SIGPL Summer School — Exercises

August 18 – 20, 2009
`http://sigpl.or.kr/school/2009s/`

1 Basic commands and tactics/tacticals

Commands

- `Section` and `End`.
- `Variable` and `Variables`.
- `Theorem` and `Lemma`.
- `Proof` and `Qed`.
- `check`, `print`, and `inspect`.

Tactics

- `intro` and `intros`.
- `apply`.
- `assumption` and `exact`.
- `split`.
- `left` and `right`.
- `elim`.
- `auto`, `trivial`, and `tauto`. (Don't use!)
- `cut`.
- `clear`.

Tacticals

- T_1`;`T_2 (T_1 then T_2).
- T`;[`T_1`|`T_2`|`$\cdots$`|`T_3`]`.

The following table shows tactics corresponding to inferences rules in the natural deduction system.

	`->`	`/\`	`\/`	`True`	`False`	`~`
Introduction	`intro`, `intros`	`split`	`left`, `right`			`intro`
Elimination	`apply`	`elim`	`elim`		`elim`	`elim`

Complete PartOne in the Coq script.

2 Negation and classical logic

Negation

Complete PartTwo (Negation) in the Coq script.

- `(A -> B) -> (~B -> ~A)`. This is known as a *contrapositive* in logic.
- `A -> ~~A`. If A is true, it is safe to assert that it is not that A is not true.
- `~~~A -> ~A`. Although `~~A -> A` is not true in general, `~~~A -> ~A` *is* true for any proposition `A`.
- `~~(~~A -> A)`. This is tantamount to saying that `~~A -> A` is not untrue.

Classical logic

Classical logic is a logic obtained by adding one of the following rules to constructive logic:

$$\frac{}{A \lor \neg A\ \mathit{true}}\ \mathsf{EM} \qquad \frac{}{\neg\neg A \supset A\ \mathit{true}}\ \mathsf{DNE} \qquad \frac{}{((A \supset B) \supset A) \supset A\ \mathit{true}}\ \mathsf{Peirce}$$

The rule EM, called *the law of excluded middle,* asserts that for any proposition A, either A *true* or $\neg A$ *true* must hold regardless of the existence of an actual proof. The rule DNE, called *the law of double-negation elimination,* asserts that if A cannot be false, it must be true. The rule Peirce, called *Peirce's law*, says that a proof of A *true* may freely assume $A \supset B$ *true* for an arbitrary proposition B. The three rules above are all equivalent to each other in that the addition of any of these rules renders the other two rules derivable.

Complete PartTwo (Classical logic) in the Coq script.

- $(A \lor \neg A) \supset (((A \supset B) \supset A) \supset A)$ *true*. That is, EM implies Peirce.
- $(((A \supset \bot) \supset A) \supset A) \supset (\neg\neg A \supset A)$ *true*. That is, Peirce implies DNE where we set $B = \bot$.
- $(\neg\neg(B \lor \neg B) \supset (B \lor \neg B)) \supset (B \lor \neg B)$ *true*. That is, DNE implies EM where we set A in DNE to $B \lor \neg B$.

3 Proof terms

Previously we exploited tactics and tacticals of Coq to prove theorems in propositional logic. Since proof terms are compact representations of proofs, we can translate all these proofs into corresponding proof terms. In fact, we can just use the Coq command `Print` to displays all such proof terms. For example, we can print the proof term for `A -> A` once we complete its proof using tactics as follows:

```
Coq < Theorem id : A -> A.
1 subgoal

  ============================
   A -> A

...

id < Qed.
intro x.
assumption.
id is defined

Coq < Print id.
id = fun x : A => x
     : A -> A
```

In order to use a proof term in proving a theorem, we use the command `Definition`. For example, we can define `id` by directly providing a proof term for it as follows:

```
Coq < Definition id : A -> A := fun x : A => x.
id is defined

Coq < Print id.
id = fun x : A => x
     : A -> A
```

`Definition` uses the following syntax:

`Definition` ⟨*identifier*⟩ `:` ⟨*proposition*⟩ `:=` ⟨*proof term*⟩`.`

Proof terms for propositional logic in Coq use slightly different syntax from the simply typed λ-calculus. The following table shows how to convert proof terms in the simply typed λ-calculus into Coq:

