CD

目录

Chapter 1

Library LF.Bib

1.1 Bib: 参考文献

1.2 本卷中出现的引用

Bertot 2004 Interactive Theorem Proving and Program Development: Coq'Art: The Calculus of Inductive Constructions, by Yves Bertot and Pierre Casteran. Springer-Verlag, 2004. https://tinyurl.com/z3o7nqu

Chlipala 2013 Certified Programming with Dependent Types, by Adam Chlipala. MIT Press. 2013. https://tinyurl.com/zqdnyg2

Lipovaca 2011 Learn You a Haskell for Great Good! A Beginner's Guide, by Miran Lipovaca, No Starch Press, April 2011. http://learnyouahaskell.com

O'Sullivan 2008 Bryan O'Sullivan, John Goerzen, and Don Stewart: Real world Haskell - code you can believe in. O'Reilly 2008. http://book.realworldhaskell.org

Pugh 1991 Pugh, William. "The Omega test: a fast and practical integer programming algorithm for dependence analysis." Proceedings of the 1991 ACM/IEEE conference on Supercomputing. ACM, 1991. https://dl.acm.org/citation.cfm?id=125848

Wadler 2015 Philip Wadler. "Propositions as types." Communications of the ACM 58, no. 12 (2015): 75-84. https://dl.acm.org/citation.cfm?id=2699407

Chapter 2

Library LF.Postscript

2.1 Postscript: 后记

恭喜，课程终于顺利结束了!

2.2 回顾一下

到目前为止，我们已经学习了很多内容。我们来快速地回顾一下：

- *'函数式编程'*：
  - "声明式" 编程风格 (在不可变的数据构造上递归，而非在可变的数组或指针结构上循环）
  - 高阶函数
  - 多态
- *'逻辑'*，软件工程的数学基础：

  逻辑微积分
  - —— ~ —————— 软件工程机械工程/土木工程
    - 归纳定义的集合和关系
    - 归纳证明
    - 证明对象

- *'Coq'*，一个强有力的证明辅助工具
  - 函数式核心语言
  - 核心策略
  - 自动化

2.3 继续前行

假如本书的内容引起了你的兴趣，你还可以阅读'软件基础'系列的：

- '编程语言基础'（《软件基础》第二卷，与本书作者组类似）覆盖了了关于编程语言理论方面的研究生课程，包括霍尔逻辑（Hoare logic）、操作语义以及类型系统。
- '函数式算法验证'（《软件基础》第三卷，Andrew Appel 著）在使用 Coq 进行程序验证和函数式编程的基础之上，讨论了一般数据结构课程中的一系列主题并着眼于其形式化验证。

2.4 资源

进一步学习的资源.....

- 本书包含了一些可选章节，其中讲述的内容或许会对你有用。你可以在目录或章节依赖简图中找到它们。
- 有关 Coq 的问题，可以查看 Stack Overflow 上的 *#coq* 板块 (https://stackoverflow.com/questions/ta
  它是个很棒的社区资源。
- 更多与函数式编程相关的内容：
  - Learn You a Haskell for Great Good, by Miran Lipovaca *Lipovaca* 2011 (in Bib.v).（《Haskell 趣学指南》https://learnyoua.haskell.sg/content/）
  - Real World Haskell, by Bryan O'Sullivan, John Goerzen, and Don Stewart *O'Sullivan* 2008 (in Bib.v).（《Real World Haskell 中文版》http://cnhaskell.com/）

  - ..... 以及其它关于 Haskell、OCaml、 Scheme、Racket、Scala、F sharp 等语言的优秀书籍。

- 更多 Coq 相关的资源：

  - Certified Programming with Dependent Types, by Adam Chlipala *Chlipala* 2013 (in Bib.v).
  - Interactive Theorem Proving and Program Development: Coq'Art: The Calculus of Inductive Constructions, by Yves Bertot and Pierre Casteran *Bertot* 2004 (in Bib.v).

- 如果你有兴趣了解现实世界中形式化验证对关键软件的应用，可以参阅'《编程语言基础》'的后记。

- 关于使用 Coq 构建形式化验证的系统，可以参考 2017 年 DeepSpec 夏令营的课程与相关资料。 https://deepspec.org/event/dsss17/index.html

Chapter 3

Library LF.AltAuto

3.1 AltAuto: More Automation

```
Set Warnings "-notation-overridden,-parsing".
From Coq Require Import omega.Omega.
From LF Require Import IndProp.
```

Up to now, we've used the more manual part of Coq's tactic facilities. In this chapter, we'll learn more about some of Coq's powerful automation features.

As a simple illustration of the benefits of automation, let's consider another problem on regular expressions, which we formalized in IndProp. A given set of strings can be denoted by many different regular expressions. For example, App EmptyString *re* matches exactly the same strings as *re*. We can write a function that "optimizes" any regular expression into a potentially simpler one by applying this fact throughout the r.e. (Note that, for simplicity, the function does not optimize expressions that arise as the result of other optimizations.)

```
Fixpoint re_opt_e {T:Type} (re: reg_exp T) : reg_exp T :=
  match re with
  | App EmptyStr re2 ⇒ re_opt_e re2
  | App re1 re2 ⇒ App (re_opt_e re1) (re_opt_e re2)
  | Union re1 re2 ⇒ Union (re_opt_e re1) (re_opt_e re2)
  | Star re ⇒ Star (re_opt_e re)
  | _ ⇒ re
  end.
```

We would like to show the equivalence of re's with their "optimized" form. One direction of this equivalence looks like this (the other is similar).

```
Lemma re_opt_e_match : ∀ T (re: reg_exp T) s,
    s =~ re → s =~ re_opt_e re.
Proof.
  intros T re s M.
  induction M
    as [| x'
        | s1 re1 s2 re2 Hmatch1 IH1 Hmatch2 IH2
        | s1 re1 re2 Hmatch IH | re1 s2 re2 Hmatch IH
        | re | s1 s2 re Hmatch1 IH1 Hmatch2 IH2].
  - simpl. apply MEmpty.
  - simpl. apply MChar.
  - simpl.
    destruct re1.
    + apply MApp. apply IH1. apply IH2.
    + inversion Hmatch1. simpl. apply IH2.
    + apply MApp. apply IH1. apply IH2.
    + apply MApp. apply IH1. apply IH2.
    + apply MApp. apply IH1. apply IH2.
    + apply MApp. apply IH1. apply IH2.
  - simpl. apply MUnionL. apply IH.
  - simpl. apply MUnionR. apply IH.
  - simpl. apply MStar0.
  - simpl. apply MStarApp. apply IH1. apply IH2.
Qed.
```

3.2 Coq Automation

The amount of repetition in this last proof is rather annoying. And if we wanted to extend the optimization function to handle other, similar, rewriting opportunities, it would start to be a real problem.

So far, we've been doing all our proofs using just a small handful of Coq's tactics and

completely ignoring its powerful facilities for constructing parts of proofs automatically. This section introduces some of these facilities, and we will see more over the next several chapters. Getting used to them will take some energy – Coq's automation is a power tool – but it will allow us to scale up our efforts to more complex definitions and more interesting properties without becoming overwhelmed by boring, repetitive, low-level details.

3.3 Tacticals

Tacticals is Coq's term for tactics that take other tactics as arguments – "higher-order tactics," if you will.

The `try` Tactical

If T is a tactic, then `try` T is a tactic that is just like T except that, if T fails, `try` T *successfully* does nothing at all (instead of failing).

```
Theorem silly1 : ∀ n, 1 + n = S n.
Proof. try reflexivity. Qed.

Theorem silly2 : ∀ (P : Prop), P → P.
Proof.
  intros P HP.
  try reflexivity.    apply HP. Qed.
```

There is no real reason to use `try` in completely manual proofs like these, but it is very useful for doing automated proofs in conjunction with the ; tactical, which we show next.

The ; Tactical (Simple Form)

In its most common form, the ; tactical takes two tactics as arguments. The compound tactic $T;T'$ first performs T and then performs T' on *each subgoal* generated by T.

For example, consider the following trivial lemma:

```
Lemma foo : ∀ n, 0 <=? n = true.
Proof.
  intros.
  destruct n eqn:E.
```

```
    - simpl. reflexivity.
    - simpl. reflexivity.
Qed.
```

We can simplify this proof using the ; tactical:

```
Lemma foo' : ∀ n, 0 <=? n = true.
Proof.
  intros.
  destruct n;

  simpl;

  reflexivity.
Qed.
```

Using `try` and ; together, we can get rid of the repetition in the proof that was bothering us a little while ago.

```
Lemma re_opt_e_match' : ∀ T (re: reg_exp T) s,
    s =~ re → s =~ re_opt_e re.
Proof.
  intros T re s M.
  induction M
    as [| x'
        | s1 re1 s2 re2 Hmatch1 IH1 Hmatch2 IH2
        | s1 re1 re2 Hmatch IH | re1 s2 re2 Hmatch IH
        | re | s1 s2 re Hmatch1 IH1 Hmatch2 IH2];

    simpl.
  - apply MEmpty.
  - apply MChar.
  -
    destruct re1;

    try (apply MApp; try apply IH1; apply IH2).
```

```
    inversion Hmatch1. simpl. apply IH2.
  - apply MUnionL. apply IH.
  - apply MUnionR. apply IH.
  - apply MStar0.
  - apply MStarApp. apply IH1. apply IH2.
Qed.
```

The ; Tactical (General Form)

The ; tactical also has a more general form than the simple *T*;*T'* we've seen above. If *T*, *T1*, ..., *Tn* are tactics, then

T; *T1* | *T2* | ... | *Tn*

is a tactic that first performs *T* and then performs *T1* on the first subgoal generated by *T*, performs *T2* on the second subgoal, etc.

So *T*;*T'* is just special notation for the case when all of the *Ti*'s are the same tactic; i.e., *T*;*T'* is shorthand for:

T; *T'* | *T'* | ... | *T'*

```
Lemma re_opt_e_match'' : ∀ T (re: reg_exp T) s,
    s =~ re → s =~ re_opt_e re.
Proof.
  intros T re s M.
  induction M
    as [| x'
        | s1 re1 s2 re2 Hmatch1 IH1 Hmatch2 IH2
        | s1 re1 re2 Hmatch IH | re1 s2 re2 Hmatch IH
        | re | s1 s2 re Hmatch1 IH1 Hmatch2 IH2];

    simpl.
  - apply MEmpty.
  - apply MChar.
  -
    destruct re1;
    try (apply MApp; [apply IH1 | apply IH2]).      inversion Hmatch1. simpl. apply
```

IH2.

- apply MUnionL. apply *IH*.
- apply MUnionR. apply *IH*.
- apply MStar0.
- apply MStarApp; [apply *IH1* | apply *IH2*]. Qed.

The repeat Tactical

The repeat tactical takes another tactic and keeps applying this tactic until it fails. Here is an example showing that 10 is in a long list using repeat.

```
Theorem In10 : In 10 [1;2;3;4;5;6;7;8;9;10].
Proof.
  repeat (try (left; reflexivity); right).
Qed.
```

The tactic repeat T never fails: if the tactic T doesn't apply to the original goal, then repeat still succeeds without changing the original goal (i.e., it repeats zero times).

```
Theorem In10' : In 10 [1;2;3;4;5;6;7;8;9;10].
Proof.
  repeat (left; reflexivity).
  repeat (right; try (left; reflexivity)).
Qed.
```

The tactic repeat T also does not have any upper bound on the number of times it applies T. If T is a tactic that always succeeds, then repeat T will loop forever (e.g., repeat simpl loops, since simpl always succeeds). While evaluation in Coq's term language, Gallina, is guaranteed to terminate, tactic evaluation is not! This does not affect Coq's logical consistency, however, since the job of repeat and other tactics is to guide Coq in constructing proofs; if the construction process diverges, this simply means that we have failed to construct a proof, not that we have constructed a wrong one.

练习：3 星, standard (re_opt) Consider this more powerful version of the regular expression optimizer.

```
Fixpoint re_opt {T:Type} (re: reg_exp T) : reg_exp T :=
  match re with
```

```
  | App re1 EmptySet ⇒ EmptySet
  | App EmptyStr re2 ⇒ re_opt re2
  | App re1 EmptyStr ⇒ re_opt re1
  | App re1 re2 ⇒ App (re_opt re1) (re_opt re2)
  | Union EmptySet re2 ⇒ re_opt re2
  | Union re1 EmptySet ⇒ re_opt re1
  | Union re1 re2 ⇒ Union (re_opt re1) (re_opt re2)
  | Star EmptySet ⇒ EmptyStr
  | Star EmptyStr ⇒ EmptyStr
  | Star re ⇒ Star (re_opt re)
  | EmptySet ⇒ EmptySet
  | EmptyStr ⇒ EmptyStr
  | Char x ⇒ Char x
  end.

Lemma re_opt_match : ∀ T (re: reg_exp T) s,
    s =~ re → s =~ re_opt re.
Proof.
  intros T re s M.
  induction M
    as [| x'
        | s1 re1 s2 re2 Hmatch1 IH1 Hmatch2 IH2
        | s1 re1 re2 Hmatch IH | re1 s2 re2 Hmatch IH
        | re | s1 s2 re Hmatch1 IH1 Hmatch2 IH2].
  - simpl. apply MEmpty.
  - simpl. apply MChar.
  - simpl.
    destruct re1.
    + inversion IH1.
    + inversion IH1. simpl. destruct re2.
      × apply IH2.
      × apply IH2.
      × apply IH2.
```

× `apply` *IH2*.

× `apply` *IH2*.

× `apply` *IH2*.

\+ `destruct` *re2*.

× `inversion` *IH2*.

× `inversion` *IH2*. `rewrite` *app_nil_r*. `apply` *IH1*.

× `apply` MApp. `apply` *IH1*. `apply` *IH2*.

× `apply` MApp. `apply` *IH1*. `apply` *IH2*.

× `apply` MApp. `apply` *IH1*. `apply` *IH2*.

× `apply` MApp. `apply` *IH1*. `apply` *IH2*.

\+ `destruct` *re2*.

× `inversion` *IH2*.

× `inversion` *IH2*. `rewrite` *app_nil_r*. `apply` *IH1*.

× `apply` MApp. `apply` *IH1*. `apply` *IH2*.

× `apply` MApp. `apply` *IH1*. `apply` *IH2*.

× `apply` MApp. `apply` *IH1*. `apply` *IH2*.

× `apply` MApp. `apply` *IH1*. `apply` *IH2*.

\+ `destruct` *re2*.

× `inversion` *IH2*.

× `inversion` *IH2*. `rewrite` *app_nil_r*. `apply` *IH1*.

× `apply` MApp. `apply` *IH1*. `apply` *IH2*.

× `apply` MApp. `apply` *IH1*. `apply` *IH2*.

× `apply` MApp. `apply` *IH1*. `apply` *IH2*.

× `apply` MApp. `apply` *IH1*. `apply` *IH2*.

\+ `destruct` *re2*.

× `inversion` *IH2*.

× `inversion` *IH2*. `rewrite` *app_nil_r*. `apply` *IH1*.

× `apply` MApp. `apply` *IH1*. `apply` *IH2*.

× `apply` MApp. `apply` *IH1*. `apply` *IH2*.

× `apply` MApp. `apply` *IH1*. `apply` *IH2*.

× `apply` MApp. `apply` *IH1*. `apply` *IH2*.

\- `simpl`.

`destruct` *re1*.

+ inversion *IH*.

+ destruct *re2*.

  × apply *IH*.

  × apply MUnionL. apply *IH*.

  × apply MUnionL. apply *IH*.

  × apply MUnionL. apply *IH*.

  × apply MUnionL. apply *IH*.

  × apply MUnionL. apply *IH*.

+ destruct *re2*.

  × apply *IH*.

  × apply MUnionL. apply *IH*.

  × apply MUnionL. apply *IH*.

  × apply MUnionL. apply *IH*.

  × apply MUnionL. apply *IH*.

  × apply MUnionL. apply *IH*.

+ destruct *re2*.

  × apply *IH*.

  × apply MUnionL. apply *IH*.

  × apply MUnionL. apply *IH*.

  × apply MUnionL. apply *IH*.

  × apply MUnionL. apply *IH*.

  × apply MUnionL. apply *IH*.

+ destruct *re2*.

  × apply *IH*.

  × apply MUnionL. apply *IH*.

  × apply MUnionL. apply *IH*.

  × apply MUnionL. apply *IH*.

  × apply MUnionL. apply *IH*.

  × apply MUnionL. apply *IH*.

+ destruct *re2*.

  × apply *IH*.

  × apply MUnionL. apply *IH*.

  × apply MUnionL. apply *IH*.

```
    × apply MUnionL. apply IH.
    × apply MUnionL. apply IH.
    × apply MUnionL. apply IH.
- simpl.
  destruct re1.
  + apply IH.
  + destruct re2.
    × inversion IH.
    × apply MUnionR. apply IH.
    × apply MUnionR. apply IH.
    × apply MUnionR. apply IH.
    × apply MUnionR. apply IH.
    × apply MUnionR. apply IH.
  + destruct re2.
    × inversion IH.
    × apply MUnionR. apply IH.
    × apply MUnionR. apply IH.
    × apply MUnionR. apply IH.
    × apply MUnionR. apply IH.
    × apply MUnionR. apply IH.
  + destruct re2.
    × inversion IH.
    × apply MUnionR. apply IH.
    × apply MUnionR. apply IH.
    × apply MUnionR. apply IH.
    × apply MUnionR. apply IH.
    × apply MUnionR. apply IH.
  + destruct re2.
    × inversion IH.
    × apply MUnionR. apply IH.
    × apply MUnionR. apply IH.
    × apply MUnionR. apply IH.
    × apply MUnionR. apply IH.
```

```
      × apply MUnionR. apply IH.
    + destruct re2.
      × inversion IH.
      × apply MUnionR. apply IH.
      × apply MUnionR. apply IH.
      × apply MUnionR. apply IH.
      × apply MUnionR. apply IH.
      × apply MUnionR. apply IH.
- simpl.
    destruct re.
    + apply MEmpty.
    + apply MEmpty.
    + apply MStar0.
    + apply MStar0.
    + apply MStar0.
    + simpl.
      destruct re.
      × apply MStar0.
      × apply MStar0.
      × apply MStar0.
      × apply MStar0.
      × apply MStar0.
      × apply MStar0.
- simpl.
   destruct re.
   + inversion IH1.
   + inversion IH1. inversion IH2. apply MEmpty.
   + apply star_app.
     × apply MStar1. apply IH1.
     × apply IH2.
   + apply star_app.
     × apply MStar1. apply IH1.
     × apply IH2.
```

+ apply star_app.
  × apply MStar1. apply *IH1*.
  × apply *IH2*.
+ apply star_app.
  × apply MStar1. apply *IH1*.
  × apply *IH2*.

Qed.

Lemma re_opt_match' : ∀ *T* (*re*: **reg_exp** *T*) *s*,
  s =~ *re* → *s* =~ re_opt *re*.

Proof.
  Admitted.

Definition manual_grade_for_re_opt : **option** (**nat**×string) := None.
  □

3.3.1 A Few More Handy Tactics

By the way, here are some miscellaneous tactics that you may find convenient as we continue.

- `clear` *H*: Delete hypothesis *H* from the context.
- `rename`... *into*...: Change the name of a hypothesis in the proof context. For example, if the context includes a variable named *x*, then `rename` *x into y* will change all occurrences of *x* to *y*.
- `subst` *x*: Find an assumption *x* = e or e = *x* in the context, replace *x* with e throughout the context and current goal, and clear the assumption.
- `subst`: Substitute away *all* assumptions of the form *x* = e or e = *x*.

We'll see examples as we go along.

3.3.2 Defining New Tactics

Coq also provides several ways of "programming" tactic scripts.

- Coq has a built-in language called `Ltac` with primitives that can examine and modify the proof state. The full details are a bit too complicated to get into here (and it is generally agreed that `Ltac` is not the most beautiful part of Coq's design!), but they can be found in the reference manual and other books on Coq. Simple use cases are not too difficult.

- There is also an OCaml API, which can be used to build tactics that access Coq's internal structures at a lower level, but this is seldom worth the trouble for ordinary Coq users.

Here is a simple `Ltac` example:

`Ltac` *impl_and_try* *c* := `simpl`; `try` *c*.

This defines a new tactical called *simpl_and_try* that takes one tactic `c` as an argument and is defined to be equivalent to the tactic `simpl`; `try c`. Now writing "*simpl_and_try* `reflexivity`." in a proof will be the same as writing "`simpl`; `try reflexivity`."

3.4 Decision Procedures

So far, the automation we have considered has primarily been useful for removing repetition. Another important category of automation consists of built-in decision procedures for specific kinds of problems. There are several of these, but the `omega` tactic is the most important to start with.

3.4.1 The Omega Tactic

The `omega` tactic implements a decision procedure for a subset of first-order logic called *Presburger arithmetic*. It is based on the Omega algorithm invented by William Pugh *Pugh* 1991 (in Bib.v).

If the goal is a universally quantified formula made out of

- numeric constants, addition (+ and S), subtraction (- and pred), and multiplication by constants (this is what makes it Presburger arithmetic),

- equality (= and $\neq$) and ordering ($\leq$), and

- the logical connectives $\wedge$, $\vee$, $\neg$, and $\rightarrow$,

then invoking `omega` will either solve the goal or fail, meaning that the goal is actually false. (If the goal is *not* of this form, `omega` will also fail.)

Note that we needed the import `Require Import Omega` at the top of this file.

```
Example silly_presburger_example : ∀ m n o p,
  m + n ≤ n + o ∧ o + 3 = p + 3 →
  m ≤ p.
Proof.
  intros. omega.
Qed.
```

3.5 Search Tactics

Another very important category of automation tactics helps us construct proofs by *searching* for relevant facts These tactics include the `auto` tactic for backwards reasoning, automated forward reasoning via the `Ltac` hypothesis matching machinery, and deferred instantiation of existential variables using `eapply` and `eauto`. Using these features together with Ltac's scripting facilities will enable us to make our proofs startlingly short! Used properly, they can also make proofs more maintainable and robust to changes in underlying definitions. A deeper treatment of `auto` and `eauto` can be found in the *UseAuto* chapter in *Programming Language Foundations*.

3.5.1 The `constructor` tactic.

A simple first example of a search tactic is `constructor`, which tries to find a constructor `c` (from some `Inductive` definition in the current environment) that can be applied to solve the current goal. If one is found, behave like `apply c`.

```
Example constructor_example: ∀ (n:nat),
    ev (n+n).
Proof.
  induction n; simpl.
  - constructor.   - rewrite plus_comm. simpl. constructor. auto.
```

```
Qed.
```

This saves us from needing to remember the names of our constructors. Warning: if more than one constructor can apply, `constructor` picks the first one (in the order in which they were defined in the `Inductive`) which is not necessarily the one we want!

3.5.2 The `auto` Tactic

Thus far, our proof scripts mostly apply relevant hypotheses or lemmas by name, and one at a time.

```
Example auto_example_1 : ∀ (P Q R: Prop),
  (P → Q) → (Q → R) → P → R.
Proof.
  intros P Q R H1 H2 H3.
  apply H2. apply H1. assumption.
Qed.
```

The `auto` tactic frees us from this drudgery by *searching* for a sequence of applications that will prove the goal:

```
Example auto_example_1' : ∀ (P Q R: Prop),
  (P → Q) → (Q → R) → P → R.
Proof.
  auto.
Qed.
```

The `auto` tactic solves goals that are solvable by any combination of

- `intros` and
- `apply` (of hypotheses from the local context, by default).

Using `auto` is always "safe" in the sense that it will never fail and will never change the proof state: either it completely solves the current goal, or it does nothing.

Here is a more interesting example showing `auto`'s power:

```
Example auto_example_2 : ∀ P Q R S T U : Prop,
  (P → Q) →
```

```
  (P → R) →
  (T → R) →
  (S → T → U) →
  ((P→Q) → (P→S)) →
  T →
  P →
  U.
Proof. auto. Qed.
```

Proof search could, in principle, take an arbitrarily long time, so there are limits to how far `auto` will search by default.

```
Example auto_example_3 : ∀ (P Q R S T U: Prop),
  (P → Q) →
  (Q → R) →
  (R → S) →
  (S → T) →
  (T → U) →
  P →
  U.
Proof.
  auto.
  auto 6.
Qed.
```

When searching for potential proofs of the current goal, `auto` considers the hypotheses in the current context together with a *hint database* of other lemmas and constructors. Some common lemmas about equality and logical operators are installed in this hint database by default.

```
Example auto_example_4 : ∀ P Q R : Prop,
  Q →
  (Q → R) →
  P ∨ (Q ∧ R).
Proof. auto. Qed.
```

If we want to see which facts `auto` is using, we can use `info_auto` instead.

```
Example auto_example_5: 2 = 2.
Proof.
  info_auto.
Qed.
```

We can extend the hint database just for the purposes of one application of `auto` by writing "`auto using ...`".

```
Lemma le_antisym : ∀ n m: nat, (n ≤ m ∧ m ≤ n) → n = m.
Proof. intros. omega. Qed.

Example auto_example_6 : ∀ n m p : nat,
  (n ≤ p → (n ≤ m ∧ m ≤ n)) →
  n ≤ p →
  n = m.
Proof.
  intros.
  auto using le_antisym.
Qed.
```

Of course, in any given development there will probably be some specific constructors and lemmas that are used very often in proofs. We can add these to the global hint database by writing

Hint Resolve T.

at the top level, where T is a top-level theorem or a constructor of an inductively defined proposition (i.e., anything whose type is an implication). As a shorthand, we can write

Hint Constructors c.

to tell Coq to do a `Hint Resolve` for *all* of the constructors from the inductive definition of `c`.

It is also sometimes necessary to add

Hint Unfold d.

where `d` is a defined symbol, so that `auto` knows to expand uses of `d`, thus enabling further possibilities for applying lemmas that it knows about.

It is also possible to define specialized hint databases that can be activated only when needed. See the Coq reference manual for more.

```
Hint Resolve le_antisym.
```

```
Example auto_example_6' : ∀ n m p : nat,
  (n≤ p → (n ≤ m ∧ m ≤ n)) →
  n ≤ p →
  n = m.
Proof.
  intros.
  auto. Qed.

Definition is_fortytwo x := (x = 42).

Example auto_example_7: ∀ x,
  (x ≤ 42 ∧ 42 ≤ x) → is_fortytwo x.
Proof.
  auto. Abort.

Hint Unfold is_fortytwo.

Example auto_example_7' : ∀ x,
  (x ≤ 42 ∧ 42 ≤ x) → is_fortytwo x.
Proof. info_auto. Qed.
```

练习：3 星, advanced (pumping_redux) Use `auto`, `omega`, and any other useful tactics from this chapter to shorten your proof (or the "official" solution proof) of the weak Pumping Lemma exercise from IndProp. `Import` *Pumping*.

```
Lemma weak_pumping : ∀ T (re : reg_exp T) s,
  s =~ re →
  pumping_constant re ≤ length s →
  ∃ s1 s2 s3,
    s = s1 ++ s2 ++ s3 ∧
    s2 ≠ [] ∧
    ∀ m, s1 ++ napp m s2 ++ s3 =~ re.
Proof.
    Admitted.
Definition manual_grade_for_pumping_redux : option (nat×string) := None.
```

□

练习：3 星, advanced, optional (pumping_redux_strong) Use `auto`, `omega`, and any other useful tactics from this chapter to shorten your proof (or the "official" solution proof) of the stronger Pumping Lemma exercise from IndProp. `Import` *Pumping*.

```
Lemma pumping : ∀ T (re : reg_exp T) s,
  s =~ re →
  pumping_constant re ≤ length s →
  ∃ s1 s2 s3,
    s = s1 ++ s2 ++ s3 ∧
    s2 ≠ [] ∧
    length s1 + length s2 ≤ pumping_constant re ∧
    ∀ m, s1 ++ napp m s2 ++ s3 =~ re.
Proof.
    Admitted.
Definition manual_grade_for_pumping_redux_strong : option (nat×string) := None.
    □
```

3.5.3 The `eapply` and `eauto` variants

To close the chapter, we'll introduce one more convenient feature of Coq: its ability to delay instantiation of quantifiers. To motivate this feature, consider again this simple example:

```
Example trans_example1: ∀ a b c d,
    a ≤ b + b×c →
    (1+c)*b ≤ d →
    a ≤ d.
Proof.
  intros a b c d H1 H2.
  apply le_trans with (b+ b×c).    + apply H1.
  + simpl in H2. rewrite mult_comm. apply H2.
Qed.
```

In the first step of the proof, we had to explicitly provide a longish expression to help Coq instantiate a "hidden" argument to the le_trans constructor. This was needed because the definition of le_trans...

le_trans : forall m n o : nat, m <= n -> n <= o -> m <= o

is quantified over a variable, n, that does not appear in its conclusion, so unifying its conclusion with the goal state doesn't help Coq find a suitable value for this variable. If we leave out the `with`, this step fails ("Error: Unable to find an instance for the variable n").

We already know one way to avoid an explicit `with` clause, namely to provide *H1* as the (first) explicit argument to le_trans. But here's another way, using the `eapply` *tactic*:

```
Example trans_example1': ∀ a b c d,
    a ≤ b + b×c →
    (1+c)*b ≤ d →
    a ≤ d.
Proof.
  intros a b c d H1 H2.
  eapply le_trans.    + apply H1.    + simpl in H2. rewrite mult_comm. apply H2.
Qed.
```

The `eapply` H tactic behaves just like `apply` H except that, after it finishes unifying the goal state with the conclusion of H, it does not bother to check whether all the variables that were introduced in the process have been given concrete values during unification.

If you step through the proof above, you'll see that the goal state at position 1 mentions the *existential variable* $?n$ in both of the generated subgoals. The next step (which gets us to position 2) replaces $?n$ with a concrete value. When we start working on the second subgoal (position 3), we observe that the occurrence of $?n$ in this subgoal has been replaced by the value that it was given during the first subgoal.

Several of the tactics that we've seen so far, including ∃, `constructor`, and `auto`, have `e...` variants. For example, here's a proof using `eauto`:

```
Example trans_example2: ∀ a b c d,
    a ≤ b + b×c →
    b + b×c ≤ d →
    a ≤ d.
Proof.
  intros a b c d H1 H2.
  info_eauto using le_trans.
Qed.
```

The `eauto` tactic works just like `auto`, except that it uses `eapply` instead of `apply`.

Pro tip: One might think that, since `eapply` and `eauto` are more powerful than `apply` and `auto`, it would be a good idea to use them all the time. Unfortunately, they are also significantly slower – especially `eauto`. Coq experts tend to use `apply` and `auto` most of the time, only switching to the `e` variants when the ordinary variants don't do the job.

Chapter 4

Library LF.Auto

4.1 Auto: 更多的自动化

```
Set Warnings "-notation-overridden,-parsing".
From Coq Require Import omega.Omega.
From LF Require Import Maps.
From LF Require Import Imp.
```

到目前为止，我们大多使用的都是 Coq 策略系统中手动的部分。在本章中，我们会学习更多 Coq 强大的自动化特性：通过 `auto` 策略进行证明搜索，通过 `Ltac` 前提搜索器进行自动正向推理，以及通过 `eapply` 和 `eauto` 推迟存在性变量的实例化。这些特性配合 Ltac 的脚本功能可以极大地缩短证明！如果使用得当，它们还会提高证明的可维护性，在以后修改证明的底层定义时也会更加健壮。对 `auto` 和 `eauto` 更加深入的探讨可参阅'《编程语言基础》'的 *UseAuto* 一章。

还有另一大类自动化方式我们所言不多，即内建的对特定种类问题的决策算法：`omega` 就是这样的例子，不过还有其它的。这一主题我们会以后再谈。

我们从以下证明开始，加上一些 Imp 中的小改动。我们将分几个阶段来简化此证明。

```
Theorem ceval_deterministic: ∀ c st st1 st2,
     st =[ c ]=> st1 →
     st =[ c ]=> st2 →
     st1 = st2.
Proof.
  intros c st st1 st2 E1 E2;
```

```
  generalize dependent st2;
  induction E1; intros st2 E2; inversion E2; subst.
  - reflexivity.
  - reflexivity.
  -
    assert (st' = st'0) as EQ1.
    { apply IHE1_1; apply H1. }
    subst st'0.
    apply IHE1_2. assumption.
  -
    apply IHE1. assumption.
  -
    rewrite H in H5. discriminate.
  -
    rewrite H in H5. discriminate.
  -
    apply IHE1. assumption.
  -
    reflexivity.
  -
    rewrite H in H2. discriminate.
  -
    rewrite H in H4. discriminate.
  -
    assert (st' = st'0) as EQ1.
    { apply IHE1_1; assumption. }
    subst st'0.
    apply IHE1_2. assumption. Qed.
```

4.2 auto 策略

迄今为止，我们的证明脚本大多是按名称来应用相关的前提或引理的，一次一个。

```
Example auto_example_1 : ∀ (P Q R: Prop),
```

```
  (P → Q) → (Q → R) → P → R.
Proof.
  intros P Q R H1 H2 H3.
  apply H2. apply H1. assumption.
Qed.
```

auto 策略可以'搜索'一系列能够证明待证目标的应用来免除这些苦役：

```
Example auto_example_1' : ∀ (P Q R: Prop),
  (P → Q) → (Q → R) → P → R.
Proof.
  auto.
Qed.
```

任何能够被以下策略的组合解决的待证目标，都能用 auto 来解决：

- intros
- apply（默认使用局部上下文中的前提）。

使用 auto 一定是"安全"的，它不会失败，也不会改变当前证明的状态： auto 要么完全解决它，要么什么也不做。

下面是个更大的例子，它展示了 auto 的强大：

```
Example auto_example_2 : ∀ P Q R S T U : Prop,
  (P → Q) →
  (P → R) →
  (T → R) →
  (S → T → U) →
  ((P→Q) → (P→S)) →
  T →
  P →
  U.
Proof. auto. Qed.
```

理论上来说，搜索可能需要任意长的时间，此时可通过参数来控制 auto 默认的搜索深度。

```
Example auto_example_3 : ∀ (P Q R S T U: Prop),
```

```
  (P → Q) →
  (Q → R) →
  (R → S) →
  (S → T) →
  (T → U) →
  P →
  U.
Proof.
  auto.
  auto 6.
Qed.
```

在搜索当前目标的潜在证明时， auto 会同时考虑当前上下文中的前提，以及一个包含其它引理或构造子的'提示数据库'。某些关于相等关系和逻辑运算符的事实默认已经安装到提示数据库中了。

```
Example auto_example_4 : ∀ P Q R : Prop,
  Q →
  (Q → R) →
  P ∨ (Q ∧ R).
Proof. auto. Qed.
```

如果我们想看 auto 用到了什么，可以使用 info_auto 。

```
Example auto_example_5: 2 = 2.
Proof.
  info_auto.
Qed.
```

我们可以为某次 auto 的调用扩展提示数据库，只需使用“auto using ...”。

```
Lemma le_antisym : ∀ n m: nat, (n ≤ m ∧ m ≤ n) → n = m.
Proof. intros. omega. Qed.

Example auto_example_6 : ∀ n m p : nat,
  (n ≤ p → (n ≤ m ∧ m ≤ n)) →
  n ≤ p →
  n = m.
```

```
Proof.
  auto using le_antisym.
Qed.
```

当然, 在证明过程中经常会用到一些特定的构造子和引理，我们可以将它们加入全局提示数据库中，方法是在顶层使用：

Hint Resolve T.

其中 T 是某个顶层的定理，或者是某个归纳定义的命题（即所有类型都是一个蕴含式）的构造子。我们也可以使用简写

Hint Constructors c.

来告诉 Coq 对归纳定义 c 的'所有'构造子都执行 `Hint Resolve`。

有时我们还需要

Hint Unfold d.

其中 d 是个已定义的符号，这样 `auto` 就知道要展开使用 d，以此来获得更多使用已知的引理的机会。

我们也可以定义特殊化的提示数据库，让它只在需要时激活。详情见 Coq 参考手册。

```
Hint Resolve le_antisym.

Example auto_example_6' : ∀ n m p : nat,
  (n≤ p → (n ≤ m ∧ m ≤ n)) →
  n ≤ p →
  n = m.
Proof.
  intros.
  auto. Qed.

Definition is_fortytwo x := (x = 42).

Example auto_example_7: ∀ x,
  (x ≤ 42 ∧ 42 ≤ x) → is_fortytwo x.
Proof.
  auto. Abort.

Hint Unfold is_fortytwo.

Example auto_example_7' : ∀ x,
  (x ≤ 42 ∧ 42 ≤ x) → is_fortytwo x.
```

```
Proof.
  auto. Qed.
```

我们来初次尝试简化 ceval_deterministic 的证明脚本。

```
Theorem ceval_deterministic': ∀ c st st1 st2,
     st =[ c ]=> st1 →
     st =[ c ]=> st2 →
     st1 = st2.
Proof.
  intros c st st1 st2 E1 E2.
  generalize dependent st2;
       induction E1; intros st2 E2; inversion E2; subst; auto.
  -
    assert (st' = st'0) as EQ1 by auto.
    subst st'0.
    auto.
  -
    +
      rewrite H in H5. discriminate.
  -
    +
      rewrite H in H5. discriminate.
  -
    +
      rewrite H in H2. discriminate.
  -
    rewrite H in H4. discriminate.
  -
    assert (st' = st'0) as EQ1 by auto.
    subst st'0.
    auto.
Qed.
```

如果在证明中我们会反复用到某个策略，呢么可以使用一个 Proof 指令的变体将它作

为证明中的默认策略。例如 `Proof with` t（其中 t 为任意一个策略）能够让我们在证明中将 *t1...* 用作 *t1*;t 的简写。作为示范，下面是以上证明的另一个版本，它用到了 `Proof with auto`。

```
Theorem ceval_deterministic'_alt: ∀ c st st1 st2,
     st =[ c ]=> st1 →
     st =[ c ]=> st2 →
     st1 = st2.
Proof with auto.
  intros c st st1 st2 E1 E2;
  generalize dependent st2;
  induction E1;
           intros st2 E2; inversion E2; subst...
  -
    assert (st' = st'0) as EQ1...
    subst st'0...
  -
    +
      rewrite H in H5. discriminate.
  -
    +
      rewrite H in H5. discriminate.
  -
    +
      rewrite H in H2. discriminate.
  -
    rewrite H in H4. discriminate.
  -
    assert (st' = st'0) as EQ1...
    subst st'0...
Qed.
```

4.3 搜索前提

证明变得更简单了，但还是有些烦人的重复。我们先从矛盾的情况着手。这些矛盾都是因为我们同时有

H1: beval st b = false

和

H2: beval st b = true

这两个前提。矛盾很显然，但证明却有点麻烦：我们必须找到 *H1* 和 *H2* 这两个前提，用一次 `rewrite` 后再用一次 `discriminate`。我们希望自动化此过程。

（实际上，Coq 有个内建的 `congruence` 策略来处理这种情况。不过我们暂时先忽略它的存在，为的是示范如何自己构建正向推理的策略。）

第一步，我们可以通过在 Ltac 中编写一个小函数来抽象出有问题的脚本。

```
Ltac rwd H1 H2 := rewrite H1 in H2; discriminate.

Theorem ceval_deterministic'': ∀ c st st1 st2,
     st =[ c ]=> st1 →
     st =[ c ]=> st2 →
     st1 = st2.
Proof.
  intros c st st1 st2 E1 E2.
  generalize dependent st2;
  induction E1; intros st2 E2; inversion E2; subst; auto.
  -
    assert (st' = st'0) as EQ1 by auto.
    subst st'0.
    auto.
  -
    +
      rwd H H5.
  -
    +
      rwd H H5.
  -
    +
```

```
    rwd H H2.
  -
    rwd H H4.
  -
    assert (st' = st'0) as EQ1 by auto.
    subst st'0.
    auto. Qed.
```

此例相比之前略有改进，但我们更希望 Coq 能替我们找到相关的假设。 Ltac 中的 `match goal` 功能可达成此目的。

```
Ltac find_rwd :=
  match goal with
    H1: ?E = true,
    H2: ?E = false
    ⊢ _ ⇒ rwd H1 H2
  end.
```

`match goal` 会查找两个不同的，形如等式的前提，其左式为两个相同的任意表达式 E，而右式为两个互相矛盾的布尔值。如果找到了这样的前提，就把 $H1$ 和 $H2$ 绑定为它们的名字，并将 rwd 策略应用到 $H1$ 和 $H2$ 上。

把此策略添加到每一个归纳证明的情况中，就能把所有的矛盾情况都解决了。

```
Theorem ceval_deterministic''': ∀ c st st1 st2,
    st =[ c ]=> st1 →
    st =[ c ]=> st2 →
    st1 = st2.
Proof.
  intros c st st1 st2 E1 E2.
  generalize dependent st2;
  induction E1; intros st2 E2; inversion E2; subst; try find_rwd; auto.
  -
    assert (st' = st'0) as EQ1 by auto.
    subst st'0.
    auto.
  -
```

```
  +
    assert (st' = st'0) as EQ1 by auto.
    subst st'0.
    auto. Qed.
```

现在我们来看看剩下的情况。每种情况都应用了带条件的前提以得到一个等式。目前我们把这些等式重述为断言，因此我们必须猜出需要的等式是什么（虽然可以用 `auto` 证明它们）。另一种方式是找出用到的有关前提，然后用它们进行 `rewrite` 改写，类似于这样：

```
Theorem ceval_deterministic'''': ∀ c st st1 st2,
     st =[ c ]=> st1 →
     st =[ c ]=> st2 →
     st1 = st2.
Proof.
  intros c st st1 st2 E1 E2.
  generalize dependent st2;
  induction E1; intros st2 E2; inversion E2; subst; try find_rwd; auto.
  -
    rewrite (IHE1_1 st'0 H1) in *. auto.
  -
    +
      rewrite (IHE1_1 st'0 H3) in *. auto. Qed.
```

现在用于改写的相关前提可以自动查找了。

```
Ltac find_eqn :=
  match goal with
    H1: ∀ x, ?P x → ?L = ?R,
    H2: ?P ?X
    ⊢ _ ⇒ rewrite (H1 X H2) in *
  end.
```

模式 $\forall\ x,\ ?P\ x \to ?L = ?\mathbf{R}$ 会匹配任何任何形如“对于所有的 x，'x 的某些性质'蕴含'某些等式'”的前提。 x 的性质被绑定为模式变量 P，而该等式的左式和右式会分别绑定为 L 和 $\mathbf{R}$。此前提的名字会被绑定为 $H1$。之后模式 $?P$?X 会匹配任何提供了“P 对于某个具体的 X 成立的证据”的前提。如果两个模式均成功，我们会在所有的前提和目标中

应用 `rewrite` 策略改写（即，用 X 来实例化量化的 x 并将 $H2$ 作为 P X 所需的证据提供）。

还剩一个问题：通常，可能有好几对前提都具有这种一般形式，而挑出我们真正需要的好像比较困难。不过关键在于我们要认识到其实可以 '全试一遍'！以下是具体方法：

- 每一个 `match goal` 在执行时都会不停地查找可行的一对前提，直到右式 RHS 匹配成功；如果没有这样的一对前提则会失败。
- `rewrite` 在得到一个形如 X = X 的平凡等式时会失败。
- 我们可以把整体策略包装在 `repeat` 中，这样就可以一直进行有用的改写，直到只剩下平凡的了。

```
Theorem ceval_deterministic''''': ∀ c st st1 st2,
     st =[ c ]=> st1 →
     st =[ c ]=> st2 →
     st1 = st2.
Proof.
  intros c st st1 st2 E1 E2.
  generalize dependent st2;
  induction E1; intros st2 E2; inversion E2; subst; try find_rwd;
    repeat find_eqn; auto.
Qed.
```

这种方法的巨大回报是，面对我们语言的适度变化时，我们的证明脚本会更加健壮。比如，我们可以为该语言增加一个 *REPEAT* 指令。

```
Module Repeat.

Inductive com : Type :=
  | CSkip
  | CAss (x : string) (a : aexp)
  | CSeq (c1 c2 : com)
  | CIf (b : bexp) (c1 c2 : com)
  | CWhile (b : bexp) (c : com)
  | CRepeat (c : com) (b : bexp).
```

REPEAT 的行为和 *WHILE* 类似，只是循环条件会在每次循环体执行完 *'之后'* 执行，且只在循环条件为*'false'*时重复执行。因此，循环体至少会被执行一次。

```
Notation "'SKIP'" :=
  CSkip.
Notation "x '::=' a" :=
  (CAss x a) (at level 60).
Notation "c1 ;; c2" :=
  (CSeq c1 c2) (at level 80, right associativity).
Notation "'WHILE' b 'DO' c 'END'" :=
  (CWhile b c) (at level 80, right associativity).
Notation "'TEST' c1 'THEN' c2 'ELSE' c3 'FI'" :=
  (CIf c1 c2 c3) (at level 80, right associativity).
Notation "'REPEAT' c 'UNTIL' b 'END'" :=
  (CRepeat c b) (at level 80, right associativity).

Reserved Notation "st '=[' c ']=>' st'"
                  (at level 40).

Inductive ceval : com → state → state → Prop :=
  | E_Skip : ∀ st,
      st =[ SKIP ]=> st
  | E_Ass : ∀ st a1 n x,
      aeval st a1 = n →
      st =[ x ::= a1 ]=> (x !-> n ; st)
  | E_Seq : ∀ c1 c2 st st' st'',
      st =[ c1 ]=> st' →
      st' =[ c2 ]=> st'' →
      st =[ c1 ;; c2 ]=> st''
  | E_IfTrue : ∀ st st' b c1 c2,
      beval st b = true →
      st =[ c1 ]=> st' →
      st =[ TEST b THEN c1 ELSE c2 FI ]=> st'
  | E_IfFalse : ∀ st st' b c1 c2,
      beval st b = false →
```

```
      st =[ c2 ]=> st' →
      st =[ TEST b THEN c1 ELSE c2 FI ]=> st'
  | E_WhileFalse : ∀ b st c,
      beval st b = false →
      st =[ WHILE b DO c END ]=> st
  | E_WhileTrue : ∀ st st' st'' b c,
      beval st b = true →
      st =[ c ]=> st' →
      st' =[ WHILE b DO c END ]=> st'' →
      st =[ WHILE b DO c END ]=> st''
  | E_RepeatEnd : ∀ st st' b c,
      st =[ c ]=> st' →
      beval st' b = true →
      st =[ REPEAT c UNTIL b END ]=> st'
  | E_RepeatLoop : ∀ st st' st'' b c,
      st =[ c ]=> st' →
      beval st' b = false →
      st' =[ REPEAT c UNTIL b END ]=> st'' →
      st =[ REPEAT c UNTIL b END ]=> st''

  where "st =[ c ]=> st'" := (ceval c st st').
```

我们对确定性证明的第一次尝试并不成功：E_RepeatEnd 和 E_RepeatLoop 这两种情况并没有被之前的自动化处理。

```
Theorem ceval_deterministic: ∀ c st st1 st2,
     st =[ c ]=> st1 →
     st =[ c ]=> st2 →
     st1 = st2.
Proof.
  intros c st st1 st2 E1 E2.
  generalize dependent st2;
  induction E1;
    intros st2 E2; inversion E2; subst; try find_rwd; repeat find_eqn; auto.
```

```
  -
    +
      find_rwd.
  -
     +
       find_rwd.
Qed.
```

幸运的是，我们只需交换 *find_eqn* 和 *find_rwd* 的调用顺序就能修复这一点。

```
Theorem ceval_deterministic': ∀ c st st1 st2,
     st =[ c ]=> st1 →
     st =[ c ]=> st2 →
     st1 = st2.
Proof.
  intros c st st1 st2 E1 E2.
  generalize dependent st2;
  induction E1;
    intros st2 E2; inversion E2; subst; repeat find_eqn; try find_rwd; auto.
Qed.

End Repeat.
```

这些例子为了让大家看看 Coq 中的“超级自动化”可以做到什么。 `match goal` 在使用时的的细节十分复杂，调试也很不方便。但至少在证明时值得加入它来简化证明，避免繁琐的工作，并为未来的修改做好准备。

4.3.1 变体 `eapply` 和 `eauto`

作为本章的结尾，我们来介绍一种更加方便的特性：它能够推迟量词的实例化。为了引出此特性，我们来回忆一下 Imp 中的这个例子：

```
Example ceval_example1:
  empty_st =[
    X ::= 2;;
    TEST X ≤ 1
      THEN Y ::= 3
```

```
      ELSE Z ::= 4
    FI
  ]=> (Z !-> 4 ; X !-> 2).
Proof.
  apply E_Seq with (X !-> 2).
  - apply E_Ass. reflexivity.
  - apply E_IfFalse. reflexivity. apply E_Ass. reflexivity.
Qed.
```

在证明的第一步，我们显式地提供了一个略长的表达式来帮助 Coq 为 E_Seq 构造子实例化一个“隐藏”的参数。需要它的原因在于 E_Seq 的定义...

E_Seq : forall c1 c2 st st' st", st = c1 => st' -> st' = c2 => st" -> st = c1 ;; c2 => st"

它是对 *st'* 的量化，而且并没有出现在结论中，因此将其结论与目标状态统一并不能帮助 Coq 为此变量找到合适的值。如果我们忽略 `with`，这一步就会失败（“Error: Unable to find an instance for the variable *st*"”）。

该错误的愚蠢指出在于适合 *st'* 的值其实在后面的步骤中会相当明显，就在我们应用 E_Ass 的地方。如果 Coq 能够等到这一步，就没必要显式地给出该值了。这正是 `eapply` 策略所能做到的：

```
Example ceval'_example1:
  empty_st =[
    X ::= 2;;
    TEST X ≤ 1
      THEN Y ::= 3
      ELSE Z ::= 4
    FI
  ]=> (Z !-> 4 ; X !-> 2).
Proof.
  eapply E_Seq.  - apply E_Ass.  reflexivity.  - apply E_IfFalse. reflexivity.
apply E_Ass. reflexivity.
Qed.
```

`eapply` H 的行为和 `apply` H 一样，只是在它统一完目标状态和 H 的结论之后，它并不会在引入所有变量的过程中，麻烦你去检查它们在统一时是否被赋予了具体的值。

如果你循着上面的证明步骤，就会看到 1 处的目标状态在生成的两个子目标中，都提

到了'存在性变量' ?st'。下一步，即把我们待带到 2 处的一步，会把 ?st' 替换成一个具体的值。这个新值包含一个新的存在性变量 ?n，它会被后面 3 处的 `reflexivity` 步骤依次实例化。当我们开始着手第二个子目标时（4 处），我们观察到此子目标中出现的 ?st' 已经被替换成了在第一个子目标中给出的值。

我们目前学过的几个策略，包括 ∃、`constructor` 和 `auto` 都有类似的变体。例如，下面是一个使用了 `eauto` 的证明：

```
Hint Constructors ceval.
Hint Transparent state.
Hint Transparent total_map.

Definition st12 := (Y !-> 2 ; X !-> 1).
Definition st21 := (Y !-> 1 ; X !-> 2).

Example eauto_example : ∃ s',
  st21 =[
    TEST X ≤ Y
      THEN Z ::= Y - X
      ELSE Y ::= X + Z
    FI
  ]=> s'.
Proof. info_eauto. Qed.
```

`eauto` 的策略和 `auto` 一样，除了它会使用 `eapply` 而非 `apply`； `info_eauto` 会显示 `auto` 使用了哪个事实。

专业提示：有人可能会想，既然 `eapply` 和 `eauto` 比 `apply` 和 `auto` 更强大，那么总是用它们不就好了。不幸的是，它们明显更慢，特别是 `eauto`。 Coq 专家倾向于主要使用 `apply` 和 `auto`，只在普通的版本无法做这些工作时才使用 `e` 开头的变体。

Chapter 5

Library LF.Extraction

5.1 Extraction: 从 Coq 中提取 ML

5.2 基本的提取方式

对于用 Coq 编写的代码而言，从中提取高效程序的最简方式是十分直白的。

首先我们需要指定提取的目标语言。可选的语言有三种：提取机制最为成熟的 OCaml，提取结果大都可以直接使用的 Haskell，以及提取机制有些过时的 Scheme。

```
Require Coq.extraction.Extraction.
Extraction Language OCaml.
```

现在我们将待提取的定义加载到 Coq 环境中。你可以直接写出定义，也可以从其它模块中加载。

```
From Coq Require Import Arith.Arith.
From Coq Require Import Init.Nat.
From Coq Require Import Arith.EqNat.
From LF Require Import ImpCEvalFun.
```

最后，我们来指定需要提取的定义，以及用于保存提取结果的文件名。

```
Extraction "imp1.ml" ceval_step.
```

Coq 在处理此指令时会生成一个名为 *imp1.ml* 的文件，其中包含了提取后的 ceval_step 以及所有递归依赖的文件。编译本章对应的 *.v* 文件，然后看看生成的 *imp1.ml* 吧！

5.3 控制提取特定的类型

我们可以让 Coq 将具体的 `Inductive` 定义提取为特定的 OCaml 类型。对于每一个定义，我们都要指明：

- 该 Coq 类型应当如何用 OCaml 来表示，以及
- 该类型的构造子应如何转换为目标语言中对应的标识符。

```
Extract Inductive bool ⇒ "bool" [ "true" "false" ].
```

对于非枚举类型（即构造器接受参数的类型），我们需要给出一个 OCaml 表达式来作为该类型元素上的“递归器”。（想想丘奇数。）

（译注：在这一部分，读者可以在为 **nat** 指定对应的类型后使用 `Extraction plus` 来得到自然数加法的提取结果，以此加深对“递归器”的理解。）

```
Extract Inductive nat ⇒ "int"
  [ "0" "(fun x -> x + 1)" ]
  "(fun zero succ n -> if n=0 then zero () else succ (n-1))".
```

我们也可以将定义的常量提取为具体的 OCaml 项或者操作符。

```
Extract Constant plus ⇒ "( + )".
Extract Constant mult ⇒ "( * )".
Extract Constant eqb ⇒ "( = )".
```

注意：保证提取结果的合理性是'你的责任'。例如，以下指定可能十分自然：

Extract Constant minus => “(-)”.

但是这样做会引起严重的混乱。（思考一下。为什么会这样呢？）

```
Extraction "imp2.ml" ceval_step.
```

检查一下生成的 *imp2.ml* 文件，留意一下此次的提取结果与 *imp1.ml* 有何不同。

5.4 一个完整的示例

为了使用提取的求值器运行 Imp 程序，我们还需要一个小巧的驱动程序来调用求值器并输出求值结果。

为简单起见，我们只取最终状态下前四个存储空间中的内容作为程序的结果。（译注：这里的存储空间指作为状态的 map。）

为了更方便地输入例子，我们将会从 ImpParser 模块中提取出语法解析器。首先需要正确建立 Coq 中的字符串与 OCaml 中字符列表的对应关系。

```
Require Import ExtrOcamlBasic.
Require Import ExtrOcamlString.
```

我们还需要翻译另一种布尔值：

```
Extract Inductive sumbool ⇒ "bool" ["true" "false"].
```

提取指令是相同的。

```
From LF Require Import Imp.
From LF Require Import ImpParser.

From LF Require Import Maps.
Extraction "imp.ml" empty_st ceval_step parse.
```

现在我们来运行一下生成的 Imp 求值器。首先你应该阅览一下 *impdriver.ml*（这并非从某个 Coq 源码提取而来，它是手写的。）

然后用下面的指令将求值器与驱动程序一同编译成可执行文件：

```
ocamlc -w -20 -w -26 -o impdriver imp.mli imp.ml impdriver.ml
./impdriver
```

（编译时所使用的 *-w* 开关只是为了避免输出一些误报的警告信息。）

5.5 讨论

由于我们证明了 ceval_step 函数的行为在适当的意义上与 **ceval** 关系一致，因此提取出的程序可视作'已验证的' Imp 解释器。当然，我们使用的语法分析器并未经过验证，因为我们并未对它进行任何证明！

5.6 更进一步

有关提取的更多详情见'软件基础'第三卷'已验证的函数式算法'中的 Extract 一章。

Chapter 6

Library LF.ImpCEvalFun

6.1 ImpCEvalFun: Imp 的求值函数

在Imp一章中我们已经见到了直接为 Imp 定义求值函数时会遇到的困难。当时为了规避这些困难，我们选择了定义求值关系而不是函数。而在这一可选的章节中，我们会再次讨论能够实现 Imp 求值函数的方法。

6.2 一个无法完成的求值器

```
From Coq Require Import omega.Omega.
From Coq Require Import Arith.Arith.
From LF Require Import Imp Maps.
```

在初次为指令编写求值函数时，我们写出了如下忽略了 *WHILE* 的代码：

```
Open Scope imp_scope.
Fixpoint ceval_step1 (st : state) (c : com) : state :=
  match c with
    | SKIP ⇒
        st
    | l ::= a1 ⇒
        (l !-> aeval st a1 ; st)
    | c1 ;; c2 ⇒
        let st' := ceval_step1 st c1 in
```

```
      ceval_step1 st' c2
  | TEST b THEN c1 ELSE c2 FI ⇒
      if (beval st b)
        then ceval_step1 st c1
        else ceval_step1 st c2
  | WHILE b1 DO c1 END ⇒
      st
  end.
Close Scope imp_scope.
```

如Imp一章中所言，在 ML 或 Haskell 这类传统的函数式语言中，我们可以这样处理 *WHILE* 指令：

| WHILE b1 DO c1 END => if (beval st b1) then ceval_step1 st (c1;; WHILE b1 DO c1 END) else st

Coq 不会接受此定义（它会提示出现错误 *Error*: *Cannot guess decreasing argument of fix*），因为我们想要定义的函数无需保证一定停机。确实，修改后的 ceval_step1 应用到 *Imp.v* 中的 loop 程序时永远不会停机。因为 Coq 不仅是一个函数式编程语言，还拥有逻辑一致性，因此任何有可能不会停机的函数都会被拒绝。下面是一段无效的(!) Coq 程序，它展示了假如 Coq 允许不停机的递归函数时会产生什么错误：

Fixpoint loop_false (n : nat) : False := loop_false n.

也就是说，像 **False** 这样的命题会变成可证的（例如 *loop_false* 0 就是个对 **False** 的证明），这对 Coq 的逻辑一致性来说是一场灾难。

由于它不会对所有的输入停机，因此至少在不借助附加技巧的情况下， ceval_step1 的完整版本无法用 Coq 写出...

6.3 一个计步的求值器

我们需要的技巧是将一个'附加'的参数传入求值函数中来告诉它需要运行多久。非正式地说，我们会在求值器的"油箱"中加一定数量的"汽油"，然后允许它运行到按一般的方式终止'或者'耗尽汽油，此时我们会停止求值并说最终结果为空内存（empty memory）。（我们也可以说当前的状态为求值器耗尽了汽油 – 这无关紧要，因为无论在哪种情况下结果都是错误的!）

```
Open Scope imp_scope.
```

```
Fixpoint ceval_step2 (st : state) (c : com) (i : nat) : state :=
  match i with
  | O ⇒ empty_st
  | S i' ⇒
    match c with
      | SKIP ⇒
          st
      | l ::= a1 ⇒
          (l !-> aeval st a1 ; st)
      | c1 ;; c2 ⇒
          let st' := ceval_step2 st c1 i' in
          ceval_step2 st' c2 i'
      | TEST b THEN c1 ELSE c2 FI ⇒
          if (beval st b)
            then ceval_step2 st c1 i'
            else ceval_step2 st c2 i'
      | WHILE b1 DO c1 END ⇒
          if (beval st b1)
          then let st' := ceval_step2 st c1 i' in
               ceval_step2 st' c i'
          else st
    end
  end.
Close Scope imp_scope.
```

'注意': 很容易想到这里的索引 i 是用来计算"求值的步数"的。然而我们仔细研究就会发现实际并非如此。例如，在串连的规则中，同一个 i 被传入了两个递归调用中。正确地理解 i 对于 ceval__ceval_step 的正式名分重要，它会在下面的练习中给出。

此求值器不太好的一点就是我们无法根据其结果说它是否停止，因为程序可能是正常停机，也可能是耗尽了汽油。我们的下下一个版本会返回一个 **option** state 而非只是一个 state，这样我们就能区分正常和异常的停机了。

```
Open Scope imp_scope.
Fixpoint ceval_step3 (st : state) (c : com) (i : nat)
```

```
                    : option state :=
  match i with
  | O ⇒ None
  | S i' ⇒
    match c with
      | SKIP ⇒
          Some st
      | l ::= a1 ⇒
          Some (l !-> aeval st a1 ; st)
      | c1 ;; c2 ⇒
          match (ceval_step3 st c1 i') with
          | Some st' ⇒ ceval_step3 st' c2 i'
          | None ⇒ None
          end
      | TEST b THEN c1 ELSE c2 FI ⇒
          if (beval st b)
            then ceval_step3 st c1 i'
            else ceval_step3 st c2 i'
      | WHILE b1 DO c1 END ⇒
          if (beval st b1)
          then match (ceval_step3 st c1 i') with
                 | Some st' ⇒ ceval_step3 st' c i'
                 | None ⇒ None
                 end
          else Some st
    end
  end.
Close Scope imp_scope.
```

我们可以引入一些辅助记法来隐藏对可选状态进行重复匹配的复杂工作，从而提高此版本的可读性。

```
Notation "'LETOPT' x <== e1 'IN' e2"
   := (match e1 with
```

```
        | Some x ⇒ e2
        | None ⇒ None
      end)
  (right associativity, at level 60).

Open Scope imp_scope.
Fixpoint ceval_step (st : state) (c : com) (i : nat)
                    : option state :=
  match i with
  | O ⇒ None
  | S i' ⇒
    match c with
      | SKIP ⇒
          Some st
      | l ::= a1 ⇒
          Some (l !-> aeval st a1 ; st)
      | c1 ;; c2 ⇒
          LETOPT st' <== ceval_step st c1 i' IN
          ceval_step st' c2 i'
      | TEST b THEN c1 ELSE c2 FI ⇒
          if (beval st b)
            then ceval_step st c1 i'
            else ceval_step st c2 i'
      | WHILE b1 DO c1 END ⇒
          if (beval st b1)
          then LETOPT st' <== ceval_step st c1 i' IN
               ceval_step st' c i'
          else Some st
    end
  end.
Close Scope imp_scope.

Definition test_ceval (st:state) (c:com) :=
  match ceval_step st c 500 with
```

```
  | None ⇒ None
  | Some st ⇒ Some (st X, st Y, st Z)
  end.
```

练习：2 星, standard, recommended (pup_to_n) 编写一个 Imp 程序对 1 到 X 求和（即 $1 + 2 + ... + X$）并赋值给 Y。确保你的解答能满足之后的测试。

```
Definition pup_to_n : com
  . Admitted.
```

□

练习：2 星, standard, optional (peven) 编写一个 Imp 程序：该程序在 X 为偶数时将 Z 置为 0，否则将 Z 置为 1。使用 test_ceval 测试你的程序。

6.4 关系求值 vs. 计步求值

对于算术表达式和布尔表达式，我们希望两种求值的定义最终都能产生同样的结果。本节将对此说明。

```
Theorem ceval_step__ceval: ∀ c st st',
      (∃ i, ceval_step st c i = Some st') →
      st =[ c ]=> st'.
Proof.
  intros c st st' H.
  inversion H as [i E].
  clear H.
  generalize dependent st'.
  generalize dependent st.
  generalize dependent c.
  induction i as [| i' ].

  -
    intros c st st' H. discriminate H.

