        nat.no_confusion h
```

Assume-show-from proof pattern

- `Assume` works on an implication, either explicit or implicit
  - `Assume` assumes the antecedent (the left-hand side of the implication)
  - New goal is now the consequent (the right-hand side)
- `show <arg>` finds the first goal matching `<arg>`. This goal now becomes the main goal, after unification (a topic we will discuss later)
- `from` is synonymous with `exact`, but is useful for demonstrating an assume/show/from pattern.

```
theorem zneqo'' : ¬ 0 = 1 :=
begin
    assume h : (0 = 1),
    show false,
    from nat.no_confusion h
end
```

We prove that 0 ≠ 1 by assuming 0 = 1 and by showing that this assumption leads to a contradiction. As that is impossible, there must be no such proof of 0 = 1. That proves ¬ 0 = 1, i.e., 0 ≠ 1.

Disjointedness with booleans

```
theorem ttneqff : ¬tt = ff :=
begin
    assume h : (tt = ff),
    show false,
    from bool.no_confusion h
end
```

How does this work?

How else could we have proved it?

```
theorem ttneqff' : tt ≠ ff.
```

Exercises

- EXERCISE: Is it true that "Hello, Lean!" ≠ "Hello Lean!"? Can you prove it? If so, how? If not, why not?

```
theorem ex1 : "Hello, Lean!" ≠ "Hello Lean!" :=
begin
    assume h : ("Hello, Lean!" = "Hello Lean!"),
    show false,
    from string.no_confusion h
end
theorem ex1': "Hello, Lean!" ≠ "Hello Lean!”.
```

- EXERCISE: What about 2 ≠ 1?

```
theorem ex2 : 2 ≠ 1.
```

Proof of negation

- To derive ¬P:
  - show that from an assumption of (a proof of P) some kind of contradiction that cannot occur would follow, and
  - thus a proof of false would follow, leading to
  - the conclusion that there must be no proof of P, that it isn't true, and that ¬P therefore is true.
  - This is called "proof by negation."

```
theorem proof_by_negation : ∀(P : Prop),
    (P → false) → ¬P :=
        λ P p, p
```

Another proof that $0 \neq 1$

```
lemma zneqo''': ¬(0 = 1) :=
begin
    apply proof_by_negation,
    assume h: (0 = 1),
    show false,
    from (nat.no_confusion h)
end
```

Compare to:

```
theorem proof_by_negation : ∀ P : Prop,
     (P → false) → ¬P :=
         λ P p, p
```

Proving Q and not Q is false

- Something cannot be both true and not true

```
theorem qAndNotQfalse{P Q: Prop}
  (pf: Q ∧ ¬Q): false :=
    pf.right pf.left
```

- `¬Q` is an implication that `Q → false`
- What do we get when we apply that implication to Q?

Non-contradiction

- The principle of non-contradiction says that a proof of any proposition, Q, and also of its negation, ¬ Q, gives rise to a contradiction.
  - Therefore such a contradiction cannot arise.
- Now consider the proof of the negation:

```
theorem no_contra :
∀(Q: Prop), ¬(Q ∧ ¬Q) :=
    λ (Q : Prop) (pf : Q ∧ ¬Q),
        pf.right pf.left
```

- Exercise: discuss how this proof works

Non-contradiction application

- Now that we've created our no_contra theorem, we can use it:

```
theorem ncab{a b: nat}: ¬((a = b) ∧ (a ≠ b)) :=
begin
    apply no_contra
end
```

- See what happens if you add a third variable, *c*, and replace one *b* in the theorem with a *c*

Manual non-contradiction proof by steps

```
theorem ncab' : ¬((a = b) ∧ (a ≠ b)) :=
begin
  assume c : ((a = b) ∧ (a ≠ b)),
  have pf_eq := c.left,
  have pf_neq := c.right,
  have f := pf_neq pf_eq,
  assumption
end
```

Negation elimination

- Does ¬¬P equal P?
- Classically, yes
- Not in constructive logic, though
  - Why not?!?
  - Consider the proposition, “the word heterological is homological”
    - We can show that this proposition cannot be proven
    - This does not mean we can prove its opposite
    - In fact, we can show that we cannot prove its opposite

Double negative elimination

```
theorem double_neg_elim: ∀{P: Prop}, ¬¬P → P :=
begin
  assume P : Prop,
  assume pfNotNotP : ¬ ¬ P,
  cases (em P) with pf_P pf_not_P,
    show P, from pf_P,

    have f: false := pfNotNotP pf_not_P,
    exact false.elim f
end
```

Application

- Derive P by double negation elimination

```
theorem prove_P: ∀{P: Prop}, ¬¬P → P :=
  λ(P)(pf_not_not_P),
    double_neg_elim pf_not_not_P
```

Proof by contradiction

- Proof by contradiction has us assume the opposite of what we want to prove and show that it is false
- I.e., assume “not P” and show it has a false truth judgment

```
theorem proof_by_contradiction : ∀(P : Prop),
  (¬P → false) → P :=
    @double_neg_elim
```

- The `@` here turns off type inferencing for this one reference to `double_neg_elim`. It is a detail here. We'll discuss `@` later.

Application

```
theorem zeqz : 0 = 0 :=
begin
  apply proof_by_contradiction,
  assume pf: 0 = 0 → false,
  show false,
  from pf (eq.refl 0)
end
```

Classical proof by contradiction

```
example{P Q: Prop}
  (pf: ¬P → (Q ∧ ¬ Q)): P :=
begin
  apply proof_by_contradiction,
  assume notP: ¬P,
  have contra := (pf notP),
  show false,
  from no_contra Q contra
end
```

Proof by contrapositive

- $(\neg Q \rightarrow \neg P) \rightarrow (P \rightarrow Q)$
- *Very* similar to modus tollens

```
theorem proof_by_contrapositive:
  ∀(P Q : Prop), (¬Q → ¬P) → (P → Q) :=
begin
  assume P Q: Prop,
  assume pf_not_Q_to_not_P: (¬Q → ¬P),
  assume pf_P : P,
  have pf_not_Q_to_false: ¬Q → false :=
    λ(pf_not_Q: ¬Q),
      no_contra P (and.intro pf_P (pf_not_Q_to_not_P pf_not_Q)),
  have pf_not_not_Q: ¬¬Q := pf_not_Q_to_false,
  show Q,
  from double_neg_elim pf_not_not_Q
end
```

Application

```
theorem zeqz' : 0 = 0 → true :=
begin
  apply proof_by_contrapositive,
  assume nt : ¬true,
  have pff := nt true.intro,
  show ¬ 0 = 0,
  from false.elim pff
end
```

Exercise

- Does it appear that one needs to use proof by contradiction (and thus classical, non-constructive, reasoning) to prove that the square root of two is irrational?
- One general proof structure:
  - Assume $\sqrt{2}$ is rational
  - Thus it can be represented as $\sqrt{2} = a/b$, with a and b relatively prime
  - Multiply both sides by b to get $b\sqrt{2} = a$
  - Square both side to get $2b^2 = a^2$
  - If is a^2 multiple of 2, then a must also be a multiple of 2, so a^2 must ber a multiple of 4
  - Thus b^2 must be a multiple of 2, so they cannot be relatively prime (since both have 2 as a divisor)

Negation relations

- Does it make sense to ask if negation is reflexive, symmetric, or transitive?
  - No, those relations require binary operators, and negation is *unary*
- Does it make sense to ask if negation is total?
  - Yes, not all unary functions are total

Fin