Simply-typed λ-calculus	Coq
$\lambda x\!:\!A.\,M$	`fun` x `:` A `=>` M
$\lambda x\!:\!A.\,\lambda y\!:\!B.\,M$	`fun (`x `:` A`) (`y `:` B`) =>` M
$\lambda x\!:\!A.\,\lambda y\!:\!B.\cdots\lambda z\!:\!C.\,M$	`fun (`x `:` A`) (`y `:` B`)` $\cdots$ `(`z `:` C`) =>` M
$M\ N$	$M\ N$
(M, N)	`conj` M N
$\mathsf{fst}\ M$ where $M : A \wedge B$	`and_ind (fun (p :` A`) (q :` B`) => p)` M
$\mathsf{snd}\ M$ where $M : A \wedge B$	`and_ind (fun (p :` A`) (q :` B`) => q)` M
$\mathsf{inl}_A\ M$	`or_introl` A M
$\mathsf{inr}_A\ M$	`or_intror` A M
$\mathsf{case}\ M\ \mathsf{of\ inl}\ x.\,N_1 \mid \mathsf{inr}\ y.\,N_2$ where $M : A \vee B$	`or_ind (fun` x `:` A `=>` N_1`) (fun` y `:` B `=>` N_2`)` M
$()$	`I`
$\mathsf{abort}_C\ M$	`False_ind` C M

Note that Coq provides just a single term `and_ind` for eliminating conjunction, which can be thought of as combining the two elimination rules for conjunction. To see how `and_ind` works, `Check` it out!

```
Coq < Check and_ind.
and_ind
     : forall A B P : Prop, (A -> B -> P) -> A /\ B -> P
```

Also `Check` out other terms such as `conj`, `or_introl`, `or_intror`, `or_ind`, and `False_ind`.

Complete PartThree in the Coq script.

4 First-order logic

Tactics for universal and existential quantifications

The following table shows tactics for universal and existential quantifications in first-order logic:

	$\forall$ (`forall`)	$\exists$ (`exists`)
Introduction	`intro`	`exists`
Elimination	`apply`, `apply ... with` *term₁* *term₂* $\cdots$ *termₙ*	`elim`

Here is my Coq program transcribing the proofs of the following examples given in the supplementary notes:

$(\forall x.A \wedge B) \supset (\forall x.A) \wedge (\forall x.B)\ \mathit{true}$
$\exists x.\neg A \supset \neg\forall x.A\ \mathit{true}$
$\forall y.(\forall x.A) \supset (\exists x.A)\ \mathit{true}$

```
Section FirstOrder.
Variable Term : Set.
Variables A B : Term -> Prop.

Theorem forall_and :
(forall x : Term, A x /\ B x) -> (forall x : Term, A x)  /\ (forall x : Term, B x).
Proof.
intro w.
split; (intro a; elim (w a); intros; assumption).
Qed.

Theorem exist_neg : (exists x : Term, ~ A x) -> (~ forall x : Term, A x).
Proof.
intro w; intro z; elim w; intros a y; elim y; apply z.
Qed.

Theorem not_weird : forall y : Term, (forall x : Term, A x) -> (exists x : Term, A x).
Proof.
intro a; intro w; exists a; apply w.
Qed.
End FirstOrder.
```

First we declare a set `Term` which we will use as the set of terms:

```
Variable Term : Set.
```

We do not actually specify elements of the set `Term` because pure first-order logic does not assume a particular set of terms.

Next we declare two propositions `A` and `B`:

```
Variables A B : Term -> Prop.
```

`A` and `B` are both given type `Term -> Prop` to indicate that they are parameterized over elements of the set `Term`, or terms. What this means in practice is that if `A` contains a term variable `x`, we write `A x`, which has type `Prop`, for the proposition. Note that all term variables in my Coq program are assigned type `Term` so that they can be used as arguments to `A` and `B`.