  -
```

```
intros c st st' H.
destruct c;
       simpl in H; inversion H; subst; clear H.
  + apply E_Skip.
  + apply E_Ass. reflexivity.
  +
    destruct (ceval_step st c1 i') eqn:Heqr1.
    ×
      apply E_Seq with s.
        apply IHi'. rewrite Heqr1. reflexivity.
        apply IHi'. simpl in H1. assumption.
    ×
      discriminate H1.
  +
    destruct (beval st b) eqn:Heqr.
    ×
      apply E_IfTrue. rewrite Heqr. reflexivity.
      apply IHi'. assumption.
    ×
      apply E_IfFalse. rewrite Heqr. reflexivity.
      apply IHi'. assumption.
  + destruct (beval st b) eqn :Heqr.
    ×
     destruct (ceval_step st c i') eqn:Heqr1.
     {
       apply E_WhileTrue with s. rewrite Heqr.
       reflexivity.
       apply IHi'. rewrite Heqr1. reflexivity.
       apply IHi'. simpl in H1. assumption. }
     { discriminate H1. }
    ×
      injection H1 as H2. rewrite ← H2.
```

```
    apply E_WhileFalse. apply Heqr. Qed.
```

练习：4 星, standard (ceval_step__ceval_inf) 按照通常的模版写出 ceval_step__ceval 的非形式化证明，（对归纳定义的值进行分类讨论的模版，除了没有归纳假设外，应当看起来与归纳证明相同。）不要简单地翻译形式化证明的步骤，请让你的证明能够将主要想法传达给读者。

```
Definition manual_grade_for_ceval_step__ceval_inf : option (nat×string) := None.
```

□

```
Theorem ceval_step_more: ∀ i1 i2 st st' c,
  i1 ≤ i2 →
  ceval_step st c i1 = Some st' →
  ceval_step st c i2 = Some st'.
Proof.
induction i1 as [|i1']; intros i2 st st' c Hle Hceval.
  -
    simpl in Hceval. discriminate Hceval.
  -
    destruct i2 as [|i2']. inversion Hle.
    assert (Hle': i1' ≤ i2') by omega.
    destruct c.
    +
      simpl in Hceval. inversion Hceval.
      reflexivity.
    +
      simpl in Hceval. inversion Hceval.
      reflexivity.
    +
      simpl in Hceval. simpl.
      destruct (ceval_step st c1 i1') eqn:Heqst1'o.
      ×
        apply (IHi1' i2') in Heqst1'o; try assumption.
        rewrite Heqst1'o. simpl. simpl in Hceval.
```

```
    apply (IHi1' i2') in Hceval; try assumption.
  ×
    discriminate Hceval.
+
  simpl in Hceval. simpl.
  destruct (beval st b); apply (IHi1' i2') in Hceval;
    assumption.
+
  simpl in Hceval. simpl.
  destruct (beval st b); try assumption.
  destruct (ceval_step st c i1') eqn: Heqst1'o.
  ×
    apply (IHi1' i2') in Heqst1'o; try assumption.
    rewrite → Heqst1'o. simpl. simpl in Hceval.
    apply (IHi1' i2') in Hceval; try assumption.
  ×
    simpl in Hceval. discriminate Hceval. Qed.
```

练习：3 星, standard, recommended (ceval__ceval_step) 请完成以下证明。你会在某些地方用到 ceval_step_more 以及一些关于 $\leq$ 和 plus 的基本事实。

```
Theorem ceval__ceval_step: ∀ c st st',
      st =[ c ]=> st' →
      ∃ i, ceval_step st c i = Some st'.
Proof.
  intros c st st' Hce.
  induction Hce.
    Admitted.
    □

Theorem ceval_and_ceval_step_coincide: ∀ c st st',
      st =[ c ]=> st'
  ↔ ∃ i, ceval_step st c i = Some st'.
Proof.
```

```
  intros c st st'.
  split. apply ceval__ceval_step. apply ceval_step__ceval.
Qed.
```

6.5 再论求值的确定性

根据关系求值和计步求值的定义等价这一事实，我们可以给出一种取巧的方式来证明求值'关系'是确定性的。

```
Theorem ceval_deterministic' : ∀ c st st1 st2,
     st =[ c ]=> st1 →
     st =[ c ]=> st2 →
     st1 = st2.
Proof.
  intros c st st1 st2 He1 He2.
  apply ceval__ceval_step in He1.
  apply ceval__ceval_step in He2.
  inversion He1 as [i1 E1].
  inversion He2 as [i2 E2].
  apply ceval_step_more with (i2 := i1 + i2) in E1.
  apply ceval_step_more with (i2 := i1 + i2) in E2.
  rewrite E1 in E2. inversion E2. reflexivity.
  omega. omega. Qed.
```

Chapter 7

Library LF.ImpParser

7.1 ImpParser: 用 Coq 实现词法分析和语法分析

在 *Imp.v* 中，我们在设计 Imp 语言时完全忽略了具体的语法问题—— 我们仍需将程序员写下的 ASCII 字符串翻译成一棵由 **aexp**、**bexp** 和 **com** 所定义成的抽象语法树。在本章中，我们将会说明如何用 Coq 的函数式编程特性来构造简单的词法分析器和语法分析器以填补这一空白。

你无需对本章中代码的所有细节了如指掌，文中对代码的解释十分简短，而且本章不包含任何练习：这一章的目的只是为了证明这是办得到的。你可以阅读这些代码，它们并不是特别复杂，只是语法分析器的代码使用了一些“单子式（Monadic）”的编程习语，可能会稍微有些难以理解；但是大部分的读者大概只会粗略看一眼，然后跳到末尾的“例子”一节。

```
Set Warnings "-notation-overridden,-parsing".
From Coq Require Import Strings.String.
From Coq Require Import Strings.Ascii.
From Coq Require Import Arith.Arith.
From Coq Require Import Init.Nat.
From Coq Require Import Arith.EqNat.
From Coq Require Import Lists.List.
Import ListNotations.
From LF Require Import Maps Imp.
```

7.2 内部结构

7.2.1 词法分析

```
Definition isWhite (c : ascii) : bool :=
  let n := nat_of_ascii c in
  orb (orb (n =? 32)
           (n =? 9))
      (orb (n =? 10)
           (n =? 13)).

Notation "x '<=?' y" := (x <=? y)
  (at level 70, no associativity) : nat_scope.

Definition isLowerAlpha (c : ascii) : bool :=
  let n := nat_of_ascii c in
    andb (97 <=? n) (n <=? 122).

Definition isAlpha (c : ascii) : bool :=
  let n := nat_of_ascii c in
    orb (andb (65 <=? n) (n <=? 90))
        (andb (97 <=? n) (n <=? 122)).

Definition isDigit (c : ascii) : bool :=
  let n := nat_of_ascii c in
     andb (48 <=? n) (n <=? 57).

Inductive chartype := white | alpha | digit | other.

Definition classifyChar (c : ascii) : chartype :=
  if isWhite c then
    white
  else if isAlpha c then
    alpha
  else if isDigit c then
    digit
  else
    other.
```

```
Fixpoint list_of_string (s : string) : list ascii :=
  match s with
  | EmptyString ⇒ []
  | String c s ⇒ c :: (list_of_string s)
  end.

Fixpoint string_of_list (xs : list ascii) : string :=
  fold_right String EmptyString xs.

Definition token := string.

Fixpoint tokenize_helper (cls : chartype) (acc xs : list ascii)
                         : list (list ascii) :=
  let tk := match acc with [] ⇒ [] | _::_ ⇒ [rev acc] end in
  match xs with
  | [] ⇒ tk
  | (x::xs') ⇒
    match cls, classifyChar x, x with
    | _, _, "(" ⇒
      tk ++ ["("]::(tokenize_helper other [] xs')
    | _, _, ")" ⇒
      tk ++ [")"]::(tokenize_helper other [] xs')
    | _, white, _ ⇒
      tk ++ (tokenize_helper white [] xs')
    | alpha,alpha,x ⇒
      tokenize_helper alpha (x::acc) xs'
    | digit,digit,x ⇒
      tokenize_helper digit (x::acc) xs'
    | other,other,x ⇒
      tokenize_helper other (x::acc) xs'
    | _,tp,x ⇒
      tk ++ (tokenize_helper tp [x] xs')
    end
  end %char.

Definition tokenize (s : string) : list string :=
```

```
  map string_of_list (tokenize_helper white [] (list_of_string s)).

Example tokenize_ex1 :
    tokenize "abc12=3 223*(3+(a+c))" %string
  = ["abc"; "12"; "="; "3"; "223";
       "*"; "("; "3"; "+"; "(";
       "a"; "+"; "c"; ")"; ")"]%string.
Proof. reflexivity. Qed.
```

7.2.2 语法分析

带错误的可选值

一个附带出错信息的 **option** 类型:

```
Inductive optionE (X:Type) : Type :=
  | SomeE (x : X)
  | NoneE (s : string).

Arguments SomeE {X}.
Arguments NoneE {X}.
```

加一些语法糖以便于编写嵌套的对 **optionE** 的匹配表达式。

```
Notation "' p <- e1 ;; e2"
   := (match e1 with
       | SomeE p ⇒ e2
       | NoneE err ⇒ NoneE err
       end)
   (right associativity, p pattern, at level 60, e1 at next level).

Notation "'TRY' ' p <- e1 ;; e2 'OR' e3"
   := (match e1 with
       | SomeE p ⇒ e2
       | NoneE _ ⇒ e3
       end)
   (right associativity, p pattern,
    at level 60, e1 at next level, e2 at next level).
```

用于构建语法分析器的通用组合子

```
Open Scope string_scope.

Definition parser (T : Type) :=
  list token → optionE (T × list token).

Fixpoint many_helper {T} (p : parser T) acc steps xs :=
  match steps, p xs with
  | 0, _ ⇒
      NoneE "Too many recursive calls"
  | _, NoneE _ ⇒
      SomeE ((rev acc), xs)
  | S steps', SomeE (t, xs') ⇒
      many_helper p (t :: acc) steps' xs'
  end.
```

一个要求符合 p 零到多次的、指定步数的词法分析器：

```
Fixpoint many {T} (p : parser T) (steps : nat) : parser (list T) :=
  many_helper p [] steps.
```

该词法分析器要求一个给定的词法标记（token），并用 p 对其进行处理：

```
Definition firstExpect {T} (t : token) (p : parser T)
                       : parser T :=
  fun xs ⇒ match xs with
            | x::xs' ⇒
              if string_dec x t
              then p xs'
              else NoneE ("expected '" ++ t ++ "'.")
            | [] ⇒
              NoneE ("expected '" ++ t ++ "'.")
            end.
```

一个要求某个特定词法标记的语法分析器：

```
Definition expect (t : token) : parser unit :=
  firstExpect t (fun xs ⇒ SomeE (tt, xs)).
```

一个 Imp 的递归下降语法分析器

标识符：

```
Definition parseIdentifier (xs : list token)
                           : optionE (string × list token) :=
match xs with
| [] ⇒ NoneE "Expected identifier"
| x::xs' ⇒
    if forallb isLowerAlpha (list_of_string x) then
      SomeE (x, xs')
    else
      NoneE ("Illegal identifier:'" ++ x ++ "'")
end.
```

数字：

```
Definition parseNumber (xs : list token)
                       : optionE (nat × list token) :=
match xs with
| [] ⇒ NoneE "Expected number"
| x::xs' ⇒
    if forallb isDigit (list_of_string x) then
      SomeE (fold_left
               (fun n d ⇒
                  10 × n + (nat_of_ascii d -
                            nat_of_ascii "0"%char))
               (list_of_string x)
               0,
             xs')
    else
      NoneE "Expected number"
end.
```

解析算术表达式：

```
Fixpoint parsePrimaryExp (steps:nat)
```

```
                                (xs : list token)
                              : optionE (aexp × list token) :=
  match steps with
  | 0 ⇒ NoneE "Too many recursive calls"
  | S steps' ⇒
      TRY ' (i, rest) ← parseIdentifier xs ;;
          SomeE (AId i, rest)
      OR
      TRY ' (n, rest) ← parseNumber xs ;;
          SomeE (ANum n, rest)
      OR
      ' (e, rest) ← firstExpect "(" (parseSumExp steps') xs ;;
      ' (u, rest') ← expect ")" rest ;;
      SomeE (e,rest')
  end

with parseProductExp (steps:nat)
                     (xs : list token) :=
  match steps with
  | 0 ⇒ NoneE "Too many recursive calls"
  | S steps' ⇒
    ' (e, rest) ← parsePrimaryExp steps' xs ;;
    ' (es, rest') ← many (firstExpect "*" (parsePrimaryExp steps'))
                         steps' rest ;;
    SomeE (fold_left AMult es e, rest')
  end

with parseSumExp (steps:nat) (xs : list token) :=
  match steps with
  | 0 ⇒ NoneE "Too many recursive calls"
  | S steps' ⇒
    ' (e, rest) ← parseProductExp steps' xs ;;
    ' (es, rest') ←
```

```
      many (fun xs ⇒
              TRY ' (e,rest') ←
                  firstExpect "+"
                              (parseProductExp steps') xs ;;
                  SomeE ( (true, e), rest')
              OR
              ' (e, rest') ←
                  firstExpect "-"
                              (parseProductExp steps') xs ;;
              SomeE ( (false, e), rest'))
        steps' rest ;;
    SomeE (fold_left (fun e0 term ⇒
                        match term with
                        | (true, e) ⇒ APlus e0 e
                        | (false, e) ⇒ AMinus e0 e
                        end)
                     es e,
           rest')
  end.

Definition parseAExp := parseSumExp.
```

解析布尔表达式：

```
Fixpoint parseAtomicExp (steps:nat)
                        (xs : list token) :=
match steps with
  | 0 ⇒ NoneE "Too many recursive calls"
  | S steps' ⇒
    TRY ' (u,rest) ← expect "true" xs ;;
        SomeE (BTrue,rest)
    OR
    TRY ' (u,rest) ← expect "false" xs ;;
        SomeE (BFalse,rest)
    OR
```

```
    TRY ' (e,rest) ← firstExpect "~"
                                    (parseAtomicExp steps')
                                    xs ;;
        SomeE (BNot e, rest)
    OR
    TRY ' (e,rest) ← firstExpect "("
                                    (parseConjunctionExp steps')
                                    xs ;;
        ' (u,rest') ← expect ")" rest ;;
        SomeE (e, rest')
    OR
    ' (e, rest) ← parseProductExp steps' xs ;;
    TRY ' (e', rest') ← firstExpect "="
                                    (parseAExp steps') rest ;;
        SomeE (BEq e e', rest')
    OR
    TRY ' (e', rest') ← firstExpect "<="
                                        (parseAExp steps') rest ;;
        SomeE (BLe e e', rest')
    OR
    NoneE "Expected '=' or '<=' after arithmetic expression"
end

with parseConjunctionExp (steps:nat)
                         (xs : list token) :=
  match steps with
  | 0 ⇒ NoneE "Too many recursive calls"
  | S steps' ⇒
    ' (e, rest) ← parseAtomicExp steps' xs ;;
    ' (es, rest') ← many (firstExpect "&&"
                 (parseAtomicExp steps'))
              steps' rest ;;
    SomeE (fold_left BAnd es e, rest')
```

```
  end.

Definition parseBExp := parseConjunctionExp.

Check parseConjunctionExp.

Definition testParsing {X : Type}
           (p : nat →
                list token →
                optionE (X × list token))
           (s : string) :=
  let t := tokenize s in
  p 100 t.
```

解析指令：

```
Fixpoint parseSimpleCommand (steps:nat)
                            (xs : list token) :=
  match steps with
  | 0 ⇒ NoneE "Too many recursive calls"
  | S steps' ⇒
    TRY ' (u, rest) ← expect "SKIP" xs ;;
        SomeE (SKIP%imp, rest)
    OR
    TRY ' (e,rest) ←
             firstExpect "TEST"
                         (parseBExp steps') xs ;;
        ' (c,rest') ←
             firstExpect "THEN"
                         (parseSequencedCommand steps') rest ;;
        ' (c',rest'') ←
             firstExpect "ELSE"
                         (parseSequencedCommand steps') rest' ;;
        ' (tt,rest''') ←
             expect "END" rest'' ;;
       SomeE(TEST e THEN c ELSE c' FI%imp, rest''')
    OR
```

```
    TRY ' (e,rest) ←
            firstExpect "WHILE"
                    (parseBExp steps') xs ;;
        ' (c,rest') ←
            firstExpect "DO"
                    (parseSequencedCommand steps') rest ;;
        ' (u,rest'') ←
            expect "END" rest' ;;
        SomeE(WHILE e DO c END%imp, rest'')
    OR
    TRY ' (i, rest) ← parseIdentifier xs ;;
        ' (e, rest') ← firstExpect "::=" (parseAExp steps') rest ;;
        SomeE ((i ::= e)%imp, rest')
    OR
        NoneE "Expecting a command"
end

with parseSequencedCommand (steps:nat)
                           (xs : list token) :=
  match steps with
  | 0 ⇒ NoneE "Too many recursive calls"
  | S steps' ⇒
    ' (c, rest) ← parseSimpleCommand steps' xs ;;
    TRY ' (c', rest') ←
            firstExpect ";;"
                    (parseSequencedCommand steps') rest ;;
        SomeE ((c ;; c')%imp, rest')
    OR
    SomeE (c, rest)
  end.

Definition bignumber := 1000.

Definition parse (str : string) : optionE com :=
```

```
  let tokens := tokenize str in
  match parseSequencedCommand bignumber tokens with
  | SomeE (c, []) ⇒ SomeE c
  | SomeE (_, t::_) ⇒ NoneE ("Trailing tokens remaining: " ++ t)
  | NoneE err ⇒ NoneE err
  end.
```

7.3 示例

```
Example eg1 : parse " TEST x = y + 1 + 2 - y * 6 + 3 THEN x ::= x * 1;; y ::= 0 ELSE
SKIP END "
=
  SomeE (
      TEST "x" = "y" + 1 + 2 - "y" × 6 + 3 THEN
        "x" ::= "x" × 1;;
        "y" ::= 0
      ELSE
        SKIP
      FI)%imp.
Proof. cbv. reflexivity. Qed.

Example eg2 : parse " SKIP;; z::=x*y*(x*x);; WHILE x=x DO TEST (z <= z*z) && ~(x
= 2) THEN x ::= z;; y ::= z ELSE SKIP END;; SKIP END;; x::=z "
=
  SomeE (
      SKIP;;
      "z" ::= "x" × "y" × ("x" × "x");;
      WHILE "x" = "x" DO
        TEST ("z" ≤ "z" × "z") && ~("x" = 2) THEN
          "x" ::= "z";;
          "y" ::= "z"
        ELSE
          SKIP
```

```
        FI;;
        SKIP
      END;;
      "x" ::= "z")%imp.
Proof. cbv. reflexivity. Qed.
```

Chapter 8

Library LF.Imp

8.1 Imp: 简单的指令式程序

在本章中，我们会更加认真地看待如何用 Coq 来研究其它东西。我们的案例研究是一个名为 Imp 的'简单的指令式编程语言'，它包含了传统主流语言（如 C 和 Java）的一小部分核心片段。下面是一个用 Imp 编写的常见数学函数：

Z ::= X;; Y ::= 1;; WHILE ~(Z = 0) DO Y ::= Y * Z;; Z ::= Z - 1 END

本章关注于如何定义 Imp 的'语法'和'语义'；'《编程语言基础》（*Programming Language Foundations*）'（'《软件基础》'第二卷）中的章节则发展了'程序等价关系（*Program Equivalence*）'并引入了'霍尔逻辑（*Hoare Logic*）'，它是一种广泛用于推理指令式程序的逻辑。

```
Set Warnings "-notation-overridden,-parsing".
From Coq Require Import Bool.Bool.
From Coq Require Import Init.Nat.
From Coq Require Import Arith.Arith.
From Coq Require Import Arith.EqNat.
From Coq Require Import omega.Omega.
From Coq Require Import Lists.List.
From Coq Require Import Strings.String.
Import ListNotations.

From LF Require Import Maps.
```

8.2 算术和布尔表达式

我们会分三部分来展示 Imp：首先是'算术和布尔表达式'的核心语言，之后是用'变量'对表达式的扩展，最后是一个包括赋值、条件、串连和循环的 '指令'语言。

8.2.1 语法

```
Module AExp.
```

以下两个定义指定了算术和布尔表达式的'抽象语法（*Abstract Syntax*）'。

```
Inductive aexp : Type :=
  | ANum (n : nat)
  | APlus (a1 a2 : aexp)
  | AMinus (a1 a2 : aexp)
  | AMult (a1 a2 : aexp).

Inductive bexp : Type :=
  | BTrue
  | BFalse
  | BEq (a1 a2 : aexp)
  | BLe (a1 a2 : aexp)
  | BNot (b : bexp)
  | BAnd (b1 b2 : bexp).
```

在本章中，我们省略了大部分从程序员实际编写的具体语法到其抽象语法树的翻译例如，它会将字符串 "1 + 2 × 3" 翻译成如下 AST：

APlus (ANum 1) (AMult (ANum 2) (ANum 3)).

可选的章节 ImpParser 中开发了一个简单的词法分析器和解析器，它可以进行这种翻译。你'无需'通过理解该章来理解本章，但如果你没有上过涵盖这些技术的课程（例如编译器课程），可能想要略读一下该章节。

作为对比，下面是用约定的 BNF（巴克斯-诺尔范式）文法定义的同样的抽象语法：

a ::= nat | a + a | a - a | a * a

b ::= true | false | a = a | a <= a | ~ b | b && b

与前面的 Coq 版本相对比...

- BNF 是非形式化的 – 例如，它给出了表达式表面上的语法的建议（例如加法运算符

写作中缀的 +），而没有指定词法分析和解析的其它方面（如 +、- 和 × 的相对优先级，用括号来明确子表达式的分组等）。在实现编译器时，需要一些附加的信息（以及人类的智慧）才能将此描述转换成形式化的定义。

Coq 版本则始终忽略了所有这些信息，只专注于抽象语法。

- 反之，BNF 版本则更加清晰易读。它的非形式化使其更加灵活，在讨论和在黑板上书写时，它有很大的优势，此时传达一般的概念要比精确定下所有细节更加重要。

  确实，存在很多种类似 BNF 的记法，人们可以随意使用它们，而无需关心具体使用了哪种 BNF，因为没有必要：大致的理解是非常重要的。

适应这两种记法都很有必要：非形式化的用语人类之间的交流，而形式化的则用于实现和证明。

8.2.2 求值

对算术表达式进行'求值（*Evaluation*）'会得到数值。

```
Fixpoint aeval (a : aexp) : nat :=
  match a with
  | ANum n ⇒ n
  | APlus a1 a2 ⇒ (aeval a1) + (aeval a2)
  | AMinus a1 a2 ⇒ (aeval a1) - (aeval a2)
  | AMult a1 a2 ⇒ (aeval a1) × (aeval a2)
  end.

Example test_aeval1:
  aeval (APlus (ANum 2) (ANum 2)) = 4.
Proof. reflexivity. Qed.
```

同样，对布尔表达式求值会得到布尔值。

```
Fixpoint beval (b : bexp) : bool :=
  match b with
  | BTrue ⇒ true
  | BFalse ⇒ false
  | BEq a1 a2 ⇒ (aeval a1) =? (aeval a2)
  | BLe a1 a2 ⇒ (aeval a1) <=? (aeval a2)
```

```
  | BNot b1 ⇒ negb (beval b1)
  | BAnd b1 b2 ⇒ andb (beval b1) (beval b2)
  end.
```

8.2.3 优化

我们尚未定义太多东西，不过从这些定义出发，已经能前进不少了。假设我们定义了一个接收算术表达式并对它稍微进行化简的函数，即将所有的 0 + e（如 (APlus (ANum 0) e）化简为 e。

```
Fixpoint optimize_0plus (a:aexp) : aexp :=
  match a with
  | ANum n ⇒ ANum n
  | APlus (ANum 0) e2 ⇒ optimize_0plus e2
  | APlus e1 e2 ⇒ APlus (optimize_0plus e1) (optimize_0plus e2)
  | AMinus e1 e2 ⇒ AMinus (optimize_0plus e1) (optimize_0plus e2)
  | AMult e1 e2 ⇒ AMult (optimize_0plus e1) (optimize_0plus e2)
  end.
```

要保证我们的优化是正确的，可以在某些示例中测试它并观察其输出出否正确。

```
Example test_optimize_0plus:
  optimize_0plus (APlus (ANum 2)
                        (APlus (ANum 0)
                               (APlus (ANum 0) (ANum 1))))
  = APlus (ANum 2) (ANum 1).
Proof. reflexivity. Qed.
```

但如果要确保该优化的正确性，即优化后的表达式与原表达式的求值结果相同，那么我们应当证明它。

```
Theorem optimize_0plus_sound: ∀ a,
  aeval (optimize_0plus a) = aeval a.
Proof.
  intros a. induction a.
  - reflexivity.
  - destruct a1 eqn:Ea1.
```

```
  + destruct n eqn:En.
    × simpl. apply IHa2.
    × simpl. rewrite IHa2. reflexivity.
  +
    simpl. simpl in IHa1. rewrite IHa1.
    rewrite IHa2. reflexivity.
  +
    simpl. simpl in IHa1. rewrite IHa1.
    rewrite IHa2. reflexivity.
  +
    simpl. simpl in IHa1. rewrite IHa1.
    rewrite IHa2. reflexivity.
-
  simpl. rewrite IHa1. rewrite IHa2. reflexivity.
-
  simpl. rewrite IHa1. rewrite IHa2. reflexivity. Qed.
```

8.3 Coq 自动化

上一个证明中的大量重复很让人烦躁。无论算术表达式的语言，还是证明优化的可靠性明显都更加复杂，因此它会成为一个真正的问题。

目前为止，我们所有的证明都只使用了一点趁手的 Coq 的策略，而它自动构造部分证明的强大功能则完全被忽略了。本节引入了这样的一些功能，而在下一章中我们会看到更多。要使用它们需要耗费点精力 – Coq 的自动化是个强大的工具 – 不过它能让我们从无聊、重复、底层的细节中解放出来，专注于更加复杂的定义和更加有趣的性质。

8.3.1 泛策略

'泛策略（*Tacticals*）'是 Coq 中的术语，它表示一个接受其它策略作为参数的策略，当然，你愿意的话也可以把它称为"高阶策略"。

try 泛策略

如果 T 是一个策略，那么 try T 是一个和 T 一样的策略，只是如果 T 失败的话，try T 就会'成功地'什么也不做（而非失败）。

```
Theorem silly1 : ∀ ae, aeval ae = aeval ae.
Proof. try reflexivity. Qed.

Theorem silly2 : ∀ (P : Prop), P → P.
Proof.
  intros P HP.
  try reflexivity.    apply HP. Qed.
```

我们并没有真正的理由在像这样的手动证明中使用 try，不过在连同 ; 泛策略一起进行自动化证明时它会非常有用，接下来我们来展示它。

; 泛策略（简单形式）

在最常用的形式中，; 泛策略会接受两个策略作为参数。复合策略 $T;T'$ 会先执行 T，然后在 T 生成的'每个子目标'中执行 T'。

例如，考虑以下平凡的引理：

```
Lemma foo : ∀ n, 0 <=? n = true.
Proof.
  intros.
  destruct n.
    - simpl. reflexivity.
    - simpl. reflexivity.
Qed.
```

我们可以用 ; 泛策略来化简它：

```
Lemma foo' : ∀ n, 0 <=? n = true.
Proof.
  intros.
  destruct n;

  simpl;
```

```
  reflexivity.
Qed.
```

try 配合 ; 一同使用，我们可以从之前证明中麻烦的重复里解脱出来。

```
Theorem optimize_0plus_sound': ∀ a,
  aeval (optimize_0plus a) = aeval a.
Proof.
  intros a.
  induction a;

    try (simpl; rewrite IHa1; rewrite IHa2; reflexivity).
  - reflexivity.
  -
    destruct a1 eqn:Ea1;

      try (simpl; simpl in IHa1; rewrite IHa1;
           rewrite IHa2; reflexivity).
    + destruct n eqn:En;
      simpl; rewrite IHa2; reflexivity. Qed.
```

Coq 专家经常在像 induction 这样的策略之后使用这种"...; try... "的习语，以便一次处理所有相似的情况。自然，在非形式化证明中也有同样的做法。例如，以下对该优化定理的非形式化证明与形式化证明的结构一致：

'定理'：对于所有的算术表达式 a，

aeval (optimize_0plus a) = aeval a.

'证明'：对 a 进行归纳。大部分情况根据即可 IH 得证。其余情况如下：

- 假设设对于某些 n 有 a = ANum n for some n。我们必须证明

  aeval (optimize_0plus (ANum n)) = aeval (ANum n).

  这一点根据 optimize_0plus 的定义即可得证。

- 假设对于某些 $a1$ 和 $a2$ 有 a = APlus $a1$ $a2$。我们必须证明

  aeval (optimize_0plus (APlus a1 a2)) = aeval (APlus a1 a2).

考虑 *a1* 可能的形式。对于大部分的情况，optimize_0plus 会对子表达式简单地递归调用自身并重建一个与 *a1* 形式相同的新表达式；在这些情况下，其结果根据 IH 即可得证。

对某些 n 有 *a1* = ANum n 是个有趣的情况。若 $n = 0$，则

optimize_0plus (APlus a1 a2) = optimize_0plus a2

而 *a2* 的 IH 正是所需的。另一方面，如果对于某些 *n'* 有 n = S *n'* 那么同样 optimize_0plus 会简单地递归调用自身，而其结果根据 IH 即可得证。 □

然而，此证明仍然可以改进：第一种情况（a = ANum n）非常平凡，甚至比我们根据归纳假设化简的那个情况还要平凡，然而我们却把它完整地写了出来。为了更加清楚，最好把它去掉，然后在最上面说："大部分情况可以立即得出，或直接从归纳假设得出。唯一有趣的情况是 APlus..." 我们也可以在形式化证明中做出这种改进，方法如下：

```
Theorem optimize_0plus_sound'': ∀ a,
  aeval (optimize_0plus a) = aeval a.
Proof.
  intros a.
  induction a;

    try (simpl; rewrite IHa1; rewrite IHa2; reflexivity);

    try reflexivity.
  -
    destruct a1; try (simpl; simpl in IHa1; rewrite IHa1;
                      rewrite IHa2; reflexivity).
    + destruct n;
      simpl; rewrite IHa2; reflexivity. Qed.
```

; 泛策略（一般形式）

; 除了我们前面见到的简单形式 *T*;*T'* 外，还有种更一般的形式。如果 *T*, *T1*, ..., *Tn* 都是策略，那么

T; *T1* | *T2* | ... | *Tn*

就是一个首先执行 T，然后在 T 生成的第一个字母表中执行 $T1$，在第二个子目标中执行 $T2$，以此类推。

因此，$T;T'$ 只是一种当所有 Ti 为相同策略时的特殊记法，即，$T;T'$ 是以下形式的简写：

T; T' | T' | ... | T'

repeat 泛策略

repeat 泛策略接受另一个测略并重复应用它直至失败。以下示例用 repeat 证明了 10 在一个长列表中。

```
Theorem In10 : In 10 [1;2;3;4;5;6;7;8;9;10].
Proof.
  repeat (try (left; reflexivity); right).
Qed.
```

策略 repeat T 永远不会失败：如果策略 T 并未应用到原始目标上，那么 repeat 仍然会成功而不改变原始目标（即，它重复了零次）。

```
Theorem In10' : In 10 [1;2;3;4;5;6;7;8;9;10].
Proof.
  repeat (left; reflexivity).
  repeat (right; try (left; reflexivity)).
Qed.
```

策略 repeat T 应用 T 的次数也没有任何上界。如果 T 策略总是成功，那么重复 T 会永远循环（例如 repeat simpl 会一直循环，因为 simpl 总是会成功）。虽然 Coq 的主语言 Gallina 中的求值保证会终止，然而策略却不会！然而这并不会影响 Coq 的逻辑一致性，因为 repeat 和其它策略的工作就是指导 Coq 去构造证明；如果构造过程发散（即不停机），那就意味着我们构造证明失败，而非构造出了错误的证明。

练习：3 星, standard (optimize_0plus_b_sound) 由于 optimize_0plus 变换不会改变 **aexp** 的值，因此我们可以将它应用到所有出现在 **bexp** 中的 **aexp** 上而不改变 **bexp** 的值。请编写一个对 **bexp** 执行此变换的函数，并证明它的可靠性。利用我们刚学过的泛策略来构造一个尽可能优雅的证明。

```
Fixpoint optimize_0plus_b (b : bexp) : bexp
  . Admitted.
```

```
Theorem optimize_0plus_b_sound : ∀ b,
  beval (optimize_0plus_b b) = beval b.
Proof.
    Admitted.
    □
```

练习：4 星, standard, optional (optimize) '设计练习'：optimize_0plus 函数只是众多算术和布尔表达式优化的方法之一。请编写一个更加聪明的优化器并证明它的正确性。（最容易的方法就是从小处着手：一开始只添加单个简单的优化并证明它的正确性，然后逐渐增加其它更有趣的优化。）

8.3.2 定义新的策略记法

Coq 还提供了几种对策略脚本进行"编程"的方式。

- 下面展示的 `Tactic Notation` 习语给出了定义"简写策略"的简便方式，它将多个策略封装成单条指令。
- 对于更加复杂的编程，Coq 提供了内建的 `Ltac` 语言，它带有可以检查和修改证明状态的原语。由于详情太过复杂，这里不便展开（`Ltac` 通常也被认为不是 Coq 的设计中最美妙的部分!），你可以在参考手册和其它关于 Coq 的书中找到它，Coq 的标准库中有很多 `Ltac` 定义的例子，你也可以参考它们。
- 还有 OCaml 的 API，它可以构建从底层访问 Coq 内部结构的策略，不过普通 Coq 用于很少需要麻烦它。

`Tactic Notation` 机制是最易于掌握的，它为很多用途提供了强大的能力。下面就是个例子。

```
Tactic Notation "simpl_and_try" tactic(c) :=
  simpl;
  try c.
```

这定义了一个新的名为 *simpl_and_try* 的泛策略，它接受一个策略 `c` 作为参数，其定义等价于策略 `simpl; try c`。现在在证明中写"*simpl_and_try* `reflexivity.`"和写"`simpl; try reflexivity.`"是一样的。

8.3.3 omega 策略

omega 实现了一种决策过程，它是名为'*Presburger* 算术'的一阶逻辑的一个子集。它基于启发自 William Pugh *Pugh* 1991 (in Bib.v) 的 Omega 算法。

如果证明目标是由以下元素构成的式子：

- 数值常量、加法（+ 和 S）、减法（- 和 pred）以及常量乘法（这就是 Presburger 算术的构成要素）
- 相等关系（= 和 $\neq$）和序（$\leq$）
- 逻辑连结 $\wedge$、$\vee$、$\neg$ 和 $\rightarrow$

那么调用 omega 要么会解决该证明目标，要么就会失败，这意味着该目标为假（目标'不满足'此形式也会失败。）

```
Example silly_presburger_example : ∀ m n o p,
  m + n ≤ n + o ∧ o + 3 = p + 3 →
  m ≤ p.
Proof.
  intros. omega.
Qed.
```

（注意本文件顶部 From *Coq* Require Import omega.Omega.。）

8.3.4 更多方便的策略

最后，下面列出一些方便的其它技巧。

- clear H：从上下文中删除前提 H。
- subst x：对于变量 x，在上下文中查找假设 x = e 或 e = x，将整个上下文和当前目标中的所有 x 替换为 e 并清除该假设。
- subst：替换掉'所有'形如 x = e 或 e = x 的假设（其中 x 为变量）。
- rename... *into*...：更改证明上下文中前提的名字。例如，如果上下文中包含名为 x 的变量，那么 rename x *into* y 就会将所有出现的 x 重命名为 y。

- `assumption`：尝试在上下文中查找完全匹配目标的前提 H。如果找到了，那么其行为与 `apply` H 相同。
- *contradiction*：尝试在当前上下文中查找逻辑等价于 **False** 的前提 H。如果找到了，就解决该目标。
- `constructor`：尝试在当前环境中的 `Inductive` 定义中查找可用于解决当前目标的构造子 `c`。如果找到了，那么其行为与 `apply c` 相同。

我们之后会看到所有它们的例子。

8.4 求值作为关系

我们已经展示了用 `Fixpoint` 定义的函数 aeval 和 beval。另一种通常更加灵活的思考求值的方式，就是把它当做表达式与其值的'关系'。（译注：求值关系不满足对称性，因为它是有方向的。）这会自然地导出如下这种算术表达式的 `Inductive` 定义...

```
Module AEVALR_FIRST_TRY.

Inductive aevalR : aexp → nat → Prop :=
  | E_ANum n :
      aevalR (ANum n) n
  | E_APlus (e1 e2: aexp) (n1 n2: nat) :
      aevalR e1 n1 →
      aevalR e2 n2 →
      aevalR (APlus e1 e2) (n1 + n2)
  | E_AMinus (e1 e2: aexp) (n1 n2: nat) :
      aevalR e1 n1 →
      aevalR e2 n2 →
      aevalR (AMinus e1 e2) (n1 - n2)
  | E_AMult (e1 e2: aexp) (n1 n2: nat) :
      aevalR e1 n1 →
      aevalR e2 n2 →
      aevalR (AMult e1 e2) (n1 × n2).

Module TooHardToRead.
```

```
Inductive aevalR : aexp → nat → Prop :=
  | E_ANum n :
      aevalR (ANum n) n
  | E_APlus (e1 e2: aexp) (n1 n2: nat)
      (H1 : aevalR e1 n1)
      (H2 : aevalR e2 n2) :
      aevalR (APlus e1 e2) (n1 + n2)
  | E_AMinus (e1 e2: aexp) (n1 n2: nat)
      (H1 : aevalR e1 n1)
      (H2 : aevalR e2 n2) :
      aevalR (AMinus e1 e2) (n1 - n2)
  | E_AMult (e1 e2: aexp) (n1 n2: nat)
      (H1 : aevalR e1 n1)
      (H2 : aevalR e2 n2) :
      aevalR (AMult e1 e2) (n1 × n2).
```

Instead, we've chosen to leave the hypotheses anonymous, just giving their types. This style gives us less control over the names that Coq chooses during proofs involving **aevalR**, but it makes the definition itself quite a bit lighter.

```
End TooHardToRead.
```

如果 **aevalR** 有中缀记法的话会很方便。我们用 `e ==> n` 表示算术表达式 `e` 求值为 n。

```
Notation "e '==>' n"
         := (aevalR e n)
            (at level 50, left associativity)
         : type_scope.
```

```
End aevalR_first_try.
```

实际上，Coq 提供了一种在 **aevalR** 自身内使用此记法的方式。这样可以避免在进行涉及 `e ==> n` 形式的证明时，必须向前引用 **aevalR** `e` n 形式的定义的情况，从而减少混淆。

具体做法是，我们先"保留"该记法，然后在给出定义的同时声明它的意义。

```
Reserved Notation "e '==>' n" (at level 90, left associativity).
```

```
Inductive aevalR : aexp → nat → Prop :=
  | E_ANum (n : nat) :
      (ANum n) ==> n
  | E_APlus (e1 e2 : aexp) (n1 n2 : nat) :
      (e1 ==> n1) → (e2 ==> n2) → (APlus e1 e2) ==> (n1 + n2)
  | E_AMinus (e1 e2 : aexp) (n1 n2 : nat) :
      (e1 ==> n1) → (e2 ==> n2) → (AMinus e1 e2) ==> (n1 - n2)
  | E_AMult (e1 e2 : aexp) (n1 n2 : nat) :
      (e1 ==> n1) → (e2 ==> n2) → (AMult e1 e2) ==> (n1 × n2)

  where "e '==>' n" := (aevalR e n) : type_scope.
```

8.4.1 推理规则的记法

在非形式化的讨论中，将 **aevalR** 和类似的关系写成更加易读的 '推理规则（*Inference Rule*）'的图像形式会十分方便，其中横线上方的前提证明了横线下方的结论是正确的（我们已经在 IndProp 一章中见过它们了）。

例如，构造子 E_APlus...

| E_APlus : forall (e1 e2: aexp) (n1 n2: nat), aevalR e1 n1 -> aevalR e2 n2 -> aevalR (APlus e1 e2) (n1 + n2)

...的推理规则写作：

e1 ==> n1 e2 ==> n2

(E_APlus) APlus e1 e2 ==> n1+n2

形式上来说，推理规则没什么深刻的含义：它们只是蕴含关系。你可以将右侧的规则名看做构造子名，将横线上每个前提见的换行（以及横线本身）看做 $\rightarrow$。规则中涉及的所有变量（如 *e1*、*n1* 等）从一开始都是全称量化的。（这种变量通常称为'元变量（*Metavariables*）'，以区别于我们在语言中定义的变量。目前，我们的算术表达式中还不包含变量，不过之后会加入它们。）整个规则集合都被认为包裹在 `Inductive` 声明中。在非正式的叙述中，这一点要么会忽略，要么会通过类似于"令 **aevalR** 为对以下规则封闭的最小关系..." 的句子来表述。

例如，==> 是对以下规则封闭的最小关系：

(E_ANum) ANum n ==> n

e1 ==> n1 e2 ==> n2

(E_APlus) APlus e1 e2 ==> n1+n2

e1 ==> n1 e2 ==> n2

(E_AMinus) AMinus e1 e2 ==> n1-n2

e1 ==> n1 e2 ==> n2

(E_AMult) AMult e1 e2 ==> n1*n2

练习：1 星, standard, optional (beval_rules) 下面是 Coq 中 beval 函数的定义：

Fixpoint beval (e : bexp) : bool := match e with | BTrue => true | BFalse => false | BEq a1 a2 => (aeval a1) =? (aeval a2) | BLe a1 a2 => (aeval a1) <=? (aeval a2) | BNot b1 => negb (beval b1) | BAnd b1 b2 => andb (beval b1) (beval b2) end.

请用推理规则记法将布尔求值的定义写成关系的形式。

```
Definition manual_grade_for_beval_rules : option (nat×string) := None.
```

□

8.4.2 定义的等价关系

求值的函数式定义与关系式定义之间的一致性证明非常直观：

```
Theorem aeval_iff_aevalR : ∀ a n,
  (a ==> n) ↔ aeval a = n.
Proof.
 split.
 -
   intros H.
   induction H; simpl.
   +
     reflexivity.
   +
     rewrite IHaevalR1. rewrite IHaevalR2. reflexivity.
```

```
  +
    rewrite IHaevalR1. rewrite IHaevalR2. reflexivity.
  +
    rewrite IHaevalR1. rewrite IHaevalR2. reflexivity.
-
  generalize dependent n.
  induction a;
    simpl; intros; subst.
  +
    apply E_ANum.
  +
    apply E_APlus.
     apply IHa1. reflexivity.
     apply IHa2. reflexivity.
  +
    apply E_AMinus.
     apply IHa1. reflexivity.
     apply IHa2. reflexivity.
  +
    apply E_AMult.
     apply IHa1. reflexivity.
     apply IHa2. reflexivity.
Qed.
```

我们可以利用泛策略将此证明缩减到很短。

```
Theorem aeval_iff_aevalR' : ∀ a n,
  (a ==> n) ↔ aeval a = n.
Proof.
  split.
  -
    intros H; induction H; subst; reflexivity.
  -
    generalize dependent n.
```

```
  induction a; simpl; intros; subst; constructor;
    try apply IHa1; try apply IHa2; reflexivity.
Qed.
```

练习：3 星, standard (bevalR) 用和 **aevalR** 同样的方式写出关系 **bevalR**，并证明它等价于 beval。

```
Inductive bevalR: bexp → bool → Prop :=

.

Lemma beval_iff_bevalR : ∀ b bv,
  bevalR b bv ↔ beval b = bv.
Proof.
  Admitted.
```

□

```
End AExp.
```

8.4.3 计算式定义与关系式定义

对于算术和布尔表达式求值的定义方式而言，函数式和关系式二者均可，选择哪种主要取决于品味。

然而在某些情况下，求值的关系式定义比函数式定义要更好。

```
Module AEvalR_division.
```

例如，假设我们想要用除法来扩展算术运算：

```
Inductive aexp : Type :=
  | ANum (n : nat)
  | APlus (a1 a2 : aexp)
  | AMinus (a1 a2 : aexp)
  | AMult (a1 a2 : aexp)
  | ADiv (a1 a2 : aexp).
```

扩展 aeval 的定义来处理此讯算并不是很直观（我们要返回什么作为 ADiv (ANum 5) (ANum 0) 的结果？）。然而扩展 **aevalR** 却很简单。

```
Reserved Notation "e '==>' n"
```

```
                    (at level 90, left associativity).
Inductive aevalR : aexp → nat → Prop :=
  | E_ANum (n : nat) :
      (ANum n) ==> n
  | E_APlus (a1 a2 : aexp) (n1 n2 : nat) :
      (a1 ==> n1) → (a2 ==> n2) → (APlus a1 a2) ==> (n1 + n2)
  | E_AMinus (a1 a2 : aexp) (n1 n2 : nat) :
      (a1 ==> n1) → (a2 ==> n2) → (AMinus a1 a2) ==> (n1 - n2)
  | E_AMult (a1 a2 : aexp) (n1 n2 : nat) :
      (a1 ==> n1) → (a2 ==> n2) → (AMult a1 a2) ==> (n1 × n2)
  | E_ADiv (a1 a2 : aexp) (n1 n2 n3 : nat) :
      (a1 ==> n1) → (a2 ==> n2) → (n2 > 0) →
      (mult n2 n3 = n1) → (ADiv a1 a2) ==> n3

where "a '==>' n" := (aevalR a n) : type_scope.
```

Notice that the evaluation relation has now become *partial*: There are some inputs for which it simply does not specify an output.

```
End AEVALR_DIVISION.

Module AEVALR_EXTENDED.
```

假设我们想要用非确定性的数值生成器 *any* 来扩展算术运算，该生成器会在求值时产生任何数。（注意，这不同于在所有可能的数值中作出 '概率上的'选择 – 我们没有为结果指定任何具体的概率分布，只是说了 '可能的结果'。）

```
Reserved Notation "e '==>' n" (at level 90, left associativity).

Inductive aexp : Type :=
  | AAny
  | ANum (n : nat)
  | APlus (a1 a2 : aexp)
  | AMinus (a1 a2 : aexp)
  | AMult (a1 a2 : aexp).
```

同样，扩展 aeval 会很棘手，因为现在的求值'并不是'一个从表达式到数值的确定性函数，而扩展 **aevalR** 则无此问题...

```
Inductive aevalR : aexp → nat → Prop :=
  | E_Any (n : nat) :
      AAny ==> n
  | E_ANum (n : nat) :
      (ANum n) ==> n
  | E_APlus (a1 a2 : aexp) (n1 n2 : nat) :
      (a1 ==> n1) → (a2 ==> n2) → (APlus a1 a2) ==> (n1 + n2)
  | E_AMinus (a1 a2 : aexp) (n1 n2 : nat) :
      (a1 ==> n1) → (a2 ==> n2) → (AMinus a1 a2) ==> (n1 - n2)
  | E_AMult (a1 a2 : aexp) (n1 n2 : nat) :
      (a1 ==> n1) → (a2 ==> n2) → (AMult a1 a2) ==> (n1 × n2)

where "a '==>' n" := (aevalR a n) : type_scope.

End AEVALR_EXTENDED.
```

这时你可能会问：默认情况下应该使用哪种风格？我们刚看到的例子表明关系式的定义反而比函数式的更加有用。对于这种定义的东西不太容易用函数表达，或者确实'不是'函数的情况来说，明显别无选择。但如果两种风格均可行呢？

关系式定义的一个优点是，它们会更优雅，更容易理解。

另一个优点是，Coq 会根据 `Inductive` 定义自动生成不错的反演函数和归纳法则。

另一方面，函数式定义通常会更方便：

- 函数的定义是确定性的，且在所有参数上定义；而对于关系式定义来说，我们需要这些性质时必须显式地'证明'它们。
- 有了函数，我们还可以利用 Coq 的计算机制在证明过程中简化表达式。

此外，函数还可以直接从 Gallina"提取"出 OCaml 或 Haskell 的可执行代码。

最终，选择视具体情况而定，或者只是品味问题。确实，在大型的 Coq 开发中，经常可以看到一个定义同时给出了函数式和关系式'两种'风格，加上一条陈述了二者等价的引理，以便在之后的证明中能够在这两种视角下随意切换。

8.5 带变量的表达式

让我们回到 Imp 的定义上来。接下来我们要为算术和布尔表达式加上变量。为简单起

见，我们会假设所有变量是都全局的，且它们只保存数值。

8.5.1 状态

由于需要查找变量来获得它们的具体值，因此我们重用了 Maps 一章中的映射。我们在 Imp 中以 string 作为变量的类型。

'机器的状态'（简称'状态'）表示程序执行中某一时刻'所有变量'的值。

虽然任何给定的程序只会涉及有限数量的变量，不过为简单起见，我们假设状态为'所有的'变量定义。状态会捕获内存中所有的信息。对 Imp 程序而言，由于每个变量都存储了一个自然数，因此我们可以将状态表示为一个从字符串到 **nat** 的映射，并且用 0 作为存储中的默认值。对于更复杂的编程语言，状态会有更多结构。

```
Definition state := total_map nat.
```

8.5.2 语法

我们只需为之前的算术表达式再加一个构造子就能添加变量：

```
Inductive aexp : Type :=
  | ANum (n : nat)
  | AId (x : string)
  | APlus (a1 a2 : aexp)
  | AMinus (a1 a2 : aexp)
  | AMult (a1 a2 : aexp).
```

为几个变量名定义简单记法能让示例更加易读：

```
Definition W : string := "W".
Definition X : string := "X".
Definition Y : string := "Y".
Definition Z : string := "Z".
```

（这种命名程序变量的约定（X、Y、Z）与我们之前使用大写字母表示类型有点冲突。由于我们在本章中开发 Imp 时没怎么使用多态，所以这种重载应该不会产生混淆。）

bexp 的定义现在除了引用了新的 **aexp** 之外并未更改：

```
Inductive bexp : Type :=
  | BTrue
```

```
| BFalse
| BEq (a1 a2 : aexp)
| BLe (a1 a2 : aexp)
| BNot (b : bexp)
| BAnd (b1 b2 : bexp).
```

8.5.3 记法

要让 Imp 程序更易读写，我们引入了一些记法和隐式转换（Coercion）。

你无需理解以下声明具体做了些什么。简言而之，Coq 中的 `Coercion` 声明规定了一个函数（或构造子）可以被类型系统隐式地用于将一个输入类型的值转换成输出类型的值。例如，AId 的转换声明在需要一个 **aexp** 时直接使用普通的字符串，该字符串会被隐式地用 AId 来包装。

下列记法在具体的'记法作用域'中声明，以避免与其它符号相同的解释相冲突。同样，你暂时也无需理解其中的细节，但要意识到到我们为 $+$、-、$\times$、$=$、$\leq$ 等运算符定义了'新的'解释十分重要。

```
Coercion AId : string >-> aexp.
Coercion ANum : nat >-> aexp.

Definition bool_to_bexp (b : bool) : bexp :=
  if b then BTrue else BFalse.
Coercion bool_to_bexp : bool >-> bexp.

Bind Scope imp_scope with aexp.
Bind Scope imp_scope with bexp.
Delimit Scope imp_scope with imp.

Notation "x + y" := (APlus x y) (at level 50, left associativity) : imp_scope.
Notation "x - y" := (AMinus x y) (at level 50, left associativity) : imp_scope.
Notation "x * y" := (AMult x y) (at level 40, left associativity) : imp_scope.
Notation "x <= y" := (BLe x y) (at level 70, no associativity) : imp_scope.
Notation "x = y" := (BEq x y) (at level 70, no associativity) : imp_scope.
Notation "x && y" := (BAnd x y) (at level 40, left associativity) : imp_scope.
Notation "'~' b" := (BNot b) (at level 75, right associativity) : imp_scope.
```

现在我们可以用 3 + (X × 2) 来代替 APlus 3 (AMult X 2) 了，同样可以用 true && !(X

≤ 4) 来代替 BAnd true (BNot (BLe X 4))。然而这只有在我们告诉 Coq 在恰当的记法作用域中用 %*imp* 解析我们的表达式时，或者当 Coq 知道该表达式的类型（**aexp** 或 **bexp**）被「限定」为使用该记法作用域时才可行。

```
Definition example_aexp : aexp := 3 + (X × 2).
Definition example_bexp : bexp := true && ~(X ≤ 4).
```

强制转换有一点不便之处，即它会略微提高人类推导表达式类型的难度。如果你感到有点困惑，请用 `Set Printing` *Coercions* 来查看具体发生了什么。

```
Set Printing Coercions.

Print example_bexp.

Unset Printing Coercions.
```

8.5.4 求值

算术和布尔求值器被扩展成以很显然的方式来处理变量，它接受一个状态作为额外的参数：

```
Fixpoint aeval (st : state) (a : aexp) : nat :=
  match a with
  | ANum n ⇒ n
  | AId x ⇒ st x
  | APlus a1 a2 ⇒ (aeval st a1) + (aeval st a2)
  | AMinus a1 a2 ⇒ (aeval st a1) - (aeval st a2)
  | AMult a1 a2 ⇒ (aeval st a1) × (aeval st a2)
  end.

Fixpoint beval (st : state) (b : bexp) : bool :=
  match b with
  | BTrue ⇒ true
  | BFalse ⇒ false
  | BEq a1 a2 ⇒ (aeval st a1) =? (aeval st a2)
  | BLe a1 a2 ⇒ (aeval st a1) <=? (aeval st a2)
  | BNot b1 ⇒ negb (beval st b1)
  | BAnd b1 b2 ⇒ andb (beval st b1) (beval st b2)
  end.
```

我们为具体状态的全映射声明具体的记法，即使用 (_ !-> 0) 作为空状态。

```
Definition empty_st := (_ !-> 0).
```

现在我们可以为“单例状态（singleton state）”添加新的记法了，即只有一个绑定到值的变量。 `Notation "x '!->' v" := (t_update empty_st x v) (at level 100).`

```
Example aexp1 :
    aeval (X !-> 5) (3 + (X × 2))
  = 13.
Proof. reflexivity. Qed.

Example bexp1 :
    beval (X !-> 5) (true && ~(X ≤ 4))
  = true.
Proof. reflexivity. Qed.
```

8.6 指令

现在我们可以定义 Imp '指令（*Command*）'（有时称作'语句（*Statement*）'）的语法和行为了。

8.6.1 语法

指令 c 可以用以下 BNF 文法非形式化地描述。

c ::= SKIP | x ::= a | c ;; c | TEST b THEN c ELSE c FI | WHILE b DO c END

（为了能够使用 Coq 的记法机制来定义 Imp 语法，我们选择了这种略尴尬的具体语法。具体来说，我们使用了 *TEST* 来避免与表中库中的 `if` 记法相冲突。）例如，下面是用 Imp 编写的阶乘：

Z ::= X;; Y ::= 1;; WHILE ~(Z = 0) DO Y ::= Y * Z;; Z ::= Z - 1 END

当此指令终止后，Y 会保存初始值 X 的阶乘。

下面是指令的抽象语法的形式化定义：

```
Inductive com : Type :=
  | CSkip
  | CAss (x : string) (a : aexp)
  | CSeq (c1 c2 : com)
```

```
  | CIf (b : bexp) (c1 c2 : com)
  | CWhile (b : bexp) (c : com).
```

至于表达式，我们可以用一些 `Notation` 声明来让 Imp 程序的读写更加方便。

```
Bind Scope imp_scope with com.
Notation "'SKIP'" :=
  CSkip : imp_scope.
Notation "x '::=' a" :=
  (CAss x a) (at level 60) : imp_scope.
Notation "c1 ;; c2" :=
  (CSeq c1 c2) (at level 80, right associativity) : imp_scope.
Notation "'WHILE' b 'DO' c 'END'" :=
  (CWhile b c) (at level 80, right associativity) : imp_scope.
Notation "'TEST' c1 'THEN' c2 'ELSE' c3 'FI'" :=
  (CIf c1 c2 c3) (at level 80, right associativity) : imp_scope.
```

例如，下面是个阶乘函数，写成 Coq 的形式化定义：

```
Definition fact_in_coq : com :=
  (Z ::= X;;
  Y ::= 1;;
  WHILE ~(Z = 0) DO
    Y ::= Y × Z;;
    Z ::= Z - 1
  END).
```

8.6.2 脱糖记法

Coq 为管理日益复杂的工作对象提供了丰富的特性，例如隐式转换和记法。然而，过度使用它们会产生繁杂的语法。为了教学，我们通常会用以下命令来“关闭”这些特性以获得对事物更加本质的描述：

- `Unset Printing` *Notations*（用 `Set Printing` *Notations* 撤销）
- `Set Printing` *Coercions*（用 `Unset Printing` *Coercions* 撤销）
- `Set Printing All`（用 `Unset Printing All` 撤销）

这些命令也可在证明过程中详述当前目标和上下文。

```
Unset Printing Notations.
Print fact_in_coq.
Set Printing Notations.
Set Printing Coercions.
Print fact_in_coq.
Unset Printing Coercions.
```

8.6.3 Locate 命令

查询记法

当遇到未知记法时，可使用 Locate 后跟一个包含其符号的'字符串'来查看其可能的解释。 `Locate "&&".`

```
Locate ";;".
Locate "WHILE".
```

查询标识符

当以标示符使用 Locate 时，它会打印作用域中同名的所有值的完成路径。它很适合解决由变量覆盖所引起的问题。 `Locate aexp.`

8.6.4 更多示例

赋值：

```
Definition plus2 : com :=
  X ::= X + 2.

Definition XtimesYinZ : com :=
  Z ::= X × Y.

Definition subtract_slowly_body : com :=
  Z ::= Z - 1 ;;
  X ::= X - 1.
```

循环

```
Definition subtract_slowly : com :=
  (WHILE ~(X = 0) DO
    subtract_slowly_body
  END).

Definition subtract_3_from_5_slowly : com :=
  X ::= 3 ;;
  Z ::= 5 ;;
  subtract_slowly.
```

无限循环：

```
Definition loop : com :=
  WHILE true DO
    SKIP
  END.
```

8.7 求值指令

接下来我们要定义对 Imp 指令进行求值是什么意思。 *WHILE* 未必会终止的事实让定义它的求值函数有点棘手...

8.7.1 求值作为函数（失败的尝试）

下面是一次为指令定义求值函数的尝试，我们忽略了 *WHILE* 的情况。

为了在模式匹配中使用记法，我们需要以下声明。 `Open Scope imp_scope.`

```
Fixpoint ceval_fun_no_while (st : state) (c : com)
                            : state :=
  match c with
    | SKIP ⇒
        st
    | x ::= a1 ⇒
        (x !-> (aeval st a1) ; st)
```

```
  | c1 ;; c2 ⇒
      let st' := ceval_fun_no_while st c1 in
      ceval_fun_no_while st' c2
  | TEST b THEN c1 ELSE c2 FI ⇒
      if (beval st b)
        then ceval_fun_no_while st c1
        else ceval_fun_no_while st c2
  | WHILE b DO c END ⇒
      st
  end.
Close Scope imp_scope.
```

在 OCaml 或 Haskell 这类传统的函数式编程语言中，我们可以像下面这样添加 *WHILE* 的情况：

Fixpoint ceval_fun (st : state) (c : com) : state := match c with ... | WHILE b DO c END => if (beval st b) then ceval_fun st (c ;; WHILE b DO c END) else st end.

Coq 会拒绝这种定义（"Error: Cannot guess decreasing argument of fix"，错误：无法猜测出固定的递减参数），因为我们想要定义的函数并不保证终止。确实，它并不'总是会终止'：例如，*ceval_fun* 函数应用到上面的 loop 程序的完整版本永远不会终止。由于 Coq 不仅是一个函数式编程语言，还是个逻辑一致的语言，因此任何潜在的不可终止函数都会被拒绝。下面是一个（无效的）程序，它展示了如果 Coq 允许不可终止的递归函数的话会产生什么错误：

Fixpoint loop_false (n : nat) : False := loop_false n.

也就是说，像 **False** 这样的假命题可以被证明（*loop_false* 0 会是 **False** 的一个证明），这对于 Coq 的逻辑一致性来说是个灾难。

由于它对于所有的输入都不会终止，因此 *ceval_fun* 无法在 Coq 中写出至少需要一些技巧和变通才行（如果你对此好奇的话请阅读 ImpCEvalFun 一章）。

8.7.2 求值作为一种关系

下面是一种更好的方法：将 **ceval** 定义成一种'关系'而非一个'函数' – 即，用 `Prop` 而非用 `Type` 来定义它，和我们前面对 **aevalR** 做的那样。

这是一个非常重要的更改。除了能将我们从尴尬的变通中解放出来之外，它还给我们的定义赋予了更多的灵活性。例如，如果我们要为该语言添加更多像 *any* 这样非确定性的

特性，我们需要让求值的定义也是非确定性的 – 即，它不仅会有不完全性，甚至还可以不是个函数！

我们将使用记法 *st* =[c]=> *st'* 来表示 **ceval** 这种关系：*st* =[c]=> *st'* 表示在开始状态 *st* 下启动程序并在结束状态 *st'* 下产生结果。它可以读作："c 将状态 *st* 变成 *st*'"。

操作语义

下面是求值的非形式化定义，为了可读性表示成推理规则：

(E_Skip) st = *SKIP* => st
aeval st a1 = n

(E_Ass) st = *x* := *a1* => (x !-> n ; st)
st = c1 => st' st' = c2 => st"

(E_Seq) st = c1;;c2 => st"
beval st b1 = true st = c1 => st'

(E_IfTrue) st = *TEST b1 THEN* c1 *ELSE* c2 *FI* => st'
beval st b1 = false st = c2 => st'

(E_IfFalse) st = *TEST b1 THEN* c1 *ELSE* c2 *FI* => st'
beval st b = false

(E_WhileFalse) st = *WHILE* b *DO* c *END* => st
beval st b = true st = c => st' st' = *WHILE* b *DO* c *END* => st"

(E_WhileTrue) st = *WHILE* b *DO* c *END* => st"

下面是它的形式化定义。请确保你理解了它是如何与以上推理规则相对应的。

```
Reserved Notation "st '=[' c ']=>' st'"
                  (at level 40).

Inductive ceval : com → state → state → Prop :=
  | E_Skip : ∀ st,
```

```
      st =[ SKIP ]=> st
  | E_Ass : ∀ st a1 n x,
      aeval st a1 = n →
      st =[ x ::= a1 ]=> (x !-> n ; st)
  | E_Seq : ∀ c1 c2 st st' st'',
      st =[ c1 ]=> st' →
      st' =[ c2 ]=> st'' →
      st =[ c1 ;; c2 ]=> st''
  | E_IfTrue : ∀ st st' b c1 c2,
      beval st b = true →
      st =[ c1 ]=> st' →
      st =[ TEST b THEN c1 ELSE c2 FI ]=> st'
  | E_IfFalse : ∀ st st' b c1 c2,
      beval st b = false →
      st =[ c2 ]=> st' →
      st =[ TEST b THEN c1 ELSE c2 FI ]=> st'
  | E_WhileFalse : ∀ b st c,
      beval st b = false →
      st =[ WHILE b DO c END ]=> st
  | E_WhileTrue : ∀ st st' st'' b c,
      beval st b = true →
      st =[ c ]=> st' →
      st' =[ WHILE b DO c END ]=> st'' →
      st =[ WHILE b DO c END ]=> st''

  where "st =[ c ]=> st'" := (ceval c st st').
```

将求值定义成关系而非函数的代价是，我们需要自己为某个程序求值成某种结束状态'构造证明'，而不能只是交给 Coq 的计算机制去做了。

```
Example ceval_example1:
  empty_st =[
     X ::= 2;;
     TEST X ≤ 1
```

```
      THEN Y ::= 3
      ELSE Z ::= 4
    FI
  ]=> (Z !-> 4 ; X !-> 2).
Proof.
  apply E_Seq with (X !-> 2).
  -
    apply E_Ass. reflexivity.
  -
    apply E_IfFalse.
    reflexivity.
    apply E_Ass. reflexivity.
Qed.
```

练习：2 星, standard (ceval_example2) `Example ceval_example2:`

```
  empty_st =[
    X ::= 0;; Y ::= 1;; Z ::= 2
  ]=> (Z !-> 2 ; Y !-> 1 ; X !-> 0).
Proof.
    Admitted.
```

□

练习：3 星, standard, optional (pup_to_n) 写一个 Imp 程序对从 1 到 X 进行求值（包括：将 $1 + 2 + ... + X$) 赋予变量 Y。证明此程序对于 $X = 2$ 会按预期执行（这可能比你预想的还要棘手）。

```
Definition pup_to_n : com
  . Admitted.

Theorem pup_to_2_ceval :
  (X !-> 2) =[
    pup_to_n
  ]=> (X !-> 0 ; Y !-> 3 ; X !-> 1 ; Y !-> 2 ; Y !-> 0 ; X !-> 2).
Proof.
    Admitted.
```

□

8.7.3 求值的确定性

将求值从计算式定义换成关系式定义是个不错的改变，因为它将我们从求值必须是全函数的人工需求中解放了出来。不过这仍然引发了一个问题：求值的第二种定义真的是一个偏函数吗？从同样的 *st* 开始，我们是否有可能按照不同的方式对某个指令 c 进行求值，从而到达两个不同的输出状态 *st'* 和 *st''*？

实际上这不可能发生，因为 **ceval** '确实'是一个偏函数：

```
Theorem ceval_deterministic: ∀ c st st1 st2,
     st =[ c ]=> st1 →
     st =[ c ]=> st2 →
     st1 = st2.
Proof.
  intros c st st1 st2 E1 E2.
  generalize dependent st2.
  induction E1; intros st2 E2; inversion E2; subst.
  - reflexivity.
  - reflexivity.
  -
    assert (st' = st'0) as EQ1.
    { apply IHE1_1; assumption. }
    subst st'0.
    apply IHE1_2. assumption.
  -
      apply IHE1. assumption.
  -
      rewrite H in H5. discriminate H5.
  -
    rewrite H in H5. discriminate H5.
  -
      apply IHE1. assumption.
  -
```

```
  reflexivity.
-
  rewrite H in H2. discriminate H2.
-
  rewrite H in H4. discriminate H4.
-
    assert (st' = st'0) as EQ1.
    { apply IHE1_1; assumption. }
    subst st'0.
    apply IHE1_2. assumption. Qed.
```

8.8 对 Imp 进行推理

我们会在'《编程语言基础》'中更加深入地探讨对 Imp 程序进行推理的系统性技术，不过目前只根据定义就能做很多工作。本节中会我们会探讨一些实例。

```
Theorem plus2_spec : ∀ st n st',
  st X = n →
  st =[ plus2 ]=> st' →
  st' X = n + 2.
Proof.
  intros st n st' HX Heval.
```

反转 *Heval* 实际上会强制让 Coq 展开 **ceval** 求值的一个步骤 – 由于 plus2 是一个赋值，因此这种情况揭示了 *st'* 一定是 *st* 通过新的值 X 扩展而来的。

```
  inversion Heval. subst. clear Heval. simpl.
  apply t_update_eq. Qed.
```

练习：3 星, standard, optional (XtimesYinZ_spec) 叙述并证明 XtimesYinZ 的规范（Specification）。

```
Definition manual_grade_for_XtimesYinZ_spec : option (nat×string) := None.
```

□

练习：3 星, standard, recommended (loop_never_stops) Theorem loop_never_stops : ∀ *st st'*,

```
  ~(st =[ loop ]=> st').
Proof.
  intros st st' contra. unfold loop in contra.
  remember (WHILE true DO SKIP END)%imp as loopdef
           eqn:Heqloopdef.
```

归纳讨论假设"*loopdef* 会终止"之构造，其中多数情形的矛盾显而易见，可用 discriminate 一步解决。

```
  Admitted.
```

□

练习：3 星, standard (no_whiles_eqv) 考虑以下函数：

```
Open Scope imp_scope.
Fixpoint no_whiles (c : com) : bool :=
  match c with
  | SKIP ⇒
      true
  | _ ::= _ ⇒
      true
  | c1 ;; c2 ⇒
      andb (no_whiles c1) (no_whiles c2)
  | TEST _ THEN ct ELSE cf FI ⇒
      andb (no_whiles ct) (no_whiles cf)
  | WHILE _ DO _ END ⇒
      false
  end.
Close Scope imp_scope.
```

此断言只对没有 *WHILE* 循环的程序产生 true。请用 Inductive 写出一个性质 **no_whilesR** 使得 **no_whilesR** c 仅当 c 是个没有 *WHILE* 循环的程序时才可以证明。之后证明它与 no_whiles 等价。

```
Inductive no_whilesR: com → Prop :=
```

.

