Coq 定理证明器

1 整数算数运算与大小比较

鸡兔同笼问题：

```
Fact chickens_and_rabbits: forall C R: Z,
  C + R = 35 ->
  2 * C + 4 * R = 94 ->
  C = 23.
```

字面意思上，这个命题说的是：对于任意整数 `C` 与 `R`，如果 `C + R = 35` 并且 `2 * C + 4 * R = 94`，那么 `C = 23`。其中，`forall` 与 `->` 是 Coq 中描述数学命题的常用符号。

Coq 表达式 1. 全称量词 forall . 在 Coq 中，`forall` 表示“任意”的意思，例如：

```
forall x: Z, x = x
```

就是一个语法上合法的 Coq 命题，在这个例子中 `Z` 表示整数集合，`forall x: Z, ...` 说的就是“对于任意整数 `x`，某性质成立”。在 `forall` 之后，可以跟一个变量也可以跟多个变量，例如：

```
forall x y: Z, x + y = y + x
```

也是合法的 Coq 命题。另外，`forall` 后的类型标注不是必须的，如果 Coq 系统能够推导出这个类型，那么就可以省略它。Coq 还允许一些 `forall` 之后的变量有类型标注，而另一些没有，例如：

```
forall (x: Z) y, x + y = y + x
```
。

Coq 表达式 2. 表示命题推导的箭头符号 `->` . 在 Coq 中，箭头符号 `->` 表示“如果... 那么...”，例如：

```
x >= 0 -> x + 1 > 0
```

就表示如果 `x` 大于等于 0，那么 `x+1` 大于 0。Coq 规定，这个箭头符号是右结合的，换言之，`P1 -> P2 -> P3` 实际是 `P1 -> (P2 -> P3)` 的简写，其表达的意思是：如果 `P1` 成立，那么 `P2` 能推出 `P3`。逻辑上，这等同于：如果 `P1` 并且 `P2`，那么 `P3`。因此，我们一般会将形如 `P1 -> P2 -> ... -> Pn -> Q` 的命题读作：如果 `P1`、`P2`、...、`Pn` 都成立，那么 `Q` 也成立。

上面的 Coq 代码中，除了逻辑符号 `forall`、`->` 与算数符号之外，还使用了保留字 `Fact`，这是一种 Coq 指令。

Coq 指令 1. Fact 指令. 在 Coq 中，Fact 指令可以用于陈述一个命题。例如，`chickens_and_rabbits` 是在上面 Coq 代码中命题的名字，之后从 `forall` 开头的逻辑算数表达式是这个命题的内容，命题的名字与命题的内容之间用冒号分隔。Coq 系统规定，只要这个命题语法上合法，那么整个 Fact 指令就是合法的。换言之，Coq 不会在执行 Fact 指令的时候检查其声明的命题是不是真命题。不过，执行 Fact 指令之后，用户需要进入 Coq 证明环境证明该结论。在 Coq 中，还有一些保留字与 Fact 功能相同，它们是：Proposition、Example、Lemma、Theorem 与 Corollary。

在 Fact 指令之后，我们可以在 Coq 中证明这个数学命题成立。如果要用中学数学知识完成这一证明，恐怕要使用加减消元法、代入消元法等代数技巧。Coq 并不需要我们掌握这些数学技巧，Coq 系统可以自动完成整数线性运算性质（linear integer arithmetic，简称 lia）的证明，`chickens_and_rabbits` 这一命题在 Coq 中的证明只需一行：

```
Proof. lia. Qed.
```

在这一行代码中，Proof 和 Qed 表示一段证明的开头与结尾，在它们之间的 `lia` 指令是证明脚本。

一般而言，编写 Coq 证明的过程是交互式的——“交互式”的意思是：在编写证明代码的同时，我们可以在 Coq 定理证明环境中运行证明脚本，获得反馈，让定理证明系统告诉我们“已经证明了哪些结论”、“还需要证明哪些结论”等等，并以此为依据编写后续的证明代码。安装 VSCoq 插件的 VSCode 编辑器、安装 proof-general 插件的 emacs 编辑器以及 CoqIDE 都是成熟易用的 Coq 定理证明环境。