Sets, propositions, and types

You might well be confused about the differences between `Set` for sets, `Prop` for propositions, and `Type` for types in Coq. To tell the truth, these are all types *and also* terms in Coq — what a convoluted system it is! For now, we only need the following facts. The invariant is that *everything in Coq has its type!*

- A proof term M, or equivalently a proof, has a certain type A, and we call A a proposition. So we have a relation $M : A$.
- A proposition A has type `Prop`, and we call `Prop` a *sort* in order to differentiate it from types in the general sense. So we have a relation A : `Prop`, which literally says that A belongs to the set `Prop` of propositions.
- A term t has a certain type τ, and we call τ a datatype. So we have a relation $t : \tau$.
- A datatype τ has type `Set`, and we also call `Set` a *sort* in order to differentiate it from types in general sense. So we have a relation τ : `Set`, which literally says that τ belongs to the set `Set` of datatypes.
- Both `Type` and `Set` have type `Type`!

We can summarize the above relations as follows:

term t	:	datatype τ	:	`Set`	:	`Type`
proof term M	:	proposition A	:	`Prop`	:	`Type`

Declarations and definitions

The following table shows how to declare term variables with only their datatypes, and how to define term variables with terms as well as their datatypes. Global declarations and definitions are exported to the outside of sections (beginning with `Section` and ending with `End`), while local declarations and definitions are not.

	declaration	definition
global	`Parameter v :` τ, `Parameters`	`Definition c :` τ `:=` t`.`
local	`Variable v :` τ, `Variables`	`Let c :` τ `:=` t`.`

It turns out that these definitions and declarations can be used not only for terms but also for proof terms and even for datatypes and propositions! For example, we have seen an example of declaring a datatype like

```
Variable Term : Set.
```

or declaring a proposition like

```
Variable P : Prop.
```

For proofs and proof terms, Coq provides the following specialized forms for declarations and definitions. An opaque definition hides its proof M and makes only `H` and A visible for later use. A transparent definition makes visible its proof M as well. If you do not understand what the difference is, just use opaque definitions in your Coq program and you will never run into trouble!

	declaration	definition
global	`Axiom H :` A (`Parameter H :` A — not recommended)	`Lemma H :` A`. Proof` M`.` — opaque `Theorem H :` A`. Proof` M`.` — opaque (`Definition H :` A `:=` M`.` — transparent, not recommended)
local	`Hypothesis H :` A, `Hypotheses` (`Variable H :` A — not recommended)	`Let H :` A `:=` M`.` — transparent

`apply`, `elim`, and `exact`

So far, we have used only variables or labels as arguments to these tactics. In general, their arguments can be proof terms as long as they have proper types. Here are a few examples.

- `apply (Ltn O (S O)).`
  Instead of specifying a label, we use a proof term `Ltn O (S O)`.
- `elim (EM (exists x,   P x)).`
  Instead of specifying a label, we use a proof term `EM (exists x,   P x)`
- `exact (Eqi a).`
  Instead of specifying a label, we use use a proof term `Eqi a`.

Properties of natural numbers

We use the following axioms to characterize natural numbers.

$$\frac{}{Nat(\mathbf{0})\ true}\ Zero \qquad \frac{}{\forall x.Nat(x) \supset Nat(\mathbf{s}(x))\ true}\ Succ$$

$$\frac{}{\forall x.Eq(x,x)\ true}\ Eq_i \qquad \frac{}{\forall x.\forall y.\forall z.(Eq(x,y) \wedge Eq(x,z)) \supset Eq(y,z)\ true}\ Eq_t$$

$$\frac{}{\forall x.Lt(x,\mathbf{s}(x))\ true}\ Lt_s \qquad \frac{}{\forall x.\forall y.Eq(x,y) \supset \neg Lt(x,y)\ true}\ Lt_\neg$$

We translate these axioms into Coq declarations as follows:

```
Variable Term : Set.

Variable O : Term.
Variable S : Term -> Term.

Variable Nat : Term -> Prop.
Variable Eq : Term -> Term -> Prop.
Variable Lt : Term -> Term -> Prop.

Hypothesis Zero : Nat O.
Hypothesis Succ : forall x : Term, Nat x -> Nat (S x).
Hypothesis Eqi : forall x : Term, Eq x x.
Hypothesis Eqt : forall (x : Term) (y : Term) (z: Term), (Eq x y /\ Eq x z) -> Eq y z.
Hypothesis Lts : forall x : Term, Lt x (S x).
Hypothesis Ltn : forall (x : Term) (y : Term), Eq x y -> ~ Lt x y.
```