Theorem no_whiles_eqv:

∀ c, no_whiles c = true ↔ **no_whilesR** c.

Proof.

Admitted.

□

练习：4 星, standard (no_whiles_terminating) 不涉及 *WHILE* 循环的 Imp 程序一定会终止。请陈述并证明定理 *no_whiles_terminating* 来说明这一点。

按照你的偏好使用 no_whiles 或 **no_whilesR**。

Definition manual_grade_for_no_whiles_terminating : **option** (**nat**×**string**) := None.

□

8.9 附加练习

练习：3 星, standard (stack_compiler) 旧式惠普计算器的编程语言类似于 Forth 和 Postscript，而其抽象机器类似于 Java 虚拟机，即所有对算术表达式的求值都使用'栈'来进行。例如，表达式

(2*3)+(3*(4-2))

会被写成

2 3 * 3 4 2 - * +

的形式，其求值过程如下（右侧为被求值的程序，左侧为栈中的内容）：

| 2 3 * 3 4 2 - * + 2 | 3 * 3 4 2 - * + 3, 2 | * 3 4 2 - * + 6 | 3 4 2 - * + 3, 6 | 4 2 - * + 4, 3, 6 | 2 - * + 2, 4, 3, 6 | - * + 2, 3, 6 | * + 6, 6 | + 12 |

此练习的目的在于编写一个小型编译器，它将 **aexp** 翻译成栈机器指令。

栈语言的指令集由以下指令构成：

- SPush n：将数 n 压栈。
- SLoad x：从存储中加载标识符 x 并将其压栈。
- SPlus：从栈顶弹出两个数，将二者相加，并将其结果压栈。

- SMinus：类似，不过执行减法。
- SMult：类似，不过执行乘法。

```
Inductive sinstr : Type :=
| SPush (n : nat)
| SLoad (x : string)
| SPlus
| SMinus
| SMult.
```

请编写一个函数对栈语言程序进行求值。它应当接受一个状态、一个表示为数字列表的栈（栈顶项在表头），以及一个表示为指令列表的程序作为输入，并在程序执行后返回该栈。请在以下示例中测试你的函数。

注意，当栈中的元素少于两个时，规范并未指定 SPlus、SMinus 或 SMult 指令的行为。从某种意义上说，这样做并无必要，因为我们的编译器永远不会产生这种畸形的程序。

```
Fixpoint s_execute (st : state) (stack : list nat)
                   (prog : list sinstr)
                 : list nat
  . Admitted.

Example s_execute1 :
     s_execute empty_st []
       [SPush 5; SPush 3; SPush 1; SMinus]
   = [2; 5].
   Admitted.

Example s_execute2 :
     s_execute (X !-> 3) [3;4]
       [SPush 4; SLoad X; SMult; SPlus]
   = [15; 4].
   Admitted.
```

接下来请编写一个将 **aexp** 编译成栈机器程序的函数。运行此程序的效果应当和将该表达式的值压入栈中一致。

```
Fixpoint s_compile (e : aexp) : list sinstr
```

. *Admitted*.

在定义完 s_compile 之后，请证明以下示例来测试它是否起作用。

```
Example s_compile1 :
  s_compile (X - (2 × Y))
  = [SLoad X; SPush 2; SLoad Y; SMult; SMinus].
```

Admitted.

□

练习：4 星, advanced (stack_compiler_correct) 现在我们将证明在之前练习中实现的编译器的正确性。记住当栈中的元素少于两个时，规范并未指定 SPlus、SMinus 或 SMult 指令的行为。（为了让正确性证明更加容易，你可能需要返回去修改你的实现！）

请证明以下定理。你需要先陈述一个更一般的引理来得到一个有用的归纳假设，由它的话主定理就只是一个简单的推论了。

```
Theorem s_compile_correct : ∀ (st : state) (e : aexp),
  s_execute st [] (s_compile e) = [ aeval st e ].
Proof.
```

Admitted.

□

练习：3 星, standard, optional (short_circuit) 大部分现代编程语言对布尔 **and** 运算提供了“短路求值”的方法：要对 BAnd *b1 b2* 进行求值，首先对 *b1* 求值。如果结果为 false，那么整个 BAnd 表达式的求值就是 false，而无需对 *b2* 求值。否则，*b2* 的求值结果就决定了 BAnd 表达式的值。

请编写 beval 的另一种版本，它能以这种方式对 BAnd 执行短路求值，并证明它等价于 beval。（注：二者等价只是因为 Imp 的表达式求值相当简单。在更大的语言中该表达式可能会发散，此时短路求值的 BAnd *'并不'* 等价于原始版本，因为它能让更多程序终止。）

```
Module BreakImp.
```

练习：4 星, advanced (break_imp) 像 C 和 Java 这样的指令式语言通常会包含 *break* 或类似地语句来中断循环的执行。在本练习中，我们考虑如何为 Imp 加上 *break*。首先，我们需要丰富语言的指令。

```
Inductive com : Type :=
  | CSkip
  | CBreak
  | CAss (x : string) (a : aexp)
  | CSeq (c1 c2 : com)
  | CIf (b : bexp) (c1 c2 : com)
  | CWhile (b : bexp) (c : com).

Notation "'SKIP'" :=
  CSkip.
Notation "'BREAK'" :=
  CBreak.
Notation "x '::=' a" :=
  (CAss x a) (at level 60).
Notation "c1 ;; c2" :=
  (CSeq c1 c2) (at level 80, right associativity).
Notation "'WHILE' b 'DO' c 'END'" :=
  (CWhile b c) (at level 80, right associativity).
Notation "'TEST' c1 'THEN' c2 'ELSE' c3 'FI'" :=
  (CIf c1 c2 c3) (at level 80, right associativity).
```

接着，我们需要定义 *BREAK* 的行为。非形式化地说，只要 *BREAK* 在指令序列中执行，它就会终止该序列的执行并发出最内层围绕它的循环应当终止的信号。（如果没有任何围绕它的循环，那么就终止整个程序。）最终状态应当与 *BREAK* 语句执行后的状态相同。

其要点之一在于当一个给定的 *BREAK* 位于多个循环中时应该做什么。此时，*BREAK* 应当只终止'*最内层*'的循环。因此，在执行完以下指令之后...

X ::= 0;; Y ::= 1;; WHILE ~(0 = Y) DO WHILE true DO BREAK END;; X ::= 1;; Y ::= Y - 1 END

...X 的值应为 1 而非 0。

表达这种行为的一种方式求值为求值关系添加一个形参，指定某个指令是否会执行 *BREAK* 语句：

```
Inductive result : Type :=
  | SContinue
```

| SBreak.

```
Reserved Notation "st '=[' c ']=>' st' '/' s"
         (at level 40, st' at next level).
```

直觉上说，st =[c]=> st' / s 表示如果 c 在 st 状况下开始，它会在 st' 状态下终止，围绕它的最内层循环（或整个程序）要么收到立即退出的信号（s = SBreak），要么继续正常执行（s = SContinue）。

"st =[c]=> st' / s"关系的定义非常类似于之前我们为一般求值关系（st =[c]=> st'）给出的定义 – 我们只需要恰当地处理终止信号。

- 若指令为 $SKIP$，则状态不变，任何围绕它的循环继续正常执行。
- 若指令为 $BREAK$，则状态保持不变但发出 SBreak 的信号。
- 若指令为赋值，则根据状态更新该变量绑定的值，并发出继续正常执行的信号。
- 若指令为 $TEST$ b $THEN$ c1 $ELSE$ c2 FI 的形式，则按照 Imp 的原始语义更新状态，除此之外我们还要从被选择执行的分支中传播信号。
- 若指令为一系列 c1 ;; c2，我们首先执行 c1。如果它产生了 SBreak，我们就跳过 c2 的执行并将 SBreak 的信号传给其外围的上下文中; 结果状态与执行 c1 后获得的相同。否则，我们在执行完 c1 后的状态下执行 c2 并继续传递它产生的信号。
- 最后，对于形如 $WHILE$ b DO c END 的循环，其语义基本和此前相同。唯一的不同是，当 b 求值为 true 时执行 c 并检查它产生的信号。若信号为 SContinue，则按照原始语义继续执行。否则，我们终止此循环的执行，而其结果状态与当前迭代执行的结果相同。对于其它情况，由于 $BREAK$ 只终止最内层的循环，因此 $WHILE$ 发出 SContinue 的信号。

基于以上描述，请完成 **ceval** 关系的定义。

```
Inductive ceval : com → state → result → state → Prop :=
  | E_Skip : ∀ st,
      st =[ CSkip ]=> st / SContinue


  where "st '=[' c ']=>' st' '/' s" := (ceval c st s st').
```

现在证明你定义的 **ceval** 的如下性质：

```
Theorem break_ignore : ∀ c st st' s,
     st =[ BREAK;; c ]=> st' / s →
     st = st'.
Proof.
     Admitted.

Theorem while_continue : ∀ b c st st' s,
  st =[ WHILE b DO c END ]=> st' / s →
  s = SContinue.
Proof.
     Admitted.

Theorem while_stops_on_break : ∀ b c st st',
  beval st b = true →
  st =[ c ]=> st' / SBreak →
  st =[ WHILE b DO c END ]=> st' / SContinue.
Proof.
     Admitted.
     □
```

练习：3 星, advanced, optional (while_break_true) `Theorem while_break_true : ∀ b c st st',`

```
  st =[ WHILE b DO c END ]=> st' / SContinue →
  beval st' b = true →
  ∃ st'', st'' =[ c ]=> st' / SBreak.
Proof.
     Admitted.
     □
```

练习：4 星, advanced, optional (ceval_deterministic) `Theorem ceval_deterministic: ∀ (c:com) st st1 st2 s1 s2,`

```
     st =[ c ]=> st1 / s1 →
     st =[ c ]=> st2 / s2 →
```

st1 = st2 ∧ s1 = s2.

Proof.

Admitted.

□ End BreakImp.

练习：4 星, standard, optional (add_for_loop) 为该语言添加 C 风格的 for 循环指令，更新 **ceval** 的定义来定义 for 循环，按需添加 for 循环的情况使得本文件中的所有证明都被 Coq 所接受。

for 循环的参数应当包含 (a) 一个初始化执行的语句； (b) 一个在循环的每次迭代中都执行的测试，它决定了循环是否应当继续； (c) 一个在循环的每次迭代最后执行的语句，以及 (d) 一个创建循环体的语句（你不必关心为 for 构造一个具体的记法，不过如果你喜欢，可以随意去做。）

Chapter 9

Library LF.Rel

9.1 Rel: 关系的性质

本章为可选章节，主要讲述了在 Coq 定义二元关系的一些基本方法和相关定理的证明。关键定义会在实际用到的地方复述（'编程语言基础'中的*Smallstep*一章），因此熟悉这些概念的读者可以略读或跳过本章。不过，这些内容也是对 Coq 的证明功能很好的练习，因此在读完 IndProp 一章后，阅读本章也许会对你有所帮助。

```
Set Warnings "-notation-overridden,-parsing".
From LF Require Export IndProp.
```

9.2 关系

集合 X 上的二元'关系（*Relation*）'指所有由两个 X 中的元素参数化的命题，即，有关一对 X 中的元素的命题。

```
Definition relation (X: Type) := X → X → Prop.
```

令人困惑的是，"关系"原本是个更为通用的词语，然而 Coq 标准库中的"关系" 却单指这一种"某个集合中的元素之间二元关系"。为了与标准库保持一致，我们将会沿用这一定义。然而"关系"一词除了指这一意义以外，也可以指代任何数量的，可能是不同集合之间的更一般的关系。在讨论中使用"关系"一词时，应该在上下文中解释具体所指的含义。

一个关系的例子是 **nat** 上的 **le**，即"小于或等于"关系，我们通常写作 $n1 \leq n2$。

```
Print le.
Check le : nat → nat → Prop.
```

```
Check le : relation nat.
```

（为什么我们不直接写成 `Inductive le : relation nat...` 呢？这是因为我们想要将第一个 **nat** 放在 : 的左侧，这能让 Coq 为 $\leq$ 更为友好的的归纳法则。）

9.3 基本性质

学过本科离散数学课的人都知道，与关系相关的东西有很多，包括对关系的性质（如自反性、传递性等），关于某类关系的一般定理，如何从一种关系构造出另一种关系等等。例如：

偏函数

对于集合 X 上的关系 **R**，如果对于任何 x 最多只有一个 y 使得 **R** x y 成立 – 即，**R** x *y1* 和 **R** x *y2* 同时成立蕴含 *y1* $=$ *y2*，则称 **R** 为*'偏函数'*。

```
Definition partial_function {X: Type} (R: relation X) :=
  ∀ x y1 y2 : X, R x y1 → R x y2 → y1 = y2.
```

例如，之前定义的 **next_nat** 关系就是个偏函数。

```
Print next_nat.
Check next_nat : relation nat.

Theorem next_nat_partial_function :
    partial_function next_nat.
Proof.
  unfold partial_function.
  intros x y1 y2 H1 H2.
  inversion H1. inversion H2.
  reflexivity. Qed.
```

然而，数值上的 $\leq$ 关系并不是个偏函数。（利用反证法，假设 $\leq$ 是一个偏函数。然而根据其定义我们有 $0 \leq 0$ 和 $0 \leq 1$，这样会推出 $0 = 1$。这是不可能的，所以原假设不成立。）

```
Theorem le_not_a_partial_function :
  ¬ (partial_function le).
Proof.
```

```
  unfold not. unfold partial_function. intros Hc.
  assert (0 = 1) as Nonsense. {
    apply Hc with (x := 0).
    - apply le_n.
    - apply le_S. apply le_n. }
  discriminate Nonsense. Qed.
```

练习：2 星, standard, optional (total_relation_not_partial) 请证明 IndProp 一章练习题中定义的 *total_relation* 不是偏函数。

练习：2 星, standard, optional (empty_relation_partial) 请证明 IndProp 一章练习题中定义的 *empty_relation* 是偏函数。

自反关系

集合 X 上的'自反关系'是指 X 的每个元素都与其自身相关。

```
Definition reflexive {X: Type} (R: relation X) :=
  ∀ a : X, R a a.

Theorem le_reflexive :
  reflexive le.
Proof.
  unfold reflexive. intros n. apply le_n. Qed.
```

传递关系

如果 **R** a b 和 **R** b c 成立时 **R** a c 也成立，则称 **R** 为'传递关系'。

```
Definition transitive {X: Type} (R: relation X) :=
  ∀ a b c : X, (R a b) → (R b c) → (R a c).

Theorem le_trans :
  transitive le.
Proof.
  intros n m o Hnm Hmo.
  induction Hmo.
```

```
  - apply Hnm.
  - apply le_S. apply IHHmo. Qed.

Theorem lt_trans:
  transitive lt.
Proof.
  unfold lt. unfold transitive.
  intros n m o Hnm Hmo.
  apply le_S in Hnm.
  apply le_trans with (a := (S n)) (b := (S m)) (c := o).
  apply Hnm.
  apply Hmo. Qed.
```

练习：2 星, standard, optional (le_trans_hard_way) 我们也可以不用 le_trans，直接通过归纳法来证明 lt_trans，不过这会耗费更多精力。请完成以下定理的证明。

```
Theorem lt_trans' :
  transitive lt.
Proof.
  unfold lt. unfold transitive.
  intros n m o Hnm Hmo.
  induction Hmo as [| m' Hm'o].
    Admitted.
```

□

练习：2 星, standard, optional (lt_trans'') 再将此定理证明一遍，不过这次要对 o 使用归纳法。

```
Theorem lt_trans'' :
  transitive lt.
Proof.
  unfold lt. unfold transitive.
  intros n m o Hnm Hmo.
  induction o as [| o'].
    Admitted.
```

□

le 的传递性反过来也能用于证明一些之后会用到的事实，例如后面对反对称性的证明：

```
Theorem le_Sn_le : ∀ n m, S n ≤ m → n ≤ m.
Proof.
  intros n m H. apply le_trans with (S n).
  - apply le_S. apply le_n.
  - apply H.
Qed.
```

练习：1 星, standard, optional (le_S_n)

```
Theorem le_S_n : ∀ n m,
  (S n ≤ S m) → (n ≤ m).
Proof.
```

Admitted.

□

练习：2 星, standard, optional (le_Sn_n_inf) 请提写出以下定理的非形式化证明：

Theorem: For every n, $\neg$ (S $n \leq n$)

此定理的形式化证明在下面作为可选的练习，不过在做形式化证明之前请先尝试写出非形式化的证明。

证明：

练习：1 星, standard, optional (le_Sn_n)

```
Theorem le_Sn_n : ∀ n,
  ¬ (S n ≤ n).
Proof.
```

Admitted.

□

在后面的章节中，我们主要会用到自反性和传递性。不过，我们先看一些其它的概念，作为在 Coq 中对关系进行操作的练习...

对称关系与反对称关系

如果 **R** a b 蕴含 **R** b a，那么 **R** 就是'对称关系'。

```
Definition symmetric {X: Type} (R: relation X) :=
  ∀ a b : X, (R a b) → (R b a).
```

练习：2 星, standard, optional (le_not_symmetric) `Theorem le_not_symmetric :`

```
  ¬ (symmetric le).
Proof.
```

Admitted.

□

如果 **R** a b 和 **R** b a 成立时有 a = b，那么 **R** 就是'反对称关系'。

```
Definition antisymmetric {X: Type} (R: relation X) :=
  ∀ a b : X, (R a b) → (R b a) → a = b.
```

练习：2 星, standard, optional (le_antisymmetric) `Theorem le_antisymmetric :`

```
  antisymmetric le.
Proof.
```

Admitted.

□

练习：2 星, standard, optional (le_step) `Theorem le_step : ∀ n m p,`

```
  n < m →
  m ≤ S p →
  n ≤ p.
Proof.
```

Admitted.

□

等价关系

如果一个关系满足自反性、对称性和传递性，那么它就是'等价关系'。

```
Definition equivalence {X:Type} (R: relation X) :=
  (reflexive R) ∧ (symmetric R) ∧ (transitive R).
```

偏序关系与预序关系

如果某个关系满足自反性、'反'对称性和传递性，那么它就一个是'*偏序关系*'。在 Coq 标准库中，它被简短地称作"order"。

```
Definition order {X:Type} (R: relation X) :=
  (reflexive R) ∧ (antisymmetric R) ∧ (transitive R).
```

预序和偏序差不多，不过它无需满足反对称性。

```
Definition preorder {X:Type} (R: relation X) :=
  (reflexive R) ∧ (transitive R).

Theorem le_order :
  order le.
Proof.
  unfold order. split.
    - apply le_reflexive.
    - split.
      + apply le_antisymmetric.
      + apply le_trans. Qed.
```

9.4 自反传递闭包

关系 **R** 的'*自反传递闭包*'是最小的包含 **R** 的自反传递关系。在 Coq 标准库的 Relations 模块中，此概念定义如下：

```
Inductive clos_refl_trans {A: Type} (R: relation A) : relation A :=
    | rt_step x y (H : R x y) : clos_refl_trans R x y
    | rt_refl x : clos_refl_trans R x x
    | rt_trans x y z
          (Hxy : clos_refl_trans R x y)
          (Hyz : clos_refl_trans R y z) :
          clos_refl_trans R x z.
```

例如，**next_nat** 关系的自反传递闭包实际上就是 **le**。

```
Theorem next_nat_closure_is_le : ∀ n m,
  (n ≤ m) ↔ ((clos_refl_trans next_nat) n m).
```

```
Proof.
  intros n m. split.
  -
    intro H. induction H.
    + apply rt_refl.
    +
      apply rt_trans with m. apply IHle. apply rt_step.
      apply nn.
  -
    intro H. induction H.
    + inversion H. apply le_S. apply le_n.
    + apply le_n.
    +
      apply le_trans with y.
      apply IHclos_refl_trans1.
      apply IHclos_refl_trans2. Qed.
```

以上对自反传递闭包的定义十分自然：它直接将自反传递闭包定义为“包含 **R** 的，同时满足自反性和传递性的最小的关系”。然而此定义对于证明来说不是很方便，因为 rt_trans 的“非确定性” 有时会让归纳变得很棘手。下面是最常用的定义：

```
Inductive clos_refl_trans_1n {A : Type}
                             (R : relation A) (x : A)
                             : A → Prop :=
  | rt1n_refl : clos_refl_trans_1n R x x
  | rt1n_trans (y z : A)
      (Hxy : R x y) (Hrest : clos_refl_trans_1n R y z) :
      clos_refl_trans_1n R x z.
```

这一新的定义将 rt_step 和 rt_trans 合并成一条。在此规则的假设中 **R** 只用了一次，这会让归纳法则更简单。

在使用这一定义并继续之前，我们需要检查这两个定义确实定义了相同的关系...

首先，我们来证明 **clos_refl_trans_1n** 模仿了两个“缺失” 的 **clos_refl_trans** 构造子的行为。

```
Lemma rsc_R : ∀ (X:Type) (R:relation X) (x y : X),
```

$R\ x\ y \to$ **clos_refl_trans_1n** $R\ x\ y$.

```
Proof.
  intros X R x y H.
  apply rtln_trans with y. apply H. apply rtln_refl. Qed.
```

练习：2 星, standard, optional (rsc_trans) `Lemma rsc_trans :`

```
  ∀ (X:Type) (R: relation X) (x y z : X),
```

clos_refl_trans_1n $R\ x\ y \to$

clos_refl_trans_1n $R\ y\ z \to$

clos_refl_trans_1n $R\ x\ z$.

```
Proof.
```

Admitted.

□

接着再用这些事实来证明这两个定义的自反性、传递性封闭确实定义了同样的关系。

练习：3 星, standard, optional (rtc_rsc_coincide) `Theorem rtc_rsc_coincide :`

```
        ∀ (X:Type) (R: relation X) (x y : X),
```

clos_refl_trans $R\ x\ y \leftrightarrow$ **clos_refl_trans_1n** $R\ x\ y$.

```
Proof.
```

Admitted.

□

Chapter 10

Library LF.IndPrinciples

10.1 IndPrinciples: 归纳法则

在理解了柯里-霍华德同构及其 Coq 实现之后，我们就可以深入学习归纳法则了。

```
Set Warnings "-notation-overridden,-parsing".
From LF Require Export ProofObjects.
```

10.2 基础

每当我们使用 `Inductive` 来声明数据类型时，Coq 就会自动为该类型生成 *'归纳法则'*。这个归纳法则也是定理：如果 t 是归纳定义的，那么对应的归纳法则被称作 t_ind。这是自然数的归纳法则：

```
Check nat_ind.
```

In English: Suppose P is a property of natural numbers (that is, P n is a `Prop` for every n). To show that P n holds of all n, it suffices to show:

- P holds of 0
- for any n, if P holds of n, then P holds of `S` n.

`induction` 利用归纳法则，执行 `apply` t_ind 等策略。为了清楚地理解这一点，让我们直接使用 `apply nat_ind` 而非 `induction` 策略来证明。例如，`Induction` 一章中 `mult_0_r'` 定理的另一种证明如下所示。

```
Theorem mult_0_r' : ∀ n:nat,
  n × 0 = 0.
Proof.
  apply nat_ind.
  - reflexivity.
  - simpl. intros n' IHn'. rewrite → IHn'.
    reflexivity. Qed.
```

这个证明基本上等同于之前的，但有几点区别值得注意。

首先，在证明的归纳步骤（"S" 情形）中，我们不得不手动管理变量名（即 `intros`），而 `induction` 会自动完成这些。

其次，在应用 nat_ind 之前我们没有在上下文中引入 n —— nat_ind 的结论是一个带有量词的公式，`apply` 需要这个结论精确地匹配当前证明目标状态的形状，包括其中的量词。相反，`induction` 策略对于上下文中的变量或目标中由量词引入的变量都适用。

第三，我们必须手动为 `apply` 提供归纳法则，而 `induction` 可以自己解决它。

相比于直接使用 nat_ind 这样的归纳法则，在实践中使用 `induction` 更加方便。但重要的是认识到除了这一点变量名的管理工作，我们在做的其实就是应用 nat_ind。

练习：2 星, standard, optional (plus_one_r') 请不使用 `induction` 策略来完成这个证明。

```
Theorem plus_one_r' : ∀ n:nat,
  n + 1 = S n.
Proof.
  Admitted.
```

□

Coq 为每一个 `Inductive` 定义的数据类型生成了归纳法则，包括那些非递归的。尽管我们不需要归纳，便可证明非递归数据类型的性质，但归纳原理仍可用来证明其性质；给定类型，及关于该类型所有值的性质，归纳原理提供了证明该性质的方法。

这些生成的原则都具有类似的模式。如果我们定义了带有构造子 `c1` ... cn 的类型 t，那么 Coq 会生成形如下文的定理：

t_ind : forall P : t -> Prop, ... case for c1 ... -> ... case for c2 ... -> ... ... case for cn ... -> forall n : t, P n

每个情形具体的形状取决于对应构造子的参数。

在尝试总结出一般规律前，让我们先来看看更多的例子。首先是一个无参数构造子的例子：

```
Inductive time : Type :=
  | day
  | night.

Check time_ind.
```

练习：1 星, standard, optional (rgb) 对如下数据类型，请写出 Coq 将会生成的归纳法则。先在纸上或注释内写下你的答案，然后同 Coq 打印出的结果比较。

```
Inductive rgb : Type :=
  | red
  | green
  | blue.
Check rgb_ind.
```

□

下文例子中，有一个构造子调取多个参数。

```
Inductive natlist : Type :=
  | nnil
  | ncons (n : nat) (l : natlist).

Check natlist_ind.
```

练习：1 星, standard, optional (natlist1) 假设我们写下的定义和上面的有一些区别：

```
Inductive natlist1 : Type :=
  | nnil1
  | nsnoc1 (l : natlist1) (n : nat).
```

现在归纳法则会是什么呢？

□

通常，为归纳类型 t 生成的归纳法则形式如下：

- 每个构造子 c 都会生成归纳法则中的一种情况
- 若 c 不接受参数，该情况为

  “P 对 c 成立”

- 若 c 接受参数 *x1*:*a1* ... *xn*:*an*，该情况为：

  “对于所有的 x1:a1 ... xn:an，若 P 对每个类型为 t 的函数都成立，则 P 对于 c *x1* ... *xn* 成立”

练习：1 星, standard, optional (booltree_ind) 对如下数据类型，请写出 Coq 将会生成的归纳法则。（与之前一样，先在纸上或注释内写下你的答案，然后同 Coq 打印出的结果比较。）

```
Inductive booltree : Type :=
 | bt_empty
 | bt_leaf (b : bool)
 | bt_branch (b : bool) (t1 t2 : booltree).
```

□

练习：1 星, standard, optional (ex_set) 这是对一个归纳定义的集合的归纳法则。

ExSet_ind : forall P : ExSet -> Prop, (forall b : bool, P (con1 b)) -> (forall (n : nat) (e : ExSet), P e -> P (con2 n e)) -> forall e : ExSet, P e

请写出使用 `Inductive` 来定义的 **ExSet**：

```
Inductive ExSet : Type :=


.
```

□

10.3 多态

多态数据结构会是怎样的呢？

归纳定义的多态列表

Inductive list (X:Type) : Type := | nil : list X | cons : X -> list X -> list X.

同 **natlist** 是十分相似的。主要的区别是，这里全部的定义是由集合 X 所’参数化’的：也即，我们定义了’一族’归纳类型 **list** X，对于每个 X 有其对应的类型在此族中。（请注意，当 **list** 出现在声明体中时，它总是被应用参数 X。）

归纳法则同样被 X 所参数化：

list_ind : forall (X : Type) (P : list X -> Prop), P □ -> (forall (x : X) (l : list X), P l -> P (x :: l)) -> forall l : list X, P l

请注意归纳法则的'所有部分'都被 X 所参数化。也即，*list_ind* 可认为是一个多态函数，当被应用类型 X 时，返回特化在类型 **list** X 上的归纳法则。

练习：1 星, standard, optional (tree) 对如下数据类型，请写出 Coq 将会生成的归纳法则，并与 Coq 打印出的结果比较。

```
Inductive tree (X:Type) : Type :=
  | leaf (x : X)
  | node (t1 t2 : tree X).
Check tree_ind.
```

□

练习：1 星, standard, optional (mytype) 请找到对应于以下归纳法则的归纳定义：

mytype_ind : forall (X : Type) (P : mytype X -> Prop), (forall x : X, P (constr1 X x)) -> (forall n : nat, P (constr2 X n)) -> (forall m : mytype X, P m -> forall n : nat, P (constr3 X m n)) -> forall m : mytype X, P m □

练习：1 星, standard, optional (foo) 请找到对应于以下归纳法则的归纳定义：

foo_ind : forall (X Y : Type) (P : foo X Y -> Prop), (forall x : X, P (bar X Y x)) -> (forall y : Y, P (baz X Y y)) -> (forall f1 : nat -> foo X Y, (forall n : nat, P (f1 n)) -> P (quux X Y f1)) -> forall f2 : foo X Y, P f2 □

练习：1 星, standard, optional (foo') 请考虑以下归纳定义：

```
Inductive foo' (X:Type) : Type :=
  | C1 (l : list X) (f : foo' X)
  | C2.
```

Coq 会为 **foo'** 生成什么归纳法则？请填写下面的空白，并使用 Coq 检查你的答案。

foo'_ind : forall (X : Type) (P : foo' X -> Prop), (forall (l : list X) (f : foo' X), _____ -> _____) -> _____ -> forall f : foo' X, _____

□

10.4 归纳假设

"归纳假设"是在什么语境下出现的呢？

对于数的归纳法则：

forall P : nat -> Prop, P 0 -> (forall n : nat, P n -> P (S n)) -> forall n : nat, P n

是一个对于所有命题 P （或更严格地说，对由数字 n 索引的所有命题 P）都成立的泛化陈述。每次使用这个原理，我们将 P 特化为一个类型为 **nat** → Prop 的表达式。

通过命名这个表达式，我们可以让归纳证明更加明确。比如，除了陈述定理 mult_0_r 为"$\forall\ n,\ n \times 0 = 0$"，我们还可以写成"$\forall\ n$, P_m0r n"，其中 O_m0r 定义为.....

```
Definition P_m0r (n:nat) : Prop :=
  n × 0 = 0.
```

.....或等价地：

```
Definition P_m0r' : nat→Prop :=
  fun n ⇒ n × 0 = 0.
```

现在看看 P_m0r 在证明中出现在哪里。

```
Theorem mult_0_r'' : ∀ n:nat,
  P_m0r n.
Proof.
  apply nat_ind.
  - reflexivity.
  -


    intros n IHn.
    unfold P_m0r in IHn. unfold P_m0r. simpl. apply IHn. Qed.
```

通常我们并不会在证明中额外地为命题命名，但有意地进行一两个练习可以帮助我们清晰地看到哪个是归纳假设。如果对 n 归纳来证明 $\forall\ n$, P_m0r n（使用 induction 或 apply nat_ind），可以看到第一个子目标要求我们证明 P_m0r 0 （"P 对零成立"），而第二个子目标要求我们证明 $\forall\ n'$, P_m0r $n' \to$ P_m0r (S n') （也即，"P 对 S n' 成立仅当其对 n' 成立"，或者说，"P 被 S 保持"）。'归纳假设'是后一个蕴含式中的前提部分，即假设 P 对 n' 成立，这是我们在证明 P 对 S n' 的过程中允许使用的。

10.5 深入 induction 策略

induction 策略事实上为我们做了更多低层次的工作。

请回忆一下自然数归纳法则的非形式化陈述：

- 如果 $P\ n$ 是某个涉及到数字 n 的命题，我们想要证明 P 对于'所有'数字 n 都成立，我们以如下方式推理：
  - 证明 $P\ 0$ 成立
  - 证明如果 $P\ n'$ 成立，那么 $P\ (\mathsf{S}\ n')$ 成立
  - 得出结论 $P\ n$ 对所有 n 成立。

因此，当以 intros n 和 induction n 开始一个证明时，我们首先在告诉 Coq 考虑一个'特殊的' n（通过引入到上下文中），然后告诉它证明一些关于 '全体'数字的性质（通过使用归纳）。

在这种情况下，Coq 事实上在内部"再次一般化（re-generalize）"了我们进行归纳的变量。比如说，起初证明 plus 的结合性时……

```
Theorem plus_assoc' : ∀ n m p : nat,
  n + (m + p) = (n + m) + p.
Proof.
  intros n m p.
  induction n as [| n'].
  - reflexivity.
  -
    simpl. rewrite → IHn'. reflexivity. Qed.
```

对目标中含有量词的变量应用 induction 同样可以工作。

```
Theorem plus_comm' : ∀ n m : nat,
  n + m = m + n.
Proof.
  induction n as [| n'].
  - intros m. rewrite ← plus_n_O. reflexivity.
  - intros m. simpl. rewrite → IHn'.
    rewrite ← plus_n_Sm. reflexivity. Qed.
```

请注意，使用 induction 后 m 在目标中仍然是绑定的，也即，归纳证明的陈述是以 $\forall\ m$ 开始的。

如果我们对目标中其他量词'后'的量化变量使用 induction，那么它会自动引入全部被量词绑定的变量到上下文中。

```
Theorem plus_comm'' : ∀ n m : nat,
  n + m = m + n.
Proof.
  induction m as [| m']. - simpl. rewrite ← plus_n_O. reflexivity.
  - simpl. rewrite ← IHm'.
    rewrite ← plus_n_Sm. reflexivity. Qed.
```

练习：1 星，standard，optional (plus_explicit_prop) 以类似 mult_0_r'' 的方式来重写 plus_assoc'，plus_comm' 和它们的证明—— 对于每个定理，给出一个明确的命题的 Definition，陈述定理并用归纳法证明这个定义的命题。

10.6 Prop 中的归纳法则

之前，我们仔细学习了 Coq 为归纳定义的'集合'生成的归纳法则。像 even 这样的归纳定义'命题'的归纳法则会复杂一点点。就全部归纳法则来说，我们想要通过使用 even 的归纳法则并归纳地考虑 even 中所有可能的形式来证明一些东西。然而，直观地讲，我们想要证明的东西并不是关于'证据'的陈述，而是关于'自然数'的陈述：因此，我们想要让归纳法则允许通过对证据进行归纳来证明关于数字的性质。例如：

根据我们前面所讲，你可能会期待这样归纳定义的 even.....

Inductive even : nat -> Prop := | ev_0 : even 0 | ev_SS : forall n : nat, even n -> even (S (S n)).

.....并给我们下面这样的归纳法则.....

ev_ind_max : forall P : (forall n : nat, even n -> Prop), P O ev_0 -> (forall (m : nat) (E : even m), P m E -> P (S (S m)) (ev_SS m E)) -> forall (n : nat) (E : even n), P n E

.....因为：

- 由于 even 被自然数 n 所索引（任何 even 对象 E 都是某个自然数 n 是偶数的证据），且命题 P 同时被 n 和 E 所参数化——因此，被用于证明断言的归纳法则同时涉及到一个偶数和这个数是偶数的证据。

- 由于有两种方法来给出偶数性质的证据（因为 even 有两个构造子），我们应用归纳法则生成了两个子目标：
  - 须证明 P 对 0 和 ev_0 成立。
  - 须证明，当 m 是一个自然数且 E 是其偶数性质的证据，如果 P 对 m 和 E 成立，那么它也对 S (S m) 和 ev_SS m E 成立。
- 如果这些子目标可以被证明，那么归纳法则告诉我们 P 对所有的偶数 n 和它们的偶数性质 E 成立。

这比我们通常需要的或想要的更灵活一点：它为我们提供了一种方式证明逻辑断言，其断言涉及到一些关于偶数的证据的性质，然而我们真正想要的是证明某些'自然数'是偶数这样的性质——我们感兴趣的是关于自然数的断言，而非关于证据。如果对于命题 P 的归纳法则仅仅被 n 所参数化，且其结论是 P 对所有偶数 n 成立，那会方便许多：

forall P : nat -> Prop, ... -> forall n : nat, even n -> P n

出于这样的原因，Coq 实际上为 even 生成了简化过的归纳法则：

```
Print ev.

Check ev_ind.
```

请特别注意，Coq 丢弃了命题 P 参数中的证据项 E。

若用自然语言来表述 ev_ind，则是说：假设 P 是关于自然数的一个性质（也即，P n 是一个在全体 n 上的 Prop）。为了证明当 n 是偶数时 P n 成立，需要证明：

- P 对 0 成立，
- 对任意 n，如果 n 是偶数且 P 对 n 成立，那么 P 对 S (S n) 成立。

正如期待的那样，我们可以不使用 induction 而直接应用 ev_ind。比如，我们可以使用它来证明 **ev'**（就是在 IndProp 一章的练习中那个有点笨拙的偶数性质的定义）等价于更简洁的归纳定义 **ev**：

```
Inductive ev' : nat → Prop :=
| ev'_0 : ev' 0
| ev'_2 : ev' 2
| ev'_sum n m (Hn : ev' n) (Hm : ev' m) : ev' (n + m).

Theorem ev_ev' : ∀ n, ev n → ev' n.
```

```
Proof.
  apply ev_ind.
  -
    apply ev'_0.
  -
    intros m Hm IH.
    apply (ev'_sum 2 m).
    + apply ev'_2.
    + apply IH.
Qed.
```

Inductive 定义的具体形式会影响到 Coq 生成的归纳法则。

```
Inductive le1 : nat → nat → Prop :=
    | le1_n : ∀ n, le1 n n
    | le1_S : ∀ n m, (le1 n m) → (le1 n (S m)).

Notation "m <=1 n" := (le1 m n) (at level 70).
```

此定义其实可以稍微简化一点，我们观察到左侧的参数 n 在定义中始终是相同的，于是可以把它变成整体定义中的一个“一般参数”，而非每个构造子的参数。

```
Inductive le2 (n:nat) : nat → Prop :=
  | le2_n : le2 n n
  | le2_S m (H : le2 n m) : le2 n (S m).

Notation "m <=2 n" := (le2 m n) (at level 70).
```

尽管第二个看起来不那么对称了，但它却更好一点。为什么呢？因为它会生成更简单的归纳法则。

```
Check le1_ind.

Check le2_ind.
```

10.7 形式化 vs. 非形式化的归纳证明

问：命题 P 的形式化证明和同一个命题 P 的非形式化证明之间是什么关系？

答：后者应当'教给'读者如何产生前者。

问：需要多少的细节？

不幸的是，并没有一个正确的答案；当然了，其实有一系列的选择。

一种选择是，我们可以为读者给出全部的形式化证明（也即，“非形式化的”证明只是把形式化的证明用文字表述出来）。这可能让读者有能力自己完成形式化的证明，但也许并没有'教给'读者什么东西。

而另一种选择则是，我们可以说“某个定理为真，如果你觉得它有些困难那么可以自己尝试把它搞明白。” 这也不是一种很好的教学策略，因为书写证明常常需要一两个对于要证明的东西的重要洞察，而多数读者往往在自己发现这些这些洞察前已经放弃了。

一种中庸之道是——我们提供含有重要洞察的证明（免去读者像我们一开始一样辛苦地寻找证明），加上模式化部分的高层次建议（比如，归纳假设是什么，以及归纳证明中每个情形的证明责任），这样帮助读者节省自己重新构造这些东西的时间，但不会有过多的细节以至于主要的概念和想法受到干扰。

我们在本章中已经仔细查看了形式化的归纳证明的“底层原理”，现在是时候来看看非形式化的归纳证明了。

在现实世界的数学交流中，证明的书写既有冗长的，也有非常简洁的。尽管理想状态是二者中间的某种形式，但从有一点冗长的证明开始学习是有好处的。同时，在学习的过程中，有一个明确的标准来进行比较也是有益的。为此，我们提供了两份模板：一份用于归纳证明'数据'（也即，`Type` 中我们进行归纳的东西），另一份用于归纳证明'证据'（也即，`Prop` 中归纳定义的东西）。

10.7.1 对归纳定义的集合进行归纳

'模板'：

- '定理'： <有形如“For all n:S, $P(n)$”的全称量化命题，其中 S 是某个归纳定义的集合。>

  '证明'：对 n 进行归纳。

  <S 中的每个构造子 c 的情形.....>

  - 假设 $n = \mathsf{c}\ a1\ \ldots\ ak$，其中 <..... 这里我们为每个具有类型 S 的 a 陈述其归纳假设（IH）> 。我们需要证明 <..... 我们在这里重新陈述 $P(\mathsf{c}\ a1\ \ldots\ ak)$>。

    <继续并证明 $P(n)$ 来完成这个情形.....>

  - <其他情形以此类推.....> □

'举例':

- '定理': 对所有集合 X，列表 l : **list** X，以及数字 n，如果 length $l = n$ 那么 $index$ (S n) l = None。

  '证明': 对 l 进行归纳。

  - 假设 l = []。我们需要证明，对于任意数字 n，如果 length [] = n，那么 $index$ (S n) [] = None。

    可从 $index$ 的定义中直接得出。

  - 假设 $l = x :: l'$ 对某个 x 和 l'，其中 length $l' = n'$ 对任意数字 n' 蕴含了 $index$ (S n') l' = None。我们需要证明，对任意数字 n，如果 length $(x::l') = n$ 那么 $index$ (S n) $(x::l')$ = None。

    设 n 为数字且 length $l = n$。因为

    length l = length (x::l') = S (length l'),

    需要证明

    index (S (length l')) l' = None.

    若选取 n' 为 length l' 这可从归纳假设中直接得出。 □

10.7.2 对归纳定义的命题进行归纳

由于归纳定义的证明对象经常被称作"导出树（derivation trees）"，这种形式的证明也被叫做'在导出式上归纳'。 '模板':

- '定理': <有形如"$Q \to P$"的命题，其中 Q 是某个归纳定义的命题（更一般地，"对任意 x y z，Q x y $z \to P$ x y z"）>

  '证明': 对 Q 的导出式进行归纳。<或者，更一般地，"假设给定 x，y 和 z。通过对 Q x y z 的导出式进行归纳，我们证明 Q x y z 蕴含 P x y z".....>

  <Q 中的每个构造子 c 的情形.....>

  - 假设被用于证明 Q 的最终规则是 c。那么<.....我们在这里陈述所有 a 的类型，从构造子的定义中得到的任何等式，以及每个具有类型 Q 的 a 的归纳假设>。我们需要证明<.....我们在这里重新陈述 P>。

    <继续并证明 P 来完成这个情形.....>

  - <其他情形以此类推.....> □

'举例'

- '定理': $\leq$ 关系是传递的，也即，对任意数字 n，m 和 o，如果 $n \leq m$ 且 $m \leq o$ 那么 $n \leq o$。

  '证明': 对 $m \leq o$ 的导出式进行归纳。

  - 假设被用于证明 $m \leq o$ 的最终规则是 le_n。那么 $m = o$ 且我们需要证明 $n \leq m$，其可从假设中直接得出。
  - 假设被用于证明 $m \leq o$ 的最终规则是 le_S。那么 $o =$ S o' 对某个 o' 且 $m \leq o'$。我们需要证明 $n \leq$ S o'。由归纳假设得出，$n \leq o'$。

    因此，根据 le_S，$n \leq$ S o'。 □

10.8 Explicit Proof Objects for Induction (Optional)

Although tactic-based proofs are normally much easier to work with, the ability to write a proof term directly is sometimes very handy, particularly when we want Coq to do something slightly non-standard.

Recall again the induction principle on naturals that Coq generates for us automatically from the Inductive declation for **nat**.

```
Check nat_ind.
```

There's nothing magic about this induction lemma: it's just another Coq lemma that requires a proof. Coq generates the proof automatically too...

```
Print nat_ind.
```

We can read this as follows: Suppose we have evidence f that P holds on 0, and evidence *f0* that $\forall$ n:**nat**, $P\ n \rightarrow P$ (S n). Then we can prove that P holds of an arbitrary nat n via a recursive function F (here defined using the expression form `Fix` rather than by a top-level `Fixpoint` declaration). F pattern matches on n:

- If it finds 0, F uses f to show that $P\ n$ holds.
- If it finds S *n0*, F applies itself recursively on *n0* to obtain evidence that P *n0* holds; then it applies *f0* on that evidence to show that P (S n) holds.

F is just an ordinary recursive function that happens to operate on evidence in `Prop` rather than on terms in `Set`.

We can adapt this approach to proving nat_ind to help prove *non-standard* induction principles too. As a motivating example, suppose that we want to prove the following lemma, directly relating the **ev** predicate we defined in IndProp to the evenb function defined in Basics.

```
Lemma evenb_ev : ∀ n: nat, evenb n = true → ev n.
Proof.
  induction n; intros.
  - apply ev_0.
  - destruct n.
    + simpl in H. inversion H.
    + simpl in H.
      apply ev_SS.
Abort.
```

Attempts to prove this by standard induction on n fail in the case for S (S n), because the induction hypothesis only tells us something about S n, which is useless. There are various ways to hack around this problem; for example, we *can* use ordinary induction on n to prove this (try it!):

Lemma *evenb_ev'* : ∀ n : **nat**, (evenb n = true → **ev** n) ∧ (evenb (S n) = true → **ev** (S n)).

But we can make a much better proof by defining and proving a non-standard induction principle that goes "by twos":

```
Definition nat_ind2 :
    ∀ (P : nat → Prop),
    P 0 →
    P 1 →
    (∀ n : nat, P n → P (S(S n))) →
    ∀ n : nat , P n :=
       fun P ⇒ fun P0 ⇒ fun P1 ⇒ fun PSS ⇒
          fix f (n:nat) := match n with
                                0 ⇒ P0
```

```
  | 1 ⇒ P1
  | S (S n') ⇒ PSS n' (f n')
end.
```

Once you get the hang of it, it is entirely straightforward to give an explicit proof term for induction principles like this. Proving this as a lemma using tactics is much less intuitive.

The `induction ... using` tactic variant gives a convenient way to utilize a non-standard induction principle like this.

```
Lemma evenb_ev : ∀ n, evenb n = true → ev n.
Proof.
 intros.
 induction n as [ | | n'] using nat_ind2.
 - apply ev_0.
 - simpl in H.
   inversion H.
 - simpl in H.
   apply ev_SS.
   apply IHn'.
   apply H.
Qed.
```

10.9 The Coq Trusted Computing Base

One issue that arises with any automated proof assistant is "why trust it?": what if there is a bug in the implementation that renders all its reasoning suspect?

While it is impossible to allay such concerns completely, the fact that Coq is based on the Curry-Howard correspondence gives it a strong foundation. Because propositions are just types and proofs are just terms, checking that an alleged proof of a proposition is valid just amounts to *type-checking* the term. Type checkers are relatively small and straightforward programs, so the "trusted computing base" for Coq – the part of the code that we have to believe is operating correctly – is small too.

What must a typechecker do? Its primary job is to make sure that in each function application the expected and actual argument types match, that the arms of a `match` ex-

pression are constructor patterns belonging to the inductive type being matched over and all arms of the `match` return the same type, and so on.

There are a few additional wrinkles:

- Since Coq types can themselves be expressions, the checker must normalize these (by using the computation rules) before comparing them.

- The checker must make sure that `match` expressions are *exhaustive*. That is, there must be an arm for every possible constructor. To see why, consider the following alleged proof object:

  Definition or_bogus : forall P Q, P \/ Q -> P := fun (P Q : Prop) (A : P \/ Q) => match A with | or_introl H => H end.

  All the types here match correctly, but the `match` only considers one of the possible constructors for **or**. Coq's exhaustiveness check will reject this definition.

- The checker must make sure that each `fix` expression terminates. It does this using a syntactic check to make sure that each recursive call is on a subexpression of the original argument. To see why this is essential, consider this alleged proof:

  Definition nat_false : forall (n:nat), False := fix f (n:nat) : False := f n.

  Again, this is perfectly well-typed, but (fortunately) Coq will reject it.

Note that the soundness of Coq depends only on the correctness of this typechecking engine, not on the tactic machinery. If there is a bug in a tactic implementation (and this certainly does happen!), that tactic might construct an invalid proof term. But when you type `Qed`, Coq checks the term for validity from scratch. Only lemmas whose proofs pass the type-checker can be used in further proof developments.

Chapter 11

Library LF.ProofObjects

11.1 ProofObjects: 柯里-霍华德对应

```
Set Warnings "-notation-overridden,-parsing".
From LF Require Export IndProp.
```

"'算法是证明的计算性内容。'" –Robert Harper

前文已讨论过 Coq 既可以用 **nat**、**list** 等归纳类型及其函数'编程'，又可以用归纳命题（如 **ev**）、蕴含式、全称量词等工具'证明'程序的性质。我们一直以来区别对待此两种用法，在很多情况下确实可以这样。但也有迹象表明在 Coq 中编程与证明紧密相关。例如，关键字 `Inductive` 同时用于声明数据类型和命题，以及 → 同时用于描述函数类型和逻辑蕴含式。这可并不是语法上的巧合！事实上，在 Coq 里面程序和证明几乎就是同一件事情。这一章我们会学习背后的原理。

我们已经知道这个基础的思想：在Coq里面，可证明性表现为拥有具体的'证据'。为基本命题构造证明，实则以树状结构表示其证据。

对于形如 A → B 的蕴含式，其证明为证据'转化装置（*transformer*）'，可将任何证明 A 的依据转化为 B 的证据。所以从根本上来讲，证明仅仅就是操纵证据的程序。

试问：如果是证据是数据，那么命题本身是什么？

答曰：类型也！

回顾一下 **ev** 这个性质的形式化定义。

```
Print ev.
```

我们可以换种方式来解读":"：用"是.....的证明"而非"具有.....类型"。例如将 **ev** 定义中第二行的 ev_0 : **ev** 0 读作"ev_0 是 **ev** 0 的证明"而非"ev_0 的类型为 **ev** 0"。

此处：既在类型层面表达“具有……类型”，又在命题层面表示“是……的证明”。这种双关称为'柯里-霍华德同构（*Curry-Howard correspondence*）'。它指出了逻辑与计算之间的深层联系：

命题 ~ 类型证明 ~ 数据值

Wadler 2015 (in Bib.v) 里有简单的历史和最新的详细介绍可供参考。

该同构启发很多看问题的新方法。首先，对 ev_SS 构造子的理解变得更加自然：

```
Check ev_SS
  : ∀ n,
    ev n →
    ev (S (S n)).
```

可以将其读作“ev_SS 构造子接受两个参数——数字 n 以及命题 **ev** n 的证明——并产生 **ev** (S (S n)) 的证明。”

现在让我们回顾一下之前有关 **ev** 的一个证明。

```
Theorem ev_4 : ev 4.
Proof.
  apply ev_SS. apply ev_SS. apply ev_0. Qed.
```

就像是处理普通的数据值和函数一样，我们可以使用 Print 指令来查看这个证明脚本所产生的'证据对象 *(proof object)*'

```
Print ev_4.
```

实际上，我们也可以不借助脚本'直接'写出表达式作为证明。

```
Check (ev_SS 2 (ev_SS 0 ev_0))
  : ev 4.
```

表达式 ev_SS 2 (ev_SS 0 ev_0) 可视为向构造子 ev_SS 传入参数 2 和 0 等参数，以及对应的 **ev** 2 与 **ev** 0 之依据所构造的证明。或言之，视 ev_SS 为“构造证明”之原语，需要给定一个数字，并进一步提供该数为偶数之依据以构造证明。其类型表明了它的功能：

forall n, ev n -> ev (S (S n)),

类似地，多态类型 ∀ X, **list** X 表明可以将 nil 视为从某类型到由该类型元素组成的空列表的函数。

我们在 Logic 这一章中已经了解到，我们可以使用函数应用的语法来实例化引理中的全称量化变量，也可以使用该语法提供引理所要求的假设。例如：

```
Theorem ev_4': ev 4.
```

```
Proof.
  apply (ev_SS 2 (ev_SS 0 ev_0)).
Qed.
```

11.2 证明脚本

我们一直在讨论的'证明对象 *(Proof Objects)*'是Coq如何运作的核心。当Coq执行一个证明脚本的时候，在内部，Coq逐渐构造出一个证明对象—— 一个类型是想要证明的命题的项。在 `Proof` 和 `Qed` 之间的策略告诉 Coq如何构造该项。为了了解这个过程是如何进行的，在下面的策略证明里，我们在多个地方使用 `Show Proof` 指令来显示当前证明树的状态。

```
Theorem ev_4'' : ev 4.
Proof.
  Show Proof.
  apply ev_SS.
  Show Proof.
  apply ev_SS.
  Show Proof.
  apply ev_0.
  Show Proof.
Qed.
```

在任意的给定时刻，Coq已经构造了一个包含一个"洞(hole)"（即 `?Goal` ）的项，并且Coq知道该洞需要什么类型的证据来填补。

每一个洞对应一个子目标。当没有子目标时，代表证明已经完成。此时，我们构造的证明将会被存储在全局环境中，其名字就是在 `Theorem` 中给定的名字

策略证明非常有用且方便，但是它们并不是必要的：原则上，我们总是能够手动构造想要的证据，如下所示。此处我们可以通过 `Definition` （而非 `Theorem`）来直接给这个证据一个全局名称。

```
Definition ev_4''' : ev 4 :=
  ev_SS 2 (ev_SS 0 ev_0).
```

所有这些构造证明的不同方式，对应的存储在全局环境中的证明是完全一样的。

```
Print ev_4.
```

```
Print ev_4'.
Print ev_4''.
Print ev_4'''.
```

练习：2 星, standard (eight_is_even) 写出对应 **ev** 8 的策略证明和证明对象。

```
Theorem ev_8 : ev 8.
Proof.
  Admitted.

Definition ev_8' : ev 8
  . Admitted.
```

□

11.3 量词，蕴含式，函数

在Coq的计算世界里（即所有的数据结构和程序存在的地方），有两种值的类型中拥有箭头：一种是*'构造子(Constructor)'*，它通过归纳地定义数据类型引入，另一种是*'函数(Function)'*。

类似地，在Coq的逻辑世界里（即我们运用证明的地方），有两种方式来给与蕴含式需要的证据：构造子，通过归纳地定义命题引入，和...函数!

例如，考虑下列陈述：

```
Theorem ev_plus4 : ∀ n, ev n → ev (4 + n).
Proof.
  intros n H. simpl.
  apply ev_SS.
  apply ev_SS.
  apply H.
Qed.
```

对应 ev_plus4 的证明对象是什么？

我们在寻找一个*'类型(Type)'*是 ∀ n, **ev** n → **ev** $(4 + n)$ 的表达式——也就是说，一个接受两个参数（一个数字和一个证据）并返回一个证据的 *'函数(Function)'*!

它的证据对象：

```
Definition ev_plus4' : ∀ n, ev n → ev (4 + n) :=
```

```
fun (n : nat) ⇒ fun (H : ev n) ⇒
  ev_SS (S (S n)) (ev_SS n H).
```

回顾 `fun n ⇒ blah` 意味着"一个函数，给定 n，产生 *blah*"，并且Coq认为 $4 + n$ 和 S (S (S (S n))) 是同义词，所以另一种写出这个定义的方式是：

```
Definition ev_plus4'' (n : nat) (H : ev n)
                      : ev (4 + n) :=
  ev_SS (S (S n)) (ev_SS n H).

Check ev_plus4''
  : ∀ n : nat,
    ev n →
    ev (4 + n).
```

当我们将 ev_plus4 证明的命题视为一个函数类型时，我们可以发现一个有趣的现象：第二个参数的类型，**ev** n，依赖于第一个参数 n 的'值'。

虽然这样的'依赖类型 *(Dependent type)*'在传统的编程语言中并不存在，但是它们对于编程来说有时候非常有用。最近它们在函数式编程社区里的活跃很好地表明了这一点。

注意到蕴含式（→）和量化（∀）都表示证据上的函数。事实上，他们是同一个东西：当我们使用∀时没有依赖，就可以简写为当→。即，我们没有必要给与箭头左边的类型一个名字：

```
forall (x:nat), nat = forall (_:nat), nat = nat -> nat
```

例如，考虑下列命题：

```
Definition ev_plus2 : Prop :=
  ∀ n, ∀ (E : ev n), ev (n + 2).
```

这个命题的一个证明项会是一个拥有两个参数的函数：一个数字n 和一个表明n是偶数的证据E。但是对于这个证据来说，名字E并没有在ev_plus2剩余的陈述里面被使用，所以还专门为它取一个名字并没有意义。因此我们可以使用虚拟标志符_来替换真实的名字：

```
Definition ev_plus2' : Prop :=
  ∀ n, ∀ (_ : ev n), ev (n + 2).
```

或者，等同地，我们可以使用更加熟悉的记号：

```
Definition ev_plus2'' : Prop :=
```

$\forall\ n,\ \mathbf{ev}\ n \to \mathbf{ev}\ (n + 2)$.

总的来说，“$P \to Q$”只是“$\forall\ (\_{:}P),\ Q$”的语法糖。

11.4 使用策略编程

如果我们可以通过显式地给出项，而不是执行策略脚本，来构造证明，你可能会好奇我们是否可以通过'策略'，而不是显式地给出项，来构造'程序'。自然地，答案是可以！

```
Definition add1 : nat → nat.
intro n.
Show Proof.
apply S.
Show Proof.
apply n. Defined.

Print add1.

Compute add1 2.
```

注意到我们通过使用.终止了`Definition`，而不是使用:=和一个项来定义它。这个记号会告诉Coq进入'证明脚本模式*(Proof Scripting Mode)*'来构造类型是**nat** → **nat**的项。并且，我们通过使用`Defined`而不是`Qed`来终止证明；这使得这个定义是'透明的*(Transparent)*'，所以它可以在计算中就像正常定义的函数一样被使用。（通过`Qed`定义的对象在计算中是不透明的。）

这个特性主要是在定义拥有依赖类型的函数时非常有用。我们不会在本书中详细讨论后者。但是它确实表明了Coq里面基本思想的一致性和正交性。

11.5 逻辑联结词作为归纳类型

归纳定义足够用于表达我们目前为止遇到的大多数的联结词。事实上，只有全称量化（以及作为特殊情况的蕴含式）是Coq内置的，所有其他的都是被归纳定义的。在这一节中我们会看到它们的定义。

```
Module Props.
```

11.5.1 合取

为了证明$P \wedge Q$成立，我们必须同时给出P和Q的证据。因此，我们可以合理地将$P \wedge Q$的证明对象定义为包含两个证明的元祖：一个是P的证明，另一个是Q的证明。即我们拥有如下定义。

```
Module And.

Inductive and (P Q : Prop) : Prop :=
| conj : P → Q → and P Q.

End And.
```

注意到这个定义与在章节 Poly 中给出的 **prod** 定义的类型的相似处；唯一的不同之处在于，**prod**的参数是Type，而**and**的类型是Prop。

```
Print prod.
```

这个定义能够解释为什么destruct和intros模式能用于一个合取前提。情况分析允许我们考虑所有$P \wedge Q$可能被证明的方式——只有一种方式（即 conj构造子）。

类似地，split策略能够用于所有只有一个构造子的归纳定义命题。特别地，它能够用于**and**：

```
Lemma and_comm : ∀ P Q : Prop, P ∧ Q ↔ Q ∧ P.
Proof.
  intros P Q. split.
  - intros [HP HQ]. split.
    + apply HQ.
    + apply HP.
  - intros [HQ HP]. split.
    + apply HP.
    + apply HQ.
Qed.
```

这解释了为什么一直以来我们能够使用策略来操作**and**的归纳定义。我们也可以使用模式匹配来用它直接构造证明。例如：

```
Definition and_comm'_aux P Q (H : P ∧ Q) : Q ∧ P :=
  match H with
  | conj HP HQ ⇒ conj HQ HP
```

```
  end.

Definition and_comm' P Q : P ∧ Q ↔ Q ∧ P :=
  conj (and_comm'_aux P Q) (and_comm'_aux Q P).
```

练习：2 星, standard, optional (conj_fact) 构造一个证明对象来证明下列命题。

```
Definition conj_fact : ∀ P Q R, P ∧ Q → Q ∧ R → P ∧ R
  . Admitted.
```

□

11.5.2 析取

析取的归纳定义有两个构造子，分别用于析取的两边：

```
Module Or.

Inductive or (P Q : Prop) : Prop :=
| or_introl : P → or P Q
| or_intror : Q → or P Q.

End Or.
```

这个声明解释了`destruct`策略在一个析取前提上的行为，产生的子类型和 or_introl以及or_intror构造子的形状相匹配。

又一次地，我们可以不使用策略，直接写出涉及**or**的定义的证明对象。

练习：2 星, standard, optional (or_commut") 尝试写下or_commut的显式证明对象。（不要使用`Print`来偷看我们已经定义的版本!）

```
Definition or_comm : ∀ P Q, P ∨ Q → Q ∨ P
  . Admitted.
```

□

11.5.3 存在量化

为了给出存在量词的证据，我们将一个证据类型x和x满足性质P的证明打包在一起：

```
Module Ex.

Inductive ex {A : Type} (P : A → Prop) : Prop :=
```

```
| ex_intro : ∀ x : A, P x → ex P.

End Ex.
```

打包之后的命题可以通过解包操作受益。这里的核心定义是为了用于构造 **ex** P命题的类型构造器**ex**，其中P自身是一个从类型为A的证据类型值到命题的*'函数(Function)'*。构造子ex_intro提供了一个给定证据类型x和$P\ x$的证明，可以构造**ex** P的证据的方式。

我们更加熟悉的类型$\exists\ x,\ P\ x$可以转换为一个涉及**ex**的表达式：

```
Check ex (fun n ⇒ ev n) : Prop.
```

下面是我们如何定义一个涉及**ex**的显式证明对象：

```
Definition some_nat_is_even : ∃ n, ev n :=
  ex_intro ev 4 (ev_SS 2 (ev_SS 0 ev_0)).
```

练习：2 星, standard, optional (ex_ev_Sn) 完成下列证明对象的定义：

```
Definition ex_ev_Sn : ex (fun n ⇒ ev (S n))
  . Admitted.
```

□

11.5.4 True和False

True命题的归纳定义很简单：

```
Inductive True : Prop :=
  | I : True.
```

它拥有一个构造子（因此**True**的所有证据都是一样的，所以给出一个 **True**的证明并没有信息量。）

False也一样的简单——事实上，它是如此简单，以致于第一眼看上去像是一个语法错误。

```
Inductive False : Prop := .
```

也就是说， **False**是一个'没有'构造子的归纳类型–即，没有任何方式能够构造一个它的证明。

```
End Props.
```

11.6 相等关系

在Coq里，甚至连相等关系都不是内置的。它拥有如下的归纳定义。（事实上，在标准库里的定义是这里给出的定义的轻微变体，前者给出了稍微容易使用一些的归纳法则。）

```
Module MyEquality.

Inductive eq {X:Type} : X → X → Prop :=
| eq_refl : ∀ x, eq x x.

Notation "x == y" := (eq x y)
                     (at level 70, no associativity)
                     : type_scope.
```

我们可以这样理解这个定义，给定一个集合X，它定义了由X的一对值 (*x*和*y*)所索引的"*x*与*y*相等"的一*'系列(Family)'*的命题。只有一种方式能够构造该系列中成员的证据：将构造子eq_refl应用到类型X 和值*x*:X，产生一个*x*等于*x*的证据。

其它形如 **eq** *x y* 的类型中的 *x* 和 *y* 并不相同，因此是非居留的。

我们可以使用eq_refl来构造证据，比如说，2 = 2。那么我们能否使用它来构造证据1 + 1 = 2呢？答案是肯定的。事实上，它就是同一个证据！

原因是如果两个项能够通过一些简单的计算规则*'可转换(convertible)'*，那么Coq认为两者"相等"。

这些计算规则，与Compute所使用的规则相似，包括函数应用的计算，定义的内联，match语句的化简。

```
Lemma four: 2 + 2 == 1 + 3.
Proof.
  apply eq_refl.
Qed.
```

至今为止我们所用来证据相等关系的reflexivity策略本质上只是apply eq_refl的简写。

在基于策略的相等关系证明中，转换规则通常隐藏在simpl的使用后面（在其他策略中或显式或隐式，例如reflexivity）。

而在如下的显式证明对象中，你可以直接看到它们：

```
Definition four' : 2 + 2 == 1 + 3 :=
  eq_refl 4.
```

```
Definition singleton : ∀ (X:Type) (x:X), []++[x] == x::[] :=
  fun (X:Type) (x:X) ⇒ eq_refl [x].
```

练习：2 星, standard (equality__leibniz_equality) 相等关系的归纳定义隐含了*'Leibniz*相等关系*(Leibniz equality)'*：当我们说"x和y相等的时候"，我们意味着所有x满足的性质P，对于y 来说也满足。

```
Lemma equality__leibniz_equality : ∀ (X : Type) (x y: X),
  x == y → ∀ P:X→Prop, P x → P y.
Proof.
    Admitted.
    □
```

练习：5 星, standard, optional (leibniz_equality__equality) 请说明，事实上，相等关系的归纳定义和Leibniz相等关系是 *'*等价的*(equivalent)'*。

```
Lemma leibniz_equality__equality : ∀ (X : Type) (x y: X),
  (∀ P:X→Prop, P x → P y) → x == y.
Proof.
    Admitted.
    □

End MyEquality.
```

11.6.1 再论反演

我们曾经见过`inversion`被同时用于相等关系前提，和关于被归纳定义的命题的前提。现在我们明白了实际上它们是同一件事情。那么我们现在可以细究一下`inversion`是如何工作的。

一般来说，`inversion`策略...

