

**Exercise 1** (Forward reasoning).

Define a tactic `fwd_modus_ponens` which repeatedly looks for two hypotheses  $H1 : A$ ,  $H2 : A \rightarrow B$  and replaces  $H2$  with  $H2' : B$ .

**Exercise 2** (Arithmetic expressions).

Consider the following inductive type of arithmetic expressions.

```
Inductive aexpr :=
| Nval : nat -> aexpr
| Avar : string -> aexpr
| Aplus : aexpr -> aexpr -> aexpr
| Amul : aexpr -> aexpr -> aexpr.

Definition state := string -> nat.
```

Variables (`Avar s`) are identified by their string names (`s`).

1. Define a function `aval : state -> aexpr -> nat` which evaluates arithmetic expressions.
2. Define a function `asimp : aexpr -> aexpr` which (recursively) simplifies arithmetic expressions by:
  - replacing `Aplus (Nval n1) (Nval n2)` with `Nval (n1 + n2)`,
  - replacing `Aplus (Nval 0) e` and `Aplus e (Nval 0)` with `e`,
  - replacing `Amul (Nval n1) (Nval n2)` with `Nval (n1 * n2)`,
  - replacing `Amul (Nval 0) e` and `Amul e (Nval 0)` with `0`,
  - replacing `Amul (Nval 1) e` and `Amul e (Nval 1)` with `e`.

For example:

```
asimp (Aplus (Nval 3) (Amul (Nval 0) (Avar "a"))) = Nval 3
```

3. Prove: `forall s e, aval s (asimp e) = aval s e`.

*Hint.* The imports from the standard library you may need are: `String`, `Bool`, `Arith`. Try to use `sauto` as much as possible. It may be helpful to split up the definition of `asimp` into several functions and separately prove helper lemmas about them.

**Exercise 3** (Dependently typed functions).

Implement the following functions on lists which take an additional proof argument that restricts the input values.

1. `head` : `forall (A : Type) (l : list A), l <> [] -> A`.
2. `tail` : `forall (A : Type) (l : list A), l <> [] -> list A`.
3. `nth` : `forall (A : Type) (n : nat) (l : list A),  
n < List.length l -> list A`.

**Exercise 4** (Computable total orders).

1. Define an inductive type `ComputableTotalOrder (A : Type) : Type` which has exactly one constructor with arguments:
  - a computable binary relation `relation` on `A`: `leb : A -> A -> bool`;
  - proofs that `leb` is total, antisymmetric and transitive.

Such a single-constructor inductive type represents a dependent record. In this case the record contains a computable binary relation and a proof that this relation is a total order.

*Hint.* There is a special syntax for dependent records. Search Coq's reference manual (<https://coq.inria.fr/distrib/current/refman/>) for `Record`.

2. Define an element `cto_nat` of type `ComputableTotalOrder nat`.