Learning to use the on-line help:

- Select the menu Help and then the menu Contents or press F1

In the menu you can find the syntax for lambda terms and the syntax and semantics of every tactic and tactical available in the system

Learning to type Unicode symbols:

- Unicode symbols are written like this: \lambda \eta \leq ...
- Optional: to get the glyph corresponding to the Unicode symbol, type Alt+L (for ligature) just after the \something stuff
- Additional ligatures (use Alt+L to get the glyph)
  := for `def`
  -> for `\to`
  => for `\Rightarrow`
  <= for `\leq`
  >= for `\geq`

- Commonly used Unicode symbols:
  \to for logical implication and function space
  \forall for universal quantifier
  \exists for existential quantifier
  \Pi for product quantifier
  \lambda for lambda abstraction
  \land for logical and
  \lor for logical or
  \bot for logical false

How to set up the environment:

- Every file must start with a line like this:
  set "baseurl" "cic:matita/nat+/".

that says that every definition and lemma in the current file will be put in the clic:matita/nat+/namespace.
For an exercise put in a foo.ma file, use the namespace cic:matita/foo/

- Files can start with inclusion lines like this:
  include "nat/plus.ma".

This is required to activate the notation given in the nat/plus.ma file. If you do not include "nat/times.ma", you will still be able to use all the definitions and lemmas given in "nat/plus.ma", but without the nice infix /\ notation for addition.

How to browse and search the library:

- Open the menu View and then New CIC Browser. You will get a browser-like window with integrated searching functionalities.
- To explore the library, type the URI "cic:" in the URI field and start browsing. Definitions will be rendered as such. Theorems will be rendered in a declarative style even if initially produced in a procedural style.
- To get a nice notation for addition and natural numbers, put in your script include "nat/plus.ma" and execute it. Then use the browser to render clic:matita/nat+/plus/associative_plus.con. The declarative proof you see is not fully expanded. Every time you see a "Proof" or a "proof of xxx" you can click on it to expand the proof. Every constant and symbol is a hyperlink. Follow the hyperlinks to see the definition of natural numbers and addition.
- The home button visualizes in declarative style the proof under development. It shows nothin when the system is not in proof mode.
- Theorems and definitions can be looked for by name using wildcards. Write

**associative** in the search bar, select "locate" and press Enter.
You will see the result list of URIs. Click on them to follow the hyperlink.
- If you know the exact statement of a theorem, but not its name, you can write its statement in the search bar and select match. Try with
  \forall n, m + n = m + n. Sometimes you can find a theorem that is just close enough to what you were looking for. Try with
  \forall n, m + 0 = 0. (0 is the letter O, not the number 0).
- Sometimes you may want to obtain an hint on what theorems can be applied to prove something. Write the statement to prove in the search bar and select Hint. Try with "S O + S 0 = 0 + S O". As before, you can get some useful results using abstraction tactics. If you want to instantiate a given statement, write the statement in the search bar using lambda abstraction in front to quantify the variables to be instantiated. Then use Instance. Try with "\forall n. \forall x. n \times x + m = x."

How to define things:

- Look in the manual for Syntax and then Definitions and declarations.
  Often you can omit the types of binders if they can be inferred. Use question marks "?" to ask the system to infer an argument of an application. Non recursive definitions must be given using "definition".
- Structural recursive definitions must be given using "let rec".

Try the following examples:

axiom f : nat \to nat
definition square := \lambda A:Type. \lambda f:A \to A. \lambda x:A. f (f x).
definition square_f : nat \to nat \to square ? f.
inductive tree (A:Type) : Type \def Empty: tree A |
  Cons: A \to tree A \to tree A.
let rec size (A:Type) (t: tree A) on t \def
  match t with
  Empty \Rightarrow 0 |
  Cons l r \Rightarrow size ? l + size ? r.