Complete PartFour in the Coq script:

$$\forall x.Nat(x) \supset (\exists y.Nat(y) \wedge Eq(x,y)) \ \mathit{true}$$
$$\forall x.\forall y.Eq(x,y) \supset Eq(y,x) \ \mathit{true}$$
$$\neg\exists x.Eq(x,\mathbf{0}) \wedge Eq(x,\mathbf{s}(\mathbf{0})) \ \mathit{true}$$

More properties of natural numbers

Complete PartFour in the Coq script:

$$\forall x.Nat(x) \supset Nat(\mathbf{s}(\mathbf{s}(x))) \ \mathit{true}$$
$$\forall x.\forall y.Lt(x,y) \supset \neg Eq(x,y) \ \mathit{true}$$
$$\neg\exists x.\exists y.Eq(x,y) \wedge Lt(x,y) \ \mathit{true}$$

5 Inductive datatypes and equality

Here is a summary of the commands and tactics that you need. Examples of using these commands and tactics are also given.

Commands

- `Fixpoint` facilitates defining primitive recursive functions.

```
Fixpoint plus (m n:nat) struct m : nat :=
match m with
| O => n
| S m' => S (plus m' n)
end.
```

- `Inductive` allows us to define inductive datatypes. Later we will use `Inductive` to define inductive predicates.

```
Inductive nat : Set :=
| O : nat
| S : nat -> nat.
```

Tactics

- `rewrite`
  `rewrite e` requires `e` to be of type `forall (x1:T1) (x2:T2) ... (xn:Tn), a = b`. Then applying `rewrite e` to a goal of the form `P(a)` rewrites it as `P(b)`.
  `rewrite Heq`
  `rewrite <- plus_n_O`
  `rewrite -> plus_n_O` (which is equivalent to `rewrite plus_n_O`)
  `rewrite <- (plus_n_O n0)`
  `rewrite -> (plus_n_O n0)` (which is equivalent to `rewrite (plus_n_O n0)`)

- `replace`
  `replace e with e'` replaces `e` in the current goal by `e'` and creates a new goal `e' = e`.
  `replace (f 1) with 0`
  `replace (f 1) with (f 0)`

- `reflexivity` (not in the Coq Tutorial)
  Applying this tactic to a goal of `t1 = t2` immediately completes the proof if `t1` and `t2` can be converted to each other (*e.g.*, `6*6=9*4`).

- `symmetry` (not in the Coq Tutorial)
  Applying this tactic to a goal of `t = s` changes the goal to `s = t`.

- `unfold`
  `unfold x` expands `x` into its definition.
  `unfold subset`
  `unfold element at 1`
  `unfold element in H`

- `red`
  `red` unfolds the head occurrence of the current goal.

- `simple induction`
  `simple induction` is an abbreviation of `intro; elim`. When applied to a goal of the form `forall x:T, A(x)`, it creates new subgoals according to the definition of type `T`.
  `simple induction n`

- `simpl`
  `simpl` simplifies terms in the current goal using the definition of its subterms. For example, it simplifies `plus O n` to `n`.
  `simpl`
  `simpl plus`
  `simpl plus at 1`

- `change`
  If the current goal can be converted to a term `e`, `change e` changes the current goal to `e`.
  `change (Is_S O)`
  `change False with (Is_S O)`
  `change False at 2 with (Is_S O)`

- `discriminate`
  Applying this tactic to a hypothesis of the form `a = b` immediately completes the proof if `a` and `b` cannot be converted to each other.

Primitive recursion

Use the `Fixpoint` command to implement the following functions as primitive recursive functions (PartFive in the Coq script).

- `plus2 : nat -> (nat -> nat)`
  `plus2` m returns a function `f` such that `f` n returns $m+n$.

- `double : nat -> nat`
  `double` m returns $2*m$.

- `mult : nat -> nat -> nat`
  `mult` m n returns $m*n$. You may use `plus` in its definition.

- `sum_n : nat -> nat`
  `sum_n` n returns $\Sigma_{i=0}^{n} i$. You may use `plus` in its definition.