- 接受一个前提H，该前提的类型P是通过归纳定义的，以及
- 对于P的定义里的每一个构造子C，
  - 产生一个新的子目标，在该子目标中我们假设H是通过C构造的，
  - 作为额外的假设，在子目标的上下文中增加C的论据（前提），

- 将C的结论（结果类型）与当前的目标相匹配，计算出为了能够应用C而必须成立的一些相等关系，
- 将这些相等关系加入上下文中（以及，为了方便，在目标中替换它们），以及
- 如果这些相等关系无法满足（例如，它们涉及到S n = O），那么立即解决这个子目标。

'例子'：如果我们反演一个使用**or**构造的前提，它有两个构造子，所以产生了两个子目标。构造子的结论（结果类型，即$P \vee Q$）并没有对于P和Q的形式有任何要求，所以在子目标的上下文中我们不会获得额外的相等关系。

'例子'：如果我们反演一个使用**and**构造的前提，它只有一个构造子，所以只产生一个子目标。再一次地，构造子的结论（结果类型，即$P \wedge Q$）并没有对于P和Q的形式有任何要求，所以在子目标的上下文中我们不会获得额外的相等关系。不过，这个构造子有两个额外的参数，我们能够在子目标的上下文中看到它们。

'例子'：如果我们反演一个使用**eq**构造的前提，它也只有一个构造子，所以只产生一个子目标。但是，现在eq_refl构造子的形式给我们带来的额外的信息：它告诉**eq**的两个参数必须是一样的。于是`inversion`策略会将这个事实加入到上下文中。

Chapter 12

Library LF.Maps

12.1 Maps: 全映射与偏映射

'映射（*Map*）'（或'字典（*Dictionary*）'）是一种非常普遍的数据结构，在编程语言理论中尤甚，而之后的章节中我们会多次用到它。映射也适合运用之前学过的概念进行研究，包括如何在高阶函数之外构建数据结构（见 Basics 和 Poly）以及通过互映来精简证明（见 IndProp）。

我们会定义两种映射：在查找的键不存在时，'全映射'会返回"默认"元素，而'偏映射'则会返回一个 **option** 来指示成功还是失败。后者根据前者来定义，它使用 None 默认元素。

12.2 Coq 标准库

开始前的小插话...

和我们目前学过的章节不同，本章无需通过 `Require Import` 导入之前的章节（自然也就不会间接导入更早的章节）。从本章开始，我们我们将直接从 Coq 标准库中导入需要的定义和定理。然而应该不会注意到多大差别，因为我们一直小心地将自己的定义和定理的命名与标准库中的部分保持一致，无论它们在哪里重复。

```
From Coq Require Import Arith.Arith.
From Coq Require Import Bool.Bool.
Require Export Coq.Strings.String.
From Coq Require Import Logic.FunctionalExtensionality.
```

```
From Coq Require Import Lists.List.
Import ListNotations.
```

标准库的文档见 https://coq.inria.fr/library/。

`Search` 指令可用于查找涉及具体类型对象的定理。我们花点时间来熟悉一下它。

12.3 标识符

首先我们需要键的类型来对映射进行索引。在 *Lists.v* 中，我们为类似的目的引入了 **id** 类型。而在'《软件基础》'后面的部分，我们会使用 Coq 标准库中的 string 类型。

为了比较字符串，我们定义了 eqb_string 函数，它在内部使用 Coq 字符串库中的 string_dec 函数。

```
Definition eqb_string (x y : string) : bool :=
  if string_dec x y then true else false.
```

（函数 string_dec 来自于 Coq 的字符串标准库。如果你查看 string_dec 的结果类型，就会发现其返回值的类型并不是 **bool**，而是一个形如 $\{x = y\} + \{x \neq y\}$ 的类型，叫做 **sumbool** 类型，它可以看做"带有证据的布尔类型"。形式上来说，一个 $\{x = y\} + \{x \neq y\}$ 类型的元素要么是 x 和 y 的相等的证明，要么就是它们不相等的证明，与一个标签一起来指出具体是哪一个。不过就目前来说，你可以把它当做一个花哨的 **bool**。）

现在我们需要一些关于字符串相等性的基本性质... `Theorem eqb_string_refl : ∀ s : string, true = eqb_string s s.`

```
Proof.
  intros s. unfold eqb_string.
  destruct (string_dec s s) as [Hs_eq | Hs_not_eq].
  - reflexivity.
  - destruct Hs_not_eq. reflexivity.
Qed.
```

两个字符串在 eqb_string 的意义上相等，当且仅当它们在 $=$ 的意义上相等。因此 eqb_string 中反映了 $=$，IndProp 一章中讨论了「互映」的意义。

```
Theorem eqb_string_true_iff : ∀ x y : string,
    eqb_string x y = true ↔ x = y.
Proof.
   intros x y.
```

```
  unfold eqb_string.
  destruct (string_dec x y) as [Hs_eq | Hs_not_eq].
  - rewrite Hs_eq. split. reflexivity. reflexivity.
  - split.
    + intros contra. discriminate contra.
    + intros H. rewrite H in Hs_not_eq. destruct Hs_not_eq. reflexivity.
Qed.
```

类似地：

```
Theorem eqb_string_false_iff : ∀ x y : string,
    eqb_string x y = false ↔ x ≠ y.
Proof.
  intros x y. rewrite ← eqb_string_true_iff.
  rewrite not_true_iff_false. reflexivity. Qed.
```

以下便于使用的变体只需通过改写就能得出：

```
Theorem false_eqb_string : ∀ x y : string,
   x ≠ y → eqb_string x y = false.
Proof.
  intros x y. rewrite eqb_string_false_iff.
  intros H. apply H. Qed.
```

12.4 全映射

在本章中，我们的主要任务就是构建一个偏映射的定义，其行为类似于我们之前在 Lists 一章中看到的那个，再加上伴随其行为的引理。

不过这一次，我们将使用'函数'而非"键-值"对的列表来构建映射。这种表示方法的优点在于它提供了映射更具'外延性'的视角，即以相同方式回应查询的两个映射将被表示为完全相同的东西（即一模一样的函数），而非只是"等价"的数据结构。这反过来简化了使用映射的证明。

我们会分两步构建偏映射。首先，我们定义一个'全映射'类型，它在某个映射中查找不存在的键时会返回默认值。

```
Definition total_map (A : Type) := string → A.
```

直观上来说，一个元素类型为 A 的全映射不过就是个根据 string 来查找 A 的函数。

给定函数 t_empty 一个默认元素，它会产生一个空的全映射。此映射在应用到任何字符串时都会返回默认元素。

```
Definition t_empty {A : Type} (v : A) : total_map A :=
  (fun _ ⇒ v).
```

更有趣的是 update 函数，它和之前一样，接受一个映射 m、一个键 x 以及一个值 v，并返回一个将 x 映射到 v 的新映射；其它键则与 m 中原来的保持一致。

```
Definition t_update {A : Type} (m : total_map A)
                    (x : string) (v : A) :=
  fun x' ⇒ if eqb_string x x' then v else m x'.
```

此定义是个高阶编程的好例子：t_update 接受一个'函数' m 并产生一个新的函数 fun x' ⇒ ...，它的表现与所需的映射一致。

例如，我们可以构建一个从 string 到 **bool** 的映射，其中 "foo" 和 "bar" 映射到 true，其它键则映射到 false，就像这样：

```
Definition examplemap :=
  t_update (t_update (t_empty false) "foo" true)
           "bar" true.
```

接下来，我们引入一些新的记法来方便映射的使用。

首先，我们会使用以下记法，根据一个默认值来创建空的全映射。 Notation "'_' '!->' v" := (t_empty v)
  (at level 100, right associativity).

```
Example example_empty := (_ !-> false).
```

然后，我们引入一种方便的记法，通过一些绑定来扩展现有的映射。 Notation "x '!->' v ';' m" := (t_update m x v)
                                  (at level 100, v at $next$ level, right associativity).

前面的 examplemap 现在可以定义如下：

```
Definition examplemap' :=
  ( "bar" !-> true;
    "foo" !-> true;
    _ !-> false
  ).
```

到这里就完成了全映射的定义。注意我们无需定义 find 操作，因为它不过就是个函数应用！

```
Example update_example1 : examplemap' "baz" = false.
Proof. reflexivity. Qed.

Example update_example2 : examplemap' "foo" = true.
Proof. reflexivity. Qed.

Example update_example3 : examplemap' "quux" = false.
Proof. reflexivity. Qed.

Example update_example4 : examplemap' "bar" = true.
Proof. reflexivity. Qed.
```

为了在后面的章节中使用映射，我们需要一些关于其表现的基本事实。

即便你没有进行下面的练习，也要确保透彻地理解以下引理的陈述！

（其中有些证明需要函数的外延性公理，我们在 Logic 一节中讨论过它）。

练习：1 星, standard, optional (t_apply_empty) 首先，空映射对于所有的键都会返回默认元素（即，空映射总是返回默认元素）：

```
Lemma t_apply_empty : ∀ (A : Type) (x : string) (v : A),
    (_ !-> v) x = v.
Proof.
    Admitted.
```

□

练习：2 星, standard, optional (t_update_eq) 接着，如果将映射 m 的键 x 关联的值更新为 v，然后在 update 产生的新映射中查找 x，就会得到 v（即，更新某个键的映射，查找它就会得到更新后的值）：

```
Lemma t_update_eq : ∀ (A : Type) (m : total_map A) x v,
    (x !-> v ; m) x = v.
Proof.
    Admitted.
```

□

练习：2 星, standard, optional (t_update_neq) 此外，如果将映射 m 的键 $x1$ 更新后在返回的结果中查找'另一个'键 $x2$，那么得到的结果与在 m 中查找它的结果相同（即，更新某个键的映射，不影响其它键的映射）：

```
Theorem t_update_neq : ∀ (A : Type) (m : total_map A) x1 x2 v,
  x1 ≠ x2 →
  (x1 !-> v ; m) x2 = m x2.
Proof.
  Admitted.
```

□

练习：2 星, standard, optional (t_update_shadow) 如果将映射 m 的键 x 关联的值更新为 $v1$ 后，又将同一个键 x 更新为另一个值 $v2$，那么产生的映射与仅将第二次 update 应用于 m 所得到的映射表现一致（即二者应用到同一键时产生的结果相同）：

```
Lemma t_update_shadow : ∀ (A : Type) (m : total_map A) x v1 v2,
  (x !-> v2 ; x !-> v1 ; m) = (x !-> v2 ; m).
Proof.
  Admitted.
```

□

对于最后两个全映射的引理而言，用 IndProp 一章中引入的互映法（Reflection idioms）来证明会十分方便。我们首先通过证明基本的'互映引理'，将字符串上的相等关系命题与布尔函数 eqb_string 关联起来。

练习：2 星, standard, optional (eqb_stringP) 请仿照 IndProp 一章中对 eqb_natP 的证明来证明以下引理：

```
Lemma eqb_stringP : ∀ x y : string,
  reflect (x = y) (eqb_string x y).
Proof.
  Admitted.
```

□

现在，给定 string 类型的字符串 $x1$ 和 $x2$，我们可以在使用策略 `destruct` (eqb_stringP $x1$ $x2$) 对 eqb_string $x1$ $x2$ 的结果进行分类讨论的同时，生成关于 $x1$ 和 $x2$ （在 $=$ 的意义上）的相等关系前提。

练习：2 星, standard (t_update_same) 请仿照 IndProp 一章中的示例，用 eqb_stringP 来证明以下定理，它陈述了：如果我们用映射 m 中已经与键 x 相关联的值更新了 x，那么其结果与 m 相等：

```
Theorem t_update_same : ∀ (A : Type) (m : total_map A) x,
    (x !-> m x ; m) = m.
Proof.
    Admitted.
```

□

练习：3 星, standard, recommended (t_update_permute) 使用 eqb_stringP 来证明最后一个 update 函数的性质：如果我们更新了映射 m 中两个不同的键，那么更新的顺序无关紧要。

```
Theorem t_update_permute : ∀ (A : Type) (m : total_map A)
                                                v1 v2 x1 x2,
    x2 ≠ x1 →
    (x1 !-> v1 ; x2 !-> v2 ; m)
    =
    (x2 !-> v2 ; x1 !-> v1 ; m).
Proof.
    Admitted.
```

□

12.5 偏映射

最后，我们在全映射之上定义'偏映射'。元素类型为 A 的偏映射不过就是个元素类型为 **option** A，默认元素为 None 的全映射。

```
Definition partial_map (A : Type) := total_map (option A).

Definition empty {A : Type} : partial_map A :=
  t_empty None.

Definition update {A : Type} (m : partial_map A)
                  (x : string) (v : A) :=
  (x !-> Some v ; m).
```

我们为偏映射引入类似的记法。 `Notation "x '|->' v ';' m" := (update m x v)`
`(at level 100, v at next level, right associativity).`

当最后一种情况为空时，我们也可以隐藏它。 `Notation "x '|->' v" := (update empty x v)`
`(at level 100).`

```
Example examplepmap :=
  ("Church" |-> true ; "Turing" |-> false).
```

现在我们将所有关于全映射的基本引理直接转换成对应的偏映射引理。

```
Lemma apply_empty : ∀ (A : Type) (x : string),
    @empty A x = None.
Proof.
  intros. unfold empty. rewrite t_apply_empty.
  reflexivity.
Qed.

Lemma update_eq : ∀ (A : Type) (m : partial_map A) x v,
    (x |-> v ; m) x = Some v.
Proof.
  intros. unfold update. rewrite t_update_eq.
  reflexivity.
Qed.

Theorem update_neq : ∀ (A : Type) (m : partial_map A) x1 x2 v,
    x2 ≠ x1 →
    (x2 |-> v ; m) x1 = m x1.
Proof.
  intros A m x1 x2 v H.
  unfold update. rewrite t_update_neq. reflexivity.
  apply H. Qed.

Lemma update_shadow : ∀ (A : Type) (m : partial_map A) x v1 v2,
    (x |-> v2 ; x |-> v1 ; m) = (x |-> v2 ; m).
Proof.
  intros A m x v1 v2. unfold update. rewrite t_update_shadow.
```

```
  reflexivity.
Qed.

Theorem update_same : ∀ (A : Type) (m : partial_map A) x v,
    m x = Some v →
    (x |-> v ; m) = m.
Proof.
  intros A m x v H. unfold update. rewrite ← H.
  apply t_update_same.
Qed.

Theorem update_permute : ∀ (A : Type) (m : partial_map A)
                                        x1 x2 v1 v2,
    x2 ≠ x1 →
    (x1 |-> v1 ; x2 |-> v2 ; m) = (x2 |-> v2 ; x1 |-> v1 ; m).
Proof.
  intros A m x1 x2 v1 v2. unfold update.
  apply t_update_permute.
Qed.
```

Chapter 13

Library LF.IndProp

13.1 IndProp: 归纳定义的命题

```
Set Warnings "-notation-overridden,-parsing".
From LF Require Export Logic.
Require Coq.omega.Omega.
```

13.2 归纳定义的命题

在 Logic 一章中，我们学习了多种方式来书写命题，包括合取、析取和存在量词。在本章中，我们引入另一种新的方式：'归纳定义（*Inductive Definitions*）'。

'注意'：为简单起见，本章中的大部分内容都用一个归纳定义的"偶数性"作为展示的例子。你可能会为此感到不解，因为我们已经有一种将偶数性定义为命题的完美方法了（若 n 等于某个整数的二倍，那么它是是偶数）。尚若如此，那么请放心，在本章末尾和以后的章节中，我们会看到更多引人入胜的归纳定义的命题示例。

在前面的章节中，我们已经见过两种表述 n 为偶数的方式了：

(1) evenb n = true，以及

(2) $\exists\, k$, n = double k。

然而还有一种方式是通过如下规则来建立 n 的偶数性质：

- 规则 ev_0: 0 是偶数。
- 规则 ev_SS: 如果 n 是偶数, 那么 S (S n) 也是偶数。

为了理解这个新的偶数性质定义如何工作，我们可想象如何证明 4 是偶数。根据规则 ev_SS，需要证明 2 是偶数。这时，只要证明 0 是偶数，我们可继续通过规则 ev_SS 确保它成立。而使用规则 ev_0 可直接证明 0 是偶数。

接下来的课程中，我们会看到很多类似方式定义的命题。在非形式化的讨论中，使用简单的记法有助于阅读和书写。 '推断规则（*Inference Rules*）'就是其中的一种。（我们为此性质取名为 **ev**，因为 even 已经用过了。）

(ev_0) ev 0

ev n

(ev_SS) ev (S (S n))

若将上面的规则重新排版成推断规则，我们可以这样阅读它，如果线上方的 '前提（*Premises*）'成立，那么线下方的'结论（*Conclusion*）'成立。比如，规则 ev_SS 读做如果 n 满足 even，那么 S (S n) 也满足。如果一条规则在线上方没有前提，则结论直接成立。

我们可以通过组合推断规则来展示证明。下面展示如何转译 4 是偶数的证明：

(ev_0) ev 0

(ev_SS) ev 2

(ev_SS) ev 4

（为什么我们把这样的证明称之为"树"而非其他，比如"栈"？因为一般来说推断规则可以有多个前提。我们很快就会看到一些例子。

13.2.1 偶数性的归纳定义

基于上述，可将偶数性质的定义翻译为在 Coq 中使用 `Inductive` 声明的定义，声明中每一个构造子对应一个推断规则：

```
Inductive ev : nat → Prop :=
| ev_0 : ev 0
| ev_SS (n : nat) (H : ev n) : ev (S (S n)).
```

这个定义与之前 Inductive 定义的用法有一个有趣的区别：一方面，我们定义的并不是一个 Type（如 **nat**），而是一个将 **nat** 映射到 Prop 的函数——即关于数的性质。然而真正要关注的是，由于 **ev** 中的 **nat** 参数出现在冒号*'右侧'*，这允许在不同的构造子类型中使用不同的值：例如 ev_0 类型中的 0 以及 ev_SS 类型中的 S (S n)。与此相应，每个构造子的类型必须在冒号后显式指定，并且对于某个自然数 n 来说，每个构造子的类型都必须有 **ev** n 的形式。

相反，回忆 **list** 的定义：

Inductive list (X:Type) : Type := | nil | cons (x : X) (l : list X).

它以*'全局的方式'*在冒号*'左侧'*引入了参数 X，强迫 nil 和 cons 的结果为同一个类型（**list** X）。如果在定义 **ev** 时将 **nat** 置于冒号左侧，就会得到如下错误：

Fail Inductive **wrong_ev** (n : **nat**) : Prop :=
| wrong_ev_0 : **wrong_ev** 0
| wrong_ev_SS (H: **wrong_ev** n) : **wrong_ev** (S (S n)).

在 Inductive 定义中，类型构造子冒号左侧的参数叫做形参（Parameter），而右侧的叫做索引（Index）或注解（Annotation）。

例如，在 Inductive **list** (X : Type) := ... 中，X 是一个形参；而在 Inductive **ev** : **nat** → Prop := ... 中，未命名的 **nat** 参数是一个索引。

在 Coq 中，我们可以认为 **ev** 定义了一个性质 **ev** : **nat** → Prop，其包括"证据构造子" ev_0 : **ev** 0 和 ev_SS : ∀ n, **ev** n → **ev** (S (S n))。

这些"证据构造子" 等同于已经证明过的定理。具体来说，我们可以使用 Coq 中的 apply 策略和规则名称来证明某个数的 **ev** 性质.....

Theorem ev_4 : **ev** 4.
Proof. apply ev_SS. apply ev_SS. apply ev_0. Qed.

.....或使用函数应用的语法：

Theorem ev_4' : **ev** 4.
Proof. apply (ev_SS 2 (ev_SS 0 ev_0)). Qed.

我们同样可以对前提中使用到 **ev** 的定理进行证明。

Theorem ev_plus4 : ∀ n, **ev** n → **ev** (4 + n).
Proof.
  intros n. simpl. intros Hn.
  apply ev_SS. apply ev_SS. apply Hn.

Qed.

更一般地，我们可以证明以任意数乘 2 是偶数：

练习：1 星, standard (ev_double) Theorem ev_double : ∀ n, **ev** (double n).

Proof.

Admitted.

□

13.3 在证明中使用证据

除了'构造'证据（evidence）来表示某个数是偶数，我们还可以'解构'这样的证据，这等于对它的构造进行论证。

对 **ev** 而言，使用 Inductive 声明来引入 **ev** 会告诉 Coq， ev_0 和 ev_SS 构造子不仅是构造偶数证明证据的有效方式，还是构造一个数满足 **ev** 的证据的'唯一'方式。

换句话说，如果某人展示了对于 **ev** n 的证据 E，那么我们知道 E 必是二者其一：

- E 是 ev_0（且 n 为 O），或
- E 是 ev_SS n' E'（且 n 为 S (S n')，E' 为 **ev** n' 的证据）.

这样的形式暗示着，我们可以像分析归纳定义的数据结构一样分析形如 **ev** n 的假设；特别地，对于这类证据使用'归纳（*induction*）'和'分类讨论（*case analysis*）'来进行论证也是可行的。让我们通过一些例子来学习实践中如何使用他们。

13.3.1 对证据进行反演

Suppose we are proving some fact involving a number n, and we are given **ev** n as a hypothesis. We already know how to perform case analysis on n using destruct or induction, generating separate subgoals for the case where n = O and the case where n = S n' for some n'. But for some proofs we may instead want to analyze the evidence that **ev** n *directly*. As a tool, we can prove our characterization of evidence for **ev** n, using destruct.

Theorem ev_inversion :

∀ (n : **nat**), **ev** n →

```
    (n = 0) ∨ (∃ n', n = S (S n')) ∧ ev n').
Proof.
  intros n E.
  destruct E as [ | n' E'].
  -
    left. reflexivity.
  -
    right. ∃ n'. split. reflexivity. apply E'.
Qed.
```

用 destruct 解构证据即可证明下述定理：

```
Theorem ev_minus2 : ∀ n,
  ev n → ev (pred (pred n)).
Proof.
  intros n E.
  destruct E as [| n' E'].
  - simpl. apply ev_0.
  - simpl. apply E'.
Qed.
```

However, this variation cannot easily be handled with just destruct.

```
Theorem evSS_ev : ∀ n,
  ev (S (S n)) → even n.
```

直观来说，我们知道支撑前提的证据不会由 ev_0 组成，因为 0 和 S 是 **nat** 类型不同的构造子；由此 ev_SS 是唯一需要应对的情况（译注：ev_0 无条件成立）。不幸的是，destruct 并没有如此智能，它仍然为我们生成两个子目标。更坏的是，于此同时最终目标没有改变，也无法为完成证明提供任何有用的信息。

```
Proof.
  intros n E.
  destruct E as [| n' E'] eqn:EE.
  -


Abort.
```

究竟发生了什么？应用 destruct 把性质的参数替换为对应于构造子的值。这对于

证明 *ev_minus2'* 是有帮助的，因为在最终目标中直接使用到了参数 n。然而，这对于 evSS_ev 并没有帮助，因为被替换掉的 S (S n) 并没有在其他地方被使用。

If we *remember* that term S (S n), the proof goes through. (We'll discuss *remember* in more detail below.)

```
Theorem evSS_ev_remember : ∀ n,
  ev (S (S n)) → ev n.
Proof.
  intros n H. remember (S (S n)) as k. destruct H as [|n' E'].
  -


    discriminate Heqk.
  -


    injection Heqk as Heq. rewrite Heq in E'. apply E'.
Qed.
```

Alternatively, the proof is straightforward using our inversion lemma.

```
Theorem evSS_ev : ∀ n, ev (S (S n)) → ev n.
Proof. intros n H. apply ev_inversion in H. destruct H.
 - discriminate H.
 - destruct H as [n' [Hnm Hev]]. injection Hnm as Heq.
   rewrite Heq. apply Hev.
Qed.
```

Note how both proofs produce two subgoals, which correspond to the two ways of proving **ev**. The first subgoal is a contradiction that is discharged with `discriminate`. The second subgoal makes use of `injection` and `rewrite`. Coq provides a handy tactic called `inversion` that factors out that common pattern.

The `inversion` tactic can detect (1) that the first case ($n = 0$) does not apply and (2) that the n' that appears in the ev_SS case must be the same as n. It has an "`as`" variant similar to `destruct`, allowing us to assign names rather than have Coq choose them.

```
Theorem evSS_ev' : ∀ n,
  ev (S (S n)) → ev n.
Proof.
```

```
  intros n E.
  inversion E as [| n' E' EQ].
  apply E'.
Qed.
```

The `inversion` tactic can apply the principle of explosion to "obviously contradictory" hypotheses involving inductively defined properties, something that takes a bit more work using our inversion lemma. For example:

```
Theorem one_not_even : ¬ ev 1.
Proof.
  intros H. apply ev_inversion in H.
  destruct H as [ | [m [Hm _]]].
  - discriminate H.
  - discriminate Hm.
Qed.

Theorem one_not_even' : ¬ ev 1.
  intros H. inversion H. Qed.
```

练习：1 星, standard (inversion_practice) 利用 `inversion` 策略证明以下结论。（如想进一步练习，请使用反演定理证明之。）

```
Theorem SSSSev__even : ∀ n,
  ev (S (S (S (S n)))) → ev n.
Proof.
    Admitted.
```

□

练习：1 星, standard (ev5_nonsense) 请使用 `inversion` 策略证明以下结果。

```
Theorem ev5_nonsense :
  ev 5 → 2 + 2 = 9.
Proof.
    Admitted.
```

□

The `inversion` tactic does quite a bit of work. For example, when applied to an equality assumption, it does the work of both `discriminate` and `injection`. In addition, it carries out the `intros` and `rewrite`s that are typically necessary in the case of `injection`. It can also be applied, more generally, to analyze evidence for inductively defined propositions. As examples, we'll use it to reprove some theorems from chapter Tactics. (Here we are being a bit lazy by omitting the `as` clause from `inversion`, thereby asking Coq to choose names for the variables and hypotheses that it introduces.)

```
Theorem inversion_ex1 : ∀ (n m o : nat),
  [n; m] = [o; o] →
  [n] = [m].
Proof.
  intros n m o H. inversion H. reflexivity. Qed.

Theorem inversion_ex2 : ∀ (n : nat),
  S n = O →
  2 + 2 = 5.
Proof.
  intros n contra. inversion contra. Qed.
```

`inversion` 的工作原理大致如下：假设 H 指代上下文中的假设 P，且 P 由 `Inductive` 归纳定义，则对于 P 每一种可能的构造，`inversion` H 各为其生成子目标。子目标中自相矛盾者被忽略，证明其余子命题即可得证原命题。在证明子目标时，上下文中的 H 会替换为 P 的构造条件，即其构造子所需参数以及必要的等式关系。例如：倘若 **ev** n 由 *evSS* 构造，上下文中会引入参数 *n'*、**ev** *n'*，以及等式 S (S *n'*) = n。

上面的 ev_double 练习展示了偶数性质的一种新记法，其被之前的两种记法所蕴含。（因为，由 Logic 一章中的 even_bool_prop，我们已经知道他们是互相等价的。）为了展示这三种方式的一致性，我们需要下面的引理：

```
Lemma ev_even_firsttry : ∀ n,
  ev n → even n.
Proof.
```

我们可以尝试使用分类讨论或对 n 进行归纳。但由于 **ev** 在前提中出现，所以和前面章节的一些例子一样，这种策略行不通，因为如前面提到的，归纳法则会讨论 n-1，而它并不是偶数！如此我们似乎可以先试着对 **ev** 的证据进行反演。确实，第一个分类可以被平凡地证明。

```
  intros n E. inversion E as [EQ' | n' E' EQ'].
  -
    ∃ 0. reflexivity.
  - simpl.
```

然而第二个分类要困难一些。我们需要证明 ∃ k, S (S n') = double k，但唯一可用的假设是 E'，也即 **ev** n' 成立。但这对证明并没有帮助，我们似乎被卡住了，而对 E 进行分类讨论是徒劳的。

如果仔细观察第二个（子）目标，我们可以发现一些有意思的事情：对 E 进行分类讨论，我们可以把要证明的原始目标归约到另一个上，其涉及到另一个 **ev** 的证据： E'。形式化地说，我们可以通过展示如下证据来完成证明：

exists k', n' = double k',

这同原始的命题是等价的，只是 n' 被替换为 n。确实，通过这个中间结果完成证明并不困难。

```
    assert (I : (∃ k', n' = double k') →
                (∃ k, S (S n') = double k)).
    { intros [k' Hk']. rewrite Hk'. ∃ (S k'). reflexivity. }
    apply I.
Abort.
```

13.3.2 对证据进行归纳

上面的情形看起来似曾相识，但这并非巧合。在 Induction 一章中，我们曾试着用分类讨论来证明其实需要归纳才能证明的命题。这一次，解决方法仍然是.....使用归纳!

对证据和对数据使用 induction 具有同样的行为：它导致 Coq 对每个可用于构造证据的构造子生成一个子目标，同时对递归出现的问题性质提供了归纳假设。

To prove a property of n holds for all numbers for which **ev** n holds, we can use induction on **ev** n. This requires us to prove two things, corresponding to the two ways in which **ev** n could have been constructed. If it was constructed by ev_0, then n=0, and the property must hold of 0. If it was constructed by ev_SS, then the evidence of **ev** n is of the form ev_SS n' E', where n = S (S n') and E' is evidence for **ev** n'. In this case, the inductive hypothesis says that the property we are trying to prove holds for n'.

让我们再次尝试证明这个引理：

```
Lemma ev_even : ∀ n,
```

```
  ev n → even n.
Proof.
  intros n E.
  induction E as [|n' E' IH].
  -
    ∃ 0. reflexivity.
  -
    destruct IH as [k' Hk'].
    rewrite Hk'. ∃ (S k'). reflexivity.
Qed.
```

这里我们看到 Coq 对 E' 产生了 IH，而 E' 是唯一递归出现的 **ev** 命题。由于 E' 中涉及到 n'，这个归纳假设是关于 n' 的，而非关于 n 或其他数字的。

关于偶数性质的第二个和第三个定义的等价关系如下：

```
Theorem ev_even_iff : ∀ n,
  ev n ↔ even n.
Proof.
  intros n. split.
  - apply ev_even.
  - intros [k Hk]. rewrite Hk. apply ev_double.
Qed.
```

我们会在后面的章节中看到，对证据进行归纳在很多领域里是一种常用的技术，特别是在形式化程序语言的语义时，由于其中很多有趣的性质都是归纳定义的。

下面的练习提供了一些简单的例子，来帮助你熟悉这项技术。

练习：2 星, standard (ev_sum) `Theorem ev_sum : ∀ n m, ev n → ev m → ev (n + m).`

```
Proof.
  Admitted.
```

□

练习：4 星, advanced, optional (ev'_ev) 一般来说，有很多种方式来归纳地定义一个性质。比如说，下面是关于 **ev** 的另一种（蹩脚的）定义：

```
Inductive ev' : nat → Prop :=
```

| ev'_0 : **ev'** 0

| ev'_2 : **ev'** 2

| ev'_sum n m (Hn : **ev'** n) (Hm : **ev'** m) : **ev'** (n + m).

请证明这个定义在逻辑上等同于前述定义。为了精简证明，请使用 Logic 一章中将定理应用到参数的技术，注意同样的技术也可用于归纳定义的命题的构造子。

Theorem ev'_ev : $\forall$ n, **ev'** n $\leftrightarrow$ **ev** n.

Proof.

Admitted.

□

练习：3 星, advanced, recommended (ev_ev__ev) 这里有两个可以试着在其上进行归纳的证据，一个不行就换另一个。

Theorem ev_ev__ev : $\forall$ n m,

ev (n+m) $\rightarrow$ **ev** n $\rightarrow$ **ev** m.

Proof.

Admitted.

□

练习：3 星, standard, optional (ev_plus_plus) 本练习只需要使用前述引理，而无需使用归纳或分类讨论，虽然一些改写可能会比较乏味。

Theorem ev_plus_plus : $\forall$ n m p,

ev (n+m) $\rightarrow$ **ev** (n+p) $\rightarrow$ **ev** (m+p).

Proof.

Admitted.

□

13.4 归纳关系

我们可以认为被一个数所参数化的命题（比如 **ev**）是一个'性质'，也即，它定义了 **nat** 的一个子集，其中的数可以被证明满足此命题。以同样的方式，我们可认为有两个参数的命题是一个'关系'，也即，它定义了一个可满足此命题的序对集合。

Module PLAYGROUND.

和命题一样，关系也可以归纳地定义。一个很有用的例子是整数的“小于等于”关系。

下面的定义应当是比较直观的。它提供了两种方法来描述一个数小于等于另一个数的证据：要么可观察到两个数相等，或提供证据显示第一个数小于等于第二个数的前继。

```
Inductive le : nat → nat → Prop :=
  | le_n (n : nat) : le n n
  | le_S (n m : nat) (H : le n m) : le n (S m).

Notation "m <= n" := (le m n).
```

类似于证明 **ev** 这样的性质，使用 le_n 和 le_S 构造子来证明关于 $\leq$ 的事实遵循了同样的模式。我们可以对构造子使用 `apply` 策略来证明 $\leq$ 目标（比如证明 3<=3 或 3<=6），也可以使用 `inversion` 策略来从上下文中 $\leq$ 的假设里抽取信息（比如证明 (2<=1) → 2+2=5）。

这里提供一些完备性检查。（请注意，尽管这同我们在开始课程时编写的函数“单元测试”类似，但我们在这里必须明确地写下他们的证明——`simpl` 和 `reflexivity` 并不会有效果，因为这些证明不仅仅是对表达式进行简化。）

```
Theorem test_le1 :
  3 ≤ 3.
Proof.
  apply le_n. Qed.

Theorem test_le2 :
  3 ≤ 6.
Proof.
  apply le_S. apply le_S. apply le_S. apply le_n. Qed.

Theorem test_le3 :
  (2 ≤ 1) → 2 + 2 = 5.
Proof.
  intros H. inversion H. inversion H2. Qed.
```

现在“严格小于”关系 $n < m$ 可以使用 **le** 来定义。

```
End Playground.

Definition lt (n m:nat) := le (S n) m.

Notation "m < n" := (lt m n).
```

这里展示了一些定义于自然数上的关系：

```
Inductive square_of : nat → nat → Prop :=
  | sq n : square_of n (n × n).

Inductive next_nat : nat → nat → Prop :=
  | nn n : next_nat n (S n).

Inductive next_ev : nat → nat → Prop :=
  | ne_1 n (H: ev (S n)) : next_ev n (S n)
  | ne_2 n (H: ev (S (S n))) : next_ev n (S (S n)).
```

练习：2 星, standard, optional (total_relation) 请定一个二元归纳关系 *total_relation* 对每一个自然数的序对成立。

练习：2 星, standard, optional (empty_relation) 请定一个二元归纳关系 *empty_relation* 对自然数永远为假。

From the definition of **le**, we can sketch the behaviors of `destruct`, `inversion`, and `induction` on a hypothesis H providing evidence of the form **le** *e1 e2*. Doing `destruct` H will generate two cases. In the first case, *e1* = *e2*, and it will replace instances of *e2* with *e1* in the goal and context. In the second case, *e2* = S *n'* for some *n'* for which **le** *e1 n'* holds, and it will replace instances of *e2* with S *n'*. Doing `inversion` H will remove impossible cases and add generated equalities to the context for further use. Doing `induction` H will, in the second case, add the induction hypothesis that the goal holds when *e2* is replaced with *n'*.

练习：3 星, standard, optional (le_exercises) 这里展示一些 $\leq$ 和 $<$ 关系的事实，我们在接下来的课程中将会用到他们。证明他们将会是非常有益的练习。

```
Lemma le_trans : ∀ m n o, m ≤ n → n ≤ o → m ≤ o.
Proof.
    Admitted.

Theorem O_le_n : ∀ n,
  0 ≤ n.
Proof.
    Admitted.
```

```
Theorem n_le_m__Sn_le_Sm : ∀ n m,
  n ≤ m → S n ≤ S m.
Proof.
    Admitted.

Theorem Sn_le_Sm__n_le_m : ∀ n m,
  S n ≤ S m → n ≤ m.
Proof.
    Admitted.

Theorem le_plus_l : ∀ a b,
  a ≤ a + b.
Proof.
    Admitted.

Theorem plus_le : ∀ n1 n2 m,
  n1 + n2 ≤ m →
  n1 ≤ m ∧ n2 ≤ m.
Proof.
    Admitted.
```

Hint: the next one may be easiest to prove by induction on n.

```
Theorem add_le_cases : ∀ n m p q,
    n + m ≤ p + q → n ≤ p ∨ m ≤ q.
Proof.
    Admitted.

Theorem lt_S : ∀ n m,
  n < m →
  n < S m.
Proof.
    Admitted.

Theorem plus_lt : ∀ n1 n2 m,
  n1 + n2 < m →
  n1 < m ∧ n2 < m.
Proof.
```

Admitted.

```
Theorem leb_complete : ∀ n m,
  n <=? m = true → n ≤ m.
Proof.
```

Admitted.

提示：在下面的问题中，对 m 进行归纳会使证明容易一些。

```
Theorem leb_correct : ∀ n m,
  n ≤ m →
  n <=? m = true.
Proof.
```

Admitted.

提示：以下定理可以不使用 `induction` 而证明。

```
Theorem leb_true_trans : ∀ n m o,
  n <=? m = true → m <=? o = true → n <=? o = true.
Proof.
```

Admitted.

□

练习：2 星, standard, optional (leb_iff) `Theorem leb_iff : ∀ n m,`

```
  n <=? m = true ↔ n ≤ m.
Proof.
```

Admitted.

□

```
Module R.
```

练习：3 星, standard, recommended (R_provability) 通过同样的方式，我们可以定义三元关系、四元关系等。例如，考虑以下定义在自然数上的三元关系：

```
Inductive R : nat → nat → nat → Prop :=
  | c1 : R 0 0 0
  | c2 m n o (H : R m n o) : R (S m) n (S o)
  | c3 m n o (H : R m n o) : R m (S n) (S o)
  | c4 m n o (H : R (S m) (S n) (S (S o))) : R m n o
```

| c5 m n o (H : **R** m n o) : **R** n m o.

- 下列哪个命题是可以被证明的？
  - **R** 1 1 2
  - **R** 2 2 6
- 如果在 **R** 的定义中我们丢弃 c5 构造子，可被证明的集合会发生变化吗？简要（一句话）解释你的答案。
- 如果在 **R** 的定义中我们丢弃 c4 构造子，可被证明的集合会发生变化吗？简要（一句话）解释你的答案。

```
Definition manual_grade_for_R_provability : option (nat×string) := None.
```
□

练习：3 星, standard, optional (R_fact) 关系 **R** 其实编码了一个熟悉的函数。请找出这个函数，定义它并在 Coq 中证明他们等价。

```
Definition fR : nat → nat → nat
  . Admitted.

Theorem R_equiv_fR : ∀ m n o, R m n o ↔ fR m n = o.
Proof.
    Admitted.
    □

End R.
```

练习：2 星, advanced (subsequence) 如果一个列表的所有元素以相同的顺序出现在另一个列表之中（但允许其中出现其他额外的元素），我们把第一个列表称作第二个列表的'子序列'。例如：

1;2;3
是以下所有列表的子序列
1;2;3 1;1;1;2;2;3 1;2;7;3 5;6;1;9;9;2;7;3;8
但'不是'以下列表的子序列
1;2 1;3 5;6;2;1;7;3;8.

- 在 **list nat** 上定一个归纳命题 **subseq**，其表达了子序列的涵义。（提示：你需要三个分类。）
- 证明子序列的自反关系 subseq_refl，也即任何列表是它自身的子序列。
- 证明关系 subseq_app 对任意列表 *l1*，*l2* 和 *l3*，如果 *l1* 是 *l2* 的子序列，那么 *l1* 也是 *l2* ++ *l3* 的子序列。
- （可选的，困难）证明子序列的传递关系 subseq_trans——也即，如果 *l1* 是 *l2* 的子序列，且 *l2* 是 *l3* 的子序列，那么 *l1* 是 *l3* 的子序列。（提示：仔细选择进行归纳的项！）

```
Inductive subseq : list nat → list nat → Prop :=


.

Theorem subseq_refl : ∀ (l : list nat), subseq l l.
Proof.
    Admitted.

Theorem subseq_app : ∀ (l1 l2 l3 : list nat),
  subseq l1 l2 →
  subseq l1 (l2 ++ l3).
Proof.
    Admitted.

Theorem subseq_trans : ∀ (l1 l2 l3 : list nat),
  subseq l1 l2 →
  subseq l2 l3 →
  subseq l1 l3.
Proof.
    Admitted.
□
```

练习：2 星, standard, optional (R_provability2) 假设我们在 Coq 中有如下定义：

```
Inductive R : nat -> list nat -> Prop := | c1 : R 0 □ | c2 n l (H: R n l) : R (S n) (n :: l) | c3 n l (H: R (S n) l) : R n l.
```

下列命题哪个是可被证明的？

- **R** 2 [1;0]
- **R** 1 [1;2;1;0]
- **R** 6 [3;2;1;0]

13.5 案例学习：正则表达式

性质 **ev** 提供了一个简单的例子来展示归纳定义和其基础的推理技巧，但这还不是什么激动人心的东西——毕竟，even 等价于我们之前见过的两个非归纳的定义，而看起来归纳定义并没有提供什么好处。为了更好地展示归纳定义的表达能力，我们继续使用它来建模计算机科学中的一个经典概念——正则表达式。

正则表达式是用来描述字符串集合的一种简单语言，定义如下：

```
Inductive reg_exp (T : Type) : Type :=
  | EmptySet
  | EmptyStr
  | Char (t : T)
  | App (r1 r2 : reg_exp T)
  | Union (r1 r2 : reg_exp T)
  | Star (r : reg_exp T).

Arguments EmptySet {T}.
Arguments EmptyStr {T}.
Arguments Char {T} _.
Arguments App {T} _ _.
Arguments Union {T} _ _.
Arguments Star {T} _.
```

请注意这个定义是'多态的'：**reg_exp** T 中的正则表达式描述了字符串，而其中的字符取自 T——也即，T 的元素构成的列表。

（同一般的实践略有不同，我们不要求类型 T 是有限的。由此可形成一些不同的正则表达式理论，但对于我们在本章的目的而言并无不同。）

我们通过以下规则来构建正则表达式和字符串，这些规则定义了正则表达式何时匹配一个字符串：

- 表达式 EmptySet 不匹配任何字符串。
- 表达式 EmptyStr 匹配空字符串 [].
- 表达式 Char x 匹配单个字符构成的字符串 $[x]$.
- 如果 $re1$ 匹配 $s1$, 且 $re2$ 匹配 $s2$, 那么 App $re1$ $re2$ 匹配 $s1$ ++ $s2$.
- 如果 $re1$ 和 $re2$ 中至少一个匹配 s, 那么 Union $re1$ $re2$ 匹配 s.
- 最后，如果我们写下某个字符串 s 作为一个字符串序列的连接 $s = s\_1$ ++ ... ++ s_k，且表达式 re 匹配其中每一个字符串 s_i，那么 Star re 匹配 s。

  特别来说，此字符串序列可能为空，因此无论 re 是什么 Star re 总是匹配空字符串 []。

容易把非形式化的定义翻译为使用 `Inductive` 的定义。我们用记法 s =~ re 表示，通过把该记法用 *Reserved* "预留"在 `Inductive` 定义之前，我们就能在定义中使用它！

```
Reserved Notation "s =~ re" (at level 80).

Inductive exp_match {T} : list T → reg_exp T → Prop :=
  | MEmpty : [] =~ EmptyStr
  | MChar x : [x] =~ (Char x)
  | MApp s1 re1 s2 re2
               (H1 : s1 =~ re1)
               (H2 : s2 =~ re2)
             : (s1 ++ s2) =~ (App re1 re2)
  | MUnionL s1 re1 re2
                 (H1 : s1 =~ re1)
               : s1 =~ (Union re1 re2)
  | MUnionR re1 s2 re2
                 (H2 : s2 =~ re2)
               : s2 =~ (Union re1 re2)
  | MStar0 re : [] =~ (Star re)
```

```
  | MStarApp s1 s2 re
                 (H1 : s1 =~ re)
                 (H2 : s2 =~ (Star re))
               : (s1 ++ s2) =~ (Star re)
  where "s =~ re" := (exp_match s re).

Lemma quiz : ∀ T (s:list T), ~(s =~ EmptySet).
Proof. intros T s Hc. inversion Hc. Qed.
```

出于可读性的考虑，在此我们也展示使用推断规则表示的定义。

(MEmpty) □ =~ EmptyStr

(MChar) x =~ Char x
    s1 =~ re1 s2 =~ re2

(MApp) s1 ++ s2 =~ App re1 re2
    s1 =~ re1

(MUnionL) s1 =~ Union re1 re2
    s2 =~ re2

(MUnionR) s2 =~ Union re1 re2

(MStar0) □ =~ Star re
    s1 =~ re s2 =~ Star re

(MStarApp) s1 ++ s2 =~ Star re

请注意这些规则不'完全'等同于之前给出的非形式化定义。首先，我们并不需要一个规则来直接表述无字符串匹配 EmptySet；我们做的仅仅是不囊括任何可能导致有字符串被 EmptySet 所匹配的规则。（的确，归纳定义的语法并不'允许'我们表达类似的"否定规则"（negative rule））。

其次，非形式化定义中的 Union 和 Star 各自对应了两个构造子： MUnionL / MUnionR，和 MStar0 / MStarApp。这在逻辑上等价于原始的定义，但在 Coq 中这样更加方便，因为递归出现的 **exp_match** 是作为构造子的直接参数给定的，这在对证据进行归纳时更简

单。（练习 *exp_match_ex1* 和 *exp_match_ex2* 会要求你证明归纳定义中的构造子和从非形式化规则的表述中提炼的规则确实是等价的。）

接下来我们对一些例子使用这些规则：

```
Example reg_exp_ex1 : [1] =~ Char 1.
Proof.
  apply MChar.
Qed.

Example reg_exp_ex2 : [1; 2] =~ App (Char 1) (Char 2).
Proof.
  apply (MApp [1] _ [2]).
  - apply MChar.
  - apply MChar.
Qed.
```

（请注意，后一个例子对字符串 [1] 和 [2] 直接应用了 MApp。由于目标的形式是 [1; 2] 而非 [1] ++ [2]，Coq 并不知道如何分解这个字符串。）

使用 `inversion`，我们还可以证明某些字符串'不'匹配一个正则表达式：

```
Example reg_exp_ex3 : ¬ ([1; 2] =~ Char 1).
Proof.
  intros H. inversion H.
Qed.
```

我们可以定义一些辅助函数来简化正则表达式的书写。函数 reg_exp_of_list 接受一个列表做参数，并构造一个正则表达式来精确地匹配这个列表：

```
Fixpoint reg_exp_of_list {T} (l : list T) :=
  match l with
  | [] ⇒ EmptyStr
  | x :: l' ⇒ App (Char x) (reg_exp_of_list l')
  end.

Example reg_exp_ex4 : [1; 2; 3] =~ reg_exp_of_list [1; 2; 3].
Proof.
  simpl. apply (MApp [1]).
  { apply MChar. }
```

```
  apply (MApp [2]).
  { apply MChar. }
  apply (MApp [3]).
  { apply MChar. }
  apply MEmpty.
Qed.
```

我们还可以证明一些关于 **exp_match** 的性质。比如，下面的引理显示任意一个匹配 re 的字符串 s 也匹配 Star re。

```
Lemma MStar1 :
  ∀ T s (re : reg_exp T) ,
    s =~ re →
    s =~ Star re.
Proof.
  intros T s re H.
  rewrite ← (app_nil_r _ s).
  apply (MStarApp s [] re).
  - apply H.
  - apply MStar0.
Qed.
```

（请注意对 app_nil_r 的使用改变了目标，以此可匹配 MStarApp 所需要的形式。）

练习：3 星, standard (exp_match_ex1) 下面的引理显示从形式化的归纳定义中可以得到本章开始的非形式化匹配规则。

```
Lemma empty_is_empty : ∀ T (s : list T),
  ¬ (s =~ EmptySet).
Proof.
    Admitted.

Lemma MUnion' : ∀ T (s : list T) (re1 re2 : reg_exp T),
  s =~ re1 ∨ s =~ re2 →
  s =~ Union re1 re2.
Proof.
    Admitted.
```

接下来的引理使用了 Poly 一章中出现的 `fold` 函数：如果 ss : **list** (**list** T) 表示一个字符串序列 $s1$, ..., sn，那么 `fold app` ss [] 是将所有字符串连接的结果。

```
Lemma MStar' : ∀ T (ss : list (list T)) (re : reg_exp T),
  (∀ s, In s ss → s =~ re) →
  fold app ss [] =~ Star re.
Proof.
```

Admitted.

□

练习：4 星, standard, optional (reg_exp_of_list_spec) 请证明 reg_exp_of_list 满足以下规范：

```
Lemma reg_exp_of_list_spec : ∀ T (s1 s2 : list T),
  s1 =~ reg_exp_of_list s2 ↔ s1 = s2.
Proof.
```

Admitted.

□

由于 **exp_match** 以递归方式定义，我们可能会发现关于正则表达式的证明常常需要对证据进行归纳。

比如，假设我们想要证明以下显然的结果：如果正则表达式 re 匹配某个字符串 s，那么 s 中的所有元素必在 re 中某处以字符字面量的形式出现。

为了表达这个定理，我们首先定义函数 re_chars 来列举一个正则表达式中出现的所有字符：

```
Fixpoint re_chars {T} (re : reg_exp T) : list T :=
  match re with
  | EmptySet ⇒ []
  | EmptyStr ⇒ []
  | Char x ⇒ [x]
  | App re1 re2 ⇒ re_chars re1 ++ re_chars re2
  | Union re1 re2 ⇒ re_chars re1 ++ re_chars re2
  | Star re ⇒ re_chars re
  end.
```

接下来我们这样陈述此定理：

```
Theorem in_re_match : ∀ T (s : list T) (re : reg_exp T) (x : T),
  s =~ re →
  In x s →
  In x (re_chars re).
Proof.
  intros T s re x Hmatch Hin.
  induction Hmatch
    as [| x'
        | s1 re1 s2 re2 Hmatch1 IH1 Hmatch2 IH2
        | s1 re1 re2 Hmatch IH | re1 s2 re2 Hmatch IH
        | re | s1 s2 re Hmatch1 IH1 Hmatch2 IH2].
  -
    simpl in Hin. destruct Hin.
  -
    simpl. simpl in Hin.
    apply Hin.
  -
    simpl.
```

Something interesting happens in the MApp case. We obtain *two* induction hypotheses: One that applies when x occurs in *s1* (which matches *re1*), and a second one that applies when x occurs in *s2* (which matches *re2*).

```
    simpl. rewrite In_app_iff in *.
    destruct Hin as [Hin | Hin].
    +
      left. apply (IH1 Hin).
    +
      right. apply (IH2 Hin).
  -
    simpl. rewrite In_app_iff.
    left. apply (IH Hin).
  -
    simpl. rewrite In_app_iff.
```

```
    right. apply (IH Hin).
  -
    destruct Hin.
  -
    simpl.
```

我们再次得到了两个归纳假设，它们表明了为什么我们需要对 **exp_match** 的证据而非 *re* 进行归纳：对后者的归纳仅仅提供匹配 *re* 的字符串的归纳假设，却无法允许我们对 In *x s2* 分类进行推理。

```
    rewrite In_app_iff in Hin.
    destruct Hin as [Hin | Hin].
    +
      apply (IH1 Hin).
    +
      apply (IH2 Hin).
Qed.
```

练习：4 星, standard (re_not_empty) 请编写一个递归函数 re_not_empty 用来测试某个正则表达式是否会匹配一些字符串。并证明你的函数是正确的。

```
Fixpoint re_not_empty {T : Type} (re : reg_exp T) : bool
  . Admitted.

Lemma re_not_empty_correct : ∀ T (re : reg_exp T),
  (∃ s, s =~ re) ↔ re_not_empty re = true.
Proof.
    Admitted.
    □
```

13.5.1 *remember* 策略

`induction` 策略让人困惑的一个特点是它会接受任意一个项并尝试归纳，即使这个项不够一般（general）。其副作用是会丢失掉一些信息（类似没有 *eqn*: 从句的 `destruct`），并且使你无法完成证明。比如：

```
Lemma star_app: ∀ T (s1 s2 : list T) (re : reg_exp T),
```

```
  s1 =~ Star re →
  s2 =~ Star re →
  s1 ++ s2 =~ Star re.
Proof.
  intros T s1 s2 re H1.
```

仅仅对 *H1* 反演并不会对处理含有递归的分类有太多帮助。（尝试一下！）因此我们需要对证据进行归纳！下面是一个朴素的尝试：

```
  generalize dependent s2.
  induction H1
    as [|x'|s1 re1 s2' re2 Hmatch1 IH1 Hmatch2 IH2
        |s1 re1 re2 Hmatch IH|re1 s2' re2 Hmatch IH
        |re''|s1 s2' re'' Hmatch1 IH1 Hmatch2 IH2].
```

现在，尽管我们得到了七个分类（正由我们从 **exp_match** 的定义中期待的那样），但 *H1* 还是丢失了一个非常重要的信息：*s1* 事实上匹配了某种形式的 Star *re*。这意味着对于'全部'的七个构造子分类我们都需要给出证明，尽管除了其中两个（MStar0 和 MStarApp）之外，剩下的五个构造子分类是和前提矛盾的。我们仍然可以在一些构造子上继续证明，比如 MEmpty……

```
  -
    simpl. intros s2 H. apply H.
```

……但有一些分类我们却卡住了。比如，对于 MChar 我们需要证明

s2 =~ Char x' -> x' :: s2 =~ Char x',

这显然是不可能完成的。

```
  - intros s2 H. simpl. Abort.
```

问题是，只有当 `Prop` 的假设是完全一般的时候，对其使用 `induction` 的才会起作用，也即，我们需要其所有的参数都是变量，而非更复杂的表达式，比如 Star *re*。

（由此，对证据使用 `induction` 的行为更像是没有 *eqn:* 的 `destruct` 而非 `inversion`。）

解决此问题的一种直接的方式是"手动推广"这个有问题的表达式，即为此引理添加一个显式的等式：

```
Lemma star_app: ∀ T (s1 s2 : list T) (re re' : reg_exp T),
  re' = Star re →
```

```
  s1 =~ re' →
  s2 =~ Star re →
  s1 ++ s2 =~ Star re.
```

我们现在可以直接对证据进行归纳，因为第一个假设的参数已经足够一般，这意味着我们可以通过反演当前上下文中的 *re'* = Star *re* 来消解掉多数分类。

这在 Coq 中是一种常用的技巧，因此 Coq 提供了策略来自动生成这种等式，并且我们也不必改写定理的陈述。

```
Abort.
```

在 Coq 中调用 *remember* e as *x* 策略会（1）替换所有表达式 e 为变量 *x*，（2）在当前上下文中添加一个等式 *x* = e。我们可以这样使用 *remember* 来证明上面的结果：

```
Lemma star_app: ∀ T (s1 s2 : list T) (re : reg_exp T),
  s1 =~ Star re →
  s2 =~ Star re →
  s1 ++ s2 =~ Star re.
Proof.
  intros T s1 s2 re H1.
  remember (Star re) as re'.
```

我们现在有 *Heqre'* : *re'* = Star *re*.

```
  generalize dependent s2.
  induction H1
    as [|x'|s1 re1 s2' re2 Hmatch1 IH1 Hmatch2 IH2
        |s1 re1 re2 Hmatch IH|re1 s2' re2 Hmatch IH
        |re''|s1 s2' re'' Hmatch1 IH1 Hmatch2 IH2].
```

Heqre' 与多数分类相互矛盾，因此我们可以直接结束这些分类。

```
  - discriminate.
  - discriminate.
  - discriminate.
  - discriminate.
  - discriminate.
```

值得注意的分类是 Star。请注意 MStarApp 分类的归纳假设 *IH2* 包含到一个额外的前提 Star *re''* = Star *re*，这是由 *remember* 所添加的等式所产生的。

```
  -
    injection Heqre'. intros Heqre'' s H. apply H.
  -
    injection Heqre'. intros H0.
    intros s2 H1. rewrite ← app_assoc.
    apply MStarApp.
    + apply Hmatch1.
    + apply IH2.
      × rewrite H0. reflexivity.
      × apply H1.
Qed.
```

练习：4 星, standard, optional (exp_match_ex2) 下面的引理 MStar''（以及它的逆，之前的练习题中的 MStar'）显示 **exp_match** 中 Star 的定义等价于前面给出的非形式化定义。

```
Lemma MStar'' : ∀ T (s : list T) (re : reg_exp T),
  s =~ Star re →
  ∃ ss : list (list T),
    s = fold app ss []
    ∧ ∀ s', In s' ss → s' =~ re.
Proof.
    Admitted.
    □
```

练习：5 星, advanced (weak_pumping) 正则表达式中一个非常有趣的定理叫做'泵引理（*Pumping Lemma*）'，非形式化地来讲，它陈述了任意某个足够长的字符串 s 若匹配一个正则表达式 re，则可以被抽取（pumped）——将 s 的某个中间部分重复任意次产生的新字符串仍然匹配 re。（为了简单起见，我们考虑一个比自动机理论课上陈述的定理稍微弱一点的定理。）

我们首先定义什么是"足够长"。由于 Coq 中使用的是构造性逻辑，我们事实上需要计算对于任何一个正则表达式 re 其最小的"可被抽取（pumpability）"长度。

```
Module PUMPING.
```

```
Fixpoint pumping_constant {T} (re : reg_exp T) : nat :=
  match re with
  | EmptySet ⇒ 1
  | EmptyStr ⇒ 1
  | Char _ ⇒ 2
  | App re1 re2 ⇒
      pumping_constant re1 + pumping_constant re2
  | Union re1 re2 ⇒
      pumping_constant re1 + pumping_constant re2
  | Star r ⇒ pumping_constant r
  end.
```

在证明后面的泵引理时，你会发现关于抽取常量的引理十分有用。

```
Lemma pumping_constant_ge_1 :
  ∀ T (re : reg_exp T),
    pumping_constant re ≥ 1.
Proof.
  intros T re. induction re.
  -
    apply le_n.
  -
    apply le_n.
  -
    apply le_S. apply le_n.
  -
    simpl.
    apply le_trans with (n:=pumping_constant re1).
    apply IHre1. apply le_plus_l.
  -
    simpl.
    apply le_trans with (n:=pumping_constant re1).
    apply IHre1. apply le_plus_l.
  -
```

```
    simpl. apply IHre.
Qed.

Lemma pumping_constant_0_false :
  ∀ T (re : reg_exp T),
    pumping_constant re = 0 → False.
Proof.
  intros T re H.
  assert (Hp1 : pumping_constant re ≥ 1).
  { apply pumping_constant_ge_1. }
  inversion Hp1 as [Hp1' | p Hp1' Hp1''].
  - rewrite H in Hp1'. discriminate Hp1'.
  - rewrite H in Hp1''. discriminate Hp1''.
Qed.
```

接下来，定义辅助函数 napp 来重复（连接到它自己）一个字符串特定次数。

```
Fixpoint napp {T} (n : nat) (l : list T) : list T :=
  match n with
  | 0 ⇒ []
  | S n' ⇒ l ++ napp n' l
  end.
```

这个辅助引理在你证明泵引理时也非常有用。

```
Lemma napp_plus: ∀ T (n m : nat) (l : list T),
  napp (n + m) l = napp n l ++ napp m l.
Proof.
  intros T n m l.
  induction n as [|n IHn].
  - reflexivity.
  - simpl. rewrite IHn, app_assoc. reflexivity.
Qed.

Lemma napp_star :
  ∀ T m s1 s2 (re : reg_exp T),
    s1 =~ re → s2 =~ Star re →
    napp m s1 ++ s2 =~ Star re.
```

```
Proof.
  intros T m s1 s2 re Hs1 Hs2.
  induction m.
  - simpl. apply Hs2.
  - simpl. rewrite ← app_assoc.
    apply MStarApp.
    + apply Hs1.
    + apply IHm.
Qed.
```

（弱化的）泵引理是说，如果 $s =\tilde{} \ re$ 且 s 的长度最小是 re 的抽取常数（pumping constant），那么 s 可分割成三个子字符串 $s1 \ ++ \ s2 \ ++ \ s3$，其中 $s2$ 可被重复任意次，其结果同 $s1$ 和 $s3$ 合并后仍然匹配 re。由于 $s2$ 必须为非空字符串，这是一种（构造性的）方式来以我们想要的长度生成匹配 re 的字符串。

```
Lemma weak_pumping : ∀ T (re : reg_exp T) s,
  s =~ re →
  pumping_constant re ≤ length s →
  ∃ s1 s2 s3,
    s = s1 ++ s2 ++ s3 ∧
    s2 ≠ [] ∧
    ∀ m, s1 ++ napp m s2 ++ s3 =~ re.
```

You are to fill in the proof. Several of the lemmas about **le** that were in an optional exercise earlier in this chapter may be useful. `Proof.`

```
  intros T re s Hmatch.
  induction Hmatch
    as [ | x | s1 re1 s2 re2 Hmatch1 IH1 Hmatch2 IH2
       | s1 re1 re2 Hmatch IH | re1 s2 re2 Hmatch IH
       | re | s1 s2 re Hmatch1 IH1 Hmatch2 IH2 ].
  -
    simpl. intros contra. inversion contra.
    Admitted.
    □
```

练习：5 星, advanced, optional (pumping) Now here is the usual version of the pumping lemma. In addition to requiring that $s2 \neq []$, it also requires that length $s1$ + length $s2$ $\leq$ pumping_constant re.

```
Lemma pumping : ∀ T (re : reg_exp T) s,
  s =~ re →
  pumping_constant re ≤ length s →
  ∃ s1 s2 s3,
    s = s1 ++ s2 ++ s3 ∧
    s2 ≠ [] ∧
    length s1 + length s2 ≤ pumping_constant re ∧
    ∀ m, s1 ++ napp m s2 ++ s3 =~ re.
```

You may want to copy your proof of weak_pumping below.

```
Proof.
  intros T re s Hmatch.
  induction Hmatch
    as [ | x | s1 re1 s2 re2 Hmatch1 IH1 Hmatch2 IH2
       | s1 re1 re2 Hmatch IH | re1 s2 re2 Hmatch IH
       | re | s1 s2 re Hmatch1 IH1 Hmatch2 IH2 ].
  -
    simpl. intros contra. inversion contra.
    Admitted.

End PUMPING.
```

□

13.6 案例学习：改进互映

在 Logic 一章中，我们经常需要关联起对布尔值的计算和 Prop 中的陈述，然而进行这样的关联往往会导致冗长的证明。请考虑以下定理的证明：

```
Theorem filter_not_empty_In : ∀ n l,
  filter (fun x ⇒ n =? x) l ≠ [] →
  In n l.
Proof.
  intros n l. induction l as [|m l' IHl'].
```

```
  -
    simpl. intros H. apply H. reflexivity.
  -
    simpl. destruct (n =? m) eqn:H.
    +
      intros _. rewrite eqb_eq in H. rewrite H.
      left. reflexivity.
    +
      intros H'. right. apply IHl'. apply H'.
Qed.
```

在 `destruct` 后的第一个分支中，我们解构 $n =? m$ 后生成的等式显式地使用了 eqb_eq 引理，以此将假设 $n =? m$ 转换为假设 $n = m$；接着使用 `rewrite` 策略和这个假设来完成此分支的证明。

为了简化这样的证明，我们可定义一个归纳命题，用于对 $n =? m$ 产生更好的分类讨论原理。它不会生成类似 $(n =? m) =$ true这样的等式，因为一般来说对证明并不直接有用，其生成的分类讨论原理正是我们所需要的假设: $n = m$。

```
Inductive reflect (P : Prop) : bool → Prop :=
| ReflectT (H : P) : reflect P true
| ReflectF (H : ¬ P) : reflect P false.
```

性质 **reflect** 接受两个参数：一个命题 P 和一个布尔值 b。直观地讲，它陈述了性质 P 在布尔值 b 中所'映现'（也即，等价）：换句话说，P 成立当且仅当 $b = $ true。为了理解这一点，请注意定义，我们能够产生 **reflect** P true 的证据的唯一方式是证明 P 为真并使用 ReflectT 构造子。如果我们反转这个陈述，意味着从 **reflect** P true 的证明中抽取出 P 的证据也是可能的。与此类似，证明 **reflect** P false 的唯一方式是合并 $\neg P$ 的证据和 ReflectF 构造子。

形式化这种直觉并证明 $P \leftrightarrow b = $ true 和 **reflect** P b 这两个表述确实等价是十分容易的。首先是从左到右的蕴含：

```
Theorem iff_reflect : ∀ P b, (P ↔ b = true) → reflect P b.
Proof.
  intros P b H. destruct b.
  - apply ReflectT. rewrite H. reflexivity.
  - apply ReflectF. rewrite H. intros H'. discriminate.
```

```
Qed.
```

Now you prove the right-to-left implication:

练习：2 星, standard, recommended (reflect_iff) `Theorem reflect_iff : ∀ P b, reflect P b → (P ↔ b = true).`

```
Proof.
```

Admitted.