How to prove things:

- Example: Notary proofs can be done by directly writing the lambda-terms (as in Agda or Epigram). Try to complete the following proofs:

  lemma ex1:
  \forall A,B:Prop. \forall x:Prop.X \to Y \to A \to B)
  \lambda \mathrm{\forall x:Prop.X \to Y \to A \to B}
  \lambda H:
  \lambda H.

  lemma ex2:
  \forall n,m. m + n = m + (n + 0) \def

  Hint: to solve ex2 use eq_n and plus_n_0. Look for their types using the browser.

- The usual way to write proofs is by using either the procedural style (as in Coq and Isabelle) or the still experimental declarative style (as in Lean and Mizar). Let's start with the declarative style.

Look in the manual for the following declarative tactics:

  assume id: type (= new assumption)
  suppose formula (id) (= new hypothesis)
  by \lambda-term done (* intermediate step *)
  by _ done (* intermediate step *)
  by _ we proved formula (id) (* intermediate step *)

Declarative tactics must always be terminated by a dot. When automation fails (last two tactics), you can always help the system by adding new intermediate steps or by writing the lambda-term by hand.
Prove again ex1 and ex2 in declarative style. A proof in declarative style starts with

```
lemma id: formula.
theorem id: formula.
```

(the two forms are totally equivalent) and ends with

```
qed.
```

Hint: you can select well-formed sub-formulae in the sequents window, copy them (using the Edit/Paste menu item or the contextual menu item) and paste them in the text (using the Edit/Copy menu item or the contextual menu item).

* The most used style to write proofs in Matita is the procedural one.

In the rest of this tutorial we will only present the procedural style.

Look in the manual for the following procedural tactics:

```
intros.
apply lambda-term.
autobatch (* in the manual autobatch is called auto *)
```

Prove again ex1 and ex2 in procedural style. A proof in procedural style starts and ends as a proof in declarative style. The two styles can be mixed.

* Some tactics open multiple new goals. For instance, copy the following lemma:

```
lemma ex3: \forall A,B:Prop. A \to B \to (A \land B) \land (A \land B). 
intros split;
```

Look for the split tactic in the manual. The split tactic of the previous script has created two new goals, both of type \( (A \land B) \). Notice that the labels 78 and 79 of both goals are now in bold. This means that both goals are currently active and that the next tactic will be applied to both goals. The \"1\" tactical used after \"intros\" and \"split\" has exactly this meaning: it activates all goals created by the previous tactic.

Look for it in the manual, then execute \"split\" again. Now you can see four active goals. The first and third one ask to prove \( A \) and \( A \land B \). If you activate the first goal with \"split\" then the remaining ones ask to prove \( B \). To apply different tactics to the selected goal, we need to branch over the selected goals. This is achieved by using the tactical \"[\]\" (branching). Now type \"[\]\" and exec it. Only the first goal is now active (in bold), and all the previously active goals have now subscripts ranging from 1 to 4. Use the \"apply \( A \)\" tactical to solve the goal.

No goals are now selected. Use the \"[\]\" (next goal) tactical to activate the next goal. Since we are able to solve the new active goal and the last goal at once, we want to select the two branches at the same time. Use the \"2,4 [\]\" tactical to select the goals having as subscripts 2 and 4. Now solve the goals with \"apply \( A \)\" and select the last remaining goal with \"[\]\". Solve the goal with \"apply \( B \)\". Finally, close the branching section using the tactical \"[\]\" and complete the proof with \"qed\".

Look for all this tactics in the manual. The \"1\" tactical is also useful: it is used just after a \"[\] or \"[\]\" tactical to activate all the remaining goals with a subscript (i.e. all the goals in the innermost branch).

If a tactic \"T\" opens multiple goals, then \"T\" activates all the new goals opened by \"T\". Instead \"1\" just activates the first goal opened by \"T\", postponing the remaining goals without marking them with subscripts.