Properties of `plus`

Prove the following lemmas in Coq (PartFive in the Coq script). *Do not use the* ***auto*** *tactic or any similar tactic.*

```
Lemma plus_n_0 : forall n:nat, n = plus n O.
```

```
Lemma plus_n_S : forall n m:nat, S (plus n m) = plus n (S m).
```

```
Lemma plus_com : forall n m:nat, plus n m = plus m n.
```

```
Lemma plus_assoc : forall (m n l:nat), plus (plus m n) l = plus m (plus n l).
```

Proving $2*\Sigma_{i=0}^{n} i = n + n*n$

Prove the following lemmas in Coq (PartFive in the Coq script). *Do not use the* ***auto*** *tactic or any similar tactic.*

```
Theorem sum_n_plus : forall n:nat, double (sum_n n) = plus n (mult n n).
```

Your proof may use any lemma from the previous part. You will need to introduce extras lemmas to complete the proof. The sample solution, for examples, introduces three lemmas, one of which is:

```
Lemma double_plus2 : forall n:nat, double n = plus n n.
```

6 Inductive predicates

Commands

- `Inductive` allows us to define inductive datatypes.

  ```
  Inductive nat : Set :=
  | O : nat
  | S : nat -> nat.
  ```

  This definition automatically declares a term `nat_ind` of the following type:

  ```
  nat_ind
       : forall P : nat -> Prop,
         P O -> (forall n : nat, P n -> P (S n)) -> forall n : nat, P n
  ```

  Be sure to understand that `nat_ind` corresponds to the rule natE_I given in the supplementary notes.

- `Inductive` also allows us to define inductive predicates.

```
Inductive eq : nat -> nat -> Prop :=
| eq_O : eq O O
| eq_S : forall (m n:nat), eq m n -> eq (S m) (S n).
```

This definition is a transcription of the definition of the predicate $m =_{\mathsf{N}} n$ given in the supplementary notes. It also automatically declares a term `eq_ind` of the following type:

```
eq_ind
     : forall P : nat -> nat -> Prop,
       P O O ->
       (forall m n : nat, eq m n -> P m n -> P (S m) (S n)) ->
       forall n n0 : nat, eq n n0 -> P n n0
```

Be sure to understand that `eq_ind` corresponds to the rule $=_{\mathsf{N}}\mathsf{E}_I$ given in the supplementary notes. Also remember that we can use `eq_O` and `eq_S` as terms of the types specified in the above definition.

Section 2.2 of the Coq Tutorial gives an example of an inductive predicate `le` (a parameterized inductive type in the Coq terminology), which might be challenging to understand at first reading. Section 1.3.3 (Inductive definitions) of the Coq Reference Manual might be more helpful where you can find a simpler example of inductive datatype `even : nat -> Prop`.

```
Inductive even : nat -> Prop :=
| even_O : even O
| even_SS : forall n:nat, even n -> even (S (S n)).

even_ind
     : forall P : nat -> Prop,
       P O ->
       (forall n : nat, even n -> P n -> P (S (S n))) ->
       forall n : nat, even n -> P n
```

This part uses another inductive predicate `lt` whose definition is a transcription of the definition of the predicate $m < n$ given in the supplementary notes.

```
Inductive lt : nat -> nat -> Prop :=
| lt_O : forall n:nat, lt O (S n)
| lt_S : forall (m:nat) (n:nat), lt m n -> lt (S m) (S n).

lt_ind
     : forall P : nat -> nat -> Prop,
       (forall n : nat, P O (S n)) ->
       (forall m n : nat, lt m n -> P m n -> P (S m) (S n)) ->
       forall n n0 : nat, lt n n0 -> P n n0
```

Tactics

- `inversion`
  In Coq, you give only introduction rules and not elimination rules because Coq provides the tactic `inversion`.
  Let us assume that `e` holds a proof of a predicate `A`. `inversion e` basically applies appropriate elimination rules to the predicate `A` and generates new hypotheses. Since elimination rules are all derived from introduction rules, we can think of `inversion e` as inverting the introduction rules to derive all the necessary conditions that should hold in order for the predicate `A` to be proved. Thus, whenever you need to apply an elimination rule to a judgment, you may need to consider this tactic.