以上面的证明为例，执行 `lia` 指令前，证明窗口显示了当前还需证明的性质（亦称为证明目标，proof goal）：

```
------------------------------------------
(1/1)
forall C R : Z,
C + R = 35 -> 2 * C + 4 * R = 94 -> C = 23
```

这里横线上方的是目前可以使用的前提，横线下方的是目前要证明的结论，目前，前提集合为空。横线下方的 (1/1) 表示总共还有 1 个证明目标需要证明，当前显示的是其中的第一个证明目标。利用证明脚本证明的过程中，每一条证明指令可以将一个证明目标规约为 0 个，1 个或者更多的证明目标。执行 `lia` 指令之后，证明窗口显示：Proof finished。表示证明已经完成。一般情况下，Coq 证明往往是不能只靠一条证明指令完成证明的。

Coq 指令 2. Proof 指令与 Qed 指令. Proof 与 Qed 是一段证明的首尾标识，在它们之间的 Coq 指令都是证明脚本。在 Coq 中，用户通过证明脚本完成证明。一般情况下，Coq 证明脚本都能保证其进行的逻辑变换与逻辑规约都是合法的，特殊情况下，Coq 定理证明系统还需要在 Qed 指令时进行额外检验。经过 Qed 检验后，一个数学命题的 Coq 证明才算完成。

Coq 证明脚本 1. lia 指令. 证明指令 `lia` 表示自动证明有关整数线性运算与大小关系的性质，lia 这三个字母是 linear integer arithmetic 的缩写。证明指令 `lia` 是完备的，换言之，所有正确的整数线性运算性质都能够通过这一指令设定的算法完成自动证明。当然，在实际使用中，可能由于待证明的命题规模太大（变量个数太多、约束条件中的表达式太长或约束条件数量太多），算法所需运行时间太长，Coq 系统将其提前终止，因而无法完成自动证明。

Coq 证明指令 `lia` 除了能够证明关于线性运算的等式，也可以证明关于线性运算的不等式。下面这个例子选自熊斌教授所著《奥数教程》的小学部分：幼儿园的小朋友去春游，有男孩、女孩、男老师与女老师共 16 人，已知小朋友比老师人数多，但是女老师比女孩人数多，女孩又比男孩人数多，男孩比男老师人数多，请证明幼儿园只有一位男老师。

```
Fact teachers_and_children: forall MT FT MC FC: Z,
  MT > 0 ->
  FT > 0 ->
  MC > 0 ->
  FC > 0 ->
  MT + FT + MC + FC = 16 ->
  MC + FC > MT + FT ->
  FT > FC ->
  FC > MC ->
  MC > MT ->
  MT = 1.
Proof. lia. Qed.
```

习题 1. 请在 Coq 中描述下面结论并证明：如果今年甲的年龄是乙 5 倍，并且 5 年后甲的年龄是乙的 3 倍，那么今年甲的年龄是 25 岁。

除了线性性质之外，Coq 中还可以证明的一些更复杂的性质。例如下面就可以证明，任意两个整数的平方和总是大于它们的乘积。证明中使用的指令`nia`表示的是非线性整数运算（nonlinear integer arithmetic）求解。

```
Fact sum_of_sqr1: forall x y: Z,
  x * x + y * y >= x * y.
Proof. nia. Qed.
```

不过，`nia`与`lia`不同，后者能够保证关于线性运算的真命题总能被自动证明（规模过大的除外），但是有不少非线性的算数运算性质是`nia`无法自动求解的。例如，下面例子说明，一些很简单的命题`nia`都无法完成自动验证。

```
Fact sum_of_sqr2: forall x y: Z,
  x * x + y * y >= 2 * x * y.
Proof. Fail nia. Abort.
```