In case of doubt, always use \"[\" in declarative scripts and only all the other tactics in procedural scripts.

```
-------------------------------
Computationary:
-------------------------------
```

* State the following theorem:

```
lemma ex4: \forall n,m. S (S n) + m = S (S (n + m)).
```

and introduce the hypotheses with \"intros\". To complete the proof, we can simply compute \( S (S n) + m = S (S (n + m)) \) to obtain \( S (S (n + m)) \). Using the browser (click on the \"\#\" hyperlink), look at the definition of addition: since addition is defined by recursion on the first argument, and since the first argument starts with two constructors \"S\", computation can be made. Look for the \"simplify\" tactic in the manual and use it to obtain a trivial equality. Solve the equality using \"reflexivity\", after having looked for it in the manual.

* State the following theorem:

```
lemma ex5: \forall n,m. n + S (S m) = S (n + m).
```

Try to use simplify to complete the proof as before. Why is \"simplify\" not useful in this case? To progress in the proof we need a lemma stating \( \forall n,m. S (n + m) = n + S m \). Using the browser, look for its name in the library. Since the lemma states an equality, it is possible to use it to replace an instance of its left hand side with an instance of its right hand side (or the other way around) in the current sequent. Look for \"rewrite\" tactic in the manual, and use it to solve the exercise. There are two possible solutions: one only uses rewriting from left to right \("rewrite \)\", the other rewriting from right to left \("rewrite *\")

It may happen that \"simplify\" fails to yield the simplified form you expect. In some situations, simplify can even make your goal more complex. In these cases you can use the \"change\" tactic to convert the goal into any other goal which is equivalent by computation only. State again exercise ex4 and solve the goal without using \"simplify\" by means of \"change with \( S (S n) + m = S (S (n + m)) \)\"

\[ \begin{align*}
\text{Simplify does nothing to expand definitions that are not given by structural recursion. To expand definition } & \text{X} \text{ in the goal, use the } \text{unfold X} \text{ tactic.}
\end{align*}\]

State the following lemma and use \"unfold Not\" to unfold the definition of negation in terms of implication and False. Then complete the proof of the theorem.

```
lemma ex6: \forall A:Prop. \not A \to \not A \to False.
```

* Sometimes you may be interested in simplifying, changing, unfolding or even substituting (by means of rewrite) only a sub-expression of the goal. Nothing you may be interested in simplifying, changing, unfolding or substituting a (sub-)expression of one hypothesis. Look in the manual for these tactics: all of them have an optional argument that is a pattern. You can generate a pattern by: 1) selecting the sub-expression you want to act on in the sequent; 2) copying it (using the Edit/Copy menu item or the contextual menu); 3) pasting it as a pattern using the \"rewrite as pattern\" menu item. Other tactics also have pattern arguments.

State and solve the following exercise:

```
lemma ex7: \forall n. (n + 0) + (n + 0) = n + (n + 0).
```

The proof of the lemma must rewrite the conclusion of the sequent to \( n + (n + 0) = n + (n + 0) \) and prove it by reflexivity.

Hint: use the browser to look for the theorem that proves \( \forall n. n + 0 = n \) and then use a pattern to control the behaviour of \"rewrite\".

```
-------------------------------
Proofs by induction:
-------------------------------
```

* Functions can be defined by structural recursion over arguments whose type is inductive. To prove properties of these functions, a common strategy is to proceed by induction over the recursive argument of the function. To proceed by induction over an inductive argument \"X\", use the \"elim X\" tactic.

Now include \"nat/orders.ma\" to activate the notation \\eqref{nat}. Then state and prove the following lemma by induction over \( n \):

```
lemma ex8: \forall n,m. m \leq n + m.
```

* Hint 1: use \"autobatch\" to automatically prove trivial facts

* Hint 2: \"autobatch\" never performs computations. In inductive proofs you often need to \"simplify\" the inductive step before using...
"autobatch". Indeed, the goal of proceeding by induction over the recursive argument of a structural recursive definition is exactly that of allowing computation both in the base and inductive cases.

- Using the browser, look at the definition of addition over natural numbers. You can notice that all the parameters are fixed during recursion, but the one we are recurring on. This is the reason why it is possible to prove a property of addition using a simple induction over the recursive argument. When other arguments of the structural recursive functions change in recursive calls, it is necessary to proceed by induction over generalized predicates where the additional arguments are universally quantified.