  Here is an example:

```
Lemma test_inversion : forall (x y:nat), eq (S x) (S y) -> eq x y.
Proof.
intros x y H.
inversion H.
assumption.
Qed.
```

At the time when we apply the `inversion` tactic, we have `H : eq (S x) (S y)`. As we want to apply the elimination rule $=_{\mathsf{N}}\mathsf{E}_{\mathsf{s}}$ to `H`, we apply the `inversion` tactic, which will generate a new hypothesis of type `x = y`, which is the only necessary and sufficient condition for `eq (S x) (S y)` to hold. Try it yourself!

- `elim`
  The `elim` tactic can be applied to *any* term of an inductive type. For example, it may be applied to a term of type `nat` which is defined using the `Inductive` command, or to a term of type `eq m n` which is also defined using the `Inductive` command. (The reason that we can use this tactic extensively in propositional logic and pure first-order logic is that it is actually applied to a term whose type is inductively defined.)

  When applied to a term of an inductive type, the `elim` tactic applies the corresponding elimination rule based on induction after analyzing the current goal. For example, when applied to a term of type `nat`, it automatically applies `nat_ind`, or the rule $\mathsf{nat}\mathsf{E}_I$ in effect. Or when applied to a term of type `eq m n`, it automatically applies `eq_ind`, or the rule $=_{\mathsf{N}}\mathsf{E}_I$ in effect. So, in order to learn how this tactic works, you want to understand the two kinds of elimination rules based on induction that are given in the supplementary notes!

For this part, do not use the `auto` *tactic or any similar tactic.*

Examples from the supplementary notes

Prove the following lemmas in Coq (PartSix in the Coq script). For `exists_greater` and `eq_nat`, *do not use the* `elim` *and* `induction` *tactics.* For others, you may use the `elim` and `induction` tactics.

```
Lemma lt_one_two : lt (S O) (S (S O)).

Lemma no_lt_zero : forall (m:nat), ~(lt m O).

Lemma exists_greater : forall (x:nat), exists y:nat, lt x y.
(* use nat_ind; do not use elim/induction. *)

Lemma exists_greater' : forall (x:nat), exists y:nat, lt x y.
(* may use elim/induction. *)

Lemma eq_nat : forall x:nat, eq x x.
(* use nat_ind; do not use elim/induction. *)

Lemma eq_nat' : forall x:nat, eq x x.
(* may use elim/induction. *)

Lemma eq_trans : forall (x y z:nat), eq x y -> eq y z -> eq x z.

Lemma eq_succ : forall x:nat, ~(eq x O) -> exists y:nat, eq (S y) x.
```

Inductive predicates

Assume the following inductive predicate `le` (standing for "less than or equal to"), and prove the following lemmas in Coq (PartSix in the Coq script). For `le_n_S` and `lt_le`, *do not use the* `elim` *and* `induction` *tactics.* For others, you may use the `elim` and `induction` tactics.

```
Inductive le : nat -> nat -> Prop :=
| le_n : forall n, le n n
| le_S : forall (m n:nat), le m n -> le m (S n).

Lemma le_zero : forall n:nat, le O n.

Lemma le_n_S : forall n m:nat, le n m -> le (S n) (S m).
(* use le_ind; do not use elim/induction. *)

Lemma lt_le : forall (m n:nat), lt m n -> le m n.
(* use lt_ind; do not use elim/induction. *)

Lemma lt_le' : forall (m n:nat), lt m n -> le m n.
(* may use elim/induction. *)
```

Less-than-or-equal-to means less-than or equal-to.

Prove the following theorem in Coq (PartSix in the Coq script). You may introduce a few lemmas if needed. You may use the `elim` and `induction` tactics.

```
Theorem le_lt_eq : forall (m n:nat), le m n -> lt m n \/ eq m n.
```

Another definition of less-than

Here is a copy of the definition of `le` from Section 2.2 of the Coq Tutorial:

```
Inductive le' (n:nat) : nat -> Prop :=
| le_n' : le' n n
| le_S' : forall m:nat, le' n m -> le' n (S m).
```

Show that `le` given above and `le'` are logically equivalent. You may use the `elim` and `induction` tactics.

```
Lemma le_le' : forall (m n:nat), le m n -> le' m n.

Lemma le'_le : forall (m n:nat), le' m n -> le m n.
```