这是就需要我们通过编写证明脚本，给出中间证明步骤。证明过程中，可以使用 Coq 标准库中的结论，也可以使用我们自己实现证明的结论。例如，Coq 标准库中，`sqr_pos`定理证明了任意一个整数`x`的平方都是非负数，即：

```
sqr_pos: forall x: Z, x * x >= 0
```

我们可以借助这一性质完成上面`sum_of_sqr2`的证明。

```
Fact sum_of_sqr2: forall x y: Z,
  x * x + y * y >= 2 * x * y.
Proof.
  intros.
  pose proof sqr_pos (x - y).
  nia.
Qed.
```

这段证明有三个证明步骤。证明指令`intros`将待证明结论中的逻辑结构“对于任意整数 x 与 y”移除，并在前提中的“x 与 y 是整数”的假设。第二条指令`pose proof`表示对 x-y 使用标准库中的定理`sqr_pos`。从 Coq 定理证明界面中可以看到，使用该定理得到的命题会被添加到证明目标的前提中去，Coq 系统将这个新命题自动命名为 H（表示 Hypothesis）。最后，`nia`可以自动根据当前证明目标中的前提证明结论。

下面证明演示了如何使用我们刚刚证明的性质`sum_of_sqr1`。

```
Example quad_ex1: forall x y: Z,
  x * x + 2 * x * y + y * y + x + y + 1 >= 0.
Proof.
  intros.
  pose proof sum_of_sqr1 (x + y) (-1).
  nia.
Qed.
```

下面这个例子说的是：如果`x < y`，那么`x * x + x * y + y * y`一定大于零。

```
Fact sum_of_sqr_lt: forall x y: Z,
  x < y ->
  x * x + x * y + y * y > 0.
```

我们可以利用下面两式相等证明：

```
4 * (x * x + x * y + y * y)
3 * (x + y) * (x + y) + (x - y) * (x - y)
```

于是，在`x < y`的假设下，等式右边的两个平方式一个恒为非负，一个恒为正。因此，等式的左边也恒为正。

```
Proof.
  intros.
  pose proof sqr_pos (x + y).
  nia.
Qed.
```

可以看到，在`x < y`的前提下，Coq 的`nia`指令可以自动推断得知`(x - y)`的平方恒为正。不过，我们仍然需要手动提示 Coq，`(x + y)`的平方恒为非负。

Coq 证明脚本 2. intros 指令. 证明指令`intros`表示将待证明结论中的假设移动到证明目标的前提中去。例如，在上面`sum_of_sqr_lt`中，`intros`指令移动了三项前提：`x: Z`、`y: Z`与`H: x < y`。其中`H`是 Coq 定理证明系统自动引入的命名，字母 H 表示 Hypothesis 的简写，当`intros`要添加若干个命题作为前提的时候，Coq 会依次选择`H`、`H0`、`H1`等名字。有时，我们在 Coq 中编写证明脚本代码时，希望能够手动控制这些前提的命名，这只需要在`intros`后添加参数就可以了。例如，`sum_of_sqr_lt`中的`intros`指令就等效于`intros x y H`。Coq 允许我们在`intros`的同时对`forall`后的变量重命名，例如，将`sum_of_sqr_lt`中的`intros`指令改为`intros x1 x2 H`后效果如下。Coq 也允许对一部分前提手动命名，而同时对另一部分前提自动命名，只需用问号占位符表示自动命名的前提即可，例如`intros ? ? H`。

Coq 证明脚本 3. pose proof 指令. 证明指令`pose proof`表示在当前证明中使用一条已经证明过定理或者使用当前证明目标中的一条前提。例如，标准库中已有定理`sqr_pos`

```
sqr_pos: forall x: Z, x * x >= 0
```

那么`pose proof sqr_pos (x + 1)`就会得到`(x + 1) * (x + 1) >= 0`。类似的，假设当前证明目标中有下述前提，

```
x: Z
H: x >= 0
H0: x >= 0 -> x + 1 > 0
```

那么，就可以通过`pose proof H H0`得到`x + 1 > 0`。另外，使用`pose proof`指令时未必需要将所有的前提全部填上，如 Coq 标准库中的`Zmult_ge_compat_r`是下面定理：

```
forall n m p : Z, n >= m -> p >= 0 -> n * p >= m * p
```