□

使用 **reflect** 而非“当且仅当”连词的好处是，通过解构一个形如 **reflect** P b 的假设或引理，我们可以对 b 进行分类讨论，同时为两个分支（第一个子目标中的 P 和第二个中的 $\neg P$）生成适当的假设。

```
Lemma eqbP : ∀ n m, reflect (n = m) (n =? m).
Proof.
  intros n m. apply iff_reflect. rewrite eqb_eq. reflexivity.
Qed.
```

filter_not_empty_In 的一种更流畅证明如下所示。请注意对 `destruct` 和 `rewrite` 的使用是如何合并成一个 `destruct` 的使用。

（为了更清晰地看到这点，使用 Coq 查看 filter_not_empty_In 的两个证明，并观察在 `destruct` 的第一个分类开始时证明状态的区别。）

```
Theorem filter_not_empty_In' : ∀ n l,
  filter (fun x ⇒ n =? x) l ≠ [] →
  In n l.
Proof.
  intros n l. induction l as [|m l' IHl'].
  -
    simpl. intros H. apply H. reflexivity.
  -
    simpl. destruct (eqbP n m) as [H | H].
    +
      intros _. rewrite H. left. reflexivity.
    +
      intros H'. right. apply IHl'. apply H'.
```

```
Qed.
```

练习：3 星, standard, recommended (eqbP_practice) 使用上面的 eqbP 证明以下定理：

```
Fixpoint count n l :=
  match l with
  | [] ⇒ 0
  | m :: l' ⇒ (if n =? m then 1 else 0) + count n l'
  end.

Theorem eqbP_practice : ∀ n l,
  count n l = 0 → ~(In n l).
Proof.
```

Admitted.

□

这个小例子展示了互映证明可以怎样为我们提供一些便利。在大型的开发中，使用 **reflect** 往往更容易写出清晰和简短的证明脚本。我们将会在后面的章节和*'编程语言基础'*一卷中看到更多的例子。

对 **reflect** 性质的使用已被 *'SSReflect'* 推广开来，这是一个 Coq 程序库，用于形式化一些数学上的重要结果，包括四色定理和法伊特－汤普森定理。 SSReflect 的名字代表着 *'small-scale reflection'*，也即，普遍性地使用互映来简化与布尔值计算有关的证明。

13.7 额外练习

练习：3 星, standard, recommended (nostutter_defn) 写出性质的归纳定义是本课程中你需要的重要技能。请尝试去独立解决以下的练习。

列表连续地重复某元素称为 "百叶窗式" (stutter)。（此概念不同于不包含重复元素：1;4;1 虽然包含重复元素 1，但因其未连续出现，故不是百叶窗式列表）。 **nostutter** mylist 表示 mylist 不是百叶窗式列表。请写出 **nostutter** 的归纳定义。

```
Inductive nostutter {X:Type} : list X → Prop :=


.
```

请确保以下测试成功，但如果你觉得我们建议的证明（在注释中）并不有效，也可随

意更改他们。若你的定义与我们的不同，也可能仍然是正确的，但在这种情况下可能需要不同的证明。（你会注意到建议的证明中使用了一些我们尚未讨论过的策略，这可以让证明适用于不同的 **nostutter** 定义方式。你可以取消注释并直接使用他们，也可以用基础的策略证明这些例子。）

```
Example test_nostutter_1: nostutter [3;1;4;1;5;6].
```

Admitted.

```
Example test_nostutter_2: nostutter (@nil nat).
```

Admitted.

```
Example test_nostutter_3: nostutter [5].
```

Admitted.

```
Example test_nostutter_4: not (nostutter [3;1;1;4]).
```

Admitted.

```
Definition manual_grade_for_nostutter : option (nat×string) := None.
```

□

练习：4 星, advanced (filter_challenge) 让我们证明在 Poly 一章中 filter 的定义匹配某个抽象的规范。可以这样非形式化地描述这个规范：

列表 *l* 是一个 *l1* 和 *l2* 的“顺序合并”（in-order merge），如果它以 *l1* 和 *l2* 中元素的顺序包含 *l1* 和 *l2* 中的所有元素，尽管可能是交替的。比如：

1;4;6;2;3

是

1;6;2

和

4;3.

的顺序合并。

现在，假设我们有集合 X，函数 *test*: X→**bool** 和一个类型为 **list** X 的列表 *l*。接着接设如果 *l* 是 *l1* 和 *l2* 的顺序合并，且 *l1* 中的每个元素满足 *test*，而 *l2* 中没有元素满足 *test*，那么 filter *test l* = *l1*。

请将这段规范翻译为 Coq 中的定理并证明它。（你首先需要定义合并两个列表的含义是什么。请使用归纳关系而非 `Fixpoint` 来完成。）

```
Definition manual_grade_for_filter_challenge : option (nat×string) := None.
```

□

练习：5 星, advanced, optional (filter_challenge_2) 另一种刻画 filter 行为的方式是：在 l 的所有其元素满足 *test* 的子序列中， filter *test* l 是最长的那个。请形式化这个命题并证明它。

练习：4 星, standard, optional (palindromes) 回文是倒序排列与正序排列相同的序列。

- 在 *listX* 上定义一个归纳命题 *pal* 来表达回文的含义。（提示：你需要三个分类。定义应当基于列表的结构；仅仅使用一个构造子，例如

  c : forall l, l = rev l -> pal l

  看起来十分显而易见，但并不会很好的工作。）

- 证明（*pal_app_rev*）

  forall l, pal (l ++ rev l).

- 证明（*pal_rev* that）

  forall l, pal l -> l = rev l.

```
Definition manual_grade_for_pal_pal_app_rev_pal_rev : option (nat×string) := None.
```
□

练习：5 星, standard, optional (palindrome_converse) 由于缺乏证据，反方向的证明要困难许多。使用之前练习中定义的 *pal* 来证明

forall l, l = rev l -> pal l.

练习：4 星, advanced, optional (NoDup) 请回忆一下 Logic 章节中性质 In 的定义，其断言值 x 在列表 l 中至少出现一次：

你的第一个任务是使用 In 来定义命题 *disjoint* X *l1 l2*：仅当列表 *l1* 和 *l2*（元素的类型为 X）不含有相同的元素时其可被证明。

接下来，使用 In 定义归纳命题 *NoDup* X l，其可被证明仅当列表 l （元素类型为 X）的每个元素都不相同。比如，*NoDup* **nat** [1;2;3;4] 和 *NoDup* **bool** [] 是可被证明的，然而 *NoDup* **nat** [1;2;1] 和 *NoDup* **bool** [true;true] 是不行的。

最后，使用 *disjoint*， *NoDup* 和 ++（列表连接）陈述并证明一个或多个有趣的定理。

```
Definition manual_grade_for_NoDup_disjoint_etc : option (nat×string) := None.
```

□

_

练习：4 星, advanced, optional (pigeonhole_principle) 鸽笼原理（Pigeonhole Principle）'是一个关于计数的基本事实：将超过 n 个物体放进 n 个鸽笼，则必有鸽笼包含至少两个物体。与此前诸多情形相似，这一数学事实看似乏味，但其证明手段并不平凡，如下所述：

首先容易证明一个有用的引理。

```
Lemma in_split : ∀ (X:Type) (x:X) (l:list X),
  In x l →
  ∃ l1 l2, l = l1 ++ x :: l2.
Proof.
```

Admitted.

现在请定一个性质 **repeats**，使 **repeats** X l 断言 l 包含至少一个（类型为 X 的）重复的元素。

```
Inductive repeats {X:Type} : list X → Prop :=
```

.

现在，我们这样来形式化鸽笼原理。假设列表 *l2* 表示鸽笼标签的列表，列表 *l1* 表示标签被指定给一个列表里的元素。如果元素的个数多于标签的个数，那么至少有两个元素被指定了同一个标签——也即，列表 *l1* 含有重复元素。

如果使用 *excluded_middle* 假设并展示 In 是可判定的（decidable），即 ∀ x l, (In x l) ∨ ¬ (In x l)，那么这个证明会容易很多。然而，若'不'假设 In 的可判定性也同样可以证明它；在这样的情况下便不必使用 excluded_middle 假设。

```
Theorem pigeonhole_principle: ∀ (X:Type) (l1 l2:list X),
   excluded_middle →
   (∀ x, In x l1 → In x l2) →
   length l2 < length l1 →
```

repeats *l1*.

Proof.

intros *X l1*. induction *l1* as [|*x l1' IHl1'*].

Admitted.

13.7.1 扩展练习：经验证的正则表达式匹配器

我们现在已经定义了正则表达式的匹配关系和多态列表。我们可以使用这些定义来手动地证明给定的正则表达式是否匹配某个字符串，但这并不是一个可以自动地判断是否匹配的程序。

有理由期待，用于构造匹配关系证据的归纳规则可以被翻译为一个递归函数，其在正则表达式上的递归对应于这种关系。然而，定义这样的函数并没有那么直接，由于给定的正则表达式会被 Coq 识别为递归变量，作为结果，Coq 并不会接受这个函数，即使它总是停机。

重度优化的匹配器会将正则表达式翻译为一个状态机，并决定状态机是否接受某个字符串。然而，正则表达式匹配也可以通过一个算法来实现，其仅仅操作字符串和正则表达式，无需定义和维护额外的数据类型，例如状态机。我们将会实现这样的算法，并验证其值与匹配关系是互映的。

我们将要实现的正则表达式匹配器会匹配由 ASCII 字符构成的列表：Require Import Coq.Strings.Ascii.

Definition string := **list** **ascii**.

Coq 标准库中包含了一个不同的 ASCII 字符串的归纳定义。然而，为了应用之前定义的匹配关系，我们在此使用刚刚给出的 ASCII 字符列表作为定义。

我们也可以定义工作在多态列表上的正则表达式匹配器，而非特定于 ASCII 字符列表。我们将要实现的匹配算法需要知道如何对列表中的元素判断相等，因此需要给定一个相等关系测试函数。一般化我们给出的定义、定理和证明有一点枯燥，但是可行的。

正则表达式匹配器的正确性证明会由匹配函数的性质和 match 关系的性质组成，match 关系并不依赖匹配函数。我们将会首先证明后一类性质。他们中的多数将会是很直接的证明，已经被直接给出；少部分关键的引理会留给你来证明。

每个可被证明的 Prop 等价于 **True**。 Lemma provable_equiv_true : $\forall$ (P : Prop), $P \rightarrow$ ($P \leftrightarrow$ **True**).

Proof.

intros.

```
  split.
  - intros. constructor.
  - intros _. apply H.
Qed.
```

其逆可被证明的 Prop 等价于 **False**。 Lemma not_equiv_false : ∀ (P : Prop), ¬P → (P ↔ **False**).

```
Proof.
  intros.
  split.
  - apply H.
  - intros. destruct H0.
Qed.
```

EmptySet 不匹配字符串。 Lemma null_matches_none : ∀ (s : string), (s =~ EmptySet) ↔ **False**.

```
Proof.
  intros.
  apply not_equiv_false.
  unfold not. intros. inversion H.
Qed.
```

EmptyStr 仅匹配空字符串。 Lemma empty_matches_eps : ∀ (s : string), s =~ EmptyStr ↔ s = [ ].

```
Proof.
  split.
  - intros. inversion H. reflexivity.
  - intros. rewrite H. apply MEmpty.
Qed.
```

EmptyStr 不匹配非空字符串。 Lemma empty_nomatch_ne : ∀ (a : **ascii**) s, (a :: s =~ EmptyStr) ↔ **False**.

```
Proof.
  intros.
  apply not_equiv_false.
  unfold not. intros. inversion H.
```

```
Qed.
```

Char a 不匹配不以 a 字符开始的字符串。 Lemma char_nomatch_char :
∀ (*a b* : **ascii**) *s*, *b* ≠ *a* → (*b* :: *s* =~ Char *a* ↔ **False**).

```
Proof.
  intros.
  apply not_equiv_false.
  unfold not.
  intros.
  apply H.
  inversion H0.
  reflexivity.
Qed.
```

如果 Char a 匹配一个非空字符串，那么这个字符串的尾（tail）为空。 Lemma char_eps_suffix : ∀ (*a* : **ascii**) *s*, *a* :: *s* =~ Char *a* ↔ *s* = [ ].

```
Proof.
  split.
  - intros. inversion H. reflexivity.
  - intros. rewrite H. apply MChar.
Qed.
```

App *re0 re1* 匹配字符串 *s* 当且仅当 *s* = *s0* ++ *s1*，其中 *s0* 匹配 *re0* 且 *s1* 匹配 *re1*。

```
Lemma app_exists : ∀ (s : string) re0 re1,
    s =~ App re0 re1 ↔
    ∃ s0 s1 , s = s0 ++ s1 ∧ s0 =~ re0 ∧ s1 =~ re1.
Proof.
  intros.
  split.
  - intros. inversion H. ∃ s1, s2. split.
    × reflexivity.
    × split. apply H3. apply H4.
  - intros [ s0 [ s1 [ Happ [ Hmat0 Hmat1 ] ] ] ].
    rewrite Happ. apply (MApp s0 _ s1 _ Hmat0 Hmat1).
Qed.
```

练习：3 星, standard, optional (app_ne) App *re0 re1* 匹配 a::*s* 当且仅当 *re0* 匹配空字符串且 a::*s* 匹配 *re1* 或 *s*=*s0*++*s1*，其中 a::*s0* 匹配 *re0* 且 *s1* 匹配 *re1*。

尽管这个性质由纯粹的匹配关系构成，它是隐藏在匹配器的设计背后的一个重要观察。因此（1）花一些时间理解它，（2）证明它，并且（3）留心后面你会如何使用它。

```
Lemma app_ne : ∀ (a : ascii) s re0 re1,
  a :: s =~ (App re0 re1) ↔
  ([ ] =~ re0 ∧ a :: s =~ re1) ∨
  ∃ s0 s1, s = s0 ++ s1 ∧ a :: s0 =~ re0 ∧ s1 =~ re1.
Proof.
  Admitted.
```

□

s 匹配 Union *re0 re1* 当且仅当 *s* 匹配 *re0* 或 *s* 匹配 *re1*。

```
Lemma union_disj : ∀ (s : string) re0 re1,
  s =~ Union re0 re1 ↔ s =~ re0 ∨ s =~ re1.
Proof.
  intros. split.
  - intros. inversion H.
    + left. apply H2.
    + right. apply H1.
  - intros [ H | H ].
    + apply MUnionL. apply H.
    + apply MUnionR. apply H.
Qed.
```

练习：3 星, standard, optional (star_ne) a::*s* 匹配 Star *re* 当且仅当 *s* = *s0* ++ *s1*，其中 a::*s0* 匹配 *re* 且 *s1* 匹配 Star *re*。同 app_ne一样，这个观察很重要，因此理解，证明并留意它。

提示：你需要进行归纳。的确是有几个合理的候选 Prop 来进行归纳。但唯一其作用的方式是首先拆分 iff 为两个蕴含式，然后在 a :: *s* =~ Star *re* 的证据上进行归纳来证明其中一个。另一个蕴含式可以无需使用归纳来证明。

为了在正确的性质上归纳，你需要使用 *remember* 策略来重新表述 a :: *s* =~ Star *re*，使其成为一般变量上的 Prop。

```
Lemma star_ne : ∀ (a : ascii) s re,
```

a :: s =~ Star re ↔

∃ $s0$ $s1$, s = $s0$ ++ $s1$ ∧ a :: $s0$ =~ re ∧ $s1$ =~ Star re.

Proof.

Admitted.

□

我们的正则表达式匹配器定义包括两个不动点函数。第一个函数对给定的正则表达式 re 进行求值，结果映射了 re 是否匹配空字符串。这个函数满足以下性质： Definition refl_matches_eps m :=

∀ re : **reg_exp ascii**, **reflect** ([ ] =~ re) (m re).

练习：2 星, standard, optional (match_eps) 完成 match_eps 的定义，其测试给定的正则表达式是否匹配空字符串： Fixpoint match_eps (re: **reg_exp ascii**) : **bool**

. *Admitted.*

□

练习：3 星, standard, optional (match_eps_refl) 现在，请证明 match_eps 确实测试了给定的正则表达式是否匹配空字符串。（提示：你会使用到互映引理 ReflectT 和 ReflectF。） Lemma match_eps_refl : refl_matches_eps *match_eps*.

Proof.

Admitted.

□

我们将会定义其他函数也使用到 match_eps。然而，这些函数的证明中你唯一会用到的 match_eps 的性质是 match_eps_refl。

我们匹配器所进行的关键操作是迭代地构造一个正则表达式生成式的序列。对于字符 a 和正则表达式 re，re 在 a 上的生成式是一个正则表达式，其匹配所有匹配 re 且以 a 开始的字符串的后缀。也即，re' 是 re 在 a 上的一个生成式如果他们满足以下关系：

Definition is_der re (a : **ascii**) re' :=

∀ s, a :: s =~ re ↔ s =~ re'.

函数 d 生成字符串如果对于给定的字符 a 和正则表达式 re，它求值为 re 在 a 上的生成式。也即，d 满足以下关系： Definition derives d := ∀ a re, is_der re a (d a re).

练习：3 星, standard, optional (derive) 请定义 derive 使其生成字符串。一个自然的实现是在某些分类使用 match_eps 来判断正则表达式是否匹配空字符串。 Fixpoint derive

(*a* : **ascii**) (*re* : **reg_exp ascii**) : **reg_exp ascii**

. *Admitted.*

□

derive 函数应当通过以下测试。每个测试都在将被匹配器所求值的表达式和最终被匹配器返回的结果之间确立一种相等关系。每个测试也被添加了它所反映的匹配事实的注解。 `Example c := ascii_of_nat 99.`

`Example d := ascii_of_nat 100.`

"c" =~ EmptySet: `Example test_der0 : match_eps (derive c (EmptySet)) = false.`

`Proof.`

Admitted.

"c" =~ Char c: `Example test_der1 : match_eps (derive c (Char c)) = true.`

`Proof.`

Admitted.

"c" =~ Char d: `Example test_der2 : match_eps (derive c (Char d)) = false.`

`Proof.`

Admitted.

"c" =~ App (Char c) EmptyStr: `Example test_der3 : match_eps (derive c (App (Char c) EmptyStr)) = true.`

`Proof.`

Admitted.

"c" =~ App EmptyStr (Char c): `Example test_der4 : match_eps (derive c (App EmptyStr (Char c))) = true.`

`Proof.`

Admitted.

"c" =~ Star c: `Example test_der5 : match_eps (derive c (Star (Char c))) = true.`

`Proof.`

Admitted.

"cd" =~ App (Char c) (Char d): `Example test_der6 :`
`match_eps (derive d (derive c (App (Char c) (Char d)))) = true.`

`Proof.`

Admitted.

"cd" =~ App (Char d) (Char c): Example test_der7 :
match_eps (*derive* d (*derive* c (App (Char d) (Char c)))) = false.
Proof.
Admitted.

练习：4 星, standard, optional (derive_corr) 请证明 derive 确实总是会生成字符串。

提示：一种证明方法是对 re 归纳，尽管你需要通过归纳和一般化合适的项来仔细选择要证明的性质。

提示：如果你定义的 derive 对某个正则表达式 re 使用了 match_eps，那么可对 re 应用 match_eps_refl，接着对结果解构并生成分类，其中你可以假设 re 匹配或不匹配空字符串。

提示：通过使用之前证明过的引理可以帮助一点你的工作。特别是，在证明归纳的许多分类时，通过之前的引理，你可以用一个复杂的正则表达式（比如， $s =\tilde{}$ Union *re0 re1*）来重写命题，得到一个简单正则表达式上的命题构成的布尔表达式（比如，$s =\tilde{}$ *re0* $\vee$ $s =\tilde{}$ *re1*）。你可以使用 intro 和 destruct 来对这些命题进行推理。 Lemma derive_corr : derives *derive*.
Proof.
Admitted.
□

我们将会使用 derive 来定义正则表达式匹配器。然而，在匹配器的性质的证明中你唯一会用到的 derive 的性质是 derive_corr。

函数 m 匹配正则表达式如果对给定的字符串 s 和正则表达式 re，它求值的结果映射了 s 是否被 re 匹配。也即，m 满足以下性质： Definition matches_regex m : Prop :=
$\forall$ (s : string) re, **reflect** (s =~ re) (m s re).

练习：2 星, standard, optional (regex_match) 完成 regex_match 的定义，使其可以匹配正则表达式。 Fixpoint regex_match (s : string) (re : **reg_exp ascii**) : **bool**
. *Admitted.*
□

练习：3 星, standard, optional (regex_refl) 最后，证明 regex_match 确实可以匹配正则表达式。

提示：如果你定义的 regex_match 对正则表达式 re 使用了 match_eps，那么可对 re 应

用 match_eps_refl，接着对结果解构并生成分类，其中你可以假设 re 匹配或不匹配空字符串。

提示：如果你定义的 regex_match 对字符 x 和正则表达式 re 使用了 derive，那么可对 x 和 re 应用 derive_corr，以此证明 $x :: s =\tilde{} re$ 当给定 $s =\tilde{}$ derive x re 时，反之亦然。

Theorem regex_refl : matches_regex *regex_match*.

Proof.

Admitted.

□

Chapter 14

Library LF.Logic

14.1 Logic: Coq 中的逻辑系统

```
Set Warnings "-notation-overridden,-parsing".
From LF Require Export Tactics.
```

我们已经见过很多对事实的断言（即'命题'）以及如何用证据展示其正确性（即'证明'）的例子了。特别是，我们证明了大量的'相等性命题'（*e1* = *e2*）、蕴含式（$P \to Q$）和量化命题（$\forall\ x,\ P\ x$）。在本章中，我们将会看到如何用 Coq 解决类似形式的逻辑推理。

在深入细节之前，我们先来探讨一下 Coq 中数学表达式的地位。回忆一下，Coq 是一门拥有'类型'的语言，也就是说，一切有意义的表达式都具有一个相应的类型。逻辑表达也不例外，我们试图在 Coq 中证明的一切语句都有名为 `Prop` 的类型，即'命题类型'。我们可以用 `Check` 指令来查看：

```
Check 3 = 3 : Prop.

Check ∀ n m : nat, n + m = m + n : Prop.
```

注意：'所有'语法形式良好的命题，无论是否为真，其类型均为 `Prop`。

简单来说，'是'一个命题与该命题'可以证明'是两回事。

```
Check 2 = 2 : Prop.

Check 3 = 2 : Prop.

Check ∀ n : nat, n = 2 : Prop.
```

除了拥有类型之外，命题还是'一等的（*First-Class*）'实体，即在 Coq 的世界中，我们可以像操作其它实体那样操作命题。

到目前为止，我们已经知道命题可以出现在 Theorem（还有 Lemma 以及 Example）的声明中了。

```
Theorem plus_2_2_is_4 :
  2 + 2 = 4.
Proof. reflexivity. Qed.
```

不过命题还可以用在其它地方。例如，我们可以用 Definition 为命题取名，就像为其它表达式取名一样。

```
Definition plus_claim : Prop := 2 + 2 = 4.
Check plus_claim : Prop.
```

之后我们可以在任何需要此命题的地方使用它们名字——例如，作为一个 Theorem 声明中的断言：

```
Theorem plus_claim_is_true :
  plus_claim.
Proof. reflexivity. Qed.
```

我们也可以写出'参数化'的命题 – 也就是一个接受某些类型的参数，然后返回一个命题的函数。

例如，以下函数接受某个数字，返回一个命题断言该数字等于 3：

```
Definition is_three (n : nat) : Prop :=
  n = 3.
Check is_three : nat → Prop.
```

在 Coq 中，返回命题的函数可以说是定义了其参数的'性质'。例如，以下（多态的）性质定义了常见的'单射函数'的概念。

```
Definition injective {A B} (f : A → B) :=
  ∀ x y : A, f x = f y → x = y.

Lemma succ_inj : injective S.
Proof.
  intros n m H. injection H as H1. apply H1.
Qed.
```

相等关系运算符 = 也是一个返回 Prop 的函数。

表达式 $n = m$ 只是 **eq** n m 的语法糖（它使用 Notation 机制定义在 Coq 标准库中）。由于 **eq** 可被用于任何类型的元素，因此它也是多态的：

```
Check @eq : ∀ A : Type, A → A → Prop.
```

（注意我们写的是 @**eq** 而非 **eq**：**eq** 的类型参数 A 是隐式声明的，因此我们需要关掉隐式参数的类型推断以便看到 **eq** 的完整类型。）

14.2 逻辑联结词

14.2.1 合取

命题 A 与 B 的'合取'（即'逻辑与'）写作 A ∧ B，表示一个 A 与 B 均为真的断言。

```
Example and_example : 3 + 4 = 7 ∧ 2 × 2 = 4.
```

证明合取的命题通常使用 `split` 策略。它会分别为语句的两部分生成两个子目标：

```
Proof.
  split.
  - reflexivity.
  - reflexivity.
Qed.
```

对于任意命题 A 和 B，如果我们假设 A 为真且 B 为真，那么就能得出 A ∧ B 也为真的结论。

```
Lemma and_intro : ∀ A B : Prop, A → B → A ∧ B.
Proof.
  intros A B HA HB. split.
  - apply HA.
  - apply HB.
Qed.
```

由于按照前提对某个目标应用定理会产生与该定理的前提一样多的子目标。因此我们可以应用 and_intro 来达到和 `split` 一样的效果。

```
Example and_example' : 3 + 4 = 7 ∧ 2 × 2 = 4.
Proof.
  apply and_intro.
  - reflexivity.
  - reflexivity.
```

```
Qed.
```

练习：2 星, standard (and_exercise) `Example and_exercise :`

```
  ∀ n m : nat, n + m = 0 → n = 0 ∧ m = 0.
Proof.
```

Admitted.

□

以上就是证明合取语句的方法。要反过来使用，即'使用'合取前提来帮助证明时，我们会采用 `destruct` 策略。

如果当前证明上下文中存在形如 A ∧ B 的前提 H，那么 `destruct` H `as` $[HA\ HB]$ 将会从上下文中移除 H 并增加 HA 和 HB 两个新的前提，前者断言 A 为真，而后者断言 B 为真。

```
Lemma and_example2 :
  ∀ n m : nat, n = 0 ∧ m = 0 → n + m = 0.
Proof.
  intros n m H.
  destruct H as [Hn Hm] eqn:HE.
  rewrite Hn. rewrite Hm.
  reflexivity.
Qed.
```

和往常一样，我们也可以在引入 H 的同时对其进行解构，而不必先引入然后再解构：

```
Lemma and_example2' :
  ∀ n m : nat, n = 0 ∧ m = 0 → n + m = 0.
Proof.
  intros n m [Hn Hm].
  rewrite Hn. rewrite Hm.
  reflexivity.
Qed.
```

为什么我们要麻烦地将 $n = 0$ 和 $m = 0$ 这两个前提放一条合取语句中呢？完全可以用两条独立的前提来陈述此定理啊：

```
Lemma and_example2'' :
```

```
  ∀ n m : nat, n = 0 → m = 0 → n + m = 0.
Proof.
  intros n m Hn Hm.
  rewrite Hn. rewrite Hm.
  reflexivity.
Qed.
```

就此定理而言，两种方式都可以。不过理解如何证明合取前提非常重要，因为合取语句通常会在证明的中间步骤中出现，特别是在做大型开发的时候。下面是个简单的例子：

```
Lemma and_example3 :
  ∀ n m : nat, n + m = 0 → n × m = 0.
Proof.
  intros n m H.
  assert (H' : n = 0 ∧ m = 0).
  { apply and_exercise. apply H. }
  destruct H' as [Hn Hm] eqn:HE.
  rewrite Hn. reflexivity.
Qed.
```

另一种经常遇到合取语句的场景是，我们已经知道了 A ∧ B，但在某些上下文中只需要 A 或者 B。此时我们可以用 `destruct` 进行解构（或许是作为 `intros` 的一部分）并用下划线模式 `_` 来丢弃不需要的合取分式。

```
Lemma proj1 : ∀ P Q : Prop,
  P ∧ Q → P.
Proof.
  intros P Q HPQ.
  destruct HPQ as [HP _].
  apply HP. Qed.
```

练习：1 星, standard, optional (proj2) `Lemma proj2 : ∀ P Q : Prop, P ∧ Q → Q.`

```
Proof.
  Admitted.
```

□

最后，我们有时需要重新排列合取语句的顺序，或者对多部分的合取语句进行分组。此时使用下面的交换律和结合律会很方便。

```
Theorem and_commut : ∀ P Q : Prop,
  P ∧ Q → Q ∧ P.
Proof.
  intros P Q [HP HQ].
  split.
    - apply HQ.
    - apply HP. Qed.
```

练习：2 星, standard (and_assoc) （在以下结合律的证明中，注意'嵌套'的 `intros` 模式是如何将 $H : P \wedge (Q \wedge \mathbf{R})$ 拆分为 $HP : P$、$HQ : Q$ 和 $HR : \mathbf{R}$ 的。请从那里开始完成证明。）

```
Theorem and_assoc : ∀ P Q R : Prop,
  P ∧ (Q ∧ R) → (P ∧ Q) ∧ R.
Proof.
  intros P Q R [HP [HQ HR]].
    Admitted.
```

□

顺便一提，中缀记法 ∧ 只是 **and** A B 的语法糖而已； **and** 是 Coq 中将两个命题合并成一个命题的运算符。

```
Check and : Prop → Prop → Prop.
```

14.2.2 析取

另一个重要的联结词是'析取'，即两个命题的'逻辑或'：若 A 或 B 二者之一为真，则 A ∨ B 为真。（这种中缀记法表示 **or** A B，其中 **or** : Prop → Prop → Prop。）

为了在证明中使用析取前提，我们需要分类讨论（它与 **nat** 之类的数据类型一样，都可以显式地通过 `destruct` 或隐式地通过 `intros` 模式来拆分：

```
Lemma eq_mult_0 :
  ∀ n m : nat, n = 0 ∨ m = 0 → n × m = 0.
Proof.
  intros n m [Hn | Hm].
```

```
  -
    rewrite Hn. reflexivity.
  -
    rewrite Hm. rewrite ← mult_n_O.
    reflexivity.
Qed.
```

相应地，要证明某个析取命题成立，只需证明其任意一边的命题成立就够了。我们可以用 `left` 和 `right` 策略来选取命题。顾名思义，`left` 会选取待析取证命题的左边，而 `right` 则会选取它的右边。下面是一种平凡的用法...

```
Lemma or_intro_l : ∀ A B : Prop, A → A ∨ B.
Proof.
  intros A B HA.
  left.
  apply HA.
Qed.
```

...而这个更有趣的例子则同时需要 `left` 和 `right`：

```
Lemma zero_or_succ :
  ∀ n : nat, n = 0 ∨ n = S (pred n).
Proof.
  intros [|n'].
  - left. reflexivity.
  - right. reflexivity.
Qed.
```

练习：1 星, standard (mult_eq_0)

```
Lemma mult_eq_0 :
  ∀ n m, n × m = 0 → n = 0 ∨ m = 0.
Proof.
    Admitted.
```

□

练习：1 星, standard (or_commut)

```
Theorem or_commut : ∀ P Q : Prop,
  P ∨ Q → Q ∨ P.
```

Proof.

Admitted.

□

14.2.3 假命题与否定

目前为止，我们主要都在证明某些东西是'真'的：加法满足结合律，列表的连接满足结合律，等等。当然，我们也关心'否定'的结果，即证明某些给定的命题'不是'真的。在 Coq 中，这样的否定语句使用逻辑否定运算符 ¬ 来表达。

为了理解否定背后的原理，我们需要回想一下Tactics一章中的'爆炸原理'。爆炸原理断言，当我们假设了矛盾存在时，就能推出任何命题。遵循这一直觉，我们可以可以将 ¬ P（即非 P）定义为 $\forall\ Q,\ P \to Q$。

不过 Coq 选择了稍有些不同（但等价）的做法，它将 ¬ P 定义为 $P \to$ **False**，而 **False** 是在标准库中特别定义的矛盾性命题。

Module MyNot.

Definition not (P:Prop) := $P \to$ **False**.

Notation "~ x" := (not x) : *type_scope*.

Check not : Prop → Prop.

End MyNot.

由于 **False** 是个矛盾性命题，因此爆炸原理对它也适用。如果我们让 **False** 进入到了证明的上下文中，可以对它使用 destruct 来完成任何待证目标。

Theorem ex_falso_quodlibet : ∀ (P:Prop),
  False → P.
Proof.
  intros P *contra*.
  destruct *contra*. Qed.

拉丁文 *'ex falso quodlibet'* 的字面意思是"从谬误出发，你能够证明任何你想要的"，这也是爆炸原理的另一个广为人知的名字。

练习：2 星, standard, optional (not_implies_our_not) 证明 Coq 对否定的定义蕴含前面提到的直觉上的定义：

```
Fact not_implies_our_not : ∀ (P:Prop),
  ¬ P → (∀ (Q:Prop), P → Q).
Proof.
```

Admitted.

□

不等性是十分常见的否定句的例子，，它有一个特别的记法 $x \neq y$：

```
Notation "x <> y" := (~(x = y)).
```

我们可以用 not 来陈述 0 和 1 是不同的 **nat** 元素：

```
Theorem zero_not_one : 0 ≠ 1.
Proof.
```

性质 $0 \neq 1$ 就是 ~(0 = 1)，即 not (0 = 1)，它会展开为 (0 = 1) → **False**。（这里显式地用 unfold not 展示了这一点，不过一般可以忽略。

```
  unfold not.
```

要证明不等性，我们可以反过来假设其相等...

```
  intros contra.
```

... 然后从中推出矛盾。在这里，等式 O = S O 与构造子 O 和 S 的不交性相矛盾，因此用 discriminate 就能解决它。

```
  discriminate contra.
Qed.
```

为了习惯用 Coq 处理否定命题，我们需要一些练习。即便你十分清楚为什么某个否定命题成立，但能让 Coq 一下就理解则需要点小技巧。以下常见事实的证明留给你热身。

```
Theorem not_False :
  ¬ False.
Proof.
  unfold not. intros H. destruct H. Qed.

Theorem contradiction_implies_anything : ∀ P Q : Prop,
  (P ∧ ¬P) → Q.
Proof.
  intros P Q [HP HNA]. unfold not in HNA.
  apply HNA in HP. destruct HP. Qed.

Theorem double_neg : ∀ P : Prop,
```

```
  P → ~~P.
Proof.
  intros P H. unfold not. intros G. apply G. apply H. Qed.
```

练习：2 星, advanced (double_neg_inf) 请写出 double_neg 的非形式化证明：

'定理'：对于任何命题 P 而言，P 蕴含 $\sim\sim P$。

```
Definition manual_grade_for_double_neg_inf : option (nat×string) := None.
```

□

练习：2 星, standard, recommended (contrapositive) Theorem contrapositive : ∀ (P Q : Prop),

```
  (P → Q) → (¬Q → ¬P).
Proof.
```

Admitted.

□

练习：1 星, standard (not_both_true_and_false) Theorem not_both_true_and_false : ∀ P : Prop,

```
  ¬ (P ∧ ¬P).
Proof.
```

Admitted.

□

练习：1 星, advanced (informal_not_PNP) 请写出 ∀ P : Prop, ~(P ∧ ¬P) 的非形式化证明。

```
Definition manual_grade_for_informal_not_PNP : option (nat×string) := None.
```

□

由于不等性包含一个否定，因此在能够熟练地使用它前还需要一些练习。这里有个有用的技巧：如果你需要证明某个目标不可能时（例如当前的目标陈述为 false = true），请使用 ex_falso_quodlibet 将该目标转换为 **False**。如果在当前上下文中存在形如 ¬P 的假设（特别是形如 $x \neq y$ 的假设），那么此技巧会让这些假设用起来更容易些。

```
Theorem not_true_is_false : ∀ b : bool,
```

```
  b ≠ true → b = false.
Proof.
  intros b H.
  destruct b eqn:HE.
  -
    unfold not in H.
    apply ex_falso_quodlibet.
    apply H. reflexivity.
  -
    reflexivity.
Qed.
```

由于用 ex_falso_quodlibet 推理十分常用，因此 Coq 提供了内建的策略 *exfalso*。

```
Theorem not_true_is_false' : ∀ b : bool,
  b ≠ true → b = false.
Proof.
  intros [] H.    -
    unfold not in H.
    exfalso.        apply H. reflexivity.
  - reflexivity.
Qed.
```

14.2.4 真值

除 **False** 外，Coq 的标准库中还定义了 **True**，一个明显真的命题。为了证明它，我们使用了预定义的常量 I : **True**：

```
Lemma True_is_true : True.
Proof. apply I. Qed.
```

与经常使用的 **False** 不同，**True** 很少使用，因为它作为证明目标来说过于平凡，而作为前提又不携带任何有用的信息。

然而在使用条件从句定义复杂的 Prop，或者作为高阶 Prop 的参数时，它还是挺有用的。之后我们会看到一些例子。

14.2.5 逻辑等价

联结词“当且仅当”用起来十分方便，它是两个蕴含式的合取，断言了两个命题拥有同样的真值。

```
Module MyIff.

Definition iff (P Q : Prop) := (P → Q) ∧ (Q → P).

Notation "P <-> Q" := (iff P Q)
                      (at level 95, no associativity)
                      : type_scope.

End MyIff.

Theorem iff_sym : ∀ P Q : Prop,
  (P ↔ Q) → (Q ↔ P).
Proof.
  intros P Q [HAB HBA].
  split.
  - apply HBA.
  - apply HAB. Qed.

Lemma not_true_iff_false : ∀ b,
  b ≠ true ↔ b = false.
Proof.
  intros b. split.
  - apply not_true_is_false.
  -
    intros H. rewrite H. intros H'. discriminate H'.
Qed.
```

练习：1 星, standard, optional (iff_properties) 参照上面对 ↔ 对称性（iff_sym）的证明，请证明它同时也有自反性和传递性。

```
Theorem iff_refl : ∀ P : Prop,
  P ↔ P.
Proof.
    Admitted.
```

```
Theorem iff_trans : ∀ P Q R : Prop,
  (P ↔ Q) → (Q ↔ R) → (P ↔ R).
Proof.
    Admitted.
    □
```

练习：3 星, standard (or_distributes_over_and) `Theorem or_distributes_over_and : ∀ P Q R : Prop, P ∨ (Q ∧ R) ↔ (P ∨ Q) ∧ (P ∨ R).`

```
Proof.
    Admitted.
    □
```

14.2.6 广集与逻辑等价

Coq 的某些策略会特殊对待 iff 语句，以此来避免操作某些底层的证明状态。特别来说，`rewrite` 和 `reflexivity` 不仅可以用于相等关系，还可用于 iff 语句。为了开启此行为，我们需要导入 Coq 库来支持它：

```
From Coq Require Import Setoids.Setoid.
```

“广集（Setoid）”指配备了等价关系的集合，即满足自反性、对称性和传递性的关系。当一个集合中的两个元素在这种关系上等价时，可以用 `rewrite` 将其中一个元素替换为另一个。我们已经在 Coq 中见过相等性关系 $=$ 了：当 $x = y$ 时，我们可以用 `rewrite` 将 x 替换为 y，反之亦可。

同样，逻辑等价关系 $\leftrightarrow$ 也满足自反性、对称性和传递性，因此我们可以用它将替换命题中的一部分替换为另一部分：若 $P \leftrightarrow Q$，那么我们可以用 `rewrite` 将 P 替换为 Q，反之亦可。

下面是一个简单的例子，它展示了这些策略如何使用 iff。首先，我们来证明一些基本的 iff 等价关系命题...

```
Lemma mult_0 : ∀ n m, n × m = 0 ↔ n = 0 ∨ m = 0.
Proof.
  split.
  - apply mult_eq_0.
  - apply eq_mult_0.
```

```
Qed.

Lemma or_assoc :
  ∀ P Q R : Prop, P ∨ (Q ∨ R) ↔ (P ∨ Q) ∨ R.
Proof.
  intros P Q R. split.
  - intros [H | [H | H]].
    + left. left. apply H.
    + left. right. apply H.
    + right. apply H.
  - intros [[H | H] | H].
    + left. apply H.
    + right. left. apply H.
    + right. right. apply H.
Qed.
```

现在我们可以用这些事实配合 `rewrite` 与 `reflexivity` 对涉及等价关系的陈述给出流畅的证明了。以下是之前 mult_0 包含三个变量的版本：

```
Lemma mult_0_3 :
  ∀ n m p, n × m × p = 0 ↔ n = 0 ∨ m = 0 ∨ p = 0.
Proof.
  intros n m p.
  rewrite mult_0. rewrite mult_0. rewrite or_assoc.
  reflexivity.
Qed.
```

`apply` 策略也可以用在 ↔ 上。当给定一个等价关系命题作为 `apply` 的参数时，它会试图猜出正确的方向。

```
Lemma apply_iff_example :
  ∀ n m : nat, n × m = 0 → n = 0 ∨ m = 0.
Proof.
  intros n m H. apply mult_0. apply H.
Qed.
```

14.2.7 存在量化

'存在量化'也是十分重要的逻辑联结词。我们说存在某个类型为 T 的 x，使得某些性质 P 对于 x 成立，写作 $\exists\, x : T,\, P$。和 $\forall$ 一样，如果 Coq 能从上下文中推断出 x 的类型，那么类型标注 $: T$ 就可以省略。

为了证明形如 $\exists\, x,\, P$ 的语句，我们必须证明 P 对于某些特定的 x 成立，这些特定的 x 被称作存在性的'例证'。证明分为两步：首先，我们调用 $\exists\, t$ 策略向 Coq 指出已经知道了使 P 成立的例证 t，然后证明将所有出现的 x 替换成 t 的命题 P。

```
Definition even x := ∃ n : nat, x = double n.

Lemma four_is_even : even 4.
Proof.
  unfold even. ∃ 2. reflexivity.
Qed.
```

反之，如果我们的的上下文中有形如 $\exists\, x,\, P$ 的存在前提，可以将其解构得到一个例证 x 和一个陈述 P 对于 x 成立的前提。

```
Theorem exists_example_2 : ∀ n,
  (∃ m, n = 4 + m) →
  (∃ o, n = 2 + o).
Proof.
  intros n [m Hm].   ∃ (2 + m).
  apply Hm. Qed.
```

练习：1 星, standard, recommended (dist_not_exists) 请证明"P 对所有 x 成立"蕴含"不存在 x 使 P 不成立。"（提示：`destruct H as [x E]` 可以用于存在假设！）

```
Theorem dist_not_exists : ∀ (X:Type) (P : X → Prop),
  (∀ x, P x) → ¬ (∃ x, ¬ P x).
Proof.
    Admitted.
```

□

练习：2 星, standard (dist_exists_or) 请证明存在量化对析取满足分配律。

```
Theorem dist_exists_or : ∀ (X:Type) (P Q : X → Prop),
```

```
  (∃ x, P x ∨ Q x) ↔ (∃ x, P x) ∨ (∃ x, Q x).
Proof.
    Admitted.
    □
```

14.3 使用命题编程

我们学过的逻辑联结词为我们提供了丰富的用简单命题构造复杂命题的词汇。为了说明，我们来看一下如何表达“元素 x 出现在列表 l 中”这一断言。注意此性质有着简单的递归结构：

- 若 l 为空列表，则 x 无法在其中出现，因此性质“x 出现在 l 中” 为假。
- 否则，若 l 的形式为 $x' :: l'$，此时 x 是否出现在 l 中，取决于它是否等于 x' 或出现在 l' 中。

我们可以将此定义直接翻译成递归函数，它接受一个元素和一个列表，返回一个命题!:

```
Fixpoint In {A : Type} (x : A) (l : list A) : Prop :=
  match l with
  | [] ⇒ False
  | x' :: l' ⇒ x' = x ∨ In x l'
  end.
```

当 In 应用于具体的列表时，它会被展开为一系列具体的析取式。

```
Example In_example_1 : In 4 [1; 2; 3; 4; 5].
Proof.
  simpl. right. right. right. left. reflexivity.
Qed.

Example In_example_2 :
  ∀ n, In n [2; 4] →
  ∃ n', n = 2 × n'.
Proof.
  simpl.
```

```
  intros n [H | [H | []]].
  - ∃ 1. rewrite ← H. reflexivity.
  - ∃ 2. rewrite ← H. reflexivity.
Qed.
```

（注意我们用空模式'无视'了最后一种情况。）

我们也可证明关于 In 的更一般，更高阶的引理。

注意，首先 In 会被应用到一个变量上，只有当我们对它进行分类讨论时，它才会被展开：

```
Lemma In_map :
  ∀ (A B : Type) (f : A → B) (l : list A) (x : A),
    In x l →
    In (f x) (map f l).
Proof.
  intros A B f l x.
  induction l as [|x' l' IHl'].
  -
    simpl. intros [].
  -
    simpl. intros [H | H].
    + rewrite H. left. reflexivity.
    + right. apply IHl'. apply H.
Qed.
```

虽然递归定义命题在某些情况下会很方便，但这种方式也有其劣势。特别是，这类命题会受到 Coq 对递归函数要求的限制，例如，在 Coq 中递归函数必须是"明显会终止"的。在下一章中，我们会了解如何'归纳地'定义命题，这是一种与之不同的技巧，有着其独特的优势和限制。

练习：3 星, standard (In_map_iff)

```
Lemma In_map_iff :
  ∀ (A B : Type) (f : A → B) (l : list A) (y : B),
    In y (map f l) ↔
    ∃ x, f x = y ∧ In x l.
Proof.
  intros A B f l y. split.
```

Admitted.

□

练习：2 星, standard (In_app_iff) Lemma In_app_iff : ∀ *A l l' (a:A)*,
In *a* (*l*++*l'*) ↔ In *a l* ∨ In *a l'*.

Proof.
intros *A l*. induction *l* as [|*a' l' IH*].

Admitted.

□

练习：3 星, standard, recommended (All) 回忆一下，返回命题的函数可以视作对其参数'性质'的定义。例如，若 *P* 的类型为 **nat** → Prop，那么 *P n* 就陈述了性质 *P* 对 *n* 成立。

以 In 作为参考，请写出递归函数 All，它陈述某个 *P* 对列表 *l* 中的所有元素成立。为了确定你的定义是正确的，请在下方证明 All_In 引理。（当然，你的定义'不应该'为了通过测试就把 All_In 的左边复述一遍。）

Fixpoint All {*T* : Type} (*P* : *T* → Prop) (*l* : **list** *T*) : Prop
. *Admitted.*

Lemma All_In :
∀ *T* (*P* : *T* → Prop) (*l* : **list** *T*),
(∀ *x*, In *x l* → *P x*) ↔
All P l.

Proof.
Admitted.

□

练习：3 星, standard (combine_odd_even) 完成以下 combine_odd_even 函数的定义。它接受两个对数字成立的性质 *Podd* 与 *Peven*，返回性质 *P* 使得当 *n* 为奇数时 *P n* 等价于 *Podd n*，否则等价于 *Peven n*。

Definition combine_odd_even (*Podd Peven* : **nat** → Prop) : **nat** → Prop
. *Admitted.*

为了测试你的定义，请证明以下事实：

```
Theorem combine_odd_even_intro :
  ∀ (Podd Peven : nat → Prop) (n : nat),
    (oddb n = true → Podd n) →
    (oddb n = false → Peven n) →
    combine_odd_even Podd Peven n.
Proof.
    Admitted.

Theorem combine_odd_even_elim_odd :
  ∀ (Podd Peven : nat → Prop) (n : nat),
    combine_odd_even Podd Peven n →
    oddb n = true →
    Podd n.
Proof.
    Admitted.

Theorem combine_odd_even_elim_even :
  ∀ (Podd Peven : nat → Prop) (n : nat),
    combine_odd_even Podd Peven n →
    oddb n = false →
    Peven n.
Proof.
    Admitted.
    □
```

14.4 对参数应用定理

Coq 不同于其它证明助理（如 ACL2 和 Isabelle）的一个特性是，它将'证明'本身也作为一等对象。

关于这一点有很多地方值得着墨，不过了解所有的细节对于使用 Coq 来说不是必须的。本节点到为止，深入的探讨参见 ProofObjects 和 IndPrinciples。

我们已经知道 `Check` 命令可以用来显式表达式的类型了，不过它还可以用来查找某个标识符所指代的定理。

```
Check plus_comm : ∀ n m : nat, n + m = m + n.
```

在检查定理 plus_comm 的'陈述'时，Coq 使用了与检查某项的'类型'一样的方式（如果我们保留以冒号开始的部分，那么它会被打印出来）。这是为什么？

原因在于标识符 plus_comm 其实指代的是被称作'证明对象'的数据结构，它表示在命题 $\forall\ n\ m : \textbf{nat}, n + m = m + n$ 的真实性上建立的逻辑推导。此对象的类型'就是'其所证命题的陈述。

从直觉上来说，这很有道理，因为对定理的陈述说明了该定理可用来做什么，正如可计算对象的类型告诉了我们可以对它做什么。例如，若我们有一个类型为 **nat** → **nat** → **nat** 的项，就可以给它两个 **nat** 作为参数并得到一个 **nat**。类似地，如果我们有一个类型为 $n = m \to n + n = m + m$ 的对象，就能为它提供一个类型为 $n = m$ 的"参数"并推导出 $n + n = m + m$。

从操作上来说，这种类比可以更进一步：由于定理可以作为函数，被应用到对应类型的前提上，因此我们可以直接产生结论而不必在途中使用断言。例如，假设我们想要证明以下结论：

```
Lemma plus_comm3 :
  ∀ x y z, x + (y + z) = (z + y) + x.
```

乍看起来，我们似乎可以用 plus_comm 改写两次使两边匹配来证明它。然而问题是，第二次 `rewrite` 会抵消第一次的效果。

```
Proof.
  intros x y z.
  rewrite plus_comm.
  rewrite plus_comm.
Abort.
```

我们之前在 `Induction` 一章中见过类似的问题，绕过它的一种简单方法是使用 `assert` 导出 plus_comm 的特殊版本，这样我们就能用它按照预期来改写。

```
Lemma plus_comm3_take2 :
  ∀ x y z, x + (y + z) = (z + y) + x.
Proof.
  intros x y z.
  rewrite plus_comm.
  assert (H : y + z = z + y).
  { rewrite plus_comm. reflexivity. }
  rewrite H.
```

```
  reflexivity.
Qed.
```

一种更优雅的方式是直接把我们想要实例化的参数应用到 plus_comm 上，就像我们将一个多态函数应用到类型参数上那样。

```
Lemma plus_comm3_take3 :
  ∀ x y z, x + (y + z) = (z + y) + x.
Proof.
  intros x y z.
  rewrite plus_comm.
  rewrite (plus_comm y z).
  reflexivity.
Qed.
```

我们来看看另一个像函数那样使用定理或引理的例子。以下定理说明：任何包含元素的列表 l 一定非空。

```
Theorem in_not_nil :
  ∀ A (x : A) (l : list A), In x l → l ≠ [].
Proof.
  intros A x l H. unfold not. intro Hl. destruct l eqn:HE.
  - simpl in H. destruct H.
  - discriminate Hl.
Qed.
```

有趣的地方是一个量化的变量（x）没有出现在结论（$l \neq []$）中。

我们可以用此引理来证明 x 为 42 的特殊情况。直接用 apply in_not_nil 会失败，因为它无法推出 x 的值。

```
Lemma in_not_nil_42 :
  ∀ l : list nat, In 42 l → l ≠ [].
Proof.
  intros l H.
  Fail apply in_not_nil.
Abort.
```

有一些方法可以绕开它：

Use apply ... with ... Lemma in_not_nil_42_take2 :
  ∀ l : **list** **nat**, In 42 l → $l \neq$ [].
Proof.
  intros l H.
  apply in_not_nil with ($x := 42$).
  apply H.
Qed.

Use apply ... in ... Lemma in_not_nil_42_take3 :
  ∀ l : **list** **nat**, In 42 l → $l \neq$ [].
Proof.
  intros l H.
  apply in_not_nil in H.
  apply H.
Qed.

显式地对 x 的值应用引理。 Lemma in_not_nil_42_take4 :
  ∀ l : **list** **nat**, In 42 l → $l \neq$ [].
Proof.
  intros l H.
  apply (in_not_nil **nat** 42).
  apply H.
Qed.

显式地对假设应用引理。 Lemma in_not_nil_42_take5 :
  ∀ l : **list** **nat**, In 42 l → $l \neq$ [].
Proof.
  intros l H.
  apply (in_not_nil _ _ _ H).
Qed.

对于几乎所有将定理名作为参数的策略而言，你都可以“将定理作为函数”来使用。注意，定理应用与函数应用使用了同样的类型推导机制，所以你可以将通配符作为定理的参数，或者为定理声明默认的隐式前提。这些特性在以下证明中展示。（此证明如何工作的细节不必关心，这里的目标只是为了展示它的用途。）

Example lemma_application_ex :

```
  ∀ {n : nat} {ns : list nat},
    In n (map (fun m ⇒ m × 0) ns) →
    n = 0.
Proof.
  intros n ns H.
  destruct (proj1 _ _ (In_map_iff _ _ _ _ _) H)
           as [m [Hm _]].
  rewrite mult_0_r in Hm. rewrite ← Hm. reflexivity.
Qed.
```

在以后的章节中我们将会看到更多这方面的例子。

14.5 Coq vs. 集合论

Coq 的逻辑核心，即'归纳构造演算（*Calculus of Inductive Constructions*）'系统，在很多重要的方面不同于数学家用来写下精确而严谨的定义和证明的形式化系统。例如，在主流的纸笔数学家中使用最普遍的'策梅洛-弗兰克尔集合论（*ZFC*）'中，一个数学对象可同时属于不同的集合；而在 Coq 的逻辑中，一个项最多只属于一个类型。这些不同之处需要人们用稍微不同的方式来描述非形式化的数学概念，但总的来说，它们都是非常自然而易于使用的。例如，在 Coq 中我们一般不说某个自然数 n 属于偶数集合，而是说 even n 成立，其中的 even : **nat** → Prop 描述了偶数的性质。

然而在某些情况下，将标准的数学论证翻译到 Coq 中会十分繁琐甚至是不可能的，除非我们引入新的公理来丰富其逻辑核心。作为本章的结尾，我们将探讨这两个世界之间最显著的区别。

14.5.1 函数的外延性

目前为止我们所看见的相等关系断言基本上都只考虑了归纳类型的元素（如 **nat**、**bool** 等等）。然而由于 Coq 的相等关系运算符是多态的，因此我们可以在'任何'类型上使用它。特别是，我们可以写出断言'两个函数相等'的命题：

```
Example function_equality_ex1 :
  (fun x ⇒ 3 + x) = (fun x ⇒ (pred 4) + x).
Proof. reflexivity. Qed.
```

在一般的数学研究中，对于任意两个函数 f 和 g，只要它们对相同的输入产生相等的结果，那么它们就被认为相等：

(forall x, f x = g x) -> f = g

这被称作'函数的外延性原理'。

不甚严谨地说，所谓"外延性"是指某个对象可观察到的行为。因此，函数的外延性就是指函数的标识完全由其行为来决定。用 Coq 的术语来说，就是函数的身份视其被应用后的结果而定。

然而，函数的外延性并不在 Coq 的基本公理之内，因此某些"合理"的命题是不可证明的：

```
Example function_equality_ex2 :
  (fun x ⇒ plus x 1) = (fun x ⇒ plus 1 x).
Proof.
Abort.
```

不过我们可以用 `Axiom` 指令将函数的外延性添加到 Coq 的核心逻辑系统中。

```
Axiom functional_extensionality : ∀ {X Y: Type}
                                    {f g : X → Y},
  (∀ (x:X), f x = g x) → f = g.
```

将某个东西用 `Axiom` 定义为公理的效果与陈述一个定理并用 *Admitted* 跳过其证明相同，不过它会提醒读者这是一个公理，我们无需证明！

现在我们可以在证明中调用函数的外延性了：

```
Example function_equality_ex2 :
  (fun x ⇒ plus x 1) = (fun x ⇒ plus 1 x).
Proof.
  apply functional_extensionality. intros x.
  apply plus_comm.
Qed.
```

当然，在为 Coq 添加公理时必须十分小心，因为这有可能会导致系统'不一致'，而当系统不一致的，任何命题都能在其中证明，包括 **False** 和 2+2=5！

不幸的是，并没有一种简单的方式能够判断添加某条公理是否安全：一般来说，确认任何一组公理的一致性都需要训练有素的数学家付出艰辛的努力。

然而，我们已经知道了添加函数外延性后的公理系统'确实是'一致的。

我们可以用 `Print Assumptions` 指令查看某个证明依赖的所有附加公理。

```
Print Assumptions function_equality_ex2.
```

练习：4 星, standard (tr_rev_correct) 列表反转函数 rev 的定义有一个问题，它会在每一步都执行一次 app 调用，而运行 app 所需时间与列表的大小线性渐近，也就是说 rev 的时间复杂度与列表长度呈渐进平方关系。我们可以用以下定义来改进它：

```
Fixpoint rev_append {X} (l1 l2 : list X) : list X :=
  match l1 with
  | [] ⇒ l2
  | x :: l1' ⇒ rev_append l1' (x :: l2)
  end.

Definition tr_rev {X} (l : list X) : list X :=
  rev_append l [].
```

此版本的 rev 是'*尾递归*（*tail-recursive*）'的，因为对函数自身的递归调用是需要执行的最后一步操作（即，在递归调用之后我们并不执行 ++ ）。一个足够好的编译器会为此生成十分高效的代码。

请证明以下两个定义等价。

```
Lemma tr_rev_correct : ∀ X, @tr_rev X = @rev X.
Proof.
  Admitted.
```

□

14.5.2 命题 vs. 布尔值

我们已经知道在 Coq 中有两种编码逻辑事实的方式了，即使用'布尔值'（类型为 **bool**）和'命题'（类型为 Prop）。

例如，我们可以通过以下两种方式来断言 n 为偶数：

evenb n 求值为 true：

```
Example even_42_bool : evenb 42 = true.
Proof. reflexivity. Qed.
```

或者存在某个 k 使得 n = double k：

```
Example even_42_prop : even 42.
Proof. unfold even. ∃ 21. reflexivity. Qed.
```

当然，如果二者刻画的偶数性描述的不是同一个自然数集合，那么会非常奇怪！幸运的是，我们确实可以证明二者相同...

首先我们需要两个辅助引理。 `Lemma evenb_double : ∀ k, evenb (double k) = true.`

```
Proof.
  intros k. induction k as [|k' IHk'].
  - reflexivity.
  - simpl. apply IHk'.
Qed.
```

练习：3 星, standard (evenb_double_conv) `Lemma evenb_double_conv : ∀ n, ∃ k,`

`  n = if evenb n then double k else S (double k).`

```
Proof.
```

Admitted.

□

Now the main theorem: `Theorem even_bool_prop : ∀ n,`

`  evenb n = true ↔ even n.`

```
Proof.
  intros n. split.
  - intros H. destruct (evenb_double_conv n) as [k Hk].
    rewrite Hk. rewrite H. ∃ k. reflexivity.
  - intros [k Hk]. rewrite Hk. apply evenb_double.
Qed.
```

此定理说明，布尔计算 evenb n 的真假会从命题 $\exists\ k,\ n =$ double k 的真假中'反映（*reflected*）'出来。

类似地，以下两种 n 与 m 相等的表述等价：

- (1) $n =?\ m$ 值为 true；
- (2) $n = m$。

同样，二者的记法也等价。

```
Theorem eqb_eq : ∀ n1 n2 : nat,
  n1 =? n2 = true ↔ n1 = n2.
Proof.
  intros n1 n2. split.
  - apply eqb_true.
```

```
  - intros H. rewrite H. rewrite ← eqb_refl. reflexivity.
Qed.
```

然而，即便布尔值和命题式在逻辑上是等价的，但它们的方便性从某些特定的目上来看并不一样。

在前面的偶数例子中，证明 even_bool_prop 的反向部分（即 evenb_double，从命题到布尔表达式的方向）时，我们对 k 进行了简单的归纳。而反方向的证明（即练习 evenb_double_conv）则需要一种聪明的一般化方法，因为我们无法直接证明 (evenb n = true) → even n。

对于这些例子来说，命题式的声明比与之对应的布尔表达式要更为有用，但并非总是如此。例如，我们无法在函数的定义中测试一般的命题是否为真，因此以下代码片段会被拒绝：

Fail

```
Definition is_even_prime n :=
  if n = 2 then true
  else false.
```

Coq 会抱怨 $n = 2$ 的类型是 Prop，而它想要一个 **bool** 类型的元素（或其它带有两个元素的归纳类型）。原因与 Coq 核心语言的'可计算性'特质有关，即它能表达的所有函数都是可计算且完全的。这样设计的的原因之一是为了能从 Coq 开发的代码中提取出可执行程序。因此，在 Coq 中 Prop '并没有'一种通用的情况分析操作来确定任意给定的命题是否为真，一旦存在这种操作，我们就能写出不可计算的函数。

尽管一般的不可计算性质无法表述为布尔计算，但值得注意的是，很多 '可计算的'性质更容易通过 Prop 而非 **bool** 来表达，因为在 Coq 中定义递归函数中会受到很大的限制。例如，下一章会展示如何用 Prop 来定义"某个正则表达式可以匹配给定的字符串"这一性质。如果使用 **bool** 来定义，就需要写一个真正的正则表达式匹配器了，比起该性质的简单定义（即非算法的定义）来说，这样会更加复杂，更难以理解，也更难以对它进行推理。

另一方面，通过布尔值来陈述事实会带来一点重要的优势，即通过对 Coq 中的项进行计算可以实现一些自动推理，这种技术被称为'互映证明（*Proof by Reflection*）'。考虑以下陈述：

```
Example even_1000 : even 1000.
```

对此命题而言，最直接的证明方式就是直接给出 k 的值。

```
Proof. unfold even. ∃ 500. reflexivity. Qed.
```

而使用与之对应的布尔语句的证明则更加简单（因为我们不必给出证据，Coq 的计算机制会帮我们搞定它！）

```
Example even_1000' : evenb 1000 = true.
Proof. reflexivity. Qed.
```

有趣的是，由于这两种定义是等价的，因此我们无需显式地给出 500，而是使用布尔等价式来证明彼此：

```
Example even_1000'' : even 1000.
Proof. apply even_bool_prop. reflexivity. Qed.
```

尽管此例的证明脚本的长度并未因此而减少，然而更大的证明通常可通过这种互映的方式来显著化简。举一个极端的例子，在用 Coq 证明著名的 '四色定理'时，人们使用互映技巧将几百种不同的情况归约成了一个布尔计算。

另一点明显的不同是"布尔事实"的否定可以被直白地陈述并证明，只需翻转预期的布尔值结果即可。

```
Example not_even_1001 : evenb 1001 = false.
Proof.
  reflexivity.
Qed.
```

相反，命题的否定形式可能更难以直接证明。

```
Example not_even_1001' : ~(even 1001).
Proof.
  rewrite ← even_bool_prop.
  unfold not.
  simpl.
  intro H.
  discriminate H.
Qed.
```

相等性提供了另一个互补的例子，在命题的世界中它有时更容易处理。在涉及 n 和 m 的证明中，知道 $n =? m = \text{true}$ 通常没什么直接的帮助。然而如果我们将该语句转换为等价的 $n = m$ 形式，则可利用该等式改写证明目标。

```
Lemma plus_eqb_example : ∀ n m p : nat,
    n =? m = true → n + p =? m + p = true.
```

```
Proof.
  intros n m p H.
    rewrite eqb_eq in H.
  rewrite H.
  rewrite eqb_eq.
  reflexivity.
Qed.
```

我们不会在这里详细讨论互映技巧，然而对于展示布尔计算与一般命题的互补优势而言，它是个很好的例子，在后面的章节中，能够在布尔和命题的世界之间来回穿梭通常会非常方便。

练习：2 星, standard (logical_connectives) 以下引理将本章中讨论的命题联结词与对应的布尔操作关联了起来。

```
Lemma andb_true_iff : ∀ b1 b2:bool,
  b1 && b2 = true ↔ b1 = true ∧ b2 = true.
Proof.
    Admitted.

Lemma orb_true_iff : ∀ b1 b2,
  b1 || b2 = true ↔ b1 = true ∨ b2 = true.
Proof.
    Admitted.
    □
```

练习：1 星, standard (eqb_neq) 以下定理为等价式 eqb_eq 的“否定”版本，在某些场景中使用它会更方便些（后面的章节中会讲到这方面的例子）。

```
Theorem eqb_neq : ∀ x y : nat,
  x =? y = false ↔ x ≠ y.
Proof.
    Admitted.
    □
```

练习：3 星, standard (eqb_list) 给定一个用于测试类型为 A 的元素相等关系的布尔操

作符 eqb，我们可以定义函数 eqb_list 来测试元素类型为 A 的列表的相等关系。请完成以下 eqb_list 函数的定义。要确定你的定义是否正确，请证明引理 eqb_list_true_iff。

```
Fixpoint eqb_list {A : Type} (eqb : A → A → bool)
                             (l1 l2 : list A) : bool
  . Admitted.

Lemma eqb_list_true_iff :
  ∀ A (eqb : A → A → bool),
    (∀ a1 a2, eqb a1 a2 = true ↔ a1 = a2) →
    ∀ l1 l2, eqb_list eqb l1 l2 = true ↔ l1 = l2.
Proof.
    Admitted.
    □
```

练习：2 星, standard, recommended (All_forallb) 回忆一下Tactics一章中练习 *forall_exists_challeng* 的函数 forallb：

```
Fixpoint forallb {X : Type} (test : X → bool) (l : list X) : bool :=
  match l with
  | [] ⇒ true
  | x :: l' ⇒ andb (test x) (forallb test l')
  end.
```

请证明以下定理，它将 forallb 与之前的定义中 `All` 的性质联系了起来。

```
Theorem forallb_true_iff : ∀ X test (l : list X),
    forallb test l = true ↔ All (fun x ⇒ test x = true) l.
Proof.
    Admitted.
```

（未分级的思考题）函数 forallb 是否还存在尚未被此规范刻画到的重要性质？

14.5.3 经典逻辑 vs. 构造逻辑

我们已经知道了，在定义 Coq 函数时是无法判断命题 P 是否成立。然而'证明'也存在类似的限制！换句话说，以下推理原则即便符合直觉，不过在 Coq 中它是不可证明的：

```
Definition excluded_middle := ∀ P : Prop,
```

$P \vee \neg P$.