Give the following tail recursive definition of addition between natural numbers:

```haskell
let rec plus' n m = n \{def
match n with
| 0 \{Rightarrow m \} S n \{Rightarrow plus' n' (S m) \}.
```

Note that both parameters of plus’ change during recursion.

Now state the following lemma, and try to prove it copying the proof given for ex8 (that started with "intros; elim n;"):

```
lemma ex9: \forall n,m. n \leq O \to n = O.
```

Why is it impossible to prove the goal in this way?

- Now start the proof with "intros l;", obtaining the generalized goal \forall n,m. l \leq plus' n m, and proceed by induction on n using "elim n" as before. Complete the proof by means of simplification and autobatch. Why is it now possible to prove the goal in this way?

- Some predicates can be given as inductive predicates. In this case, remember that you can proceed by induction over the proof of the predicate. In particular, if \( H \) is a proof of False/And/Or/Exists, then "elim \( H \)" corresponds to False/And/Or/Exists elimination.

State and prove the following lemma:

```
lemma ex10: \forall A B. Prop. A \lor (False \land B) \to A.
```

Proofs by inversion:

- Some predicates defined by induction are really defined as dependent families of predicates. For instance, the \( \leq \) relation over natural numbers is defined as follow:

```
inductive le (n: nat) : nat \to Prop \{def
le_0 n = le n 0 |
le_S m n = le m n \to le (S m) n.
```

In Matita we say that the first parameter of le is a left parameter (since it is at the left of the "\( \)" sign), and that the second parameter is a right parameter. Dependent families of predicates are inductive definitions having a right parameter.

Now, consider a proof \( H \) of \( (le n E) \) for some expression \( E \).

Differently from what happens in Agda, proceeding by elimination of \( H \) (i.e. doing an "elim \( H \)"") ignores the fact that \( le n E \) holds. Equivalently, eliminating \( H \) of type \( (le n E) \) and \( H' \) of type \( (le n E') \), you obtain exactly the same new goals even if \( E \) and \( E' \) are different.

State the following exercise and try to prove it by elimination of the first premisse (i.e. by doing an "intros; elim \( H \)").
Following the hyperlink, look at the type inferred for leb. What interpretation Matita chose for the first and second \ lor sign?
* The basic idea behind overloading of mathematical notations is the following:
  1. during pretty printing of formulae, the internal logical representation of
     mathematical notions is mapped to MathML Content (an infinitary XML
     based standard for the description of abstract syntax tree of mathematical
     formulae). E.g. both Or (a predicate former) and orb (a function over
     boolean) are mapped to the same MathML Content symbol "or".
  2. then, the MathML Content abstract syntax tree of a formula is mapped
     to concrete syntax in MathML Presentation (a finitary XML based standard
     for the description of concrete syntax trees of mathematical formulae).
     E.g. the ""or x y" abstract syntax tree is mapped to "x \lor y". The sequent
     window and the browser are based on a widget that is able to
     render and interact MathML Presentation.
  3. during parsing, the two phases are reversed: starting from the concrete
     syntax tree (which is in plain Unicode text), the abstract syntax tree
     in MathML Content is computed unambiguously. Then the abstract syntax tree
     is efficiently translated to every well-typed logical representation.
     E.g. "x \lor y" is first translated to "or x y" and then interpreted as
     "or x y" or "orb x y", depending on which interpretation finally yields
     well-typed lambda-terms.

* Using leb and cases analysis over booleana, define the two new non
  recursive predicates:

  min: nat \to nat \to nat
  max: nat \to nat \to nat

Now overload the \land notation (associated to the "\land x y" MathML
  Content formula) to work also for min:

interpretation "min of two natural numbers" \land x y =
  (cic/matita/exercise/min.con x y).

Note: you have to substitute "cic/matita/exercise/min.con" with the URI
determined by the baseurl you picked at the beginning of the file.