7 Strings of matched parentheses

We use the following inference rules to prove the two theorems shown below:

$$\frac{}{\epsilon \ \mathsf{mparen}}\ Meps \qquad \frac{s \ \mathsf{mparen}}{(s) \ \mathsf{mparen}}\ Mpar \qquad \frac{s_1 \ \mathsf{mparen} \quad s_2 \ \mathsf{mparen}}{s_1 \ s_2 \ \mathsf{mparen}}\ Mseq$$

$$\frac{}{\epsilon \ \mathsf{lparen}}\ Leps \qquad \frac{s_1 \ \mathsf{lparen} \quad s_2 \ \mathsf{lparen}}{(s_1) \ s_2 \ \mathsf{lparen}}\ Lseq$$

Theorem 7.1. *If s mparen, then s lparen.*

Theorem 7.2. *If s lparen, then s mparen.*

We provide a definition for strings of parentheses (`S`) and a function for concatenating two strings of parentheses (`concat`). Your task is to define two inductive judgments `mparen` and `lparen` according to the inference rules shown above, and to give proofs of theorems `mparen2lparen` and `lparen2mparen`.

```
Inductive E : Set :=
| LP : E
| RP : E.
```

```
Inductive S : Set :=
| eps : S
| cons : E -> S -> S.

Fixpoint concat (s1 s2:S) {struct s1} : S :=
match s1 with
| eps => s2
| cons e s2' => cons e (concat s2' s2) end.

Inductive mparen : S -> Prop := ...

Inductive lparen : S -> Prop := ...

Theorem mparen2lparen : forall s:S, mparen s -> lparen s.

Theorem lparen2mparen : forall s:S, lparen s -> mparen s.
```

You may introduce additional lemmas to simplify the proof. You may also need to prove some properties of `concat`, *e.g.*, `concat s eps = s`. Feel free to introduce any auxiliary definitions that are necessary to complete the proofs. All that I care about is your definitions of `mparen` and `lparen` and your proofs of `mparen2lparen` and `lparen2mparen`.

8 Complete induction

We have learned the principle of complete induction, which appears to be more powerful than mathematical induction, but turns out to be a derived notion. In this part, you will give a proof of the principle of complete induction in Coq. The goal is to give a proof of the theorem `nat_complete_ind` shown below:

```
Inductive nat : Set :=
| O : nat
| S : nat -> nat.

Inductive lt : nat -> nat -> Prop :=
| lt_O : forall n:nat, lt O (S n)
| lt_S : forall (m:nat) (n:nat), lt m n -> lt (S m) (S n).

Variable P : nat -> Prop.

Theorem nat_complete_ind :
  P O -> (forall n:nat, (forall z:nat, lt z n -> P z) -> P n) -> forall x:nat, P x.
```

The theorem can be written in our notation as follows:

$$P(\mathbf{0}) \supset (\forall n \in \mathsf{nat}.(\forall z \in \mathsf{nat}.z < n \supset P(z)) \supset P(n)) \supset \forall x \in \mathsf{nat}.P(x) \ \mathit{true}$$

Here are a few hints that you might find useful.

- Remember that complete induction is a principle derived from mathematical induction. This implies that your proof should contain an application of `nat_ind` somewhere.
- Then the whole problem boils down to finding an appropriate predicate, say `A n` where `n` is a natural number, for the application of `nat_ind`. Then this application of `nat_ind` will prove `forall n:nat, A n`. This is the key part of your proof.
- `A n` should *not* be `P n`. Instead you have to generalize the goal statement so that `forall n:nat, A n` would *imply* `forall n:nat, P n`. Letting `A n = P n` will fail!

- Before starting to write a proof in Coq, try to find a mathematical proof. Without a solid understanding of how the proof works, it might be very difficult to complete the proof in Coq in an interactive manner. That is, Coq helps you a lot especially when you know how to complete the proof yourself.
- You can simplify the presentation by explicitly defining the predicate `A`, as in:

  ```
  Let A : nat -> Prop := fun k:nat => ...
  ```

- This is a line copied directly from the sample solution:

  ```
  apply (nat_ind A A0 (Aind H)).
  ```

- You will have to prove some properties of `lt`.