为了在操作上理解为何如此，回忆一下，在证明形如 $P \vee Q$ 的陈述时，我们使用了 `left` 与 `right` 策略，它们能够有效地知道析取的哪边成立。然而在 excluded_middle 中，P 是被全称量化的'任意'命题，我们对它一无所知。我们没有足够的信息来选择使用 `left` 或 `right` 中的哪一个。就像 Coq 因为缺乏信息而无法在函数内部机械地确定 P 是否成立一样。

然而，如果我们恰好知道 P 与某个布尔项互映，那么就能很轻易地知道它是否成立了：我们只需检查 b 的值即可。

```
Theorem restricted_excluded_middle : ∀ P b,
  (P ↔ b = true) → P ∨ ¬ P.
Proof.
  intros P [] H.
  - left. rewrite H. reflexivity.
  - right. rewrite H. intros contra. discriminate contra.
Qed.
```

特别来说，对于自然数 n 和 m 的 $n = m$ 而言，排中律是成立的。

```
Theorem restricted_excluded_middle_eq : ∀ (n m : nat),
  n = m ∨ n ≠ m.
Proof.
  intros n m.
  apply (restricted_excluded_middle (n = m) (n =? m)).
  symmetry.
  apply eqb_eq.
Qed.
```

一般的排中律在 Coq 中默认并不可用，因为它是常见的逻辑系统（如 ZFC）中的标准特性。尽管如此，不假设排中律的成立仍有其独特的优点：Coq 中的陈述可以构造出比标准数学中同样陈述更强的断言。特别是，当存在 $\exists\ x,\ P\ x$ 的 Coq 证明时，我们可以直接给出一个使 $P\ x$ 得证的值 x。换言之，任何关于存在性的证明必定是'构造性'的。

像 Coq 一样不假设排中律成立的逻辑系统被称作'构造逻辑'。

像 ZFC 这样更加传统的，排中律对于任何命题都成立的逻辑系统则被称作'经典逻辑'。

以下示例展示了为何假设排中律成立会导致非构造性证明：

'命题'：存在无理数 a 和 b 使得 a ^ b（a 的 b 次方）为有理数。

'证明'：易知 $sqrt\ 2$ 为无理数。若 $sqrt\ 2\ \hat{}\ sqrt\ 2$ 为有理数，那么可以取 $a = b = sqrt\ 2$ 证明结束；否则 $sqrt\ 2\ \hat{}\ sqrt\ 2$ 为无理数。此时，我们可以取 $a = sqrt\ 2\ \hat{}\ sqrt\ 2$ 和 $b = sqrt\ 2$，因为 $a\ \hat{}\ b = sqrt\ 2\ \hat{}\ (sqrt\ 2 \times sqrt\ 2) = sqrt\ 2\ \hat{}\ 2 = 2$. □

看到发生什么了吗？我们使用排中律在不知道 $sqrt\ 2\ \hat{}\ sqrt\ 2$ 是否为有理数的情况下就分别考虑了这两种情况！因此，我们通过证明知道了这样的 a 和 b 存在，但却无法确切知道它们的值（至少无法从此论据中获知）。

即便构造逻辑很有用，它也有自身的限制：存在很多容易用经典逻辑证明的命题，用构造证明只会更加复杂，而对于某些已知的命题而言这样的构造性证明甚至不存在！幸运的是，排中律和函数外延性一样都是与 Coq 的逻辑系统兼容的，我们可以安全地将它作为公理添加到 Coq 中。然而，在本书中我们不必如此：我们所涉及的结构都可以完全用构造逻辑得到，所需的额外代价则微不足道。

我们需要一定的实践才能理解哪些证明技巧不应在构造推理中使用，而其中的反证法尤为臭名昭著，因为它会导向非构造性证明。这里有个典型的例子：假设我们想要证明存在 x 具有某种性质 P，即存在 $P\ x$。我们先假设结论为假，也就是说 $\neg\ \exists\ x,\ P\ x$。根据此前提，不难推出 $\forall\ x,\ \neg\ P\ x$。如果我们能够根据此中间事实得到矛盾，就能得到一个存在性证明而完全不必指出一个 x 的值使得 $P\ x$ 成立！

从构造性的角度来看，这里存在着技术上的瑕疵，即我们试图通过对 $\neg\ \neg\ (\exists\ x,\ P\ x)$ 的证明来证明 $\exists\ x,\ P\ x$。从以下练习中我们会看到，允许自己从任意陈述中去掉双重否定等价于引入排中律。因此，只要我们不引入排中律，就无法在 Coq 中编码此推理。

练习：3 星, standard (excluded_middle_irrefutable) 证明通用排中律公理与 Coq 的一致性需要复杂的推理，而且并不能在 Coq 自身中进行。然而，以下定理蕴含了假设可判定性公理（即排中律的一个特例）成立对于任何'具体的'命题 P 而言总是安全的。之所以如此，是因为我们无法证明这类公理的否定命题。假如我们可以的话，就会同时有 $\neg\ (P \vee \neg P)$ 和 $\neg\ \neg\ (P \vee \neg P)$（因为根据引理 double_neg，$P$ 蕴含 $\neg\ \neg\ P$），而这会产生矛盾。但因为我们不能，所以将 $P \vee \neg P$ 作为公理加入是安全的。

```
Theorem excluded_middle_irrefutable: ∀ (P:Prop),
  ¬ ¬ (P ∨ ¬ P).
Proof.
```

Admitted.

□

练习：3 星, advanced (not_exists_dist) 在经典逻辑中有这样一条定理，它断言以下两条命题是等价的：

~ (exists x, ~ P x) forall x, P x

之前的 dist_not_exists 证明了此等价式的一个方向。有趣的是，我们无法用构造逻辑证明另一个方向。你的任务就是证明排中律蕴含此方向的证明。

```
Theorem not_exists_dist :
  excluded_middle →
  ∀ (X:Type) (P : X → Prop),
    ¬ (∃ x, ¬ P x) → (∀ x, P x).
Proof.
```

Admitted.

□

练习：5 星, standard, optional (classical_axioms) 对于喜欢挑战的读者，以下练习来自于 Bertot 与 Casteran 所著的 Coq'Art 一书中第 123 页。以下四条陈述的每一条，加上 excluded_middle 可以认为刻画了经典逻辑。我们无法在 Coq 中证明其中的任意一条，不过如果我们希望在经典逻辑下工作的话，可以安全地将其中任意一条作为公理添加到 Coq 中而不会造成不一致性。

请证明所有五个命题都是等价的（这四个再加上 excluded_middle）。

提示：不要去分别考虑每一对命题，而是证明一条将它们连接起来的单向蕴含环链。

```
Definition peirce := ∀ P Q: Prop,
  ((P→Q)->P)->P.

Definition double_negation_elimination := ∀ P:Prop,
  ~~P → P.

Definition de_morgan_not_and_not := ∀ P Q:Prop,
  ~(~P ∧ ¬Q) → P∨Q.

Definition implies_to_or := ∀ P Q:Prop,
  (P→Q) → (¬P∨Q).
```

Chapter 15

Library LF.Tactics

15.1 Tactics: 更多基本策略

本章额外介绍了一些证明策略和手段，它们能用来证明更多关于函数式程序的有趣性质。

我们会看到：

- 如何在“向前证明”和“向后证明”两种风格中使用辅助引理；
- 如何对数据构造子进行论证，特别是，如何利用它们单射且不交的事实；
- 如何增强归纳假设，以及何时需要增强；
- 还有通过分类讨论进行论证的更多细节。

```
Set Warnings "-notation-overridden,-parsing".
From LF Require Export Poly.
```

15.2 apply 策略

我们经常会遇到待证目标与上下文中的前提或已证引理'刚好相同'的情况。

```
Theorem silly1 : ∀ (n m o p : nat),
  n = m →
  [n;o] = [n;p] →
```

```
    [n;o] = [m;p].
Proof.
  intros n m o p eq1 eq2.
  rewrite ← eq1.
```

我们可以像之前那样用“`rewrite → eq2. reflexivity.`”来完成。不过如果我们使用 `apply` 策略，只需一步就能完成此证明：

```
  apply eq2. Qed.
```

`apply` 策略也可以配合'条件（*Conditional*）'假设和引理来使用：如果被应用的语句是一个蕴含式，那么该蕴含式的前提就会被添加到待证子目标列表中。

```
Theorem silly2 : ∀ (n m o p : nat),
    n = m →
    (n = m → [n;o] = [m;p]) →
    [n;o] = [m;p].
Proof.
  intros n m o p eq1 eq2.
  apply eq2. apply eq1. Qed.
```

通常，当我们使用 `apply H` 时，语句 H 会以一个引入了某些'通用变量（*Universal Variables*）'的 ∀ 开始。在 Coq 针对 H 的结论匹配当前目标时，它会尝试为这些变量查找适当的值。例如，当我们在以下证明中执行 `apply eq2` 时，*eq2* 中的通用变量 q 会以 n 实例化，而 r 会以 m 实例化。

```
Theorem silly2a : ∀ (n m : nat),
    (n,n) = (m,m) →
    (∀ (q r : nat), (q,q) = (r,r) → [q] = [r]) →
    [n] = [m].
Proof.
  intros n m eq1 eq2.
  apply eq2. apply eq1. Qed.
```

练习：2 星, standard, optional (silly_ex) 请只用 `intros` 和 `apply` 完成以下证明。

```
Theorem silly_ex :
    (∀ n, evenb n = true → oddb (S n) = true) →
```

```
    evenb 4 = true →
    oddb 3 = true.
Proof.
  Admitted.
```

□

要使用 apply 策略，被应用的事实（的结论）必须精确地匹配证明目标：例如, 当等式的左右两边互换后，apply 就无法起效了。

```
Theorem silly3_firsttry : ∀ (n : nat),
    true = (n =? 5) →
    (S (S n)) =? 7 = true.
Proof.
  intros n H.
```

在这里，我们无法直接使用 apply，不过我们可以用 symmetry 策略它会交换证明目标中等式的左右两边。

```
  symmetry.
  simpl.  (此处的 simpl 是可选的，因为 apply 会在需要时先进行化简。)  apply H.
Qed.
```

练习：3 星, standard (apply_exercise1) （'提示'：你可以配合之前定义的引理来使用 apply，不仅限于当前上下文中的前提。记住 Search 是你的朋友。）

```
Theorem rev_exercise1 : ∀ (l l' : list nat),
    l = rev l' →
    l' = rev l.
Proof.
  Admitted.
```

□

练习：1 星, standard, optional (apply_rewrite) 简述 apply 与 rewrite 策略之区别。哪些情况下二者均可有效利用？

15.3 apply with 策略

以下愚蠢的例子在一行中使用了两次改写来将 [a;b] 变成 [e;f]。

```
Example trans_eq_example : ∀ (a b c d e f : nat),
     [a;b] = [c;d] →
     [c;d] = [e;f] →
     [a;b] = [e;f].
Proof.
  intros a b c d e f eq1 eq2.
  rewrite → eq1. rewrite → eq2. reflexivity. Qed.
```

由于这种模式十分常见，因此我们希望一劳永逸地把它作为一条引理记录下来，即等式具有传递性。

```
Theorem trans_eq : ∀ (X:Type) (n m o : X),
  n = m → m = o → n = o.
Proof.
  intros X n m o eq1 eq2. rewrite → eq1. rewrite → eq2.
  reflexivity. Qed.
```

现在，按理说我们应该可以用 trans_eq 来证明前面的例子了。然而，为此我们还需要稍微改进一下 apply 策略。

```
Example trans_eq_example' : ∀ (a b c d e f : nat),
     [a;b] = [c;d] →
     [c;d] = [e;f] →
     [a;b] = [e;f].
Proof.
  intros a b c d e f eq1 eq2.
```

如果此时我们只是告诉 Coq apply trans_eq，那么它会（根据该引理的结论对证明目标的匹配）说 X 应当实例化为 [**nat**]、n 实例化为 [a,b]、以及 o 实例化为 [e,f]。然而，匹配过程并没有为 m 确定实例：我们必须在 apply 的调用后面加上 "with (m:=[c,d])" 来显式地提供一个实例。

```
  apply trans_eq with (m:=[c;d]).
  apply eq1. apply eq2. Qed.
```

（实际上，我们通常不必在 `with` 从句中包含名字 m，Coq 一般足够聪明来确定我们实例化的变量。我们也可以写成： `apply trans_eq with [c;d]`。）

练习：3 星, standard, optional (apply_with_exercise) `Example trans_eq_exercise : ∀ (n m o p : nat),`

```
  m = (minustwo o) →
  (n + p) = m →
  (n + p) = (minustwo o).
Proof.
  Admitted.
```

□

15.4 The `injection` and `discriminate` Tactics

回想自然数的定义：

Inductive nat : Type := | O | S (n : nat).

我们可从该定义中观察到，所有的数都是两种形式之一：要么是构造子 O，要么就是将构造子 S 应用到另一个数上。不过这里还有无法直接看到的：自然数的定义中还蕴含了两个事实：

- 构造子 S 是'单射（*Injective*）'或'一一对应'的。即，如果 S n = S m，那么 $n = m$ 必定成立。
- 构造子 O 和 S 是'不相交（*Disjoint*）'的。即，对于任何 n，O 都不等于 S n。

类似的原理同样适用于所有归纳定义的类型：所有构造子都是单射的，而不同构造子构造出的值绝不可能相等。对于列表来说，cons 构造子是单射的，而 nil 不同于任何非空列表。对于布尔值来说，true 和 false 是不同的。因为 true 和 false 二者都不接受任何参数，它们既不在这边也不在那边。其它归纳类型亦是如此。

例如，我们可以使用定义在 *Basics.v* 中的 pred 函数来证明 S 的单射性。 .

```
Theorem S_injective : ∀ (n m : nat),
  S n = S m →
  n = m.
Proof.
```

```
  intros n m H1.
  assert (H2: n = pred (S n)). { reflexivity. }
  rewrite H2. rewrite H1. reflexivity.
Qed.
```

这个技巧可以通过编写等价的 pred 来推广到任意的构造子上—— 即编写一个“撤销”一次构造子调用的函数。为此，Coq 提供了更加简便的 injection 策略，它能让我们利用任意构造子的单射性。下面是使用 injection 对上面定理的另一种证法：

```
Theorem S_injective' : ∀ (n m : nat),
  S n = S m →
  n = m.
Proof.
  intros n m H.
```

通过在此处编写 injection H as Hmn，我们让 Coq 利用构造子的单射性来产生所有它能从 H 所推出的等式（本例中为等式 $n = m$）。每一个这样的等式都作为假设（本例中为 Hmn）被添加到上下文中。

```
  injection H as Hnm. apply Hnm.
Qed.
```

以下例子展示了一个 injection 如何直接得出多个等式。

```
Theorem injection_ex1 : ∀ (n m o : nat),
  [n; m] = [o; o] →
  [n] = [m].
Proof.
  intros n m o H.
  injection H as H1 H2.
  rewrite H1. rewrite H2. reflexivity.
Qed.
```

另一方面，如果你只使用 injection H 而不带 as 从句，那么所有的等式都会在目标的开头被转变为假设。

```
Theorem injection_ex2 : ∀ (n m o : nat),
  [n; m] = [o; o] →
  [n] = [m].
```

```
Proof.
  intros n m o H.
  injection H.
  intros H1 H2. rewrite H1. rewrite H2. reflexivity.
Qed.
```

练习：3 星, standard (injection_ex3) `Example injection_ex3 : ∀ (X : Type) (x y z : X) (l j : list X),`

```
  x :: y :: l = z :: j →
  j = z :: l →
  x = y.
Proof.
```

Admitted.

□

So much for injectivity of constructors. What about disjointness?

The principle of disjointness says that two terms beginning with different constructors (like O and S, or true and false) can never be equal. This means that, any time we find ourselves in a context where we've *assumed* that two such terms are equal, we are justified in concluding anything we want, since the assumption is nonsensical.

The `discriminate` tactic embodies this principle: It is used on a hypothesis involving an equality between different constructors (e.g., S n = O), and it solves the current goal immediately. Here is an example:

```
Theorem eqb_0_l : ∀ n,
   0 =? n = true → n = 0.
Proof.
  intros n.
```

我们可以通过对 n 进行分类讨论来继续。第一种分类是平凡的。

```
  destruct n as [| n'] eqn:E.
  -
    intros H. reflexivity.
```

However, the second one doesn't look so simple: assuming 0 =? (S n') = true, we must show S n' = 0! The way forward is to observe that the assumption itself is nonsensical:

-

  simpl.

如果我们对这个假设使用 discriminate ， Coq 便会确认我们当前正在证明的目标不可行，并同时移除它，不再考虑。

  intros H. discriminate H.

Qed.

本例是逻辑学原理'爆炸原理'的一个实例，它断言矛盾的前提会推出任何东西，甚至是假命题!

Theorem discriminate_ex1 : $\forall$ (n : **nat**),

  S n = O $\rightarrow$

  2 + 2 = 5.

Proof.

  intros n *contra*. discriminate *contra*. Qed.

Theorem discriminate_ex2 : $\forall$ (n m : **nat**),

  false = true $\rightarrow$

  [n] = [m].

Proof.

  intros n m *contra*. discriminate *contra*. Qed.

爆炸原理可能令你费解，那么请记住上述证明'并不'肯定其后件，而是说明：'如果'荒谬的前件成立，'那么'就会得出荒谬的结论，如此一来我们将生活在一个不一致的宇宙中，这里每个陈述都是正确的。下一章将进一步讨论爆炸原理。

练习：1 星, standard (discriminate_ex3) Example discriminate_ex3 :

  $\forall$ (X : Type) (x y z : X) (l j : **list** X),

    x :: y :: l = [] $\rightarrow$

    x = z.

Proof.

    Admitted.

    □

构造子的单射性能让我们论证 $\forall$ (n m : **nat**), S n = S m $\rightarrow$ n = m。此蕴含式的逆形式是一个构造子和函数的更一般的实例，在后面我们会发现它用起来很方便：

Theorem f_equal : $\forall$ (A B : Type) (f: $A \rightarrow B$) (x y: A),

$x = y \to f\ x = f\ y.$

```
Proof. intros A B f x y eq. rewrite eq. reflexivity. Qed.
```

```
Theorem eq_implies_succ_equal : ∀ (n m : nat),
  n = m → S n = S m.
Proof. intros n m H. apply f_equal. apply H. Qed.
```

There is also a tactic named 'f_equal' that can prove such theorems. Given a goal of the form $f\ a1\ ...\ an = g\ b1\ ...\ bn$, the tactic `f_equal` will produce subgoals of the form $f = g$, $a1 = b1$, ..., $an = bn$. At the same time, any of these subgoals that are simple enough (e.g., immediately provable by `reflexivity`) will be automatically discharged by `f_equal`.

```
Theorem eq_implies_succ_equal' : ∀ (n m : nat),
  n = m → S n = S m.
Proof. intros n m H. f_equal. apply H. Qed.
```

15.5 对假设使用策略

默认情况下，大部分策略会作用于目标公式并保持上下文不变。然而，大部分策略还有对应的变体来对上下文中的语句执行类似的操作。

例如，策略 `simpl in` H 会对上下文中的假设 H 执行化简。

```
Theorem S_inj : ∀ (n m : nat) (b : bool),
  (S n) =? (S m) = b →
  n =? m = b.
Proof.
  intros n m b H. simpl in H. apply H. Qed.
```

类似地，`apply` L `in` H 会针对上下文中的假设 H 匹配某些（形如 X → Y 中的）条件语句 L。然而，与一般的 `apply` 不同（它将匹配 Y 的目标改写为子目标 X），`apply` L `in` H 会针对 X 匹配 H，如果成功，就将其替换为 Y。

换言之，`apply` L `in` H 给了我们一种"正向推理"的方式：根据 X → Y 和一个匹配 X 的假设，它会产生一个匹配 Y 的假设。作为对比，`apply` L 是一种"反向推理"：它表示如果我们知道 X → Y 并且试图证明 Y，那么证明 X 就足够了。

下面是前面证明的一种变体，它始终使用正向推理而非反向推理。

```
Theorem silly3' : ∀ (n : nat),
```

```
  (n =? 5 = true → (S (S n)) =? 7 = true) →
  true = (n =? 5) →
  true = ((S (S n)) =? 7).
Proof.
  intros n eq H.
  symmetry in H. apply eq in H. symmetry in H.
  apply H. Qed.
```

正向推理从'给定'的东西开始（即前提、已证明的定理），根据它们迭代地刻画结论直到抵达目标。反向推理从'目标'开始，迭代地推理蕴含目标的东西，直到抵达前提或已证明的定理。

你在数学或计算机科学课上见过的非形式化证明可能倾向于正向推理。通常，Coq 习惯上倾向于使用反向推理，但在某些情况下，正向推理更易于思考。

15.6 变换归纳假设

在 Coq 中进行归纳证明时，有时控制归纳假设的确切形式是十分重要的。特别是，在调用 `induction` 策略前，我们有时需要用 `intros` 将假设从目标移到上下文中时要十分小心。例如，假设我们要证明 double 函数是单射的 – 即，它将不同的参数映射到不同的结果：

Theorem double_injective: forall n m, double n = double m -> n = m.

此证明的开始方式有点微妙：如果我们以

intros n. induction n.

开始，那么一切都好。然而假如以

intros n m. induction n.

开始，就会卡在归纳情况中...

```
Theorem double_injective_FAILED : ∀ n m,
     double n = double m →
     n = m.
Proof.
  intros n m. induction n as [| n' IHn'].
  - simpl. intros eq. destruct m as [| m'] eqn:E.
    + reflexivity.
```

+ discriminate *eq*.

- intros *eq*. destruct *m* as [| *m'*] *eqn*:*E*.

+ discriminate *eq*.

+ apply f_equal.

此时，归纳假设 *IHn'* '不会'给出 $n' = m'$ – 会有个额外的 S 阻碍 – 因此该目标无法证明。

Abort.

哪里出了问题？

问题在于，我们在调用归纳假设的地方已经将 m 引入了上下文中 – 直观上，我们已经告诉了 Coq"我们来考虑具体的 n 和 m..."，而现在必须为那些'具体的' n 和 m 证明 double n = double m，然后才有 $n = m$。

下一个策略 induction n 告诉 Coq：我们要对 n 归纳来证明该目标。也就是说，我们要证明对于'所有的' n，命题

- $P\ n$ = "if double n = double m, then $n = m$"

成立，需通过证明

- P O

（即，若"double O = double m 则 O = m"）和

- $P\ n \to P$ (S n)

（即，若"double n = double m 则 $n = m$"蕴含"若 double (S n) = double m 则 S n = m"）来得出。

如果我们仔细观察第二个语句，就会发现它说了奇怪的事情：即，对于一个'具体的' m，如果我们知道

- "若 double n = double m 则 $n = m$"

那么我们就能证明

- "若 double (S n) = double m 则 S $n = m$"。

要理解为什么它很奇怪，我们来考虑一个具体的（任意但确定的）m – 比如说 5。该语句就会这样说：如果我们知道

- Q = “若 double $n = 10$ 则 $n = 5$”

  那么我们就能证明

- **R** = “若 double (S n) $= 10$ 则 S $n = 5$”。

但是知道 Q 对于证明 **R** 来说并没有任何帮助！（如果我们试着根据 Q 证明 **R**，就会以“假设 double (S n) $= 10$..”这样的句子开始，不过之后我们就会卡住：知道 double (S n) 为 10 并不能告诉我们 double n 是否为 10。（实际上，它强烈地表示 double n '不是' 10！）因此 Q 是没有用的。）

当 m 已经在上下文中时，试图对 n 进行归纳来进行此证明是行不通的，因为我们之后要尝试证明涉及'每一个' n 的命题，而不只是'单个' m。

对 double_injective 的成功证明将 m 留在了目标语句中 `induction` 作用于 n 的地方：

```
Theorem double_injective : ∀ n m,
  double n = double m →
  n = m.
Proof.
  intros n. induction n as [| n' IHn'].
  - simpl. intros m eq. destruct m as [| m'] eqn:E.
    + reflexivity.
    + discriminate eq.
  - simpl.
```

注意，此时的证明目标和归纳假设是不同的：证明目标要求我们证明更一般的事情（即，为'每一个' m 证明该语句），而归纳假设 IH 相应地更加灵活，允许我们在应用归纳假设时选择任何想要的 m。

```
    intros m eq.
```

现在我们选择了一个具体的 m 并引入了假设 double n = double m。由于我们对 n 做了情况分析，因此还要对 m 做情况分析来保持两边“同步”。

```
    destruct m as [| m'] eqn:E.
    + simpl.
```

0 的情况很显然：

```
      discriminate eq.
```

+

  apply f_equal.

到这里，由于我们在 destruct m 的第二个分支中，因此上下文中涉及到的 m' 就是我们开始讨论的 m 的前趋。由于我们也在归纳的 S 分支中，这就很完美了：如果我们在归纳假设中用当前的 m'（此实例由下一步的 apply 自动产生）实例化一般的 m，那么 *IHn'* 就刚好能给出我们需要的来结束此证明。

  apply *IHn'*. injection *eq* as goal. apply goal. Qed.

What you should take away from all this is that we need to be careful, when using induction, that we are not trying to prove something too specific: When proving a property involving two variables n and m by induction on n, it is sometimes crucial to leave m generic.

以下练习遵循同样的模式。

练习：2 星, standard (eqb_true) Theorem eqb_true : $\forall$ n m,

  n =? m = true $\rightarrow$ n = m.

Proof.

  Admitted.

□

练习：2 星, advanced (eqb_true_informal) 给出一个详细的 eqb_true 的非形式化证明，量词要尽可能明确。

Definition manual_grade_for_informal_proof : **option** (**nat**×**string**) := None.

□

练习：3 星, standard, recommended (plus_n_n_injective) In addition to being careful about how you use intros, practice using "in" variants in this proof. (Hint: use plus_n_Sm.) Theorem plus_n_n_injective : $\forall$ n m,

  n + n = m + m $\rightarrow$

  n = m.

Proof.

  intros n. induction n as [| n'].

  Admitted.

□

在 induction 之前做一些 intros 来获得更一般归纳假设并不总是奏效。有时需要对量化的变量做一下'重排'。例如，假设我们想要通过对 m 而非 n 进行归纳来证明 double_injective。

```
Theorem double_injective_take2_FAILED : ∀ n m,
     double n = double m →
     n = m.
Proof.
  intros n m. induction m as [| m' IHm'].
  - simpl. intros eq. destruct n as [| n'] eqn:E.
    + reflexivity.
    + discriminate eq.
  - intros eq. destruct n as [| n'] eqn:E.
    + discriminate eq.
    + apply f_equal.
Abort.
```

问题在于，要对 m 进行归纳，我们首先必须对 n 归纳。（而如果我们不引入任何东西就执行 induction m，Coq 就会自动为我们引入 n!）

我们可以对它做什么？一种可能就是改写该引理的陈述使得 m 在 n 之前量化。这样是可行的，不过它不够好：我们不想调整该引理的陈述来适应具体的证明策略！我们更想以最清晰自然的方式陈述它。

我们可以先引入所有量化的变量，然后'重新一般化（*re-generalize*）'其中的一个或几个，选择性地从上下文中挑出几个变量并将它们放回证明目标的开始处。用 generalize dependent 策略就能做到。

```
Theorem double_injective_take2 : ∀ n m,
     double n = double m →
     n = m.
Proof.
  intros n m.
  generalize dependent n.
  induction m as [| m' IHm'].
  - simpl. intros n eq. destruct n as [| n'] eqn:E.
    + reflexivity.
```

```
    + discriminate eq.
  - intros n eq. destruct n as [| n'] eqn:E.
    + discriminate eq.
    + apply f_equal.
      apply IHm'. injection eq as goal. apply goal. Qed.
```

我们来看一下此定理的非形式化证明。注意我们保持 n 的量化状态并通过归纳证明的命题，对应于我们形式化证明中依赖的一般化。

'定理'：对于任何自然数 n 和 m，若 double n = double m，则 $n = m$。

'证明'：令 m 为一个 **nat**。我们通过对 m 进行归纳来证明，对于任何 n，若 double n = double m，则 $n = m$。

- 首先，设 $m = 0$，而 n 是一个数使得 double n = double m。我们必须证明 $n = 0$。

  由于 $m = 0$，根据 double 的定义，我们有 double $n = 0$。此时对于 n 需要考虑两种情况。若 $n = 0$，则得证，因为 $m = 0 = n$，正如所需。否则，若对于某个 n' 有 n = S n'，我们就会导出矛盾：根据 double 的定义，我们可得出 double n = S (S (double n'))，但它与 double $n = 0$ 相矛盾。

- 其次，设 m = S m'，而 n 同样是一个数使得 double n = double m。我们必须证明 n = S m'，根据归纳假设，对于任何数 s，若 double s = double m'，则 $s = m'$。

  根据 m = S m' 的事实以及 double 的定义我们有 double n = S (S (double m'))。此时对于 n 需要考虑两种情况。

  若 $n = 0$，则根据 double $n = 0$ 的定义会得出矛盾。

  故存在 n' 使得 n = S n'。再次根据 double 之定义，可得 S (S (double n')) = S (S (double m'))。再由构造子单射可知 double n' = double m'。以n' 代入归纳假设，推得 $n' = m'$，故显然 S n' = S m'，其中 S n' = n，S m' = m，所以原命题得证。 □

在结束本节之前，我们先稍微跑个题，使用 eqb_true 来证明一个标识符的类似性质以备后用：

```
Theorem eqb_id_true : ∀ x y,
  eqb_id x y = true → x = y.
Proof.
  intros [m] [n]. simpl. intros H.
  assert (H' : m = n). { apply eqb_true. apply H. }
```

```
  rewrite H'. reflexivity.
Qed.
```

练习：3 星, standard, recommended (gen_dep_practice) 通过对 l 进行归纳来证明它。

```
Theorem nth_error_after_last: ∀ (n : nat) (X : Type) (l : list X),
     length l = n →
     nth_error l n = None.
Proof.
    Admitted.
```

□

15.7 展开定义

It sometimes happens that we need to manually unfold a name that has been introduced by a `Definition` so that we can manipulate the expression it denotes. For example, if we define...

```
Definition square n := n × n.
```

...并试图证明一个关于 square 的简单事实...

```
Lemma square_mult : ∀ n m, square (n × m) = square n × square m.
Proof.
  intros n m.
  simpl.
```

...那么就会卡住：`simpl` 无法化简任何东西，而由于我们尚未证明任何关于 square 的事实，也就没有任何可以用来 `apply` 或 `rewrite` 的东西。

为此，我们可以手动用 `unfold` 展开 square 的定义：

```
  unfold square.
```

现在我们有很多工作要做：等式两边都是涉及乘法的表达式，而我们有很多可用的关于乘法的事实。特别是，我们知道它满足交换性和结合性，该引理据此不难证明。

```
  rewrite mult_assoc.
  assert (H : n × m × n = n × n × m).
```

```
  { rewrite mult_comm. apply mult_assoc. }
  rewrite H. rewrite mult_assoc. reflexivity.
Qed.
```

现在，是时候讨论下展开和化简了。

我们已经观察到，像 simpl、reflexivity 和 apply 这样的策略，通常总会在需要时自动展开函数的定义。例如，若我们将 foo m 定义为常量 5...

```
Definition foo (x: nat) := 5.
```

那么在以下证明中 simpl（或 reflexivity，如果我们忽略 simpl）就会将 foo m 展开为 (fun $x \Rightarrow 5$) m 并进一步将其化简为 5。

```
Fact silly_fact_1 : ∀ m, foo m + 1 = foo (m + 1) + 1.
Proof.
  intros m.
  simpl.
  reflexivity.
Qed.
```

然而，这种自动展开有些保守。例如，若我们用模式匹配定义稍微复杂点的函数...

```
Definition bar x :=
  match x with
  | O ⇒ 5
  | S _ ⇒ 5
  end.
```

...那么类似的证明就会被卡住：

```
Fact silly_fact_2_FAILED : ∀ m, bar m + 1 = bar (m + 1) + 1.
Proof.
  intros m.
  simpl. Abort.
```

simpl 没有进展的原因在于，它注意到在试着展开 bar m 之后会留下被匹配的 m，它是一个变量，因此 match 无法被进一步化简。它还没有聪明到发现 match 的两个分支是完全相同的。因此它会放弃展开 bar m 并留在那。类似地，在试着展开 bar $(m+1)$ 时会留下一个 match，被匹配者是一个函数应用（即它本身，即便在展开 $+$ 的定义后也无法被化简），因此 simpl 会留下它。

此时有两种方法可以继续。一种是用 `destruct` m 将证明分为两种情况，每一种都关注于更具体的 m（O vs S _）。在这两种情况下， bar 中的 `match` 可以继续了，证明则很容易完成。

```
Fact silly_fact_2 : ∀ m, bar m + 1 = bar (m + 1) + 1.
Proof.
  intros m.
  destruct m eqn:E.
  - simpl. reflexivity.
  - simpl. reflexivity.
Qed.
```

这种方法是可行的，不过它依赖于我们能发现隐藏在 bar 中的 `match` 阻碍了证明的进展。

一种更直白的方式就是明确地告诉 Coq 去展开 bar。

```
Fact silly_fact_2' : ∀ m, bar m + 1 = bar (m + 1) + 1.
Proof.
  intros m.
  unfold bar.
```

现在很明显，我们在 $=$ 两边的 `match` 上都卡住了，不用多想就能用 `destruct` 来结束证明。

```
  destruct m eqn:E.
  - reflexivity.
  - reflexivity.
Qed.
```

15.8 对复合表达式使用 destruct

我们已经见过许多通过 `destruct` 进行情况分析来处理一些变量的值了。有时我们需要根据某些'表达式'的结果的情况来进行推理。我们也可以用 `destruct` 来做这件事。

下面是一些例子：

```
Definition sillyfun (n : nat) : bool :=
  if n =? 3 then false
```

```
  else if n =? 5 then false
  else false.

Theorem sillyfun_false : ∀ (n : nat),
  sillyfun n = false.
Proof.
  intros n. unfold sillyfun.
  destruct (n =? 3) eqn:E1.
    - reflexivity.
    - destruct (n =? 5) eqn:E2.
      + reflexivity.
      + reflexivity. Qed.
```

在前面的证明中展开 sillyfun 后，我们发现卡在 `if (n =? 3) then ... else ...` 上了。但由于 n 要么等于 3 要么不等于，因此我们可以用 `destruct (eqb n 3)` 来对这两种情况进行推理。

通常，`destruct` 策略可用于对任何计算结果进行情况分析。如果 `e` 是某个表达式，其类型为归纳定义的类型 T，那么对于 T 的每个构造子 `c`，`destruct e` 都会生成一个子目标，其中（即目标和上下文中）所有的 `e` 都会被替换成 `c`。

练习：3 星, standard, optional (combine_split) 以下是 Poly 一章中出现过的 `split` 函数的实现：

```
Fixpoint split {X Y : Type} (l : list (X×Y))
               : (list X) × (list Y) :=
  match l with
  | [] ⇒ ([], [])
  | (x, y) :: t ⇒
      match split t with
      | (lx, ly) ⇒ (x :: lx, y :: ly)
      end
  end.
```

请证明 `split` 和 combine 在以下概念下互为反函数：

```
Theorem combine_split : ∀ X Y (l : list (X × Y)) l1 l2,
  split l = (l1, l2) →
```

```
  combine l1 l2 = l.
Proof.
    Admitted.
    □
```

`destruct` 策略的 *eqn*: 部分是可选的：目前，我们大部分时间都会包含它，只是为了文档的目的。

然而在用 `destruct` 结构复合表达式时，*eqn*: 记录的信息是十分关键的：如果我们丢弃它，那么 `destruct` 会擦除我们完成证明时所需的信息。

例如，假设函数 sillyfun1 定义如下：

```
Definition sillyfun1 (n : nat) : bool :=
  if n =? 3 then true
  else if n =? 5 then true
  else false.
```

Now suppose that we want to convince Coq that sillyfun1 n yields true only when n is odd. If we start the proof like this (with no *eqn*: on the `destruct`)...

```
Theorem sillyfun1_odd_FAILED : ∀ (n : nat),
     sillyfun1 n = true →
     oddb n = true.
Proof.
  intros n eq. unfold sillyfun1 in eq.
  destruct (n =? 3).
Abort.
```

... then we are stuck at this point because the context does not contain enough information to prove the goal! The problem is that the substitution performed by `destruct` is quite brutal – in this case, it thows away every occurrence of $n =? 3$, but we need to keep some memory of this expression and how it was destructed, because we need to be able to reason that, since $n =? 3 =$ true in this branch of the case analysis, it must be that $n = 3$, from which it follows that n is odd.

What we want here is to substitute away all existing occurences of $n =? 3$, but at the same time add an equation to the context that records which case we are in. This is precisely what the *eqn*: qualifier does.

```
Theorem sillyfun1_odd : ∀ (n : nat),
```

```
      sillyfun1 n = true →
      oddb n = true.
Proof.
  intros n eq. unfold sillyfun1 in eq.
  destruct (n =? 3) eqn:Heqe3.
    - apply eqb_true in Heqe3.
      rewrite → Heqe3. reflexivity.
    -


      destruct (n =? 5) eqn:Heqe5.
        +
          apply eqb_true in Heqe5.
          rewrite → Heqe5. reflexivity.
        + discriminate eq. Qed.
```

练习：2 星, standard (destruct_eqn_practice)

```
Theorem bool_fn_applied_thrice :
  ∀ (f : bool → bool) (b : bool),
  f (f (f b)) = f b.
Proof.
    Admitted.
```

□

15.9 复习

现在我们已经见过 Coq 中最基础的策略了。未来的章节中我们还会介绍更多，之后我们会看到一些更加强大的'自动化'策略，它能让 Coq 帮我们处理底层的细节。不过基本上我们已经有了完成工作所需的东西。

下面是我们已经见过的：

- `intros`：将前提/变量从证明目标移到上下文中
- `reflexivity`：（当目标形如 `e = e` 时）结束证明
- `apply`：用前提、引理或构造子证明目标

- `apply`... `in` H：将前提、引理或构造子应用到上下文中的假设上（正向推理）
- `apply`... `with`...：为无法被模式匹配确定的变量显式指定值
- `simpl`：化简目标中的计算
- `simpl in` H：化简前提中的计算
- `rewrite`：使用相等关系假设（或引理）来改写目标
- `rewrite` ... `in` H：使用相等关系假设（或引理）来改写前提
- `symmetry`：将形如 $t{=}u$ 的目标改为 $u{=}t$
- `symmetry in` H：将形如 $t{=}u$ 的前提改为 $u{=}t$
- `unfold`：用目标中的右式替换定义的常量
- `unfold`... `in` H：用前提中的右式替换定义的常量
- `destruct`... `as`...：对归纳定义类型的值进行情况分析
- `destruct`... *eqn*:...：为添加到上下文中的等式指定名字，记录情况分析的结果
- `induction`... `as`...: 对归纳定义类型的值进行归纳
- `injection`: reason by injectivity on equalities between values of inductively defined types
- `discriminate`: reason by disjointness of constructors on equalities between values of inductively defined types
- `assert` (H: `e`)（或 `assert` (`e`) `as` H）：引入“局部引理”`e` 并称之为 H
- `generalize dependent` x：将变量 x（以及任何依赖它的东西）从上下文中移回目标公式内的前提中

15.10 附加练习

练习：3 星, standard (eqb_sym) Theorem eqb_sym : ∀ (*n m* : **nat**),
(*n* =? *m*) = (*m* =? *n*).

Proof.

Admitted.

□

练习：3 星, advanced, optional (eqb_sym_informal) 根据前面你对该引理的形式化证明，给出与它对应的非形式化证明：

定理：对于任何自然数 *n m*，*n* =? *m* = *m* =? *n*.

证明：

练习：3 星, standard, optional (eqb_trans) Theorem eqb_trans : ∀ *n m p*,
n =? *m* = true →
m =? *p* = true →
n =? *p* = true.

Proof.

Admitted.

□

练习：3 星, advanced (split_combine) 在前面的练习中，我们证明了对于所有序对的列表，combine 是 split 的反函数。你如何形式化陈述 split 是 combine 的反函数？何时此性质成立？

请完成下面 split_combine_statement 的定义，其性质指出 split 是 combine 的反函数。之后，证明该性质成立。（除必要的 intros 之外，不要进行更多的 intros，以此来保证你的归纳假设的一般性。提示：你需要 *l1* 和 *l2* 的什么性质来保证 split (combine *l1 l2*) = (*l1*,*l2*) 成立？）

Definition split_combine_statement : Prop

. *Admitted.*

Theorem split_combine : *split_combine_statement*.

Proof.

Admitted.

```
Definition manual_grade_for_split_combine : option (nat×string) := None.
```

□

练习：3 星, advanced (filter_exercise) 本练习有点难度。为你的归纳假设的形式花点心思。

```
Theorem filter_exercise : ∀ (X : Type) (test : X → bool)
                            (x : X) (l lf : list X),
  filter test l = x :: lf →
  test x = true.
Proof.
  Admitted.
```

□

练习：4 星, advanced, recommended (forall_exists_challenge) 定义两个递归的 *Fixpoints*，forallb 和 existsb。第一个检查列表中的每一个元素是否均满足给定的断言：

forallb oddb 1;3;5;7;9 = true

forallb negb false;false = true

forallb evenb 0;2;4;5 = false

forallb (eqb 5) □ = true

第二个检查列表中是否存在一个元素满足给定的断言：

existsb (eqb 5) 0;2;3;6 = false

existsb (andb true) true;true;false = true

existsb oddb 1;0;0;0;0;3 = true

existsb evenb □ = false

接着，用 forallb 和 negb 定义一个 existsb 的非递归版本，名为 existsb'。

最后，证明定理 existsb_existsb' 指出 existsb' 和 existsb 的行为相同。

```
Fixpoint forallb {X : Type} (test : X → bool) (l : list X) : bool
  . Admitted.

Example test_forallb_1 : forallb oddb [1;3;5;7;9] = true.
Proof. Admitted.

Example test_forallb_2 : forallb negb [false;false] = true.
```

```
Proof. Admitted.

Example test_forallb_3 : forallb evenb [0;2;4;5] = false.
Proof. Admitted.

Example test_forallb_4 : forallb (eqb 5) [] = true.
Proof. Admitted.

Fixpoint existsb {X : Type} (test : X → bool) (l : list X) : bool
  . Admitted.

Example test_existsb_1 : existsb (eqb 5) [0;2;3;6] = false.
Proof. Admitted.

Example test_existsb_2 : existsb (andb true) [true;true;false] = true.
Proof. Admitted.

Example test_existsb_3 : existsb oddb [1;0;0;0;0;3] = true.
Proof. Admitted.

Example test_existsb_4 : existsb evenb [] = false.
Proof. Admitted.

Definition existsb' {X : Type} (test : X → bool) (l : list X) : bool
  . Admitted.

Theorem existsb_existsb' : ∀ (X : Type) (test : X → bool) (l : list X),
  existsb test l = existsb' test l.
Proof. Admitted.

    □
```

Chapter 16

Library LF.Poly

16.1 Poly: 多态与高阶函数

```
Set Warnings "-notation-overridden,-parsing".
From LF Require Export Lists.
```

16.2 多态

在本章中，我们会继续发展函数式编程的基本概念，其中最关键的新概念就是 '多态'（在所处理的数据类型上抽象出函数）和'高阶函数'（函数作为数据）。

16.2.1 多态列表

在上一章中，我们使用了只包含数的列表。很明显，有趣的程序还需要能够处理其它元素类型的列表，如字符串列表、布尔值列表、列表的列表等等。我们'可以'分别为它们定义新的归纳数据类型，例如...

```
Inductive boollist : Type :=
  | bool_nil
  | bool_cons (b : bool) (l : boollist).
```

...不过这样很快就会变得乏味。部分原因在于我们必须为每种数据类型都定义不同的构造子，然而主因还是我们必须为每种数据类型再重新定义一遍所有的列表处理函数（length、rev 等）以及它们所有的性质（rev_length、app_assoc 等）。

为避免这些重复，Coq 支持定义'多态'归纳类型。例如，以下就是'多态列表'数据类型。

```
Inductive list (X:Type) : Type :=
  | nil
  | cons (x : X) (l : list X).
```

这和上一章中 **natlist** 的定义基本一样，只是将 cons 构造子的 **nat** 参数换成了任意的类型 X，函数头的第一行添加了 X 的绑定，而构造子类型中的 **natlist** 则换成了 **list** X。（我们可以重用构造子名 nil 和 cons，因为之前定义的 **natlist** 在当前作用之外的一个 Module 中。）

list 本身又是什么类型？一种不错的思路就是把 **list** 当做从 Type 类型到 Inductive 归纳定义的'函数'；或者换种更简明的思路，即 **list** 是个从 Type 类型到 Type 类型的函数。对于任何特定的类型 X，类型 **list** X 是一个 Inductive 归纳定义的，元素类型为 X 的列表的集合。

```
Check list : Type → Type.
```

list 的定义中的参数 X 自动成为构造子 nil 和 cons 的参数——也就是说，nil 和 cons 在这里是多态的构造子；现在我们调用它们的时候必须要提供一个参数，就是它们要构造的列表的具体类型。例如，nil **nat** 构造的是 **nat** 类型的空列表。

```
Check (nil nat) : list nat.
```

cons **nat** 与此类似，它将类型为 **nat** 的元素添加到类型为 **list nat** 的列表中。以下示例构造了一个只包含自然数 3 的列表：

```
Check (cons nat 3 (nil nat)) : list nat.
```

nil 的类型会是什么呢？也许我们可以（根据定义）说它是 **list** X，不过这样它就不是接受 X 返回 **list** 的函数了。再提出一种：Type → **list** X 并没有解释 X 是什么，(X : Type) → **list** X 则比较接近。 Coq 对这种情况的记法为 ∀ X : Type, **list** X：

```
Check nil : ∀ X : Type, list X.
```

类似地，定义中 cons 的类型看起来像 X → **list** X → **list** X 然而以上述约定来解释 X 的含义则可以得到类型 ∀ X, X → **list** X → **list** X。

```
Check cons : ∀ X : Type, X → list X → list X.
```

（关于记法的附注：在 .v 文件中，量词"forall"会写成字母的形式，而在生成的 HTML 和一些设置了显示控制的 IDE 中，∀ 通常会渲染成一般的"倒 A"数学符号，虽然你偶尔还是会看到英文拼写的"forall"。这只是排版上的效果，它们的含义没有任何区别。）

如果在每次使用列表构造子时，都要为它提供类型参数，那样会很麻烦。不过我们很快就会看到如何省去这种麻烦。

```
Check (cons nat 2 (cons nat 1 (nil nat)))
      : list nat.
```

现在我们可以回过头来定义之前写下的列表处理函数的多态版本了。例如 `repeat`：

```
Fixpoint repeat (X : Type) (x : X) (count : nat) : list X :=
  match count with
  | 0 ⇒ nil X
  | S count' ⇒ cons X x (repeat X x count')
  end.
```

同 nil 与 cons 一样，我们可以通过将 `repeat` 应用到一个类型、一个该类型的元素以及一个数字来使用它：

```
Example test_repeat1 :
  repeat nat 4 2 = cons nat 4 (cons nat 4 (nil nat)).
Proof. reflexivity. Qed.
```

要用 `repeat` 构造其它种类的列表，我们只需通过对应类型的参数将它实例化即可：

```
Example test_repeat2 :
  repeat bool false 1 = cons bool false (nil bool).
Proof. reflexivity. Qed.
```

练习：2 星, standard (mumble_grumble) 考虑以下两个归纳定义的类型：

```
Module MumbleGrumble.

Inductive mumble : Type :=
  | a
  | b (x : mumble) (y : nat)
  | c.

Inductive grumble (X:Type) : Type :=
  | d (m : mumble)
  | e (x : X).
```

对于某个类型 X，以下哪些是 **grumble** X 良定义的元素？（在各选项后填“是”或“否”。）

- d (b a 5)
- d **mumble** (b a 5)
- d **bool** (b a 5)
- e **bool** true
- e **mumble** (b c 0)
- e **bool** (b c 0)
- c

End MumbleGrumble.

Definition manual_grade_for_mumble_grumble : **option** (**nat**×**string**) := None.
□

类型标注的推断

我们再写一遍 repeat 的定义，不过这次不指定任何参数的类型。 Coq 还会接受它么？

```
Fixpoint repeat' X x count : list X :=
  match count with
  | 0 ⇒ nil X
  | S count' ⇒ cons X x (repeat' X x count')
  end.
```

确实会。我们来看看 Coq 赋予了 repeat' 什么类型：

```
Check repeat'
  : ∀ X : Type, X → nat → list X.
Check repeat
  : ∀ X : Type, X → nat → list X.
```

它与 repeat 的类型完全一致。Coq 可以使用'类型推断'基于它们的使用方式来推出 X、x 和 count 一定是什么类型。例如，由于 X 是作为 cons 的参数使用的，因此它必定是个 Type 类型，因为 cons 期望一个 Type 作为其第一个参数，而用 0 和 S 来匹配 count 意味着它必须是个 **nat**，诸如此类。

这种强大的功能意味着我们不必总是在任何地方都显式地写出类型标注，不过显式的类型标注对于文档和完整性检查来说仍然非常有用，因此我们仍会继续使用它。在代码中使用类型标注时，你应当把握好一个度，太多会导致混乱并分散注意力，太少则会迫使读者为理解你的代码而在大脑中进行类型推断。

类型参数的推断

要使用多态函数，我们需要为其参数再额外传入一个或更多类型。例如，前面 `repeat` 函数体中的递归调用必须传递类型 `X`。不过由于 `repeat` 的第二个参数为 `X` 类型的元素，第一个参数明显只能是 `X`，既然如此，我们何必显式地写出它呢？

幸运的是，Coq 允许我们避免这种冗余。在任何我们可以写类型参数的地方，我们都可以将类型参数写为“洞” `_`，可以看做是说“请 Coq 自行找出这里应该填什么。” 更确切地说，当 Coq 遇到 `_` 时，它会尝试'统一'所有的局部变量信息，包括函数应当应用到的类型，其它参数的类型，以及应用函数的上下文中期望的类型，以此来确定 `_` 处应当填入的具体类型。

这听起来很像类型标注推断。实际上，这两种个过程依赖于同样的底层机制。除了简单地忽略函数中某些参数的类型：

repeat' X x count : list X :=

我们还可以将类型换成洞：

repeat' (X : _) (x : _) (count : _) : list X :=

以此来告诉 Coq 要尝试推断出缺少的信息。

Using holes, the `repeat` function can be written like this:

```
Fixpoint repeat'' X x count : list X :=
  match count with
  | 0 ⇒ nil _
  | S count' ⇒ cons _ x (repeat'' _ x count')
  end.
```

在此例中，我们写出 `_` 并没有省略多少 `X`。然而在很多情况下，这对减少击键次数和提高可读性还是很有效的。例如，假设我们要写下一个包含数字 1、2 和 3 的列表，此时不必写成这样：

```
Definition list123 :=
  cons nat 1 (cons nat 2 (cons nat 3 (nil nat))).
```

.....我们可以用洞来这样写：

```
Definition list123' :=
  cons _ 1 (cons _ 2 (cons _ 3 (nil _))).
```

隐式参数

我们甚至可以通过告诉 Coq '总是'推断给定函数的类型参数来在大多数情况下直接避免写 _。

Arguments 用于指令指定函数或构造子的名字并列出其参数名，花括号中的任何参数都会被视作隐式参数。（如果定义中的某个参数没有名字，那么它可以用通配模式 _ 来标记。这种情况常见于构造子中。）

```
Arguments nil {X}.
Arguments cons {X} _ _.
Arguments repeat {X} x count.
```

现在我们完全不用提供类型参数了：

```
Definition list123'' := cons 1 (cons 2 (cons 3 nil)).
```

此外，我们还可以在定义函数时就声明隐式参数，只需要将某个参数两边的圆括号换成花括号。例如：

```
Fixpoint repeat''' {X : Type} (x : X) (count : nat) : list X :=
  match count with
  | 0 ⇒ nil
  | S count' ⇒ cons x (repeat''' x count')
  end.
```

（注意我们现在甚至不必在 repeat''' 的递归调用中提供类型参数了，实际上提供了反而是无效的，因为 Coq 并不想要它。）

我们会尽可能使用最后一种风格，不过还会继续在 `Inductive` 构造子中使用显式的 *Argument* 声明。原因在于如果将归纳类型的形参标为隐式的话，不仅构造子的类型会变成隐式的，类型本身也会变成隐式的。例如，考虑以下 **list** 类型的另一种定义：

```
Inductive list' {X:Type} : Type :=
  | nil'
  | cons' (x : X) (l : list').
```

由于 X 在包括 **list'** 本身的'整个'归纳定义中都是隐式声明的，因此当我们讨论数值、布尔值或其它任何类型的列表时，都只能写 **list'**，而写不了 **list' nat**、**list' bool** 等等，这

样就有点过分了。

作为本节的收尾，我们为新的多态列表重新实现几个其它的标准列表函数...

```
Fixpoint app {X : Type} (l1 l2 : list X)
                : (list X) :=
  match l1 with
  | nil ⇒ l2
  | cons h t ⇒ cons h (app t l2)
  end.

Fixpoint rev {X:Type} (l:list X) : list X :=
  match l with
  | nil ⇒ nil
  | cons h t ⇒ app (rev t) (cons h nil)
  end.

Fixpoint length {X : Type} (l : list X) : nat :=
  match l with
  | nil ⇒ 0
  | cons _ l' ⇒ S (length l')
  end.

Example test_rev1 :
  rev (cons 1 (cons 2 nil)) = (cons 2 (cons 1 nil)).
Proof. reflexivity. Qed.

Example test_rev2:
  rev (cons true nil) = cons true nil.
Proof. reflexivity. Qed.

Example test_length1: length (cons 1 (cons 2 (cons 3 nil))) = 3.
Proof. reflexivity. Qed.
```

显式提供类型参数

用 `Implicit` 将参数声明为隐式的会有个小问题：Coq 偶尔会没有足够的局部信息来确定类型参数。此时，我们需要告诉 Coq 这次我们会显示地给出参数。例如，假设我们写了如下定义：

Fail Definition mynil := nil.

（Definition 前面的 *Fail* 限定符可用于'任何'指令，它的作用是确保该指令在执行时确实会失败。如果该指令失败了，Coq 就会打印出相应的错误信息，不过之后会继续处理文件中剩下的部分。）

在这里，Coq 给出了一条错误信息，因为它不知道应该为 nil 提供何种类型。我们可以为它提供个显式的类型声明来帮助它，这样 Coq 在"应用"nil 时就有更多可用的信息了：

```
Definition mynil : list nat := nil.
```

此外，我们还可以在函数名前加上前缀 @ 来强制将隐式参数变成显式的：

```
Check @nil : ∀ X : Type, list X.

Definition mynil' := @nil nat.
```

使用参数推断和隐式参数，我们可以为列表定义和前面一样的简便记法。由于我们让构造子的的类型参数变成了隐式的，因此 Coq 就知道在我们使用该记法时自动推断它们了。

```
Notation "x :: y" := (cons x y)
                     (at level 60, right associativity).
Notation "[ ]" := nil.
Notation "[ x ; .. ; y ]" := (cons x .. (cons y []) ..).
Notation "x ++ y" := (app x y)
                     (at level 60, right associativity).
```

现在列表就能写成我们希望的方式了：

```
Definition list123''' := [1; 2; 3].
```

练习

练习：2 星, standard, optional (poly_exercises) 下面是一些简单的练习，和 Lists 一章中的一样。为了实践多态，请完成下面的证明。

```
Theorem app_nil_r : ∀ (X:Type), ∀ l:list X,
  l ++ [] = l.
Proof.
  Admitted.
```

```
Theorem app_assoc : ∀ A (l m n:list A),
  l ++ m ++ n = (l ++ m) ++ n.
Proof.
    Admitted.

Lemma app_length : ∀ (X:Type) (l1 l2 : list X),
  length (l1 ++ l2) = length l1 + length l2.
Proof.
    Admitted.
    □
```

练习：2 星, standard, optional (more_poly_exercises) 这儿有些更有趣的东西...

```
Theorem rev_app_distr: ∀ X (l1 l2 : list X),
  rev (l1 ++ l2) = rev l2 ++ rev l1.
Proof.
    Admitted.

Theorem rev_involutive : ∀ X : Type, ∀ l : list X,
  rev (rev l) = l.
Proof.
    Admitted.
    □
```

16.2.2 多态序对

按照相同的模式，我们在上一章中给出的数值序对的定义可被推广为 '多态序对（*Polymorphic Pairs*）'，它通常叫做'积（*Products*）'：

```
Inductive prod (X Y : Type) : Type :=
| pair (x : X) (y : Y).

Arguments pair {X} {Y} _ _.
```

和列表一样，我们也可以将类型参数定义成隐式的，并以此定义类似的具体记法：

```
Notation "( x , y )" := (pair x y).
```

我们也可以使用 `Notation` 来定义标准的'积类型（*Product Types*）'记法：

```
Notation "X * Y" := (prod X Y) : type_scope.
```

（标注：*type_scope* 会告诉 Coq 该缩写只能在解析类型而非表达式时使用。这避免了与乘法符号的冲突。）

一开始会很容易混淆 (*x*,*y*) 和 X×Y。不过要记住 (*x*,*y*) 是一个'值'，它由两个其它的值构造得来；而 X×Y 是一个'类型'，它由两个其它的类型构造得来。如果 *x* 的类型为 X 而 *y* 的类型为 Y，那么 (*x*,*y*) 的类型就是 X×Y。

第一元（first）和第二元（second）的射影函数（Projection Functions）现在看起来和其它函数式编程语言中的很像了：

```
Definition fst {X Y : Type} (p : X × Y) : X :=
  match p with
  | (x, y) ⇒ x
  end.

Definition snd {X Y : Type} (p : X × Y) : Y :=
  match p with
  | (x, y) ⇒ y
  end.
```

以下函数接受两个列表，并将它们结合成一个序对的列表。在其它函数式语言中，它通常被称作 *zip*。我们为了与 Coq 的标准库保持一致，将它命名为 combine。

```
Fixpoint combine {X Y : Type} (lx : list X) (ly : list Y)
                 : list (X×Y) :=
  match lx, ly with
  | [], _ ⇒ []
  | _, [] ⇒ []
  | x :: tx, y :: ty ⇒ (x, y) :: (combine tx ty)
  end.
```

练习：1 星, standard, optional (combine_checks) 请尝试在纸上回答以下问题并在 Coq 中检验你的解答：

- combine 的类型是什么？（即 `Check @combine` 会打印出什么？）
- 以下指令会打印出什么？

  ```
  Compute (combine 1;2 false;false;true;true).
  ```

□

练习：2 星, standard, recommended (split) 函数 `split` 是 combine 的右逆（right inverse）：它接受一个序对的列表并返回一个列表的序对。在很多函数式语言中，它被称作 *unzip*。

请在下面完成 `split` 的定义，确保它能够通过给定的单元测试。

```
Fixpoint split {X Y : Type} (l : list (X×Y))
                : (list X) × (list Y)
  . Admitted.

Example test_split:
  split [(1,false);(2,false)] = ([1;2],[false;false]).
Proof.
    Admitted.
```

□

16.2.3 多态候选

最后一种要介绍的多态类型是'多态候选（*Polymorphic Options*）', 它推广了上一章中的 **natoption**（由于我们之后要用标准库中定义的 **option** 版本，因此这里的定义我们把它放在模块中）：

```
Module OptionPlayground.

Inductive option (X:Type) : Type :=
  | Some (x : X)
  | None.

Arguments Some {X} _.
Arguments None {X}.

End OptionPlayground.
```

现在我们可以重写 nth_error 函数来让它适用于任何类型的列表了。

```
Fixpoint nth_error {X : Type} (l : list X) (n : nat)
                   : option X :=
  match l with
  | [] ⇒ None
```

```
  | a :: l' ⇒ if n =? O then Some a else nth_error l' (pred n)
  end.

Example test_nth_error1 : nth_error [4;5;6;7] 0 = Some 4.
Proof. reflexivity. Qed.
Example test_nth_error2 : nth_error [[1];[2]] 1 = Some [2].
Proof. reflexivity. Qed.
Example test_nth_error3 : nth_error [true] 2 = None.
Proof. reflexivity. Qed.
```

练习：1 星, standard, optional (hd_error_poly) 请完成上一章中 hd_error 的多态定义，确保它能通过下方的单元测试。

```
Definition hd_error {X : Type} (l : list X) : option X
  . Admitted.
```

再说一遍，要强制将隐式参数转为显式参数，我们可以在函数名前使用 @。

```
Check @hd_error : ∀ X : Type, list X → option X.

Example test_hd_error1 : hd_error [1;2] = Some 1.
  Admitted.
Example test_hd_error2 : hd_error [[1];[2]] = Some [1].
  Admitted.
```

□

16.3 函数作为数据

和大部分现代编程语言一样，特别是“函数式”的语言，包括 OCaml、Haskell、Racket、Scala、Clojure 等，Coq 也将函数视作“一等公民（First-Class Citizens）”，即允许将它们作为参数传入其它函数、作为结果返回、以及存储在数据结构中等等。

16.3.1 高阶函数

用于操作其它函数的函数通常叫做'高阶函数'。以下是简单的示例：

```
Definition doit3times {X:Type} (f:X→X) (n:X) : X :=
  f (f (f n)).
```

这里的参数 f 本身也是个（从 X 到 X 的）函数， doit3times 的函数体将 f 对某个值 n 应用三次。

```
Check @doit3times : ∀ X : Type, (X → X) → X → X.

Example test_doit3times: doit3times minustwo 9 = 3.
Proof. reflexivity. Qed.

Example test_doit3times': doit3times negb true = false.
Proof. reflexivity. Qed.
```

16.3.2 过滤器

下面是个更有用的高阶函数，它接受一个元素类型为 X 的列表和一个 X 的谓词（即一个从 X 到 **bool** 的函数），然后“过滤”此列表并返回一个新列表，其中仅包含对该谓词返回 true 的元素。

```
Fixpoint filter {X:Type} (test: X→bool) (l:list X)
                : (list X) :=
  match l with
  | [] ⇒ []
  | h :: t ⇒ if test h then h :: (filter test t)
                       else filter test t
  end.
```

例如，如果我们将 filter 应用到谓词 evenb 和一个数值列表 l 上，那么它就会返回一个只包含 l 中偶数的列表。

```
Example test_filter1: filter evenb [1;2;3;4] = [2;4].
Proof. reflexivity. Qed.

Definition length_is_1 {X : Type} (l : list X) : bool :=
  (length l) =? 1.

Example test_filter2:
    filter length_is_1
           [ [1; 2]; [3]; [4]; [5;6;7]; []; [8] ]
  = [ [3]; [4]; [8] ].
Proof. reflexivity. Qed.
```

我们可以使用 filter 给出 Lists 中 countoddmembers 函数的简洁的版本。

```
Definition countoddmembers' (l:list nat) : nat :=
  length (filter oddb l).

Example test_countoddmembers'1: countoddmembers' [1;0;3;1;4;5] = 4.
Proof. reflexivity. Qed.
Example test_countoddmembers'2: countoddmembers' [0;2;4] = 0.
Proof. reflexivity. Qed.
Example test_countoddmembers'3: countoddmembers' nil = 0.
Proof. reflexivity. Qed.
```

16.3.3 匿名函数

在上面这个例子中，我们不得不定义一个名为 length_is_1 的函数，以便让它能够作为参数传入到 filter 中，由于该函数可能再也用不到了，这有点令人沮丧。我们经常需要传入“一次性”的函数作为参数，之后不会再用，而为每个函数取名是十分无聊的。

幸运的是，有一种更好的方法。我们可以按需随时构造函数而不必在顶层中声明它或给它取名。

```
Example test_anon_fun':
  doit3times (fun n ⇒ n × n) 2 = 256.
Proof. reflexivity. Qed.
```

表达式 (fun $n \Rightarrow n \times n$) 可读作“一个给定 n 并返回 $n \times n$ 的函数。”

以下为使用匿名函数重写的 filter 示例：

```
Example test_filter2':
    filter (fun l ⇒ (length l) =? 1)
           [ [1; 2]; [3]; [4]; [5;6;7]; []; [8] ]
  = [ [3]; [4]; [8] ].
Proof. reflexivity. Qed.
```

练习：2 星, standard (filter_even_gt7) 使用 filter（而非 Fixpoint）来编写 Coq 函数 filter_even_gt7，它接受一个自然数列表作为输入，返回一个只包含大于 7 的偶数的列表。

```
Definition filter_even_gt7 (l : list nat) : list nat
```

. *Admitted.*

```
Example test_filter_even_gt7_1 :
  filter_even_gt7 [1;2;6;9;10;3;12;8] = [10;12;8].
```

Admitted.

```
Example test_filter_even_gt7_2 :
  filter_even_gt7 [5;2;6;19;129] = [].
```

Admitted.

□

练习：3 星, standard (partition) 使用 filter 编写一个 Coq 函数 partition：

partition : forall X : Type, (X -> bool) -> list X -> list X * list X

给定一个集合 X、一个类型为 X → **bool** 的断言和一个 **list** X， partition 应当返回一个列表的序对。该序对的第一个成员为包含原始列表中满足该测试的子列表，而第二个成员为包含不满足该测试的元素的子列表。两个子列表中元素的顺序应当与它们在原始列表中的顺序相同。

```
Definition partition {X : Type}
                     (test : X → bool)
                     (l : list X)
                   : list X × list X
```

. *Admitted.*

```
Example test_partition1: partition oddb [1;2;3;4;5] = ([1;3;5], [2;4]).
```

Admitted.

```
Example test_partition2: partition (fun x ⇒ false) [5;9;0] = ([], [5;9;0]).
```

Admitted.