Overload also the notation for \lor (associated to "or x y") in the
  same way.

To check if everything works correctly, state the following lemma:

lemma foo: \forall b : n. (false \land b) = false \land (0 \land n) = 0.

How the system interpreted the instances of \land?

Now try to state the following ill-typed statement:

lemma foo: \forall b : n. (false \land 0) = false \land (0 \land n) = 0.

Click on the three error locations before trying to read the errors.
Then click on the errors and read them in the error message window
(just below the sequent window). Which error messages did you suspect?
Which ones make sense to you? Which error message do you consider to be
the "right" one? In what sense?
* Defining a new notation (i.e., associating to a new MathML Content tree
some MathML Presentation tree) is more involved.

Suppose we want to use the "a \middot b" notation for multiplication
between natural numbers. Type:

notation "hbbox(a break \middot b)" non associative with precedence 55
  for @[ "times $a \ $b"]

interpretation "times over natural numbers" "times x y =
  (cic/matita/nat/times/times.con x y).

To check if everything was correct, state the following lemma:

lemma foo: \forallall n. n \middot 0 = 0.

The "hbbox(a break \middot b)" contains more information than just
"a \middot b". The "hbbox" tells the system to write "a", "\middot" and

"b" in an horizontal row if there is enough space, or vertically otherwise.
The "break" keyword tells the system where to break the formula in case
of need. The syntax for defining new notations is not documented in the
manual yet.

Using notions without including them:

Using the browser, look for the "fact" function.

Notice that it is defined in the "cic/matita/nat/factorial" namespace
that has not been included yet. Now state the following lemma:

lemma fact_O_S_0: fact 0 = 1.

Note that Matita automatically introduces in the script some informations
to remember where "fact" comes from. However, you do not get the nice
notation for factorial. Remove the lines automatically added by Matita
and replace them with

include "nat/factorial.ma"

before stating again the lemma. Now the lines are no longer added and you
get the nice notation. In the future we plan to activate all notation without
the need of including anything.

Few relatively simple exercises:

1) Start an empty .ma file, include the following standard files

 include "nat/factorization.ma".
 include "list/list.ma".
 include "nat/iteration2.ma".
 include "Func/util.ma".

In particular, the following notations for lists and pairs are introduced:

[] is the empty list
in the list obtained putting a new element hd in
front of the list tl @
list concatenation

Write the body of the following function, that sums all the
elements of the list:

let rec sum (l : list nat) (accumulator : nat) on l :=
  match l with
  | nil => ...
  | cons x tl => ...
.

Now write the body of the function that given a number e generates the list
of length n containing only e as element.

let rec mkI (e : nat) on n :def
  match ... with
  | 0 => ...
  | S n1 => ...
.

1.1) Prove the following theorem:

\begin{verbatim}
  theorem sum_mkI_times : \forallall n m, sum (mkI n m) = n * m.
  Now define the function that given n generates the list
  n :: (n-1) :: ... :: 1 :: []
  let rec iota (n : nat) :=
    match n with
    | 0 => ...
    | S n1 => ...
  \end{verbatim}

1.2) Prove the following theorem (medium/hard exercise):

\begin{verbatim}
  theorem sum_iota_div : \forallall n. sum (iota n) = div (n * (S n)) (S (S 0)).
\end{verbatim}
Hints:
− search the library!
− give a look at decidable_lt, le_to_or_lt_eq, div_mod_spec_div_mod, div_plus_times

2) Start an empty .ma file including the following

```ocaml
include "nat/factorization.ma".
include "list/list.ma".
include "nat/iteration2.ma".
include "Psub/util.ma".
```

Define the following datatype

```ocaml
inductive tree : nat -> Type :=
  Leaf : tree 0
  Node : forall n. tree n -> tree n -> tree (S n).
```

Define the body of the depth function

```ocaml
let rec depth (n : nat) (t : tree n) on t : nat :=
  match t with [ Leaf => 0 | Node _ t1 t2 => ... ].
```