□

16.3.4 映射

另一个方便的高阶函数叫做 map。

```
Fixpoint map {X Y: Type} (f:X→Y) (l:list X) : (list Y) :=
  match l with
  | [] ⇒ []
```

```
  | h :: t ⇒ (f h) :: (map f t)
  end.
```

它接受一个函数 f 和一个列表 $l = [n1, n2, n3, ...]$ 并返回列表 $[f\ n1, f\ n2, f\ n3, ...]$，其中 f 可分别应用于 l 中的每一个元素。例如：

```
Example test_map1: map (fun x ⇒ plus 3 x) [2;0;2] = [5;3;5].
Proof. reflexivity. Qed.
```

输入列表和输出列表的元素类型不必相同，因为 map 会接受'两个'类型参数 X 和 Y，因此它可以应用到一个数值的列表和一个从数值到布尔值的函数，并产生一个布尔值列表：

```
Example test_map2:
  map oddb [2;1;2;5] = [false;true;false;true].
Proof. reflexivity. Qed.
```

它甚至可以应用到一个数值的列表和一个从数值到布尔值列表的函数，并产生一个布尔值的'列表的列表'：

```
Example test_map3:
    map (fun n ⇒ [evenb n;oddb n]) [2;1;2;5]
  = [[true;false];[false;true];[true;false];[false;true]].
Proof. reflexivity. Qed.
```

习题

练习：3 星, standard (map_rev) 请证明 map 和 rev 可交换。你可能需要定义一个辅助引理

```
Theorem map_rev : ∀ (X Y : Type) (f : X → Y) (l : list X),
  map f (rev l) = rev (map f l).
Proof.
    Admitted.
```

□

练习：2 星, standard, recommended (flat_map) 函数 map 通过一个类型为 X → Y 的函数将 **list** X 映射到 **list** Y。我们可以定义一个类似的函数 flat_map，它通过一个类型

为 X → **list** Y 的函数 f 将 **list** X 映射到 **list** Y。你的定义应当可以"压扁"f 的结果，就像这样：

```
flat_map (fun n => n;n+1;n+2) 1;5;10 = 1; 2; 3; 5; 6; 7; 10; 11; 12.
```

```
Fixpoint flat_map {X Y: Type} (f: X → list Y) (l: list X)
                  : (list Y)
  . Admitted.

Example test_flat_map1:
  flat_map (fun n ⇒ [n;n;n]) [1;5;4]
  = [1; 1; 1; 5; 5; 5; 4; 4; 4].
    Admitted.
```

□

map 这个函数不止对列表有意义，以下是一个在 **option** 上的 map：

```
Definition option_map {X Y : Type} (f : X → Y) (xo : option X)
                      : option Y :=
  match xo with
    | None ⇒ None
    | Some x ⇒ Some (f x)
  end.
```

练习：2 星, standard, optional (implicit_args) filter 和 map 的定义和应用在很多地方使用了隐式参数。请将隐式参数外层的花括号替换为圆括号，然后在必要的地方补充显式类型形参并用 Coq 检查你做的是否正确。（本练习并不会打分，你可以在本文件的'副本'中做它，之后丢掉即可。） □

16.3.5 折叠

一个更加强大的高阶函数叫做 `fold`。本函数启发自"*reduce* 归约" 操作，它是 Google 的 map/reduce 分布式编程框架的核心。

```
Fixpoint fold {X Y: Type} (f: X→Y→Y) (l: list X) (b: Y)
                          : Y :=
  match l with
  | nil ⇒ b
  | h :: t ⇒ f h (fold f t b)
```

```
end.
```

直观上来说，`fold` 操作的行为就是将给定的二元操作符 f 插入到给定列表的每一对元素之间。例如， `fold plus` [1;2;3;4] 直观上的意思是 1+2+3+4。为了让它更精确，我们还需要一个"起始元素" 作为 f 初始的第二个输入。因此，例如

fold plus 1;2;3;4 0

就会产生

$1 + (2 + (3 + (4 + 0)))$.

以下是更多例子：

```
Check (fold andb) : list bool → bool → bool.

Example fold_example1 :
  fold mult [1;2;3;4] 1 = 24.
Proof. reflexivity. Qed.

Example fold_example2 :
  fold andb [true;true;false;true] true = false.
Proof. reflexivity. Qed.

Example fold_example3 :
  fold app [[1];[];[2;3];[4]] [] = [1;2;3;4].
Proof. reflexivity. Qed.
```

练习：1 星, advanced (fold_types_different) 我们观察到 `fold` 由 X 和 Y 这'两个'类型变量参数化，形参 f 则是个接受 X 和 Y 并返回 Y 的二元操作符。你能想出一种 X 和 Y 不同时的应用情景吗？

```
Definition manual_grade_for_fold_types_different : option (nat×string) := None.
```

□

16.3.6 用函数构造函数

目前我们讨论过的大部分高阶函数都是接受函数作为参数的。现在我们来看一些将函数作为其它函数的结果'返回'的例子。首先，下面是一个接受值 x（由某个类型 X 刻画）并返回一个从 **nat** 到 X 的函数，当它被调用时总是产生 x 并忽略其 **nat** 参数。

```
Definition constfun {X: Type} (x: X) : nat→X :=
  fun (k:nat) ⇒ x.
```

```
Definition ftrue := constfun true.

Example constfun_example1 : ftrue 0 = true.
Proof. reflexivity. Qed.

Example constfun_example2 : (constfun 5) 99 = 5.
Proof. reflexivity. Qed.
```

实际上，我们已经见过的多参函数也是讲函数作为数据传入的例子。为了理解为什么，请回想 plus 的类型。

```
Check plus : nat → nat → nat.
```

该表达式中的每个 → 实际上都是一个类型上的'二元'操作符。该操作符是'右结合'的，因此 plus 的类型其实是 **nat** → (**nat** → **nat**) 的简写，即，它可以读作"plus 是一个单参数函数，它接受一个 **nat** 并返回另一个函数，该函数接受另一个 **nat** 并返回一个 **nat**"。在上面的例子中，我们总是将 plus 一次同时应用到两个参数上。不过如果我们喜欢，也可以一次只提供一个参数，这叫做'偏应用（*Partial Application*）'。

```
Definition plus3 := plus 3.
Check plus3 : nat → nat.

Example test_plus3 : plus3 4 = 7.
Proof. reflexivity. Qed.
Example test_plus3' : doit3times plus3 0 = 9.
Proof. reflexivity. Qed.
Example test_plus3'' : doit3times (plus 3) 0 = 9.
Proof. reflexivity. Qed.
```

16.4 附加练习

```
Module Exercises.
```

练习：2 星, standard (fold_length) 列表的很多通用函数都可以通过 `fold` 来实现。例如，下面是 length 的另一种实现：

```
Definition fold_length {X : Type} (l : list X) : nat :=
  fold (fun _ n ⇒ S n) l 0.

Example test_fold_length1 : fold_length [4;7;0] = 3.
```

```
Proof. reflexivity. Qed.
```

请证明 fold_length 的正确性。（提示：知道 `reflexivity` 的化简力度比 `simpl` 更大或许会有所帮助。也就是说，你或许会遇到 `simpl` 无法解决但 `reflexivity` 可以解决的目标。）

```
Theorem fold_length_correct : ∀ X (l : list X),
  fold_length l = length l.
Proof.
```

Admitted.

□

练习：3 星，standard (fold_map) 我们也可以用 `fold` 来定义 `map`。请完成下面的 fold_map。

```
Definition fold_map {X Y: Type} (f: X → Y) (l: list X) : list Y
  . Admitted.
```

在 Coq 中写出 *fold_map_correct* 来陈述 fold_map 是正确的，然后证明它。（提示：再次提醒，`reflexivity` 的化简力度比 `simpl` 更强。）

```
Definition manual_grade_for_fold_map : option (nat×string) := None.
```

□

练习：2 星，advanced (currying) 在 Coq 中，函数 f : A → B → C 的类型其实是 A → (B → C)。也就是说，如果给 f 一个类型为 A 的值，它就会给你函数 f' : B → C。如果再给 f' 一个类型为 B 的值，它就会返回一个类型为 C 的值。这为我们提供了 plus3 中的那种偏应用能力。用返回函数的函数处理参数列表的方式被称为'柯里化（*Currying*）'，它是为了纪念逻辑学家 Haskell Curry。

反之，我们也可以将 A → B → C 解释为 (A × B) → C。这叫做'反柯里化（*Uncurrying*）'。对于反柯里化的二元函数，两个参数必须作为序对一次给出，此时它不会偏应用。

我们可以将柯里化定义如下：

```
Definition prod_curry {X Y Z : Type}
  (f : X × Y → Z) (x : X) (y : Y) : Z := f (x, y).
```

作为练习，请定义它的反函数 prod_uncurry，然后在下面证明它们互为反函数的定理。

```
Definition prod_uncurry {X Y Z : Type}
  (f : X → Y → Z) (p : X × Y) : Z
  . Admitted.
```

举一个柯里化用途的（平凡的）例子，我们可以用它来缩短之前看到的一个例子：

```
Example test_map1': map (plus 3) [2;0;2] = [5;3;5].
Proof. reflexivity. Qed.
```

思考练习：在运行以下指令之前，你能计算出 prod_curry 和 prod_uncurry 的类型吗？

```
Check @prod_curry.
Check @prod_uncurry.

Theorem uncurry_curry : ∀ (X Y Z : Type)
                        (f : X → Y → Z)
                        x y,
  prod_curry (prod_uncurry f) x y = f x y.
Proof.
  Admitted.

Theorem curry_uncurry : ∀ (X Y Z : Type)
                        (f : (X × Y) → Z) (p : X × Y),
  prod_uncurry (prod_curry f) p = f p.
Proof.
  Admitted.
```

□

练习：2 星, advanced (nth_error_informal) 回想 nth_error 函数的定义：

```
Fixpoint nth_error {X : Type} (l : list X) (n : nat) : option X := match l with | □ => None | a :: l' => if n =? O then Some a else nth_error l' (pred n) end.
```

请写出以下定理的非形式化证明：

```
forall X l n, length l = n -> @nth_error X l n = None
```

```
Definition manual_grade_for_informal_proof : option (nat×string) := None.
```

□

本练习使用'邱奇数（*Church numerals*）'探讨了另一种定义自然数的方式，它以数学家 Alonzo Church 命名。我们可以将自然数 n 表示为一个函数，它接受一个函数 f 作为参数并返回迭代了 n 次的 f。

```
Module CHURCH.
Definition cnat := ∀ X : Type, (X → X) → X → X.
```

我们来看看如何用这种记法写数。将函数迭代一次应当与将它应用一次相同。因此：

```
Definition one : cnat :=
  fun (X : Type) (f : X → X) (x : X) ⇒ f x.
```

与此类似，two 应当对其参数应用两次 f：

```
Definition two : cnat :=
  fun (X : Type) (f : X → X) (x : X) ⇒ f (f x).
```

定义 zero 有点刁钻：我们如何"将函数应用零次"？答案很简单：把参数原样返回就好。

```
Definition zero : cnat :=
  fun (X : Type) (f : X → X) (x : X) ⇒ x.
```

更一般地说，数 n 可以写作 `fun X f x ⇒ f (f ... (f x) ...)`，其中 f 出现了 n 次。要特别注意我们之前定义 doit3times 函数的方式就是 3 的邱奇数表示。

```
Definition three : cnat := @doit3times.
```

完成以下函数的定义。请用 `reflexivity` 证明来确认它们能够通过对应的单元测试。

练习：1 星, advanced (church_succ) 自然数的后继：给定一个邱奇数 n，它的后继 succ n 是一个把它的参数比 n 还多迭代一次的函数。 `Definition succ (n : cnat) : cnat . Admitted.`

```
Example succ_1 : succ zero = one.
Proof. Admitted.

Example succ_2 : succ one = two.
Proof. Admitted.

Example succ_3 : succ two = three.
Proof. Admitted.
```

□

练习：1 星, advanced (church_plus) 两邱奇数相加： `Definition plus (n m : cnat) : cnat`

. *Admitted*.

```
Example plus_1 : plus zero one = one.
Proof. Admitted.

Example plus_2 : plus two three = plus three two.
Proof. Admitted.

Example plus_3 :
  plus (plus two two) three = plus one (plus three three).
Proof. Admitted.
```

□

练习：2 星, advanced (church_mult) 乘法： `Definition mult (n m : cnat) : cnat`

. *Admitted*.

```
Example mult_1 : mult one one = one.
Proof. Admitted.

Example mult_2 : mult zero (plus three three) = zero.
Proof. Admitted.

Example mult_3 : mult two three = plus three three.
Proof. Admitted.
```

□

练习：2 星, advanced (church_exp) 乘方：

（'提示'：多态在这里起到了关键的作用。然而，棘手之处在于选择正确的类型来迭代。如果你遇到了「Universe inconsistency，全域不一致」错误，请在不同的类型上迭代。在 cnat 本身上迭代通常会有问题。）

```
Definition exp (n m : cnat) : cnat
```

. *Admitted*.

```
Example exp_1 : exp two two = plus two two.
Proof. Admitted.

Example exp_2 : exp three zero = one.
Proof. Admitted.
```

```
Example exp_3 : exp three two = plus (mult two (mult two two)) one.
Proof. Admitted.
```

□

End CHURCH.

End EXERCISES.

Chapter 17

Library LF.Lists

17.1 Lists: 使用结构化的数据

```
From LF Require Export Induction.
Module NatList.
```

17.2 数值序对

在 Inductive 类型定义中，每个构造子（Constructor）可以有任意多个参数—— 可以没有（如 true 和 O），可以只有一个（如 S），也可以更多（如 **nybble**，以及下文所示）：

```
Inductive natprod : Type :=
| pair (n1 n2 : nat).
```

此声明可以读作："构造数值序对的唯一一种方法，就是将构造子 pair 应用到两个 **nat** 类型的参数上。"

```
Check (pair 3 5) : natprod.
```

下述函数分别用于提取二元组中的第一个和第二个分量。

```
Definition fst (p : natprod) : nat :=
  match p with
  | pair x y ⇒ x
  end.
```

```
Definition snd (p : natprod) : nat :=
  match p with
  | pair x y ⇒ y
  end.

Compute (fst (pair 3 5)).
```

由于二元组十分常用，不妨以标准的数学记法 (x,y) 取代 pair x y。通过 Notation 向 Coq 声明该记法：

```
Notation "( x , y )" := (pair x y).
```

The new notation can be used both in expressions and in pattern matches.

```
Compute (fst (3,5)).

Definition fst' (p : natprod) : nat :=
  match p with
  | (x,y) ⇒ x
  end.

Definition snd' (p : natprod) : nat :=
  match p with
  | (x,y) ⇒ y
  end.

Definition swap_pair (p : natprod) : natprod :=
  match p with
  | (x,y) ⇒ (y,x)
  end.
```

Note that pattern-matching on a pair (with parentheses: (x, y)) is not to be confused with the "multiple pattern" syntax (with no parentheses: x, y) that we have seen previously. The above examples illustrate pattern matching on a pair with elements x and y, whereas, for example, the definition of minus in Basics performs pattern matching on the values n and m:

Fixpoint minus (n m : nat) : nat := match n, m with | O , _=> O | S _, O => n | S n', S m' => minus n' m' end.

The distinction is minor, but it is worth knowing that they are not the same. For instance, the following definitions are ill-formed:

Definition bad_fst (p : natprod) : nat := match p with | x, y => x end.

Definition bad_minus (n m : nat) : nat := match n, m with | (O , _) => O | (S _, O) => n | (S n', S m') => bad_minus n' m' end.

现在我们来证明一些有关二元组的简单事实。

如果我们以稍显古怪的方式陈述序对的性质，那么有时只需 `reflexivity`（及其内建的简化）即可完成证明。

```
Theorem surjective_pairing' : ∀ (n m : nat),
  (n,m) = (fst (n,m), snd (n,m)).
Proof.
  reflexivity. Qed.
```

但是，如果我们用一种更为自然的方式来陈述此引理的话，只用 `reflexivity` 还不够。

```
Theorem surjective_pairing_stuck : ∀ (p : natprod),
  p = (fst p, snd p).
Proof.
  simpl. Abort.
```

我们还需要向 Coq 展示 p 的具体结构，这样 `simpl` 才能对 fst 和 snd 做模式匹配。通过 `destruct` 可以达到这个目的。

```
Theorem surjective_pairing : ∀ (p : natprod),
  p = (fst p, snd p).
Proof.
  intros p. destruct p as [n m]. simpl. reflexivity. Qed.
```

注意：不同于解构自然数产生两个子目标，`destruct` 在此只产生一个子目标。这是因为 **natprod** 只有一种构造方法。

练习：1 星, standard (snd_fst_is_swap) `Theorem snd_fst_is_swap : ∀ (p : natprod), (snd p, fst p) = swap_pair p. Proof. Admitted.`

□

练习：1 星, standard, optional (fst_swap_is_snd) `Theorem fst_swap_is_snd : ∀ (p : natprod),`

```
  fst (swap_pair p) = snd p.
Proof.
```

Admitted.

□

17.3 数值列表

通过推广序对的定义，数值'列表'类型可以这样描述："一个列表要么是空的，要么就是由一个数和另一个列表组成的序对。"

```
Inductive natlist : Type :=
  | nil
  | cons (n : nat) (l : natlist).
```

例如，这是一个三元素列表：

```
Definition mylist := cons 1 (cons 2 (cons 3 nil)).
```

和序对一样，使用熟悉的编程记法来表示列表会更方便些。以下两个声明能让我们用 :: 作为中缀的 cons 操作符，用方括号作为构造列表的"外围（outfix）"记法。

```
Notation "x :: l" := (cons x l)
                     (at level 60, right associativity).
Notation "[ ]" := nil.
Notation "[ x ; .. ; y ]" := (cons x .. (cons y nil) ..).
```

我们不必完全理解这些声明，但如果你感兴趣的话，我会大致说明一下发生了什么。注解 "`right associativity`" 告诉 Coq 当遇到多个 :: 时如何给表达式加括号，如此一来下面三个声明做的就是同一件事：

```
Definition mylist1 := 1 :: (2 :: (3 :: nil)).
Definition mylist2 := 1 :: 2 :: 3 :: nil.
Definition mylist3 := [1;2;3].
```

"`at level 60`" 告诉 Coq 当遇到表达式和其它中缀运算符时应该如何加括号。例如，我们已经为 plus 函数定义了 + 中缀符号，它的优先级是 50：

Notation "x + y" := (plus x y) (at level 50, left associativity).

+ 会比 :: 结合的更紧密，所以 1 + 2 :: [3] 会被解析成 (1 + 2) :: [3] 而非 1 + (2 :: [3])。

(当你在 .v 文件中看到"1 + (2 :: [3])"这样的记法时可能会很疑惑。最里面那个括住了 3 的方括号，标明了它是一个列表。而外层的方括号则是用来指示"coqdoc"这部分要被显示为代码而非普通的文本；在生成的 HTML 文件中，外层的方括号是看不到的。)

上面的第二和第三个 Notation 声明引入了标准的方括号记法来表示列表；第三个声明的右半部分展示了在 Coq 中声明 n 元记法的语法，以及将它们翻译成嵌套的二元构造子序列的方法。

Repeat

接下来我们看几个用来构造和操作列表的函数。第一个是 repeat 函数，它接受一个数字 n 和一个 count，返回一个长度为 count，每个元素都是 n 的列表。

```
Fixpoint repeat (n count : nat) : natlist :=
  match count with
  | O ⇒ nil
  | S count' ⇒ n :: (repeat n count')
  end.
```

Length

length 函数用来计算列表的长度。

```
Fixpoint length (l:natlist) : nat :=
  match l with
  | nil ⇒ O
  | h :: t ⇒ S (length t)
  end.
```

Append

app 函数用来把两个列表联接起来。

```
Fixpoint app (l1 l2 : natlist) : natlist :=
  match l1 with
  | nil ⇒ l2
```

```
  | h :: t ⇒ h :: (app t l2)
  end.
```

由于下文中 app 随处可见，不妨将其定义为中缀运算符。

```
Notation "x ++ y" := (app x y)
                     (right associativity, at level 60).

Example test_app1: [1;2;3] ++ [4;5] = [1;2;3;4;5].
Proof. reflexivity. Qed.
Example test_app2: nil ++ [4;5] = [4;5].
Proof. reflexivity. Qed.
Example test_app3: [1;2;3] ++ nil = [1;2;3].
Proof. reflexivity. Qed.
```

Head 与 Tail

下面介绍列表上的两种运算：hd 函数返回列表的第一个元素（即“表头”）； tl 函数返回列表除去第一个元素以外的部分（即“表尾”）。由于空表没有表头，我们必须传入一个参数作为返回的默认值。

```
Definition hd (default:nat) (l:natlist) : nat :=
  match l with
  | nil ⇒ default
  | h :: t ⇒ h
  end.

Definition tl (l:natlist) : natlist :=
  match l with
  | nil ⇒ nil
  | h :: t ⇒ t
  end.

Example test_hd1: hd 0 [1;2;3] = 1.
Proof. reflexivity. Qed.
Example test_hd2: hd 0 [] = 0.
Proof. reflexivity. Qed.
Example test_tl: tl [1;2;3] = [2;3].
```

```
Proof. reflexivity. Qed.
```

练习

练习：2 星, standard, recommended (list_funs) 完成以下 nonzeros、oddmembers 和 countoddmembers 的定义，你可以查看测试函数来理解这些函数应该做什么。

```
Fixpoint nonzeros (l:natlist) : natlist
  . Admitted.

Example test_nonzeros:
  nonzeros [0;1;0;2;3;0;0] = [1;2;3].
    Admitted.

Fixpoint oddmembers (l:natlist) : natlist
  . Admitted.

Example test_oddmembers:
  oddmembers [0;1;0;2;3;0;0] = [1;3].
    Admitted.

Definition countoddmembers (l:natlist) : nat
  . Admitted.

Example test_countoddmembers1:
  countoddmembers [1;0;3;1;4;5] = 4.
    Admitted.

Example test_countoddmembers2:
  countoddmembers [0;2;4] = 0.
    Admitted.

Example test_countoddmembers3:
  countoddmembers nil = 0.
    Admitted.
```

□

练习：3 星, advanced (alternate) 完成以下 alternate 的定义，它从两个列表中交替地取出元素并合并为一个列表，就像把拉链“拉”起来一样。更多具体示例见后面的测试。

（注意：alternate 有一种自然而优雅的定义，但是这一定义无法满足 Coq 对于 `Fixpoint` 必须“显然会终止”的要求。如果你发现你被这种解法束缚住了，可以试着换一种稍微啰嗦一点的解法，比如同时对两个列表中的元素进行操作。有种可行的解法需要定义新的序对，但这并不是唯一的方法。）

```
Fixpoint alternate (l1 l2 : natlist) : natlist
  . Admitted.

Example test_alternate1:
  alternate [1;2;3] [4;5;6] = [1;4;2;5;3;6].
    Admitted.

Example test_alternate2:
  alternate [1] [4;5;6] = [1;4;5;6].
    Admitted.

Example test_alternate3:
  alternate [1;2;3] [4] = [1;4;2;3].
    Admitted.

Example test_alternate4:
  alternate [] [20;30] = [20;30].
    Admitted.
    □
```

17.3.1 用列表实现口袋（Bag）

bag（或者叫 *multiset* 多重集）类似于集合，只是其中每个元素都能出现不止一次。口袋的一种可行的表示是列表。

```
Definition bag := natlist.
```

练习：3 星, standard, recommended (bag_functions) 为袋子完成以下 count、sum、add、和 member 函数的定义。

```
Fixpoint count (v:nat) (s:bag) : nat
  . Admitted.
```

这些命题都能通过 `reflexivity` 来证明。

```
Example test_count1: count 1 [1;2;3;1;4;1] = 3.
```
Admitted.
```
Example test_count2: count 6 [1;2;3;1;4;1] = 0.
```
Admitted.

Multiset sum is similar to set *union*: sum a b contains all the elements of a and of b. (Mathematicians usually define *union* on multisets a little bit differently – using max instead of sum – which is why we don't call this operation *union*.) For sum, we're giving you a header that does not give explicit names to the arguments. Moreover, it uses the keyword `Definition` instead of `Fixpoint`, so even if you had names for the arguments, you wouldn't be able to process them recursively. The point of stating the question this way is to encourage you to think about whether sum can be implemented in another way – perhaps by using one or more functions that have already been defined.

```
Definition sum : bag → bag → bag
```
. *Admitted.*
```
Example test_sum1: count 1 (sum [1;2;3] [1;4;1]) = 3.
```
Admitted.
```
Definition add (v:nat) (s:bag) : bag
```
. *Admitted.*
```
Example test_add1: count 1 (add 1 [1;4;1]) = 3.
```
Admitted.
```
Example test_add2: count 5 (add 1 [1;4;1]) = 0.
```
Admitted.
```
Definition member (v:nat) (s:bag) : bool
```
. *Admitted.*
```
Example test_member1: member 1 [1;4;1] = true.
```
Admitted.
```
Example test_member2: member 2 [1;4;1] = false.
```
Admitted.

□

练习：3 星, standard, optional (bag_more_functions) 你可以把下面这些和 bag 有

关的函数当作额外的练习

倘若某口袋不包含所要移除的数字，那么将 remove_one 作用其上不应改变其内容。（本练习为选做，但高级班的学生为了完成后面的练习，需要写出 remove_one 的定义。）

```
Fixpoint remove_one (v:nat) (s:bag) : bag
  . Admitted.

Example test_remove_one1:
  count 5 (remove_one 5 [2;1;5;4;1]) = 0.
    Admitted.

Example test_remove_one2:
  count 5 (remove_one 5 [2;1;4;1]) = 0.
    Admitted.

Example test_remove_one3:
  count 4 (remove_one 5 [2;1;4;5;1;4]) = 2.
    Admitted.

Example test_remove_one4:
  count 5 (remove_one 5 [2;1;5;4;5;1;4]) = 1.
    Admitted.

Fixpoint remove_all (v:nat) (s:bag) : bag
  . Admitted.

Example test_remove_all1: count 5 (remove_all 5 [2;1;5;4;1]) = 0.
    Admitted.
Example test_remove_all2: count 5 (remove_all 5 [2;1;4;1]) = 0.
    Admitted.
Example test_remove_all3: count 4 (remove_all 5 [2;1;4;5;1;4]) = 2.
    Admitted.
Example test_remove_all4: count 5 (remove_all 5 [2;1;5;4;5;1;4;5;1;4]) = 0.
    Admitted.

Fixpoint subset (s1:bag) (s2:bag) : bool
  . Admitted.

Example test_subset1: subset [1;2] [2;1;4;1] = true.
    Admitted.
```

```
Example test_subset2: subset [1;2;2] [2;1;4;1] = false.
```

Admitted.

□

练习：2 星, standard, recommended (bag_theorem) 写一个你认为有趣的关于袋子的定理 *bag_theorem*，然后证明它；这个定理需要用到 count 和 add。注意，这是个开放性问题。也许你写下的定理是正确的，但它可能会涉及到你尚未学过的技巧因而无法证明。如果你遇到麻烦了，欢迎提问！

```
Definition manual_grade_for_bag_theorem : option (nat×string) := None.
```

□

17.4 有关列表的论证

和数字一样，有些列表处理函数的简单事实仅通过化简就能证明。例如，对于下面这个例子，`reflexivity` 所做的简化就已经足够了...

```
Theorem nil_app : ∀ l:natlist,
  [] ++ l = l.
Proof. reflexivity. Qed.
```

...由于 [] 被替换进了 app 定义中相应的“被检”分支（即经由匹配“仔细检查”过值的表达式），整个匹配得以被简化。

和数字一样，有时对一个列表做分类讨论（是否是空）是非常有用的。

```
Theorem tl_length_pred : ∀ l:natlist,
  pred (length l) = length (tl l).
Proof.
  intros l. destruct l as [| n l'].
  -
    reflexivity.
  -
    reflexivity. Qed.
```

在这里 nil 的情况能够工作是因为我们定义了 tl nil = nil，而 `destruct` 策略中 `as` 注解引入的两个名字，n 和 l'，分别对应了 cons 构造子的两个参数（正在构造的列表的头和尾）。

然而一般来说，许多关于列表的有趣定理都需要用到归纳法来证明，接下来我们就会看到证明的方法。

（一点点说教：随着不断地深入，若你只是'阅读'证明的话，并不会获得什么特别有用的东西。搞清楚每一个细节非常重要，你应该在 Coq 中单步执行这些证明并思考每一步在整个证明中的作用，否则练习题将毫无用处。啰嗦完毕。）

17.4.1 对列表进行归纳

比起对自然数的归纳，读者可能对归纳证明 **natlist** 这样的数据类型更加陌生。不过基本思路同样简单。每个 Inductive 声明定义了一组数据值，这些值可以用声明过的构造子来构造。例如，布尔值可以用 true 或 false 来构造；自然数可以用 O 或 S 应用到另一个自然数上来构造；而列表可以用 nil 或者将 cons 应用到一个自然数和另一个列表上来构造。除此以外，归纳定义的集合中元素的形式 '只能是' 构造子对其它项的应用。

这一事实同时也给出了一种对归纳定义的集合进行论证的方法：一个自然数要么是 O，要么就是 S 应用到某个'更小'的自然数上；一个列表要么是 nil，要么就是 cons 应用到某个数字和某个'更小'的列表上，诸如此类。所以，如果我们有某个命题 P 涉及列表 l，而我们想证明 P 对 '一切' 列表都成立，那么可以像这样推理：

- 首先，证明当 l 为 nil 时 $P\ l$ 成立。
- 然后，证明当 l 为 cons $n\ l'$ 时 $P\ l$ 成立，其中 n 是某个自然数，l' 是某个更小的列表，假设 $P\ l'$ 成立.

由于较大的列表总是能够分解为较小的列表，最终这个较小的列表会变成 nil，这两点合在一起就完成了 P 对一切列表 l 成立的证明。下面是个具体的例子：

```
Theorem app_assoc : ∀ l1 l2 l3 : natlist,
  (l1 ++ l2) ++ l3 = l1 ++ (l2 ++ l3).
Proof.
  intros l1 l2 l3. induction l1 as [| n l1' IHl1'].
  -
    reflexivity.
  -
    simpl. rewrite → IHl1'. reflexivity. Qed.
```

注意，和归纳自然数时一样，此处 induction 策略的 as... 从句为在"$l1$ 由构造子 cons 构造而来"这一情况时出现的"更小的列表"和归纳假设取了名字。

再次强调，如果你把 Coq 的证明当做静态的文档，那么可能不会有特别多的收获——如果你通过交互式的 Coq 会话来阅读证明，就能看到当前的目标和上下文，而这些状态在你阅读写下来的脚本时是不可见的。所以一份用自然语言写成的证明—— 写给人看的——需要包含更多的提示来帮助读者理解当前的状态，比如第二种情况下的归纳假设到底是什么。

'定理'：对所有的列表 *l1*, *l2*, 和 *l3*， $(l1 \mathrel{++} l2) \mathrel{++} l3 = l1 \mathrel{++} (l2 \mathrel{++} l3)$。

'证明': 通过对 *l1* 使用归纳法。

- 首先, 假设 $l1 = []$。我们必须证明：

  $(\square \mathrel{++} l2) \mathrel{++} l3 = \square \mathrel{++} (l2 \mathrel{++} l3)$,

  这可以通过展开 $++$ 的定义得到。

- 然后, 假设 $l1 = n::l1'$，有：

  $(l1' \mathrel{++} l2) \mathrel{++} l3 = l1' \mathrel{++} (l2 \mathrel{++} l3)$

  （归纳假设）。我们必须证明：

  $((n :: l1') \mathrel{++} l2) \mathrel{++} l3 = (n :: l1') \mathrel{++} (l2 \mathrel{++} l3)$.

  根据 $++$ 的定义, 上式等价于：

  $n :: ((l1' \mathrel{++} l2) \mathrel{++} l3) = n :: (l1' \mathrel{++} (l2 \mathrel{++} l3))$,

  该式可通过我们的归纳假设立即证得。 □

反转列表

举一个更加深入的例子来说明对列表的归纳证明：假设我们使用 app 来定义一个列表反转函数 rev：

```
Fixpoint rev (l:natlist) : natlist :=
  match l with
  | nil ⇒ nil
  | h :: t ⇒ rev t ++ [h]
  end.

Example test_rev1: rev [1;2;3] = [3;2;1].
Proof. reflexivity. Qed.
```

```
Example test_rev2: rev nil = nil.
Proof. reflexivity. Qed.
```

为了比目前所见的证明多一点挑战性，我们来证明反转一个列表不会改变它的长度。我们的首次尝试在后继这一分支上卡住了....

```
Theorem rev_length_firsttry : ∀ l : natlist,
  length (rev l) = length l.
Proof.
  intros l. induction l as [| n l' IHl'].
  -
    reflexivity.
  -


    simpl.
    rewrite ← IHl'.
Abort.
```

不妨单独提出引理，阐述 ++ 与 length 形成的等式关系，以从我们卡住的地方推进证明。

```
Theorem app_length : ∀ l1 l2 : natlist,
  length (l1 ++ l2) = (length l1) + (length l2).
Proof.
  intros l1 l2. induction l1 as [| n l1' IHl1'].
  -
    reflexivity.
  -
    simpl. rewrite → IHl1'. reflexivity. Qed.
```

注意，为了让该引理尽可能 '通用'，我们不仅关心由 rev 得到的列表，还要对 '所有' 的 **natlist** 进行全称量化。这很自然，因为这个证明目标显然不依赖于被反转的列表。除此之外，证明这个更普遍的性质也更容易些。

现在我们可以完成最初的证明了。

```
Theorem rev_length : ∀ l : natlist,
  length (rev l) = length l.
Proof.
```

```
  intros l. induction l as [| n l' IHl'].
  -
    reflexivity.
  -
    simpl. rewrite → app_length.
    simpl. rewrite → IHl'. rewrite plus_comm.
    reflexivity.
Qed.
```

作为对比，以下是这两个定理的非形式化证明：
'定理'：对于所有的列表 *l1* 和 *l2*， length (*l1* ++ *l2*) = length *l1* + length *l2*.
'证明'：对 *l1* 进行归纳。

- 首先，假设 *l1* = []。我们必须证明

  length (□ ++ l2) = length □ + length l2,

  根据 length、++ 和 plus 的定义，上式显然可得。

- 其次，假设 *l1* = *n*::*l1'*，并且

  length (l1' ++ l2) = length l1' + length l2.

  我们必须证明

  length ((n::l1') ++ l2) = length (n::l1') + length l2.

  根据 length 和 ++ 的定义以及归纳假设，上式显然可得。 □

'定理': 对于所有的列表 *l*，length (rev *l*) = length *l*。
'证明': 对 *l* 进行归纳。

- 首先，假设 *l* = []。我们必须证明

  length (rev □) = length □,

  根据 length 和 rev 的定义，上式显然可得。

- 其次，假设 *l* = *n*::*l'*，并且

  length (rev l') = length l'.

  我们必须证明

length (rev (n :: l')) = length (n :: l').

根据 rev 的定义，上式来自于

length ((rev l') ++ n) = S (length l')

根据之前的引理，此式等同于

length (rev l') + length n = S (length l').

根据归纳假设和 length 的定义，上式显然可得。 □

这些证明的风格实在是冗长而迂腐。读多了之后，我们会发现减少细枝末节，详述不太显然的步骤更有助于我们理解证明。毕竟细节更容易在大脑中思考，必要时我们还可以在草稿纸上补全。下面我们以一种更加紧凑的方式呈现之前的证明：

'定理'：对于所有 l，length (rev l) = length l。

'证明'：首先，观察到 length (l ++ [n]) = S (length l) 对一切 l 成立，这一点可通过对 l 的归纳直接得到。当 l = n'::l' 时，通过再次对 l 使用归纳，然后同时使用之前观察得到的性质和归纳假设即可证明。 □

一般而言，在不同的情况下合适的风格也会不同：读者对这个问题了解程度，以及当前的证明与读者熟悉的证明之间的相似度都会影响到这一点。对于我们现在的目的而言，最好先用更加冗长的方式。

17.4.2 Search 搜索

我们已经见过很多需要使用之前证明过的结论（例如通过 `rewrite`）来证明的定理了。但是在引用别的定理时，我们必须事先知道它们的名字。当然，即使是已被证明的定理本身我们都不能全部记住，更不用提它们的名字了。

Coq 的 `Search` 指令在这时就非常有用了。执行 `Search foo` 会让 Coq 显示所有涉及到 foo 的定理。例如，去掉下面的注释后，你会看到一个我们证明过的所有关于 rev 的定理的列表：

在接下来的学习中，你要记得使用 `Search`，它能为你节约大量的时间!

如果你正在使用 ProofGeneral，那么可以用 C-c C-a C-a 来运行 `Search`。通过 C-c C-; 可以将它返回的结果粘贴到缓冲区内。

17.4.3 列表练习，第一部分

练习：3 星, standard (list_exercises) 更多有关列表的实践：

```
Theorem app_nil_r : ∀ l : natlist,
  l ++ [] = l.
Proof.
    Admitted.

Theorem rev_app_distr: ∀ l1 l2 : natlist,
  rev (l1 ++ l2) = rev l2 ++ rev l1.
Proof.
    Admitted.

Theorem rev_involutive : ∀ l : natlist,
  rev (rev l) = l.
Proof.
    Admitted.
```

下面的练习有简短的解法，如果你开始发现情况已经复杂到你无法理清的程度，请后退一步并试着寻找更为简单的方法。

```
Theorem app_assoc4 : ∀ l1 l2 l3 l4 : natlist,
  l1 ++ (l2 ++ (l3 ++ l4)) = ((l1 ++ l2) ++ l3) ++ l4.
Proof.
    Admitted.
```

一个关于你对 nonzeros 的实现的练习：

```
Lemma nonzeros_app : ∀ l1 l2 : natlist,
  nonzeros (l1 ++ l2) = (nonzeros l1) ++ (nonzeros l2).
Proof.
    Admitted.
    □
```

练习：2 星, standard (eqblist) 填写 eqblist 的定义，它通过比较列表中的数字来判断是否相等。证明对于所有列表 l，eqblist l l 返回 true。

```
Fixpoint eqblist (l1 l2 : natlist) : bool
  . Admitted.

Example test_eqblist1 :
  (eqblist nil nil = true).
```

Admitted.

```
Example test_eqblist2 :
  eqblist [1;2;3] [1;2;3] = true.
```

Admitted.

```
Example test_eqblist3 :
  eqblist [1;2;3] [1;2;4] = false.
```

Admitted.

```
Theorem eqblist_refl : ∀ l:natlist,
  true = eqblist l l.
Proof.
```

Admitted.

□

17.4.4 列表练习, 第二部分

下面这组简单的定理用于证明你之前关于袋子的定义。

练习：1 星, standard (count_member_nonzero) `Theorem count_member_nonzero : ∀ (s : bag),`

```
  1 <=? (count 1 (1 :: s)) = true.
Proof.
```

Admitted.

□

下面这条关于 leb 的引理可助你完成下一个证明。

```
Theorem leb_n_Sn : ∀ n,
  n <=? (S n) = true.
Proof.
  intros n. induction n as [| n' IHn'].
  -
    simpl. reflexivity.
  -
    simpl. rewrite IHn'. reflexivity. Qed.
```

Before doing the next exercise, make sure you've filled in the definition of remove_one above.

练习：3 星, advanced (remove_does_not_increase_count) Theorem remove_does_not_increase_coun

```
∀ (s : bag),
  (count 0 (remove_one 0 s)) <=? (count 0 s) = true.
Proof.
```

Admitted.

□

练习：3 星, standard, optional (bag_count_sum) 写下一个用到函数 count 和 sum 的，关于袋子的有趣定理 *bag_count_sum*，然后证明它。（你可能会发现该证明的难度取决于你如何定义 count!）

练习：4 星, advanced (rev_injective) 求证 rev 是单射函数，即：

```
forall (l1 l2 : natlist), rev l1 = rev l2 -> l1 = l2.
```

（这个问题既可以用简单的方式解决也可以用繁琐的方式来解决。）

```
Definition manual_grade_for_rev_injective : option (nat×string) := None.
```

□

17.5 Options 可选类型

假设我们想要写一个返回某个列表中第 n 个元素的函数。如果我们为它赋予类型 **nat** → **natlist** → **nat**，那么当列表太短时我们仍须返回某个数...

```
Fixpoint nth_bad (l:natlist) (n:nat) : nat :=
  match l with
  | nil ⇒ 42
  | a :: l' ⇒ match n with
                | 0 ⇒ a
                | S n' ⇒ nth_bad l' n'
                end
  end.
```

这种方案并不好：如果 nth_bad 返回了 42，那么不经过进一步处理的话，我们无法得知该值是否真的出现在了输入中。（译注：我们无法判断是什么因素让它返回了 42，因为它可能是列表过短时的返回值，同时也可能是（此时列表足够长）在列表中找到的值）一种更好的方式是改变 nth_bad 的返回类型，使其包含一个错误值作为可能的结果。我们将此类型命名为 **natoption**。

```
Inductive natoption : Type :=
  | Some (n : nat)
  | None.
```

然后我们可以修改前面 nth_bad 的定义，使其在列表太短时返回 None，在列表足够长且 a 在 n 处时返回 Some a。我们将这个新函数称为 nth_error 来表明它可以产生带错误的结果。

```
Fixpoint nth_error (l:natlist) (n:nat) : natoption :=
  match l with
  | nil ⇒ None
  | a :: l' ⇒ match n with
                | O ⇒ Some a
                | S n' ⇒ nth_error l' n'
                end
  end.

Example test_nth_error1 : nth_error [4;5;6;7] 0 = Some 4.
Proof. reflexivity. Qed.
Example test_nth_error2 : nth_error [4;5;6;7] 3 = Some 7.
Proof. reflexivity. Qed.
Example test_nth_error3 : nth_error [4;5;6;7] 9 = None.
Proof. reflexivity. Qed.
```

（在 HTML 版本中隐藏了这些老套的证明。若你想看它请点击小方格。）

本例也是个介绍 Coq 编程语言更多细微特性的机会，比如条件表达式...

```
Fixpoint nth_error' (l:natlist) (n:nat) : natoption :=
  match l with
  | nil ⇒ None
  | a :: l' ⇒ if n =? O then Some a
                else nth_error' l' (pred n)
```

```
  end.
```

Coq 的条件语句和其它语言中的一样，不过加上了一点更为一般化的特性。由于 **bool** 类型不是内建的，因此 Coq 实际上支持在'任何'带有两个构造子的，归纳定义的类型上使用条件表达式。当断言（guard）求值为 `Inductive` 定义中的第一个构造子时，它被认为是真的；当它被求值到第二个构造子时，则被认为是假的。

以下函数从 **natoption** 中取出一个 **nat**，在遇到 None 时它将返回提供的默认值。

```
Definition option_elim (d : nat) (o : natoption) : nat :=
  match o with
  | Some n' ⇒ n'
  | None ⇒ d
  end.
```

练习：2 星, standard (hd_error) 用同样的思路修正之前的 hd 函数，使我们无需为 nil 的情况提供默认元素。

```
Definition hd_error (l : natlist) : natoption
  . Admitted.

Example test_hd_error1 : hd_error [] = None.
    Admitted.

Example test_hd_error2 : hd_error [1] = Some 1.
    Admitted.

Example test_hd_error3 : hd_error [5;6] = Some 5.
    Admitted.
```

□

练习：1 星, standard, optional (option_elim_hd) 此练习能帮助你在新的 hd_error 和旧的 hd 之间建立联系。

```
Theorem option_elim_hd : ∀ (l:natlist) (default:nat),
  hd default l = option_elim default (hd_error l).
Proof.
    Admitted.
```

□

```
End NatList.
```

17.6 偏映射（Partial Maps）

最后演示一下如何在 Coq 中定义基础的数据结构。这是一个简单的 '偏映射' 数据类型，它类似于大多数编程语言中的映射或字典数据结构。

首先，我们定义一个新的归纳数据类型 **id** 来用作偏映射的"键"。

```
Inductive id : Type :=
  | Id (n : nat).
```

本质上来说，**id** 只是一个数。但通过 Id 标签封装自然数来引入新的类型，能让定义变得更加可读，同时也给了我们更多的灵活性。

我们还需要一个 **id** 的相等关系测试：

```
Definition eqb_id (x1 x2 : id) :=
  match x1, x2 with
  | Id n1, Id n2 ⇒ n1 =? n2
  end.
```

练习：1 星, standard (eqb_id_refl) `Theorem eqb_id_refl : ∀ x, true = eqb_id x x.`

```
Proof.
```

Admitted.

□

现在我们定义偏映射的类型：

```
Module PartialMap.
Export NatList.

Inductive partial_map : Type :=
  | empty
  | record (i : id) (v : nat) (m : partial_map).
```

此声明可以读作："有两种方式可以构造一个 **partial_map**：用构造子 empty 表示一个空的偏映射，或将构造子 record 应用到一个键、一个值和一个既有的 **partial_map** 来构造一个带"键-值"映射的 **partial_map**。"

update 函数在部分映射中覆盖给定的键以取缔原值（如该键尚不存在，则新建其记录）。

```
Definition update (d : partial_map)
                  (x : id) (value : nat)
```

```
                    : partial_map :=
  record x value d.
```

最后，find 函数按照给定的键搜索一个 **partial_map**。若该键无法找到，它就返回 None；若该键与 *val* 相关联，则返回 Some *val*。若同一个键被映到多个值，find 就会返回它遇到的第一个值。

```
Fixpoint find (x : id) (d : partial_map) : natoption :=
  match d with
  | empty ⇒ None
  | record y v d' ⇒ if eqb_id x y
                    then Some v
                    else find x d'
  end.
```

练习：1 星, standard (update_eq)

```
Theorem update_eq :
  ∀ (d : partial_map) (x : id) (v: nat),
    find x (update d x v) = Some v.
Proof.
    Admitted.
    □
```

练习：1 星, standard (update_neq)

```
Theorem update_neq :
  ∀ (d : partial_map) (x y : id) (o: nat),
    eqb_id x y = false → find x (update d y o) = find x d.
Proof.
    Admitted.
    □ End PartialMap.
```

练习：2 星, standard (baz_num_elts) 考虑以下归纳定义：

```
Inductive baz : Type :=
  | Baz1 (x : baz)
  | Baz2 (y : baz) (b : bool).
```

有'多少'个表达式具备类型 **baz**？（以注释说明。）

```
Definition manual_grade_for_baz_num_elts : option (nat×string) := None.
```

□

Chapter 18

Library LF.Induction

18.1 Induction: 归纳证明

在开始之前，我们需要把上一章中所有的定义都导入进来：

```
From LF Require Export Basics.
```

For the `Require Export` to work, Coq needs to be able to find a compiled version of *Basics.v*, called *Basics.vo*, in a directory associated with the prefix *LF*. This file is analogous to the *.class* files compiled from *.java* source files and the *.o* files compiled from .c files.

First create a file named *_CoqProject* containing the following line (if you obtained the whole volume "Logical Foundations" as a single archive, a *_CoqProject* should already exist and you can skip this step):

-Q . LF

This maps the current directory ("."， which contains *Basics.v*, *Induction.v*, etc.) to the prefix (or "logical directory") "*LF*". PG and CoqIDE read *_CoqProject* automatically, so they know to where to look for the file *Basics.vo* corresponding to the library LF.Basics.

Once *_CoqProject* is thus created, there are various ways to build *Basics.vo*:

- In Proof General: The compilation can be made to happen automatically when you submit the `Require` line above to PG, by setting the emacs variable *coq-compile-before-require* to *t*.
- In CoqIDE: Open *Basics.v*; then, in the "Compile" menu, click on "Compile Buffer".

- From the command line: Generate a *Makefile* using the *coq_makefile* utility, that comes installed with Coq (if you obtained the whole volume as a single archive, a *Makefile* should already exist and you can skip this step):

  *coq_makefile -f _CoqProject *.v -o Makefile*

  Note: You should rerun that command whenever you add or remove Coq files to the directory.

  Then you can compile *Basics.v* by running *make* with the corresponding *.vo* file as a target:

  make Basics.vo

  All files in the directory can be compiled by giving no arguments:

  make

  Under the hood, *make* uses the Coq compiler, *coqc*. You can also run *coqc* directly:

  coqc -Q . LF Basics.v

  But *make* also calculates dependencies between source files to compile them in the right order, so *make* should generally be prefered over explicit *coqc*.

如果你遇到了问题（例如，稍后你可能会在本文件中遇到缺少标识符的提示），那可能是因为没有正确设置 Coq 的"加载路径"。指令 `Print` *LoadPath*. 能帮你理清这类问题。

特别是，如果你看到了像这样的信息：

Compiled library Foo makes inconsistent assumptions over library Bar

check whether you have multiple installations of Coq on your machine. It may be that commands (like *coqc*) that you execute in a terminal window are getting a different version of Coq than commands executed by Proof General or CoqIDE.

- Another common reason is that the library *Bar* was modified and recompiled without also recompiling *Foo* which depends on it. Recompile *Foo*, or everything if too many files are affected. (Using the third solution above: *make clean*; *make*.)

再给 CoqIDE 用户一点技巧：如果你看到了 *Error*: *Unable to locate library* Basics，那么可能的原因是用 *'CoqIDE'* 编译的代码和在指令行用 *'coqc'* 编译的不一致。这通常在系统中安装了两个不兼容的 *coqc* 时发生（一个与 CoqIDE 关联，另一个与指令行的 *coqc* 关联）。这种情况的变通方法就是只使用 CoqIDE 来编译（即从菜单中选择"make"）并完全避免使用 *coqc*。

18.2 归纳法证明

我们在上一章中通过基于化简的简单论据证明了 0 是 + 的左幺元。我们也观察到，当我们打算证明 0 也是 + 的 '右' 幺元时...

```
Theorem plus_n_O_firsttry : ∀ n:nat,
  n = n + 0.
```

...事情就没这么简单了。只应用 `reflexivity` 的话不起作用，因为 $n + 0$ 中的 n 是任意未知数，所以在 + 的定义中 `match` 匹配无法被化简。

```
Proof.
  intros n.
  simpl. Abort.
```

即便用 `destruct n` 分类讨论也不会有所改善：诚然，我们能够轻易地证明 $n = 0$ 时的情况；但在证明对于某些 n' 而言 $n = S\ n'$ 时，我们又会遇到和此前相同的问题。

```
Theorem plus_n_O_secondtry : ∀ n:nat,
  n = n + 0.
Proof.
  intros n. destruct n as [| n'] eqn:E.
  -
    reflexivity.   -
    simpl. Abort.
```

虽然还可以用 `destruct n'` 再推进一步，但由于 n 可以任意大，如果照这个思路继续证明的话，我们永远也证不完。

为了证明这种关于数字、列表等归纳定义的集合的有趣事实，我们通常需要一个更强大的推理原理：'归纳'。

回想一下 '自然数的归纳法则'，你也许曾在高中的数学课上，在某门离散数学课上或在其它类似的课上学到过它：若 $P(n)$ 为关于自然数的命题，而当我们想要证明 P 对于所有自然数 n 都成立时，可以这样推理：

- 证明 $P(O)$ 成立；
- 证明对于任何 n'，若 $P(n')$ 成立，那么 $P(S\ n')$ 也成立。
- 最后得出 $P(n)$ 对于所有 n 都成立的结论。

在 Coq 中的步骤也一样：我们以证明 $P(n)$ 对于所有 n 都成立的目标开始，然后（通过应用 induction 策略）把它分为两个子目标：一个是我们必须证明 $P(\mathsf{O})$ 成立，另一个是我们必须证明 $P(n') \to P(\mathsf{S}\ n')$。下面就是对该定理的用法：

```
Theorem plus_n_O : ∀ n:nat, n = n + 0.
Proof.
  intros n. induction n as [| n' IHn'].
  - reflexivity.
  - simpl. rewrite ← IHn'. reflexivity. Qed.
```

和 destruct 一样，induction 策略也能通过 as... 从句为引入到子目标中的变量指定名字。由于这次有两个子目标，因此 as... 有两部分，用 | 隔开。（严格来说，我们可以忽略 as... 从句，Coq 会为它们选择名字。然而在实践中这样不好，因为让 Coq 自行选择名字的话更容易导致理解上的困难。）

在第一个子目标中 n 被 0 所取代。由于没有新的变量会被引入，因此 as ... 字句的第一部分为空；而当前的目标会变成 $0 + 0 = 0$：使用化简就能得到此结论。

在第二个子目标中，n 被 $\mathsf{S}\ n'$ 所取代，而对 n' 的归纳假设（Inductive Hypothesis），即 $n' + 0 = n'$ 则以 IHn' 为名被添加到了上下文中。这两个名字在 as... 从句的第二部分中指定。在此上下文中，待证目标变成了 $(\mathsf{S}\ n') + 0 = \mathsf{S}\ n'$；它可被化简为 $\mathsf{S}\ (n' + 0) = \mathsf{S}\ n'$，而此结论可通过 IHn' 得出。

```
Theorem minus_diag : ∀ n,
  minus n n = 0.
Proof.
  intros n. induction n as [| n' IHn'].
  -
    simpl. reflexivity.
  -
    simpl. rewrite → IHn'. reflexivity. Qed.
```

（其实在这些证明中我们并不需要 intros：当 induction 策略被应用到包含量化变量的目标中时，它会自动将需要的变量移到上下文中。）

练习：2 星, standard, recommended (basic_induction) 用归纳法证明以下命题。你可能需要之前的证明结果。

```
Theorem mult_0_r : ∀ n:nat,
```

```
  n × 0 = 0.
Proof.
    Admitted.

Theorem plus_n_Sm : ∀ n m : nat,
  S (n + m) = n + (S m).
Proof.
    Admitted.

Theorem plus_comm : ∀ n m : nat,
  n + m = m + n.
Proof.
    Admitted.

Theorem plus_assoc : ∀ n m p : nat,
  n + (m + p) = (n + m) + p.
Proof.
    Admitted.
    □
```

练习：2 星, standard (double_plus) 考虑以下函数，它将其参数乘以二：

```
Fixpoint double (n:nat) :=
  match n with
  | O ⇒ O
  | S n' ⇒ S (S (double n'))
  end.
```

用归纳法证明以下关于 double 的简单事实：

```
Lemma double_plus : ∀ n, double n = n + n .
Proof.
    Admitted.
    □
```

练习：2 星, standard, optional (evenb_S) 我们的 evenb n 定义对 n - 2 的递归调用不大方便。这让证明 evenb n 时更难对 n 进行归纳，因此我们需要一个关于 n - 2 的归纳假设。以下引理赋予了 evenb (S n) 另一个特征，使其在归纳时能够更好地工作：

```
Theorem evenb_S : ∀ n : nat,
  evenb (S n) = negb (evenb n).
Proof.
    Admitted.
    □
```

练习：1 星, standard, optional (destruct_induction) 请简要说明一下 `destruct` 策略和 `induction` 策略之间的区别。

```
Definition manual_grade_for_destruct_induction : option (nat×string) := None.
```

□

18.3 证明里的证明

和在非形式化的数学中一样，在 Coq 中，大的证明通常会被分为一系列定理，后面的定理引用之前的定理。但有时一个证明会需要一些繁杂琐碎的事实，而这些事实缺乏普遍性，以至于我们甚至都不应该给它们单独取顶级的名字。此时，如果能仅在需要时简单地陈述并立即证明所需的"子定理"就会很方便。我们可以用 `assert` 策略来做到。例如，我们之前对 *mult_0_plus* 定理的证明引用了前一个名为 plus_O_n 的定理，而我们只需内联使用 `assert` 就能陈述并证明 plus_O_n：

```
Theorem mult_0_plus' : ∀ n m : nat,
  (0 + n) × m = n × m.
Proof.
  intros n m.
  assert (H: 0 + n = n). { reflexivity. }
  rewrite → H.
  reflexivity. Qed.
```

`assert` 策略引入两个子目标。第一个为断言本身，通过给它加前缀 H: 我们将该断言命名为 H。（当然也可以用 `as` 来命名该断言，与之前的 `destruct` 和 `induction` 一样。例如 `assert` $(0 + n = n)$ `as` H。）注意我们用花括号 { ... } 将该断言的证明括了起来。这样不仅方便阅读，同时也能在交互使用 Coq 时更容易看出该子目标何时得证。第二个目标与之前执行 `assert` 时一样，只是这次在上下文中，我们有了名为 H 的前提 $0 + n = n$。也就是说，`assert` 生成的第一个子目标是我们必须证明的已断言的事实，而在第二个

子目标中，我们可以使用已断言的事实在一开始尝试证明的事情上取得进展。

另一个 assert 的例子...

举例来说，假如我们要证明 $(n + m) + (p + q) = (m + n) + (p + q)$。 = 两边唯一不同的就是内层第一个子式中 + 的参数 m 和 n 交换了位置，我们似乎可以用加法交换律（plus_comm）来改写它。然而， rewrite 策略并不知道应该作用在 '哪里'。本命题中 + 用了三次，结果 rewrite → plus_comm 只对 '最外层' 起了作用...

```
Theorem plus_rearrange_firsttry : ∀ n m p q : nat,
  (n + m) + (p + q) = (m + n) + (p + q).
Proof.
  intros n m p q.
  rewrite → plus_comm.
Abort.
```

为了在需要的地方使用 plus_comm，我们可以（为此这里讨论的 m 和 n）引入一个局部引理来陈述 $n + m = m + n$，之后用 plus_comm 证明它，并用它来进行期望的改写。

```
Theorem plus_rearrange : ∀ n m p q : nat,
  (n + m) + (p + q) = (m + n) + (p + q).
Proof.
  intros n m p q.
  assert (H: n + m = m + n).
  { rewrite → plus_comm. reflexivity. }
  rewrite → H. reflexivity. Qed.
```

18.4 形式化证明 vs. 非形式化证明

"'非形式化证明是算法，形式化证明是代码。'"

数学声明的成功证明由什么构成？这个问题已经困扰了哲学家数千年，不过这儿有个还算凑合的定义：数学命题 P 的证明是一段书面（或口头）的文本，它对 P 的真实性进行无可辩驳的论证，逐步说服读者或听者使其确信 P 为真。也就是说，证明是一种交流行为。

交流活动会涉及不同类型的读者。一方面，"读者"可以是像 Coq 这样的程序，此时灌输的"确信"是 P 能够从一个确定的，由形式化逻辑规则组成的集合中机械地推导出来，而证明则是指导程序检验这一事实的方法。这种方法就是 '形式化' 证明。

另一方面，读者也可以是人类，这种情况下证明可以用英语或其它自然语言写出，因此必然是 '非形式化' 的，此时成功的标准不太明确。一个"有效的"证明是让读者相信 P。但同一个证明可能被很多不同的读者阅读，其中一些人可能会被某种特定的表述论证方式说服，而其他人则不会。有些读者太爱钻牛角尖，或者缺乏经验，或者只是单纯地过于愚钝；说服他们的唯一方法就是细致入微地进行论证。不过熟悉这一领域的读者可能会觉得所有细节都太过繁琐，让他们无法抓住整体的思路；他们想要的不过是抓住主要思路，因为相对于事无巨细的描述而言，让他们自行补充所需细节更为容易。总之，我们没有一个通用的标准，因为没有一种编写非形式化证明的方式能够说服所能顾及的每一个读者。

然而在实践中，数学家们已经发展出了一套用于描述复杂数学对象的约定和习语，这让交流（至少在特定的社区内）变得十分可靠。这种约定俗成的交流形式已然成风，它为证明的好坏给出了清晰的判断标准。

由于我们在本课程中使用 Coq，因此会重度使用形式化证明。但这并不意味着我们可以完全忽略掉非形式化的证明过程！形式化证明在很多方面都非常有用，不过它们对人类之间的思想交流而言 '并不' 十分高效。

例如，下面是一段加法结合律的证明：

```
Theorem plus_assoc' : ∀ n m p : nat,
  n + (m + p) = (n + m) + p.
Proof. intros n m p. induction n as [| n' IHn']. reflexivity.
  simpl. rewrite → IHn'. reflexivity. Qed.
```

Coq 对此表示十分满意。然而人类却很难理解它。我们可以用注释和标号让它的结构看上去更清晰一点...

```
Theorem plus_assoc'' : ∀ n m p : nat,
  n + (m + p) = (n + m) + p.
Proof.
  intros n m p. induction n as [| n' IHn'].
  -
    reflexivity.
  -
    simpl. rewrite → IHn'. reflexivity. Qed.
```

...而且如果你习惯了 Coq，你可能会在脑袋里逐步过一遍策略，并想象出每一处上下文和目标栈的状态。不过若证明再复杂一点，那就几乎不可能了。

一个（迂腐的）数学家可能把证明写成这样：

- '定理': 对于任何 n、m 和 p，

  n + (m + p) = (n + m) + p.

  '证明': 对 n 使用归纳法。

  - 首先，设 $n = 0$。我们必须证明

    0 + (m + p) = (0 + m) + p.

    此结论可从 + 的定义直接得到。
  - 然后，设 n = S n'，其中

    n' + (m + p) = (n' + m) + p.

    我们必须证明

    (S n') + (m + p) = ((S n') + m) + p.

    根据 + 的定义，该式可写成

    S (n' + (m + p)) = S ((n' + m) + p),

    它由归纳假设直接得出。'证毕'。

证明的总体形式大体类似，当然这并非偶然：Coq 的设计使其 `induction` 策略会像数学家写出的标号那样，按相同的顺序生成相同的子目标。但在细节上则有明显的不同：形式化证明在某些方面更为明确（例如对 `reflexivity` 的使用），但在其它方面则不够明确（特别是 Coq 证明中任何一处的"证明状态"都是完全隐含的，而非形式化证明则经常反复告诉读者目前证明进行的状态）。

练习：2 星, advanced, recommended (plus_comm_informal) 将你对 plus_comm 的解答翻译成非形式化证明：

定理：加法满足交换律。

Proof:

```
Definition manual_grade_for_plus_comm_informal : option (nat×string) := None.
```

□

练习：2 星, standard, optional (eqb_refl_informal) 以 plus_assoc 的非形式化证明为范本，写出以下定理的非形式化证明。不要只是用中文来解释 Coq 策略！

定理：对于任何 n，均有 true = n =? n。

证明： □

18.5 更多练习

练习：3 星, standard, recommended (mult_comm) 用 `assert` 来帮助证明此定理。你应该不需要对 plus_swap 进行归纳。

```
Theorem plus_swap : ∀ n m p : nat,
  n + (m + p) = m + (n + p).
Proof.
    Admitted.
```

现在证明乘法交换律。（你在证明过程中可能想要定义并证明一个辅助定理。提示：$n \times (1 + k)$ 是什么？）

```
Theorem mult_comm : ∀ m n : nat,
  m × n = n × m.
Proof.
    Admitted.
```

□

练习：3 星, standard, optional (more_exercises) 找一张纸。对于以下定理，首先请*'思考'* (a) 它能否能只用化简和改写来证明， (b) 它还需要分类讨论（`destruct`），以及 (c) 它还需要归纳证明。先写下你的预判，然后填写下面的证明（你的纸不用交上来，这只是鼓励你先思考再行动!）

```
Check leb.

Theorem leb_refl : ∀ n:nat,
  true = (n <=? n).
Proof.
    Admitted.

Theorem zero_nbeq_S : ∀ n:nat,
  0 =? (S n) = false.
Proof.
    Admitted.

Theorem andb_false_r : ∀ b : bool,
  andb b false = false.
Proof.
```

```
  Admitted.

Theorem plus_ble_compat_l : ∀ n m p : nat,
  n <=? m = true → (p + n) <=? (p + m) = true.
Proof.
  Admitted.

Theorem S_nbeq_0 : ∀ n:nat,
  (S n) =? 0 = false.
Proof.
  Admitted.

Theorem mult_1_l : ∀ n:nat, 1 × n = n.
Proof.
  Admitted.

Theorem all3_spec : ∀ b c : bool,
    orb
      (andb b c)
      (orb (negb b)
           (negb c))
  = true.
Proof.
  Admitted.

Theorem mult_plus_distr_r : ∀ n m p : nat,
  (n + m) × p = (n × p) + (m × p).
Proof.
  Admitted.

Theorem mult_assoc : ∀ n m p : nat,
  n × (m × p) = (n × m) × p.
Proof.
  Admitted.
  □
```

练习：2 星, standard, optional (eqb_refl) 证明以下定理。（把 true 放在等式左边可能看起来有点奇怪，不过 Coq 标准库中就是这样表示的，我们照做就是。无论按哪个方向

改写都一样好用，所以无论我们如何表示定理，用起来都没问题。）

```
Theorem eqb_refl : ∀ n : nat,
  true = (n =? n).
Proof.
```

Admitted.

□

练习：2 星, standard, optional (plus_swap') `replace` 策略允许你指定一个具体的要改写的子项和你想要将它改写成的项：`replace` (t) `with` (u) 会将目标中表达式 t（的所有副本）替换为表达式 u，并生成 $t = u$ 作为附加的子目标。在简单的 `rewrite` 作用在目标错误的部分上时这种做法通常很有用。

用 `replace` 策略来证明 plus_swap'，除了无需 `assert` $(n + m = m + n)$ 外和 plus_swap 一样。

```
Theorem plus_swap' : ∀ n m p : nat,
  n + (m + p) = m + (n + p).
Proof.
```

Admitted.

□

练习：3 星, standard, recommended (binary_commute) 回忆一下你在 Basics 中为练习 *binary* 编写的 incr 和 bin_to_nat 函数。证明下图可交换。

incr bin ——————————-> bin | | bin_to_nat | | bin_to_nat | | v v nat ——————————-> nat S

也就是说，一个二进制数先自增然后将它转换为（一进制）自然数，和先将它转换为自然数后再自增会产生相同的结果。将你的定理命名为 *bin_to_nat_pres_incr*（“pres”即“preserves”的简写，意为“保持原状”）。

在开始做这个练习之前，将你在 *binary* 练习的解中的定义复制到这里，让这个文件可以被单独评分。若你想要更改你的原始定义以便让此属性更易证明，请自便！

```
Definition manual_grade_for_binary_commute : option (nat×string) := None.
```

□

练习：5 星, advanced (binary_inverse) This is a further continuation of the previous exercises about binary numbers. You may find you need to go back and change your earlier definitions to get things to work here.

(a) First, write a function to convert natural numbers to binary numbers.

```
Fixpoint nat_to_bin (n:nat) : bin
```

. *Admitted.*

Prove that, if we start with any **nat**, convert it to binary, and convert it back, we get the same **nat** we started with. (Hint: If your definition of nat_to_bin involved any extra functions, you may need to prove a subsidiary lemma showing how such functions relate to nat_to_bin.)

```
Theorem nat_bin_nat : ∀ n, bin_to_nat (nat_to_bin n) = n.
Proof.
```

Admitted.

```
Definition manual_grade_for_binary_inverse_a : option (nat×string) := None.
```

(b) One might naturally expect that we should also prove the opposite direction – that starting with a binary number, converting to a natural, and then back to binary should yield the same number we started with. However, this is not the case! Explain (in a comment) what the problem is.

```
Definition manual_grade_for_binary_inverse_b : option (nat×string) := None.
```

(c) Define a normalization function – i.e., a function *normalize* going directly from **bin** to **bin** (i.e., *not* by converting to **nat** and back) such that, for any binary number b, converting b to a natural and then back to binary yields (*normalize* b). Prove it. (Warning: This part is a bit tricky – you may end up defining several auxiliary lemmas. One good way to find out what you need is to start by trying to prove the main statement, see where you get stuck, and see if you can find a lemma – perhaps requiring its own inductive proof – that will allow the main proof to make progress.) Don't define this using nat_to_bin and bin_to_nat!

```
Definition manual_grade_for_binary_inverse_c : option (nat×string) := None.
```

□

Chapter 19

Library LF.Basics

19.1 Basics: Coq 函数式编程

19.2 引言

函数式编程风格建立在简单的、日常的数学直觉之上：若一个过程或方法没有副作用，那么在忽略效率的前提下，我们需要理解的一切便只剩下它如何将输入映射到输出了—— 也就是说，我们只需将它视作一种计算数学函数的具体方法即可。这也是"函数式编程"中"函数式"一词的含义之一。程序与简单数学对象之间这种直接的联系，同时支撑了对程序行为进行形式化证明的正确性以及非形式化论证的可靠性。

函数式编程中"函数式"一词的另一个含义是它强调把函数作为'一等'的值—— 这类值可以作为参数传递给其它函数，可以作为结果返回，也可以包含在数据结构中等等。这种将函数当做数据的方式，产生了大量强大而有用的编程习语（Idiom）。

其它常见的函数式语言特性包括能让构造和处理丰富数据结构更加简单的'代数数据类型（*Algebraic Data Type*）'和'模式匹配（*Pattern Matching*）'，以及用来支持抽象和代码复用的'多态类型系统（*Polymorphic Type System*）'。Coq 提供了所有这些特性。

本章的前半部分介绍了 Coq 原生的函数式编程语言 *'Gallina'* 中最基本的元素，后半部分则介绍了一些基本'策略（*Tactic*）'，它可用于证明 Gallina 程序的简单性质。

19.3 数据与函数

19.3.1 枚举类型

Coq 的一个不同寻常之处在于它'极小'的内建特性集合。比如，Coq 并未提供通常的原语（atomic）类型（如布尔、整数、字符串等），而是提供了一种极为强大的，可以从头定义新的数据类型的机制—— 上面所有常见的类型都是由它定义而产生的实例。

当然，Coq 发行版同时也提供了内容丰富的标准库，其中定义了布尔值、数值，以及如列表、散列表等许多通用的数据结构。不过这些库中的定义并没有任何神秘之处，也没有原语（Primitive）的特点。为了说明这一点，我们并未在本课程中直接使用标准库中的数据类型，而是在整个教程中重新定义了其中的绝大部分。

19.3.2 一周七日

让我们从一个非常简单的例子开始，看看这种定义机制是如何工作的。以下声明会告诉 Coq 我们定义了一个数据集合，即一个'类型（*Type*）'。

```
Inductive day : Type :=
  | monday
  | tuesday
  | wednesday
  | thursday
  | friday
  | saturday
  | sunday.
```

这个新的类型名为 **day**，成员包括 monday、tuesday 等等。

定义了 **day** 之后, 我们就能写一些操作星期的函数了。

```
Definition next_weekday (d:day) : day :=
  match d with
  | monday ⇒ tuesday
  | tuesday ⇒ wednesday
  | wednesday ⇒ thursday
  | thursday ⇒ friday
  | friday ⇒ monday
```

```
  | saturday ⇒ monday
  | sunday ⇒ monday
  end.
```

注意，这里显式声明了函数的参数和返回类型。和大多数函数式编程语言一样，如果没有显式指定类型，Coq 通常会自己通过 '类型推断（*Type Inference*）' 得出。不过我们会标上类型使其更加易读。

定义了函数之后，我们接下来应该用一些例子来检验它。实际上，在 Coq 中，一共有三种不同的检验方式：第一，我们可以用 `Compute` 指令来计算包含 `next_weekday` 的复合表达式：

```
Compute (next_weekday friday).

Compute (next_weekday (next_weekday saturday)).
```

（我们在注释中写出 Coq 返回的结果。如果你身边就有电脑，不妨自己用 Coq 解释器试一试：选一个你喜欢的 IDE，CoqIde 或 Proof General 都可以。然后从本书附带的 Coq 源码中载入 *Basics.v* 文件，找到上面的例子，提交给 Coq，然后查看结果。）

第二，我们可以将'期望'的结果写成 Coq 的示例：

```
Example test_next_weekday:
  (next_weekday (next_weekday saturday)) = tuesday.
```

该声明做了两件事：首先它断言 `saturday` 之后的第二个工作日是 `tuesday`；然后它为该断言取了名字以便之后引用它。定义好断言后，我们还可以让 Coq 来验证它，就像这样：

```
Proof. simpl. reflexivity. Qed.
```

具体细节目前并不重要，不过这段代码基本上可以读作"若等式两边的求值结果相同，该断言即可得证。"

第三，我们可以让 Coq 从 `Definition` 中'提取（*Extract*）'出用其它更加常规的编程语言编写的程序（如 OCaml、Scheme、Haskell 等），它们拥有高性能的编译器。这种能力非常有用，我们可以通过它将 Gallina 编写的'证明正确'的算法转译成高效的机器码。（诚然，我们必须信任 OCaml/Haskell/Scheme 的编译器，以及 Coq 提取工具自身的正确性，然而比起现在大多数软件的开发方式，这也是很大的进步了。）实际上，这就是 Coq 最主要的使用方式之一。在之后的章节中我们会回到这一主题上来。

19.3.3 作业提交指南

如果你在课堂中使用《软件基础》，你的讲师可能会用自动化脚本来为你的作业评分。为了让这些脚本能够正常工作（这样你才能拿到全部学分!），请认真遵循以下规则：

- 评分脚本在提取你提交的 *.v* 文件时会用到其中的特殊标记。因此请勿修改练习的"分隔标记"，如练习的标题、名称、以及末尾的 [] 等等。

- 不要删除练习。如果你想要跳过某个练习（例如它标记为"可选"或你无法解决它），可以在 *.v* 文件中留下部分证明，这没关系，不过此时请确认它以 *Admitted* 结尾（不要用 `Abort` 之类的东西）。

- 你也可以在解答中使用附加定义（如辅助函数，需要的引理等）。你可以将它们放在练习的头部和你要证明的定理之间。

- 如果你为了证明某定理而需要引入一个额外引理，且未能证明该引理，请确保该引理与使用它的原定理都以 *Admitted* 而非 `Qed` 结尾。这样能使在你利用原定理解决其他练习时得到部分分数。

你或许注意到每一章都附带有一个'测试脚本'来自动计算该章节已完成的作业的分数。这些脚本一般只作为自动评分工具，但你也可以用它们在提交前再一次确认作业格式的正确性。你可以在一个终端窗口中输入 "*make BasicsTest.vo*" 或下面的命令来运行这些测试脚本

```
coqc -Q . LF Basics.v coqc -Q . LF BasicsTest.v
```

你并不需要提交 *BasicsTest.v* 这种测试脚本（也不需要提交前言 *Preface.v*）。

如果你的班级使用 Canvas 系统来提交作业。

- 如果你提交了多个不同版本的作业，你可能会注意到它们在系统中有着不同的名字。这是正常情况，只有最新的提交会被评分。

- 如果你需要同时提交多个文件（例如一次作业中包含多个不同的章节），你需要创建一个一次性包含所有文件的提交。（译者注：关于多文件提交细节请查看英文原文。）
  To hand in multiple files at the same time (if more than one chapter is assigned in the same week), you need to make a single submission with all the files at once using the button "Add another file" just above the comment box.

The `Require Export` statement on the next line tells Coq to use the String module from the standard library. We'll use strings ourselves in later chapters, but we need to `Require` it here so that the grading scripts can use it for internal purposes. `From` *Coq* `Require Export` String.

19.3.4 布尔值

通过类似的方式，我们可以为布尔值定义常见的 **bool** 类型，它包括 true 和 false 两个成员。

```
Inductive bool : Type :=
  | true
  | false.
```

布尔值的函数可按照同样的方式来定义：

```
Definition negb (b:bool) : bool :=
  match b with
  | true ⇒ false
  | false ⇒ true
  end.

Definition andb (b1:bool) (b2:bool) : bool :=
  match b1 with
  | true ⇒ b2
  | false ⇒ false
  end.

Definition orb (b1:bool) (b2:bool) : bool :=
  match b1 with
  | true ⇒ true
  | false ⇒ b2
  end.
```

（虽然我们正尝试从零开始定义布尔类型，但由于 Coq 的标准库中也提供了布尔类型的默认实现，以及大量有用的函数和引理。我们会尽量让自己的定义和定理的名字与标准库保持一致。）

其中 andb 和 orb 演示了如何定义多参函数。以下四个“单元测试”则演示了如何应用这些函数，它们构成了 orb 函数的完整规范（Specification），即真值表：

```
Example test_orb1: (orb true false) = true.
Proof. simpl. reflexivity. Qed.
Example test_orb2: (orb false false) = false.
Proof. simpl. reflexivity. Qed.
Example test_orb3: (orb false true) = true.
Proof. simpl. reflexivity. Qed.
Example test_orb4: (orb true true) = true.
Proof. simpl. reflexivity. Qed.
```

我们也可以为刚定义的布尔运算引入更加熟悉的中缀语法。 `Notation` 指令能为既有的定义赋予新的符号记法。

```
Notation "x && y" := (andb x y).
Notation "x || y" := (orb x y).

Example test_orb5: false || false || true = true.
Proof. simpl. reflexivity. Qed.
```

'关于记法的说明'：在 *.v* 文件中，我们用方括号来界定注释中的 Coq 代码片段；这种约定也在 *coqdoc* 文档工具中使用，它能让代码与周围的文本从视觉上区分开来。在 HTML 版的文件中，这部分文本会以*'不同的字体'*显示。

下面的例子展示了 Coq 的另一个特性: 条件表达式...

```
Definition negb' (b:bool) : bool :=
  if b then false
  else true.

Definition andb' (b1:bool) (b2:bool) : bool :=
  if b1 then b2
  else false.

Definition orb' (b1:bool) (b2:bool) : bool :=
  if b1 then true
  else b2.
```

Coq 的条件表达式相较于其他语言的，只有一点小小的扩展。由于 **bool** 类型并不是内建类型，Coq 实际上支持对_任何_归纳定义的双子句表达式使用 “if” 表达式（不过恰巧

在这里该表达式被称为 **bool**）。当条件求值后得到的是第一个子句的“构造子” (constructor)，那么条件就会被认为是“真” true（不过恰巧在这里第一个分支的构造子被称为“真” true，并且如果求值后得到的是第二个子句，那么条件就被认为是“假” false）。

练习：1 星, standard (nandb) 指令 *Admitted* 被用作不完整证明的占位符。我们会在练习中用它来表示你需要完成的部分。你的任务就是将 *Admitted* 替换为具体的证明。

移除“*Admitted.*”并补完以下函数的定义，然后确保下列每一个 `Example` 中的断言都能被 Coq 验证通过。（即仿照上文 orb 测试的格式补充证明，并确保 Coq 接受它。）此函数应在两个输入中的任意一个（或者都）包含 false 时返回 true 。提示：如果 `simpl` 在你的证明中未能化简目标，那是因为你可能并未使用 `match` 表达式定义你的 nandb。尝试使用另一种 nandb 的定义方式，或者直接跳过 `simpl` 直接使用 `reflexivity`。我们后面会解释为什么会发生这种情况。

```
Definition nandb (b1:bool) (b2:bool) : bool
  . Admitted.

Example test_nandb1: (nandb true false) = true.
    Admitted.
Example test_nandb2: (nandb false false) = true.
    Admitted.
Example test_nandb3: (nandb false true) = true.
    Admitted.
Example test_nandb4: (nandb true true) = false.
    Admitted.
```

□

练习：1 星, standard (andb3) 与此前相同，完成下面的 andb3 函数。此函数应在所有输入均为 true 时返回 true，否则返回 false。

```
Definition andb3 (b1:bool) (b2:bool) (b3:bool) : bool
  . Admitted.

Example test_andb31: (andb3 true true true) = true.
    Admitted.
Example test_andb32: (andb3 false true true) = false.
    Admitted.
```

```
Example test_andb33: (andb3 true false true) = false.
```

Admitted.

```
Example test_andb34: (andb3 true true false) = false.
```

Admitted.

□

19.3.5 类型

Coq 中的每个表达式都有类型，它描述了该表达式所计算的东西的类别。`Check` 指令会让 Coq 显示一个表达式的类型。

```
Check true.
```

如果在被 `Check` 的表达式后加上一个分号和你想验证的类型，那么 Coq 会验证该表达式是否属于你提供的类型。当两者不一致时，Coq 会报错并终止执行。

```
Check true
```

: **bool**.

```
Check (negb true)
```

: **bool**.

像 negb 这样的函数本身也是数据值，就像 true 和 false 一样。它们的类型被称为'*函数类型*'，用带箭头的类型表示。

```
Check negb
```

: **bool** → **bool**.

negb 的类型写作 **bool** → **bool**，读做"**bool** 箭头 **bool**"，可以理解为"给定一个 **bool** 类型的输入，该函数产生一个 **bool** 类型的输出。" 同样，andb 的类型写作 **bool** → **bool** → **bool**，可以理解为"给定两个 **bool** 类型的输入，该函数产生一个 **bool** 类型的输出。"

19.3.6 由旧类型构造新类型

到目前为止，我们定义的类型都是"枚举类型"：它们的定义显式地枚举了一个元素的有限集，其中每个元素都只是一个裸构造子（译注：即无参数构造子）。下面是一个更加有趣的类型定义，其中有个构造子接受一个参数：

```
Inductive rgb : Type :=
  | red
```

| green
| blue.

Inductive **color** : Type :=
| black
| white
| primary (p : **rgb**).

像 red、green、blue、black、white 以及 primary（还有 true、false、monday 等）这样的原子标识符叫做'构造子（*Constructor*）'。

我们可以用它们来构建'构造子表达式（*Constructor Expression*）'，其中每一个要么是一个简单的构造子，要么就是一个构造子应用于一个或多个参数（每个这样的参数也都是构造子表达式）。

我们来仔细研究一下。每个归纳定义的类型（如 **day**、**bool**、**rgb**、**color** 等）都描述了一组由'构造子'构成的'构造子表达式'。

- 我们从有限的一组'构造子'开始。例如 red、primary、true、false、monday 等等都是构造子。
- '构造子表达式'通过将构造子应用到一个或多个构造子表达式上构成。例如 red、true、primary、primary red、red primary、red true、 primary (primary primary) 等等
- 每个 Inductive 定义都刻画了一个构造子表达式的子集并赋予了它们名字，如 **bool**、**rgb** 或 **color**。

具体来说，**rgb** 和 **color** 的定义描述了如何构造这两个集合中的构造子表达式：

- 构造子表达式 red、green 和 blue 属于集合 **rgb**；
- 构造子表达式 black 和 white 属于集合 **color**；
- 若 p 是属于 **rgb** 的构造子表达式，则 primary p（读作"构造子 primary 应用于参数 p）是属于集合 **color** 的构造子表达式；且
- 通过这些方式构造的构造子表达式'只属于'集合 **rgb** 和 **color**。

我们可以像之前的 **day** 和 **bool** 那样用模式匹配为 **color** 定义函数。

Definition monochrome (c : **color**) : **bool** :=

```
  match c with
  | black ⇒ true
  | white ⇒ true
  | primary p ⇒ false
  end.
```

鉴于 primary 构造子接收一个参数，匹配到 primary 的模式应当带有一个变量或常量。变量可以取任意名称，如上文所示；常量需有适当的类型，例如：

```
Definition isred (c : color) : bool :=
  match c with
  | black ⇒ false
  | white ⇒ false
  | primary red ⇒ true
  | primary _ ⇒ false
  end.
```

这里的模式 primary _ 是“构造子 primary 应用到除 red 之外的任何 **rgb** 构造子上”的简写形式（通配模式 _ 的效果与 monochrome 定义中的哑（dummy）模式变量 p 相同。）

19.3.7 元组

一个多参数的单构造子可以用来创建元组类型。例如，为了让一个半字节（nybble）表示四个比特。我们首先定义一个 **bit** 数据类型来类比 **bool** 数据。并且使用 B0 和 B1 两种构造子来表示其可能的取值。最后定义 **nybble** 这种数据类型，也就是一个四比特的元组。

```
Inductive bit : Type :=
  | B0
  | B1.

Inductive nybble : Type :=
  | bits (b0 b1 b2 b3 : bit).

Check (bits B1 B0 B1 B0)
    : nybble.
```

这里的 **bit** 构造子起到了对它内容的包装作用。解包可以通过模式匹配来实现，就如同下面的 all_zero 函数一样，其通过解包来验证一个半字节的所有比特是否都为 B0。我们

用'通配符' _ 来避免创建不需要的变量名。

```
Definition all_zero (nb : nybble) : bool :=
  match nb with
    | (bits B0 B0 B0 B0) ⇒ true
    | (bits _ _ _ _) ⇒ false
  end.

Compute (all_zero (bits B1 B0 B1 B0)).
Compute (all_zero (bits B0 B0 B0 B0)).
```

19.3.8 模块

Coq 提供了'模块系统'来帮助组织大规模的开发。在本课程中，我们不太会用到这方面的特性。不过其中有一点非常有用：如果我们将一组定义放在 `Module X` 和 `End X` 标记之间，那么在文件中的 `End` 之后，我们就可以通过像 *X.foo* 这样的名字来引用，而不必直接用 foo 了。在这里，我们通过此特性在内部模块中引入了 **nat** 类型的定义，这样就不会覆盖标准库中的同名定义了（我们会在本书后面的部分中使用它，因为它提供了一些简便的特殊记法。）

```
Module NatPlayground.
```

19.3.9 数值

目前我们定义的所有类型都是有限的。无论是像 **day**, **bool** 和 **bit** 这样的"枚举类型"，抑或是像 **nybble** 这样基于"枚举类型"的元组类型，本质上都是有限的集合。而自然数（natural numbers）是一个无限集合，因此我们需要一种更强大的类型声明方式来表示它们。

数字的表示方法有许多种。我们最为熟悉的便是十进制（base 10），利用 0～9 十个数字来表示一个数，例如用 1，2 和 3 来表示 123 （一百二十三）。你或许也接触过十六进制（base 16），在十六进制中，它被表示为 7B。类似的还有 173 （八进制表示）和 111011（二进制表示）。我们可以使用枚举类型来定义以上任何一种数字表示方式。它们在不同的场景下有着不同的用途。

二进制表示在计算机硬件中起着举足轻重的作用。它只需要两种不同的电平来表示，因此它的硬件电路可以被设计十分简单。同样的，我们也希望选择一种自然数的表示方式，来让我们的_证明_变得更加简单。

实际上，比起二进制，还有一种更加简单的数字表示方式，一进制（base 1），也就是只使用单个数字的表示方式（就如同我们的祖先山顶洞人在洞穴上刻“痕迹”计算日子一般）。为了在 Coq 中表示一进制数，我们使用两个构造子。大写的 O 构造子用来表示“零”，而大写的 S 构造子用来表示“后继” （或者洞穴上的“痕迹”）。当 S 构造子被应用于一个自然数 n 的表示上时，结果会是自然数 n + 1 的表示。下面是完整的数据类型定义。

```
Inductive nat : Type :=
  | O
  | S (n : nat).
```

在这种定义下， 0 被表示为 O, 1 则被表示为 S O, 2 则是 S (S O)，以此类推
非形式化地说，此定义中的子句可读作：

- O 是一个自然数（注意这里是字母“O”，不是数字“0”）。
- S 可被放在一个自然数之前来产生另一个自然数—— 也就是说，如果 n 是一个自然数，那么 S n 也是。

同样，我们来仔细观察这个定义。 **nat** 的定义描述了集合 **nat** 中的表达式是如何构造的：

- 构造子表达式 O 属于集合 **nat**；
- 如果 n 是属于集合 **nat** 的构造子表达式，那么 S n 也是属于集合 **nat** 的构造子表达式；并且
- 只有以这两种产生的方式构造字表达式才属于集合 **nat**。

这些条件精确刻画了这个“归纳” `Inductive` 声明。它们意味着，构造子表达式 O、 S O、S (S O)、S (S (S O)) 等等都属于集合 **nat**，而其它的构造子表达式，如 true、andb true false、S (S false) 以及 O (O (O S)) 等则不属于 **nat**。

关键之处在于，我们目前只是定义了一种数字的’表示’方式，即一种写下它们的方式。名称 O 和 S 是任意的，在这一点上它们没有特殊的意义，它们只是我们能用来写下数字的两个不同的记号（以及一个说明了任何 **nat** 都能写成一串 S 后跟一个 O 的规则）。如果你喜欢，完全可以将同样的定义写成：

```
Inductive nat' : Type :=
  | stop
```

```
  | tick (foo : nat').
```

这些记号的'解释'完全取决于我们如何用它进行计算。

我们可以像之前的布尔值或日期那样，编写一个函数来对上述自然数的表示进行模式匹配。例如，以下为前趋函数：

```
Definition pred (n : nat) : nat :=
  match n with
    | O ⇒ O
    | S n' ⇒ n'
  end.
```

第二个分支可以读作："如果 n 对于某个 n' 的形式为 S n'，那么就返回 n'。"

下面的 End 指令会关闭当前的模块，所以 **nat** 会重新代表标准库中的类型而非我们自己定义的 **nat**。

```
End NatPlayground.
```

为了让自然数使用起来更加自然，Coq 内建了一小部分解析打印功能：普通的十进制数可视为"一进制"自然数的另一种记法，以代替 S 与 O 构造子；反过来，Coq 也会默认将自然数打印为十进制形式：

```
Check (S (S (S (S O)))).

Definition minustwo (n : nat) : nat :=
  match n with
    | O ⇒ O
    | S O ⇒ O
    | S (S n') ⇒ n'
  end.

Compute (minustwo 4).
```

构造子 S 的类型为 **nat** → **nat**，与 pred 和 minustwo 之类的函数相同：

```
Check S : nat→nat.
Check pred : nat→nat.
Check minustwo : nat→nat.
```

以上三个东西均可作用于自然数，并产生自然数结果，但第一个 S 与后两者有本质区别：pred 和 minustwo 这类函数是通过给定的'计算规则'定义的—— 例如 pred 的定义表明

pred 2 可化简为 1——但 S 的定义不包含此类行为。虽然 S 可以作用于参数这点与函数'相似'，但其作用仅限于构造数字，而并不用于计算。

（考虑标准的十进制数：数字 1 不代表任何计算，只表示一部分数据。用 111 指代数字一百一十一，实则使用三个 1 符号表示此数各位。）

现在我们来为数值定义更多的函数。

简单的模式匹配不足以描述很多有趣的数值运算，我们还需要递归定义。例如：给定自然数 n，欲判定其是否为偶数，则需递归检查 n-2 是否为偶数。关键字 `Fixpoint` 可用于定义此类函数。

```
Fixpoint evenb (n:nat) : bool :=
  match n with
  | O ⇒ true
  | S O ⇒ false
  | S (S n') ⇒ evenb n'
  end.
```

我们可以使用类似的 `Fixpoint` 声明来定义 *odd* 函数，不过还有种更简单方式：

```
Definition oddb (n:nat) : bool :=
  negb (evenb n).

Example test_oddb1: oddb 1 = true.
Proof. simpl. reflexivity. Qed.
Example test_oddb2: oddb 4 = false.
Proof. simpl. reflexivity. Qed.
```

（如果你逐步检查完这些证明，就会发现 `simpl` 其实没什么作用—— 所有工作都被 `reflexivity` 完成了。我们之后会讨论为什么会这样。）

当然，我们也可以用递归定义多参函数。

```
Module NatPlayground2.

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

三加二等于五，不出意料。

```
Compute (plus 3 2).
```

Coq 所执行的化简步骤如下所示：

为了书写方便，如果两个或更多参数具有相同的类型，那么它们可以写在一起。在下面的定义中，(*n m* : **nat**) 的意思与 (*n* : **nat**) (*m* : **nat**) 相同。

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

Example test_mult1: (mult 3 3) = 9.
Proof. simpl. reflexivity. Qed.
```

你可以在两个表达式之间添加逗号来同时匹配它们：

```
Fixpoint minus (n m:nat) : nat :=
  match n, m with
  | O , _ ⇒ O
  | S _ , O ⇒ n
  | S n', S m' ⇒ minus n' m'
  end.

End NatPlayground2.

Fixpoint exp (base power : nat) : nat :=
  match power with
    | O ⇒ S O
    | S p ⇒ mult base (exp base p)
  end.
```

练习：1 星, standard (factorial) 回想一下标准的阶乘函数：

factorial(0) = 1 factorial(n) = n * factorial(n-1) (if n>0)

把它翻译成 Coq 代码。

```
Fixpoint factorial (n:nat) : nat
  . Admitted.

Example test_factorial1: (factorial 3) = 6.
```

Admitted.

```
Example test_factorial2: (factorial 5) = (mult 10 12).
```

Admitted.

□

我们可以通过引入加法、乘法和减法的'记法（*Notation*）'来让数字表达式更加易读。

```
Notation "x + y" := (plus x y)
                    (at level 50, left associativity)
                    : nat_scope.
Notation "x - y" := (minus x y)
                    (at level 50, left associativity)
                    : nat_scope.
Notation "x * y" := (mult x y)
                    (at level 40, left associativity)
                    : nat_scope.

Check ((0 + 1) + 1) : nat.
```

（`level`、`associativity` 和 *nat_scope* 标记控制着 Coq 语法分析器如何处理上述记法。目前无需关注这些细节。有兴趣的读者可参阅本章末尾"关于记法的更多内容"一节。）

注意，这些声明并不会改变我们之前的定义，而只是让 Coq 语法分析器接受用 $x + y$ 来代替 plus x y，并在 Coq 美化输出时反过来将 plus x y 显示为 $x + y$。

Coq 几乎不包含任何内置定义，甚至连数值间的相等关系都是由用户来实现。 eqb 函数定义如下：该函数检验自然数 **nat** 间是否满足相等关系 **eq**，并以布尔值 **bool** 表示。注意该定义使用嵌套匹配 `match` （亦可仿照 minus 使用并列匹配）。

```
Fixpoint eqb (n m : nat) : bool :=
  match n with
  | O ⇒ match m with
          | O ⇒ true
          | S m' ⇒ false
          end
  | S n' ⇒ match m with
              | O ⇒ false
              | S m' ⇒ eqb n' m'
```

```
        end
  end.
```

类似地，leb 函数检验其第一个参数是否小于等于第二个参数，以布尔值表示。

```
Fixpoint leb (n m : nat) : bool :=
  match n with
  | O ⇒ true
  | S n' ⇒
      match m with
      | O ⇒ false
      | S m' ⇒ leb n' m'
      end
  end.

Example test_leb1: leb 2 2 = true.
Proof. simpl. reflexivity. Qed.
Example test_leb2: leb 2 4 = true.
Proof. simpl. reflexivity. Qed.
Example test_leb3: leb 4 2 = false.
Proof. simpl. reflexivity. Qed.
```

我们之后会经常用到它们（特别是 eqb），因此先定义好它们的中缀记法：

```
Notation "x =? y" := (eqb x y) (at level 70) : nat_scope.
Notation "x <=? y" := (leb x y) (at level 70) : nat_scope.

Example test_leb3': (4 <=? 2) = false.
Proof. simpl. reflexivity. Qed.
```

练习：1 星, standard (ltb) ltb 函数检验自然数间的小于关系，以布尔值表示。请利用前文定义的函数写出该定义，不要使用 `Fixpoint` 构造新的递归。（只需前文中的一个函数即可实现该定义，不过也可两者皆用。）

```
Definition ltb (n m : nat) : bool
  . Admitted.

Notation "x <? y" := (ltb x y) (at level 70) : nat_scope.

Example test_ltb1: (ltb 2 2) = false.
```

Admitted.

```
Example test_ltb2: (ltb 2 4) = true.
```

Admitted.

```
Example test_ltb3: (ltb 4 2) = false.
```

Admitted.

□

19.4 基于化简的证明

至此，我们已经定义了一些数据类型和函数。让我们把问题转到如何表述和证明它们行为的性质上来。其实我们已经开始这样做了：前几节中的每个 `Example` 都对几个函数在某些特定输入上的行为做出了准确的断言。这些断言的证明方法都一样：使用 `simpl` 来化简等式两边，然后用 `reflexivity` 来检查两边是否具有相同的值。

这类“基于化简的证明”还可以用来证明更多有趣的性质。例如，对于“0 出现在左边时是加法 $+$ 的‘幺元’”这一事实，我们只需读一遍 plus 的定义，即可通过观察“对于 $0 + n$，无论 n 的值为多少均可化简为 n”而得到证明。

```
Theorem plus_O_n : ∀ n : nat, 0 + n = n.
Proof.
  intros n. simpl. reflexivity. Qed.
```

（或许你会注意到以上语句在你的 IDE 中和在浏览器渲染的 HTML 中不大一样，我们用保留标识符“forall”来表示全称量词 ∀。当 .v 文件转换为 HTML 后，它会变成一个倒立的“A”。）

现在是时候说一下 `reflexivity` 了，它其实比我们想象的更为强大。在前面的例子中，其实并不需要调用 `simpl` ，因为 `reflexivity` 在检查等式两边是否相等时会自动做一些化简；我们加上 `simpl` 只是为了看到化简之后，证明结束之前的中间状态。下面是对同一定理更短的证明：

```
Theorem plus_O_n' : ∀ n : nat, 0 + n = n.
Proof.
  intros n. reflexivity. Qed.
```

此外，`reflexivity` 在某些方面做了比 `simpl` ’更多’的化简—— 比如它会尝试“展开”已定义的项，将它们替换为该定义右侧的值。了解这一点会很有帮助。产生这种差别的原因是，当自反性成立时，整个证明目标就完成了，我们不必再关心 `reflexivity` 化简和展

开了什么；而当我们必须去观察和理解新产生的证明目标时，我们并不希望盲目地展开定义，将证明目标留在混乱的声明中。这种情况下就要用到 simpl 了。

我们刚刚声明的定理形式及其证明与前面的例子基本相同，它们只有一点差别。

首先，我们使用了关键字 Theorem 而非 Example。这种差别纯粹是风格问题；在 Coq 中，关键字 Example 和 Theorem（以及其它一些，包括 Lemma、Fact 和 Remark）都表示完全一样的东西。

其次，我们增加了量词 ∀ n:**nat**，因此我们的定理讨论了'所有的'自然数 n。在非形式化的证明中，为了证明这种形式的定理，我们通常会说"'假设'存在一个任意自然数 n..."。而在形式化证明中，这是用 intros n 来实现的，它会将量词从证明目标转移到当前假设的'上下文'中。注意在 intros 从句中，我们可以使用别的标识符来代替 n （当然这可能会让阅读证明的人感到困惑）：

```
Theorem plus_O_n'' : ∀ n : nat, 0 + n = n.
Proof.
  intros m. reflexivity. Qed.
```

关键字 intros、simpl 和 reflexivity 都是'策略（*Tactic*）'的例子。策略是一条可以用在 Proof（证明）和 Qed（证毕）之间的指令，它告诉 Coq 如何来检验我们所下的一些断言的正确性。在本章剩余的部分及以后的课程中，我们会见到更多的策略。

其它类似的定理可通过相同的模式证明。

```
Theorem plus_1_l : ∀ n:nat, 1 + n = S n.
Proof.
  intros n. reflexivity. Qed.

Theorem mult_0_l : ∀ n:nat, 0 × n = 0.
Proof.
  intros n. reflexivity. Qed.
```

上述定理名称的后缀 $\_l$ 读作"在左边"。

跟进这些证明的每个步骤，观察上下文及证明目标的变化是非常值得的。你可能要在 reflexivity 前面加上 simpl 调用，以便观察 Coq 在检查它们的相等关系前进行的化简。

19.5 基于改写的证明

下面这个定理比我们之前见过的更加有趣：

```
Theorem plus_id_example : ∀ n m:nat,
```

```
  n = m →
  n + n = m + m.
```

该定理并未对自然数 n 和 m 所有可能的值做全称断言，而是讨论了仅当 $n = m$ 时这一更加特定情况。箭头符号读作“蕴含”。

与此前相同，我们需要在能够假定存在自然数 n 和 m 的基础上进行推理。另外我们需要假定有前提 $n = m$。`intros` 策略用来将这三条前提从证明目标移到当前上下文的假设中。

由于 n 和 m 是任意自然数，我们无法用化简来证明此定理，不过可以通过观察来证明它。如果我们假设了 $n = m$，那么就可以将证明目标中的 n 替换成 m 从而获得两边表达式相同的等式。用来告诉 Coq 执行这种替换的策略叫做'改写' `rewrite`。

```
Proof.
  intros n m.
  intros H.
  rewrite → H.
  reflexivity. Qed.
```

证明的第一行将全称量词变量 n 和 m 移到上下文中。第二行将前提 $n = m$ 移到上下文中，并将其命名为 H。第三行告诉 Coq 改写当前目标（$n + n = m + m$），把前提等式 H 的左边替换成右边。

(`rewrite` 中的箭头与蕴含无关：它指示 Coq 从左往右地应用改写。若要从右往左改写，可以使用 `rewrite` ←。在上面的证明中试一试这种改变，看看 Coq 的反应有何不同。)

练习：1 星, standard (plus_id_exercise) 删除“*Admitted.*”并补完证明。

```
Theorem plus_id_exercise : ∀ n m o : nat,
  n = m → m = o → n + m = m + o.
Proof.
  Admitted.
```

□

Admitted 指令告诉 Coq 我们想要跳过此定理的证明，而将其作为已知条件，这在开发较长的证明时很有用。在进行一些较大的命题论证时，我们能够声明一些附加的事实。既然我们认为这些事实对论证是有用的，就可以用 *Admitted* 先不加怀疑地接受这些事实，然后继续思考大命题的论证。直到确认了该命题确实是有意义的，再回过头去证明刚才跳

过的证明。但是要小心：每次你使用 *Admitted*，你就为 Coq 这个完好、严密、形式化且封闭的世界开了一个毫无逻辑的后门。

`Check` 命令也可用来检查以前声明的引理和定理。下面两个关于乘法引理来自于标准库。（在下一章中，我们会亲自证明它们。）

```
Check mult_n_O.

Check mult_n_Sm.
```

除了上下文中现有的假设外，我们还可以通过 `rewrite` 策略来运用前期证明过的定理。如果前期证明的定理的语句中包含量词变量，如前例所示，Coq 会通过匹配当前的证明目标来尝试实例化（Instantiate）它们。

```
Theorem mult_n_0_m_0 : ∀ n m : nat,
  (n × 0) + (m × 0) = 0.
Proof.
  intros n m.
  rewrite ← mult_n_O.
  rewrite ← mult_n_O.
  reflexivity. Qed.
```

练习：2 星, standard (mult_n_1) `Theorem mult_n_1 : ∀ n : nat,`

```
  n × 1 = n.
Proof.
    Admitted.
```

□

19.6 利用分类讨论来证明

当然，并非一切都能通过简单的计算和改写来证明。通常，一些未知的，假定的值（如任意数值、布尔值、列表等等）会阻碍化简。例如，如果我们像以前一样使用 `simpl` 策略尝试证明下面的事实，就会被卡住。（现在我们用 `Abort` 指令来放弃证明。）

```
Theorem plus_1_neq_0_firsttry : ∀ n : nat,
  (n + 1) =? 0 = false.
Proof.
  intros n.
```

```
  simpl. Abort.
```

原因在于：根据 eqb 和 + 的定义，其第一个参数先被 match 匹配。但此处 + 的第一个参数 n 未知，而 eqb 的第一个参数 $n + 1$ 是复合表达式，二者均无法化简。

欲进行规约，则需分情况讨论 n 的所有可能构造。如果 n 为 O，则可验算 $(n + 1) =? 0$ 的结果确实为 false；如果 n 由 S n' 构造，那么即使我们不知道 $n + 1$ 表示什么，但至少知道它的构造子为 S，因而足以得出 $(n + 1) =? 0$ 的结果为 false。

告诉 Coq 分别对 $n = 0$ 和 $n =$ S n' 这两种情况进行分析的策略，叫做 destruct。

```
Theorem plus_1_neq_0 : ∀ n : nat,
  (n + 1) =? 0 = false.
Proof.
  intros n. destruct n as [| n'] eqn:E.
  - reflexivity.
  - reflexivity. Qed.
```

destruct 策略会生成_两个_子目标，为了让 Coq 认可这个定理，我们必须接下来证明这两个子目标。

as [| n'] 这种标注被称为 '引入模式'。它告诉 Coq 应当在每个子目标中使用什么样的变量名。总体而言，在方括号之间的是一个由 | 隔开的 '列表的列表'（译者注：list of lists）。在上面的例子中，第一个元素是一个空列表，因为 O 构造子是一个空构造子（它没有任何参数）。第二个元素提供了包含单个变量名 n' 的列表，因为 S 是一个单构造子。

在每个子目标中，Coq 会记录这个子目标中关于 n 的假设，$n = 0$ 还是对于某个 n', $n =$ S n'。而 $eqn{:}E$ 记号则告知 Coq 以 E 来命名这些假设。省略 $eqn{:}E$ 会导致 Coq 省略这些假设。这种省略能够使得一些不需要显式用到这类假设的证明显得更加流畅。但在实践中最好还是保留他们，因为他们可以作为一种说明文档来在证明过程中指引你。

第二行和第三行中的 - 符号叫做'标号'，它标明了这两个生成的子目标所对应的证明部分。（译注：此处的"标号"应理解为一个项目列表中每个 '条目' 前的小标记，如或•。）标号后面的证明脚本是一个子目标的完整证明。在本例中，每个子目标都简单地使用 reflexivity 完成了证明。通常，reflexivity 本身会执行一些化简操作。例如，第二段证明将 at (S n' + 1) 0 化简成 false，是通过先将 (S n' + 1) 转写成 S $(n' + 1)$，接着展开 beq_nat，之后再化简 match 完成的。

用标号来区分情况是可选的：如果没有标号，Coq 只会简单地要求你依次证明每个子目标。尽管如此，使用标号仍然是一个好习惯。原因有二：首先，它能让证明的结构更加清晰易读。其次，标号能指示 Coq 在开始验证下一个目标前确认上一个子目标已完成，

防止不同子目标的证明搅和在一起。这一点在大型开发中尤为重要，因为一些证明片段会导致很耗时的排错过程。

在 Coq 中并没有既严格又便捷的规则来格式化证明—— 尤其指应在哪里断行，以及证明中的段落应如何缩进以显示其嵌套结构。然而，无论格式的其它方面如何布局，只要在多个子目标生成的地方为每行开头标上标号，那么整个证明就会有很好的可读性。

这里也有必要提一下关于每行代码长度的建议。Coq 的初学者有时爱走极端，要么一行只有一个策略语句，要么把整个证明都写在一行里。更好的风格则介于两者之间。一个合理的习惯是给自己设定一个每行 80 个字符的限制。更长的行会很难读，也不便于显示或打印。很多编辑器都能帮你做到。

`destruct` 策略可用于任何归纳定义的数据类型。比如，我们接下来会用它来证明布尔值的取反是对合（Involutive）的—— 即，取反是自身的逆运算。

```
Theorem negb_involutive : ∀ b : bool,
  negb (negb b) = b.
Proof.
  intros b. destruct b eqn:E.
  - reflexivity.
  - reflexivity. Qed.
```

注意这里的 `destruct` 没有 `as` 子句，因为此处 `destruct` 生成的子分类均无需绑定任何变量，因此也就不必指定名字。实际上，我们也可以省略 '任何' `destruct` 中的 `as` 子句， Coq 会自动填上变量名。不过这通常是个坏习惯，因为如果任其自由决定的话， Coq 经常会选择一些容易令人混淆的名字。

有时在一个子目标内调用 `destruct`，产生出更多的证明义务（Proof Obligation）也非常有用。这时候，我们使用不同的标号来标记目标的不同"层级"，比如：

```
Theorem andb_commutative : ∀ b c, andb b c = andb c b.
Proof.
  intros b c. destruct b eqn:Eb.
  - destruct c eqn:Ec.
    + reflexivity.
    + reflexivity.
  - destruct c eqn:Ec.
    + reflexivity.
    + reflexivity.
```

```
Qed.
```

每一对 `reflexivity` 调用和紧邻其上的 `destruct` 执行后生成的子目标对应。

除了 `-` 和 `+` 之外，还可以使用 `*`（星号）或任何重复的标号符（如 `--` 或 `***`）作为标号。我们也可以用花括号将每个子证明目标括起来：

```
Theorem andb_commutative' : ∀ b c, andb b c = andb c b.
Proof.
  intros b c. destruct b eqn:Eb.
  { destruct c eqn:Ec.
    { reflexivity. }
    { reflexivity. } }
  { destruct c eqn:Ec.
    { reflexivity. }
    { reflexivity. } }
Qed.
```

由于花括号同时标识了证明的开始和结束，因此它们可以同时用于不同的子目标层级，如上例所示。此外，花括号还允许我们在一个证明中的多个层级下使用同一个标号。使用大括号、标号还是二者结合都纯粹是个人偏好问题。

```
Theorem andb3_exchange :
  ∀ b c d, andb (andb b c) d = andb (andb b d) c.
Proof.
  intros b c d. destruct b eqn:Eb.
  - destruct c eqn:Ec.
    { destruct d eqn:Ed.
      - reflexivity.
      - reflexivity. }
    { destruct d eqn:Ed.
      - reflexivity.
      - reflexivity. }
  - destruct c eqn:Ec.
    { destruct d eqn:Ed.
      - reflexivity.
      - reflexivity. }
```

```
  { destruct d eqn:Ed.
    - reflexivity.
    - reflexivity. }
Qed.
```

在本章结束之前，我们最后再说一种简便写法。或许你已经注意到了，很多证明在引入变量之后会立即对它进行情况分析：

intros x y. destruct y as [|y] eqn:E.

这种写法是如此的常见以至于 Coq 为它提供了一种简写：我们可以在引入一个变量的同时对他使用'引入模式'来进行分类讨论。例如，下面是一个对 plus_1_neq_0 的更简短证明。（这种简写的缺点也显而易见，我们无法再记录在每个子目标中所使用的假设，而之前我们可以通过 *eqn*:*E* 将它们标注出来。）

```
Theorem plus_1_neq_0' : ∀ n : nat,
  (n + 1) =? 0 = false.
Proof.
  intros [|n].
  - reflexivity.
  - reflexivity. Qed.
```

如果没有需要命名的构造子参数，我们只需写上 [] 即可进行情况分析。

```
Theorem andb_commutative'' :
  ∀ b c, andb b c = andb c b.
Proof.
  intros [] [].
  - reflexivity.
  - reflexivity.
  - reflexivity.
  - reflexivity.
Qed.
```

练习：2 星, standard (andb_true_elim2) 证明以下断言, 当使用 `destruct` 时请用标号标出情况（以及子情况）。

```
Theorem andb_true_elim2 : ∀ b c : bool,
  andb b c = true → c = true.
```

Proof.

Admitted.

□

练习：1 星, standard (zero_nbeq_plus_1) Theorem zero_nbeq_plus_1 : ∀ n : nat, 0 =? (n + 1) = false.

Proof.

Admitted.

□

19.7 关于记法的更多内容 (可选)

（通常，标为可选的部分对于跟进本书其它部分的学习来说不是必须的，除了那些也标记为可选的部分。在初次阅读时，你可以快速浏览这些部分，以便在将来遇到时能够想起来这里讲了些什么。）

回忆一下中缀加法和乘法的记法定义：

```
Notation "x + y" := (plus x y)
                    (at level 50, left associativity)
                    : nat_scope.
Notation "x * y" := (mult x y)
                    (at level 40, left associativity)
                    : nat_scope.
```

对于 Coq 中的每个记法符号，我们可以指定它的 '优先级' 和 '结合性'。优先级 n 用 at level n 来表示，这样有助于 Coq 分析复合表达式。结合性的设置有助于消除表达式中相同符号出现多次时产生的歧义。比如，上面这组对 + 和 × 的参数定义的表达式 1+2*3*4 是 (1+((2*3)*4)) 的简写。Coq 使用 0 到 100 的优先级等级，同时支持 '左结合'、'右结合' 和 '不结合' 三种结合性。之后我们在别的章节会看到更多与此相关的例子，比如 Lists 一章。

每个记法符号还与 '记法范围（*Notation Scope*）'相关。Coq 会尝试根据上下文来猜测你所指的范围，因此当你写出 S(0*0) 时，它猜测是 *nat_scope*；而当你写出积（元组）类型 **bool**×**bool** 时，它猜测是 *type_scope*。有时你可能必须百分号记法 (x×y)%**nat** 来帮助 Coq 确定范围。另外，有时 Coq 打印的结果中也用 %**nat** 来指示记法所在的范围。

记法范围同样适用于数值记法（3、4、5、42 等等），因此你有时候会看到 0%**nat**，表示 0（即我们在本章中使用的自然数零 0），而 0%Z 表示整数零（来自于标准库中的另一个部分）。

专业提示：Coq 的符号机制不是特别强大，别期望太多。

19.8 不动点 Fixpoint 和结构化递归 (可选)

以下是加法定义的一个副本：

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

当 Coq 查看此定义时，它会注意到“plus' 的第一个参数是递减的”。这意味着我们对参数 n 执行了'结构化递归'。换言之，我们仅对严格递减的 n 值进行递归调用。这一点蕴含了“对 plus' 的调用最终会停止”。 Coq 要求每个 Fixpoint 定义中的某些参数必须是“递减的”。

这项要求是 Coq 的基本特性之一，它保证了 Coq 中定义的所有函数对于所有输入都会终止。然而，由于 Coq 的“递减分析”不是非常精致，因此有时必须用一点不同寻常的方式来编写函数。

练习：2 星, standard, optional (decreasing) 为了更好的理解这一点，请尝试写一个对于所有输入都_的确_终止的 Fixpoint 定义。但这个定义需要违背上述的限制，以此来让 Coq 拒绝。（如果您决定将这个可选练习作为作业，请确保您将您的解答注释掉以防止 Coq 拒绝执行整个文件。）

19.9 更多练习

Each SF chapter comes with a tester file (e.g. *BasicsTest.v*), containing scripts that check most of the exercises. You can run *make BasicsTest.vo* in a terminal and check its output to make sure you didn't miss anything.

练习：1 星, standard (identity_fn_applied_twice) 用你学过的策略证明以下关于布尔函数的定理。

```
Theorem identity_fn_applied_twice :
  ∀ (f : bool → bool),
  (∀ (x : bool), f x = x) →
  ∀ (b : bool), f (f b) = b.
Proof.
```

Admitted.

□

练习：1 星, standard (negation_fn_applied_twice) 现在声明并证明定理 *negation_fn_applied_twice*，与上一个类似，但是第二个前提说明函数 f 有 $f\ x = \mathsf{negb}\ x$ 的性质。

```
Definition manual_grade_for_negation_fn_applied_twice : option (nat×string) := None.
```

(The last definition is used by the autograder.)

□

练习：3 星, standard, optional (andb_eq_orb) 请证明下列定理。（提示：此定理的证明可能会有点棘手，取决于你如何证明它。或许你需要先证明一到两个辅助引理。或者，你要记得未必要同时引入所有前提。）

```
Theorem andb_eq_orb :
  ∀ (b c : bool),
  (andb b c = orb b c) →
  b = c.
Proof.
```

Admitted.

□

练习：3 星, standard (binary) 我们可以将对于自然数的一进制表示推广成更高效的二进制数表达方式。对于一个二进制数，我们可以将它看成一个由 A 构造子和 B 构造子组成的序列（它们分别表示 0 和 1），而这个序列的结束符为 Z。类似的，一个数的一进制表示可以看成一个由 S 构造子组成，并由 O 构造子结束的序列。 *)

For example:

decimal binary unary 0 Z O 1 B Z S O 2 A (B Z) S (S O) 3 B (B Z) S (S (S O)) 4 A (A (B Z)) S (S (S (S O))) 5 B (A (B Z)) S (S (S (S (S O)))) 6 A (B (B Z)) S (S (S (S (S (S O))))) 7 B (B (B Z)) S (S (S (S (S (S (S O)))))) 8 A (A (A (B Z))) S (S (S (S (S (S (S (S O))))))))

注意到在上面的表示中，二进制数的低位被写在左边而高位写在右边。（与通常的二进制写法相反，这种写法可以让我们在证明中更好的操作他们。）

```
Inductive bin : Type :=
  | Z
  | A (n : bin)
  | B (n : bin).
```

补全下面二进制自增函数 incr 的定义。并且补全二进制数与一进制自然数转换的函数 bin_to_nat。

```
Fixpoint incr (m:bin) : bin
  . Admitted.

Fixpoint bin_to_nat (m:bin) : nat
  . Admitted.
```

下面这些针对单增函数和二进制转换函数的“单元测试”可以验算你的定义的正确性。当然，这些单元测试并不能确保你的定义在所有输入下都是正确的！我们在下一章的末尾会重新回到这个话题。

```
Example test_bin_incr1 : (incr (B Z)) = A (B Z).
    Admitted.

Example test_bin_incr2 : (incr (A (B Z))) = B (B Z).
    Admitted.

Example test_bin_incr3 : (incr (B (B Z))) = A (A (B Z)).
    Admitted.

Example test_bin_incr4 : bin_to_nat (A (B Z)) = 2.
    Admitted.

Example test_bin_incr5 :
        bin_to_nat (incr (B Z)) = 1 + bin_to_nat (B Z).
    Admitted.

Example test_bin_incr6 :
```

bin_to_nat (*incr* (*incr* (B Z))) = 2 + *bin_to_nat* (B Z).

Admitted.

□

Chapter 20

Library LF.Preface

20.1 Preface: 前言

20.2 欢迎

这里是'软件基础'系列书籍的起点，本书阐明了可靠软件背后的数学根基。书中的主题包括基本的逻辑概念、计算机辅助定理证明、Coq 证明助理、函数式编程、操作语义、用于论证软件的逻辑和技术、静态类型系统、基于性质的随机测试、以及对实践中 C 代码的验证。本书可供高年级本科生、研究生、科研工作者及同等学力的广大读者学习参考。阅读本书无需具备逻辑学、程序语言等背景知识，但一定的数学基础有助于理解书中内容。

本书最大的创新之处在于，书中全文均已形式化并由机器检验。换言之，本书内容即为 Coq 脚本，可在 Coq 的交互界面下阅读。书中大部分习题也在 Coq 中完成。

本书中的文件都经过了精心组织：核心章节作为主线贯穿始终，涵盖了一学期的内容；"支线"中则包含附加的主题。所有核心章节都适合高年级本科生和研究生学习。

本书为第一卷'《逻辑基础》'，它向读者介绍了函数式编程的基本概念、构造逻辑以及 Coq 证明助理，为其余诸卷的学习奠定了基础。

20.3 概览

构建可靠的软件非常，非常地困难。现代系统的规模、复杂度、参与构建过程的人数，还有置于系统之上的需求范围，让构建或多或少地正确的软件变得极为困难，更不用说百

分之百地正确了。同时，由于信息处理技术继续渗透到社会的各个层面，人们为程序错误和漏洞付出的代价变得越来越高昂。

为了应对这些挑战，计算机科学家和软件工程师们发展了一套完整的提升软件质量的方法，从为管理软件项目的团队提供建议（如极限编程，Extreme Programming），到库的设计原理（如模型-视图-控制器，Model-View-Controller；发布-订阅模式， Publish-Subscribe）以及编程语言的设计哲学（面向对象编程，Object Oriented Programmin；面向剖面编程，Aspect Oriented Programming；函数式编程，Functional Programming），还有用于阐明和论证软件性质的数学技术，以及验证这些性质的工具。'《软件基础》'系列着重于最后一种方法。

本书将以下三种概念穿插在一起：

（1）'逻辑学'中的基本工具，用于准确地提出并论证关于程序的假设；

（2）'证明助理'用于构造严谨的逻辑论据；

（3）'函数式编程'思想，同时作为一种编程方法来简化程序的论证，以及架起程序和逻辑学之间的桥梁。

20.3.1 逻辑学

逻辑学是研究'证明'的领域，即对特定命题的真伪性进行不容置疑的论证。有关逻辑学在计算机科学中核心作用的书卷汗牛充栋。Manna 和 Waldinger 称之为"计算机科学的微积分"，而 Halpern 的论文 *'On the Unusual Effectiveness of Logic in Computer Science'* 中则收录了大量逻辑学为计算机科学提供的洞察力和至关重要的工具。的确，他们发现："实际上，逻辑学对计算机科学来说远比在数学中更加有效。这相当引人注目，特别是过去一百年来，逻辑学发展的动力大都来自于数学。"

具体来说，'归纳证明'的基本概念在计算机科学中无处不在。你以前肯定见过它们，比如说在离散数学或算法分析中。不过在本课程中，我们会在你未曾涉及的深度下对它进行探讨。

20.3.2 证明助理

逻辑学和计算机科学之间的思想交流并不是单向的，计算机科学也为逻辑学做出了重要的贡献，其中之一就是发展了帮助逻辑命题构造证明的软件工具。这些工具分为两大类：

- '自动定理证明器' 提供了一键式操作：它们接受一个命题，然后返回'真'或'假'（有时为'未知：超时'）。尽管它们的能力仅限于特定种类的推理，然而在近几年却快速

成熟，并应用到了多种场景中。此类工具包括 SAT 求解器，SMT 求解器以及模型检查器（Model Checker）。

- *'证明助理'* 是一种混合式工具，它能将证明的构建中比较常规的部分自动化，而更加困难的部分则依赖人类来解决。广泛使用的证明助理包括 Isabelle、Agda、Twelf、ACL2、PVS 以及 Coq 等等。

本课程围绕 Coq 展开，它是个自 1983 年以来主要在法国开发的证明助理，近年来吸引了大量来自研究机构和业界的社区用户。 Coq 为机器验证的形式化论证的交互式开发提供了丰富的环境。Coq 系统的内核是一个简单的证明检查器，它保证只会执行正确的推理步骤。在此内核之上，Coq 环境提供了高级的证明开发功能，包括一个庞大的库，其中包含各种定义和引理；强大策略，用于半自动化构造证明；还有一个专用的编程语言，能够为特殊情况定义新的自动证明策略。

Coq 已成为跨计算机科学和数学研究的关键推动者：

- 作为一个*'编程语言的建模平台'*， Coq 成为了研究员对复杂的语言定义进行描述和论证的标准工具。例如，它被用来检查 JavaCard 平台的安全性，得到了最高等级的通用准则验证，它还被用在 x86 和 LLVM 指令集以及 C 等编程语言的形式化规范中。

- 作为一个*'形式化软件验证的开发环境'*，Coq 被用来构建： CompCert，一个完全验证的 C 优化编译器； CertiKos，一个完全验证的工具，用于证明涉及浮点数的精妙算法的正确性； Coq 也是 CertiCrypt 的基础，一个用于论证密码学算法安全性的环境。 Coq 还被用来构建开源 RISC-V 处理器架构的验证实现。

- 作为一个*'依赖类型函数式编程的现实环境'*，Coq 激发了大量的创新。例如 Ynot 系统嵌入了"关系式霍尔推理"（一个 *'霍尔逻辑'* 的扩展，我们之后会看到它）。

- 作为一个*'高阶逻辑的证明助理'*，Coq 被用来验证数学中一些重要的结果。例如 Coq 可在证明中包含复杂计算的能力，使其开发出了第一个形式化验证的四色定理证明。此前数学家们对该证明颇有争议，因为它需要用程序对大量组态进行检验。在 Coq 的形式化中，所有东西都被检验过，自然也包括计算的正确性。近年来，Feit-Thompson 定理经过了更大的努力用 Coq 形式化了，它是对有限单群进行分类的十分重要的第一步。

顺便一提，如果你对 Coq 这个名字感到好奇，INRIA (法国国家研究实验室，Coq 主要在这里开发）上的 Coq 官方网站给出了解释："一些法国计算机科学家有用动物命名软

件的传统：像 Caml、Elan、Foc、Phox 都心照不宣地遵循这种默契。在法国，“Coq”是雄鸡，发音也像构造演算（Calculus of Constructions）的首字母缩写（CoC），它是 Coq 的基础。” 高卢雄鸡是法国的象征。C-o-q 还是 Thierry Coquand 名字的前三个字母，他是 Coq 的早期开发者之一。

20.3.3 函数式编程

*'函数式编程'*不仅表示可以在几乎任何编程语言中使用的各种习语（Idiom），还代表着一族以这些习语为侧重点设计的编程语言，包括 Haskell、OCaml、Standard ML、F#、Scala、Scheme、Racket、Common Lisp、Erlang 还有 Coq。

函数式编程已经有数十年的历史了–实际上，它甚至可以追溯到 1930 年代 Church 发明的 λ-演算，那时还没有电子计算机呢！自 90 年代初以来，函数式编程激起了业内软件工程师和语言设计者浓厚的兴趣，它们还在 Jane Street Capital、Microsoft、Facebook、Twitter 和 Ericsson 等公司的高价值系统中发挥着关键的作用。

函数式编程最根本的原则是，计算应当尽可能地 *'纯粹'*，也就是说，执行代码的唯一作用应当是只产生一个结果。计算应当没有 *'副作用'*，即它与输入/输出、可变量的赋值、指针重定向等相分离。例如，*'指令式'* 的排序函数会接受一个数字列表，通过重组指针使列表得以排序；而一个纯粹的排序函数则会接受一个列表，返回一个含有同样数字，但是已经排序的新列表。

这种编程风格最明显的好处之一，就是它能让程序变得更容易理解和论证。如果对某个数据结构的所有操作都会返回新的数据结构，而旧有的结构没有变动，那么我们便无需担心它的共享方式，因为程序中一部分的改变并不会破坏另一部分的属性。在并发程序中，线程间共享的每一个可变状态都是致命 Bug 的潜在来源，因此这方面的考虑尤为关键。事实上，业界最近对函数式编程的兴趣大部分来源于此，即它在并发中表现出的简单行为。

人们对函数式编程感到兴奋的另一原因与前文所述的原因相关：函数式程序通常比指令式程序更容易并行化和物理分布式化。如果一个计算除了产生结果之外没有其它的作用，那么它在 *'何时'* 执行便不再重要。同样，如果一个数据结构不会被破坏性地修改，那么它可以跨核心或网络地被随意复制。其实，“映射-归约”（Map-Reduce）的惯用法就是函数式编程的经典例子，它在大规模分布式查询处理器（如 Hadoop）中处于核心地位，并被 Google 用来索引整个互联网。

对本课程而言，函数式编程还有另一个重要的吸引力：它在逻辑和计算机科学之间架起了一座桥梁。事实上，Coq 本身即可视作一个小巧却有着极强表达能力的函数式编程语

言，以及一组用于陈述和证明逻辑断言的工具的结合体。进而言之，当我们更加深入地审视它时，会发现 Coq 的这两方面其实基于完全相同的底层机制 – '命题即类型，程序即证明'，可谓殊途同归。

20.3.4 扩展阅读

本书旨在自成一体，不过想要对特定主题进行深入研究的读者，可以在 *Postscript* 一章中找到推荐的扩展阅读。所有引用的参考文献可在 *Bib* 文件中找到。

20.4 实践指南

20.4.1 系统要求

Coq 可以在 Windows、Linux 和 macOS 上运行。我们需要：

- 安装近期版本的 Coq，它可以从 Coq 主页获得。本书中的文件均已通过了 Coq 8.9.1 的测试。
- 一个能跟 Coq 交互的 IDE。目前有两种选择：
  - Proof General 是一个基于 Emacs 的 IDE，Emacs 用户应该更喜欢这个。它需要单独安装（Google 搜索"Proof General"）。

    爱冒险的 Emacs 用户也可以试试 *company-coq* 和 *control-lock* 之类的扩展。
  - CoqIDE 是个更加简单的独立 IDE。它随 Coq 一起发布，所以如果你安装了 Coq，它应该就能用。你也可以从头编译安装它，不过在某些平台上还需要额外安装 GUI 之类的库。

    用户在运行 CoqIDE 时可以考虑关闭"异步"和"错误恢复"模式：

```
coqide -async-proofs off -async-proofs-command-error-resilience off Foo.v &
```

20.4.2 练习

每一章都包含大量的习题。每个习题都有标有"星级"，其含义是：

- 一星：很简单习题，强调课程的重点。对于大部分读者而言，一两分钟应该足够了。养成看到一个做一个的习惯。

- 二星：直截了当的习题（5 到 10 分钟）。
- 三星：需要一点思考的习题（10 分钟到半小时）。
- 四或五星：更加困难的习题（半小时以上）。

Those using SF in a classroom setting should note that the autograder assigns extra points to harder exercises:

1 star = 1 point 2 stars = 2 points 3 stars = 3 points 4 stars = 6 points 5 stars = 10 points

有些习题标注为“进阶”，有些习题标注为“可选”。只做非进阶、非可选的习题已经能将核心概念掌握得很不错了。可选习题会对一些关键概念提供额外的练习，还有一些读者可能会感兴趣的次级主题。进阶练习则留给想要更多挑战和更深理解的读者。

'请勿公布习题解答！'

《软件基础》已被广泛地用作自学教程和大学课程。如果习题解答很容易获得，那么本书的效用将大打折扣，对于会为作业评分的大学课程来说尤其如此。我们特别请求读者，切勿将习题答案放在任何能够被搜索引擎找到的地方。

20.4.3 下载 Coq 文件

本书的“英文发布版”以及所有源代码的压缩包（其中包含一组 Coq 脚本和 HTML 文件）可访问 https://softwarefoundations.cis.upenn.edu 获取。

本书的中文版和压缩包可访问 https://github.com/Coq-zh/SF-zh 获取。

如果你是在一门课程中使用本书的，那么你的教授可能会让你使用本地的修改版，此时你应当使用它们而非发布版，这样你可以获得所有该学期的本地更新。

20.4.4 章节依赖

章节之间的依赖关系图以及建议的学习路线可以在文件 *deps.html* 中查看。

20.4.5 推荐的引用格式

如果你想在自己的作品中引用本书，请采用以下格式：

@book {$FIRSTAUTHOR$: SFVOLUMENUMBER, author = {$AUTHORS, title =$ "VOLUMENAME“, series = "Software Foundations“, volume = "$VOLUMENUMBER$", $year =$ "VOLUMEYEAR“, publisher = "Electronic textbook“, note = {Version $VERSION, : //softwarefoundati$

20.5 资源

20.5.1 模拟题

宾夕法尼亚大学的 CIS500（软件基础）课程提供了大量的考试大纲，可访问 https://www.seas.upen
获取。近年来书中的记法有所变动，但大部分问题仍与本文对应。

20.5.2 课程视频

'《逻辑基础》'夏季加强班（DeepSpec 夏季班系列之一）的课程讲义可访问 https://deepspec.org/ev
和 https://deepspec.org/event/dsss18/ 获取。2017 年的视频清晰度不高，但在之后的课程
中会更好。

20.6 对授课员的要求

如果您有意用这些课件授课，那肯定会发现希望改进、提高或增加的材料。我们欢迎您的贡献!

为保证法律上的简单性和单一责任制，任何情况下都不应出现许可协议条款的的调整，授权的转移等等，我们要求所有贡献者（即，任何可访问开发者仓库的人）根据"作者记录"为他们的贡献赋予如下版权信息：

- I hereby assign copyright in my past and future contributions to the Software Foundations project to the Author of Record of each volume or component, to be licensed under the same terms as the rest of Software Foundations. I understand that, at present, the Authors of Record are as follows: For Volumes 1 and 2, known until 2016 as "Software Foundations" and from 2016 as (respectively) "Logical Foundations" and "Programming Foundations," and for Volume 4, "QuickChick: Property-Based Testing in Coq," the Author of Record is Benjamin C. Pierce. For Volume 3, "Verified Functional Algorithms", the Author of Record is Andrew W. Appel. For components outside of designated volumes (e.g., typesetting and grading tools and other software infrastructure), the Author of Record is Benjamin Pierce.

要参与贡献，请向 Benjamin Pierce 发送一封电子邮件，介绍一下自己，说明你打算如何使用这些材料，信中需包含 (1) 上面的版权移交文本，以及 (2) 你的 Github 用户名。

我们会赋予你访问 Git 源码库和开发者邮件列表的权限。你可以在源码库中找到 *INSTRUCTORS* 文件获取更多指示。

20.7 译本

感谢翻译志愿者团队的努力，'《软件基础》'已有日文版可以享用 http://proofcafe.org/sf。中文版还在填坑= =||

20.8 鸣谢

'《软件基础》'系列的开发，部分由国家科学基金会（National Science Foundation）在 NSF 科研赞助 1521523 号 '深度规范科学'下提供支持。