Define the body of the size function

```ocaml
let rec size (n : nat) (t : tree n) on t : nat :=
  match t with [ Leaf => 0 | Node _ t1 t2 => ... ].
```

2.1) Prove the following theorem

```ocaml
forall n.forall t:tree n. S (size t) = (S (S O)) sup n.
```

Define the balanced predicate by recursion

```ocaml
let rec balanced (n : nat) (t : tree n) on t : Prop :=
  match t with [ Leaf => True | Node _ t1 t2 => ... ]
```

2.2) Prove the following (easy, if you define a good balanced predicate):

```ocaml
forall n. forall t:(tree n). balanced ? t.
```

Define a new type treel, without the depath-in-type annotation.

```ocaml
inductive treel : Type :=
  Leaf1 : treel
  Node1 : treel -> treel -> treel.
```

2.3) Try to define depth, size, and balanced and then prove the same formula as before (medium difficulty):

```ocaml
forall t:treel. balanced t -> S (size t) = (S (S O)) sup (depth t)
```

3) Start from an empty .ma file, change the baseuri and include the following files for auxiliary notation:

```ocaml
include "nat/plus.ma".
include "nat/compare.ma".
include "list/sort.ma".
include "datatype/constructor.ma".
```

In particular, the following notations for lists and pairs are introduced:

```ocaml
[] is the empty list
hd : t1 is the list obtained putting a new element hd in front of the list tl
@ times is the cartesian product
\langle l, r \rangle is the pair (l,r)
```

Define an inductive data type of propositional formulae built from a denumerable set of atoms, conjunction, disjunction, negation, truth and falsity (no primitive implication).

Hint: complete the following inductive definition.

```ocaml
inductive Formula : Type \def
FTrue: Formula
FFalse: Formula
FAtom: nat \to Formula
FAnd: Formula \to Formula \to Formula
...
```

Define a classical interpretation as a function from atom indexes to booleans:

```ocaml
definition interp \def nat \to bool.
```

Define by structural recursion over formulas an evaluation function parameterized over an interpretation.

Hint: complete the following definition. The order of the different cases should be exactly the order of the constructors in the definition of the inductive type.

```ocaml
let rec eval (i:interp) F on F : bool \def
  match F with
  | FTrue \rightarrow true
  | FFalse \rightarrow false
  | FAtom n \rightarrow interp n ...
```

We are interested in formulas in a particular normal form where only atoms can be negated. Define the "being in normal form" not_nf predicate as an inductive predicate with one right parameter.

Hint: complete the following definition.

```ocaml
inductive not_nf : Formula \to Prop \def
NTrue: not_nf FTrue
NFalse: not_nf FFalse
NAtom: \forall n. not_nf (FAtom n)
...
NNot: \forall n. not_nf (FNot (FAtom n))
```

We want to describe a procedure that reduces a formula to an equivalent not_nf normal form. Define two mutually recursive functions elim_not and negate over formulas that respectively 1: put the formula in normal form and 2: put the negated of a formula in normal form.

Hint: complete the following definition.

```ocaml
let rec negate F \def
  match F with
  | FTrue \rightarrow FFalse
  | FFalse \rightarrow FTrue
  | FNot f \rightarrow elim_not \(elim_not f)
  ...
  | FNot f \rightarrow negate f
```

Why is not possible to only define elim_not by changing the FNot case to "FNot f \(elim_not (FNot f))"?

3.1) Prove that the procedures just given correctly produce normal forms. I.e. prove the following theorem.

```ocaml
theorem not_nf_elim_not:
forall F,not_nf (elim_not F) \land not_nf (negate F).
```
Why is not possible to prove that one function produces normal forms without proving the other part of the statement? Try and see what happens.

Hint: use the "ni,...,nm:" tactical to activate similar cases and solve all of them at once.

3.2) Finally prove that the procedures just given preserve the semantics of the formula. I.e. prove the following theorem.

\[\text{thm eq_eval_elim_not_eval:}\]

\[\\forall i, F.\\ \text{eval i (elim not F) = eval i F \land evalu (\neg \text{not} F)} = \text{eval i (F\text{Not} F)} .\]

Hint: you may need to prove (or assume axiomatically) additional lemmas on booleans such as the two demorgan laws.

==============================================================================
A moderately difficult exercise:
==============================================================================

4) Consider the inductive type of propositional formulae of the previous exercise. Describe with an inductive type the set of well types derivation trees for classical propositional sequent calculus without implication.

Hint: complete the following definitions.

definition sequent \def (list Formula) \to \text{Prop} \def

inductive derive: sequent \to Prop \def

ExchangeL: \forall l, l1, l2, f. derive \langle l1,l@1\rangle \to derive \langle l1@0,l\rangle \to derive \langle l1@0,l\rangle \to derive \langle f\rangle \to derive \langle f\rangle \to derive \langle f\rangle \to derive \langle f\rangle \to derive \langle f\rangle \to derive \langle f\rangle \to derive \langle f\rangle

ExchangeR: \forall l, l1, l2, f. derive \langle l1,l\rangle \to derive \langle l1,l\rangle \to derive \langle l1,l\rangle \to derive \langle l1,l\rangle \to derive \langle l1,l\rangle \to derive \langle l1,l\rangle \to derive \langle l1,l\rangle \to derive \langle l1,l\rangle \to derive \langle l1,l\rangle \to derive \langle l1,l\rangle

AndR: \forall l, l1, f1, f2. derive \langle l1,f1\rangle \to derive \langle l1,f2\rangle \to derive \langle l1,f\rangle \to derive \langle l1,f\rangle \to derive \langle l1,f\rangle \to derive \langle l1,f\rangle

Note that while the exchange rules are explicit, weakening and contraction are embedded in the other rules.

Define two functions that transform the left hand side and the right hand side of a sequent into a logically equivalent formula obtained by making the conjunction (respectively disjunction) of all formulae in the left hand side (respectively right hand side). From those, define a function that folds a sequent into a logically equivalent formula obtained by negating the conjunction of all formulae in the left hand side and putting the result in disjunction with the disjunction of all formulae in the right hand side.

Define a predicate is_tautology for formulae.

4.1) Prove the soundness of the sequent calculus. I.e. prove

\[\text{thm soundness:}\]

\[\\forall F. \text{derive F } \to \text{is_tautology (formula_of_sequent F)} .\]

Hint: you may need to axiomatically assume or prove several lemmas on booleans that are missing from the library. You also need to prove some lemmas on the functions you have just defined.

==============================================================================
A long and tough exercise:
==============================================================================

5) Prove the completeness of the sequent calculus studied in the previous exercise. I.e. prove

\[\text{thm completeness:}\]

\[\\forall S. \text{is_tautology (formula_of_sequent S) } \to \text{derive S}.\]

Hint: the proof is by induction on the size of the sequent, defined as the size of all formulae in the sequent. The size of a formula is the number of unary and binary connectives in the formula. In the inductive case you have to pick one formula with a positive size, bring it in front using the exchange rule, and construct the tree applying the appropriate elimination rules. The subtrees are obtained by inductive hypotheses. In the base case, since the formula is a tautology, either there is a False formula in the left hand side of the sequent, or there is a True formula in the right hand side, or there is a formula both in the left and right hand sides. In all cases you can construct a tree by applying once or twice the exchange rules and once the FalseL/TrueR/Axiom rule. The computational content of the proof is a search strategy.

The main difficulty of the proof is to proceed by induction on something other than the size of the sequent (made of a pair of lists). Moreover, from the fact that the size of the sequent is greater than 0, you need to detect the exact positions of a non atomic formula in the sequent and this needs to be done by structural recursion on the appropriate side, which is a list. Finally, from the fact that a sequent of size 0 is a tautology, you need to detect the False premise or the True conclusion or the two occurrences of a formula that form an axiom, excluding all other cases. This last proof is already quite involved, and finding the right inductive predicate is quite challenging.