假设当前证明目标中有下述前提，

```
k1: Z
k2: Z
x: Z
H: k1 >= k2
H0: x * x >= 0
```

那么，就可以通过以下`pose proof`指令

```
pose proof Zmult_ge_compat_r k1 k2 (x * x) H
pose proof Zmult_ge_compat_r k1 k2 (x * x) H H0
pose proof Zmult_ge_compat_r (x * x) 0 5 H0 ltac:(lia)
```

分别得到以下结论：

```
x * x >= 0 -> k1 * (x * x) >= k2 * (x * x)
k1 * (x * x) >= k2 * (x * x)
x * x * 5 >= 0 * 5
```

可以看到，填写前提中的命题部分时，既可以填写已有前提的名称（如`H`、`H0`等），也可以填写一条证明指令，如`ltac:(lia)`。除此之外，如果`pose proof`指令的一些参数可以由另一些参数推导出来，那么可以用下划线省去这些参数。例如，下面这几条证明指令的效果和上面证明指令的效果时相同的。

```
pose proof Zmult_ge_compat_r _ _ (x * x) H
pose proof Zmult_ge_compat_r _ _ _ H H0
pose proof Zmult_ge_compat_r _ _ 5 H0 ltac:(lia)
```

最后，在 Coq 中还可以指明`pose proof`所生成新命题的名称。例如，

```
pose proof Zmult_ge_compat_r _ _ 5 H0 ltac:(lia) as H5xx
```

得到的新命题是：`H5xx: x * x * 5 >= 0 * 5`。

Coq 证明脚本 4. nia 指令. 证明指令`nia`表示自动证明有关整数非线性算数运算的性质，nia 这三个字母是 nonlinear integer arithmetic 的缩写。证明指令`nia`是不完备的，但是它能够自动完成多项式的展开与线性性质的推理。另外，它也能自动推到乘法与正负数之间的关系。

2 函数与谓词

函数是一类重要的数学对象。例如，“加一”这个函数往往可以写作：$f(x) = x + 1$。在 Coq 中，我们可以用以下代码将其定义为`plus_one`函数。

```
Definition plus_one (x: Z): Z := x + 1.
```

在类型方面，`plus_one (x: Z): Z`表示这个函数的自变量和值都是整数，而`:=`符号右侧的表达式`x + 1`也描述了函数值的计算方法。

我们知道，“在 1 的基础上加一”结果是 2，“在 1 的基础上加一再加一”结果是 3。这些简单论断都可以用 Coq 命题表达出来并在 Coq 中证明。

```
Example One_plus_one: plus_one 1 = 2.
Proof. unfold plus_one. lia. Qed.
```

```
Example One_plus_one_plus_one: plus_one (plus_one 1) = 3.
Proof. unfold plus_one. lia. Qed.
```

Coq 表达式 3. 包含函数的表达式. 在 Coq 中，某函数`F`作用于某参数`X`写作`F X`，不需要写括号。这一语法类似于 Ocaml 等函数式编程语言。另外，这一语法是左结合的。换言之，表达式`F X Y`是`(F X) Y`的简写，而表达`F (G X)`时必须添加括号。

Coq 证明脚本 5. unfold 指令. 证明指令`unfold`表示在待证明的结论中展开某项定义。如果要在证明目标的某前提`H`中展开`X`的定义，可以使用证明指令`unfold X in H`。

下面是更多函数的例子，我们可以采用类似的方法定义平方函数。

```
Definition square (x: Z): Z := x * x.
```

```
Example square_5: square 5 = 25.
Proof. unfold square. lia. Qed.
```

Coq 中也可以定义多元函数。

```
Definition smul (x y: Z): Z := x * y + x + y.
```

```
Example smul_ex1: smul 1 1 = 3.
Proof. unfold smul. lia. Qed.
```

```
Example smul_ex2: smul 2 3 = 11.
Proof. unfold smul. lia. Qed.
```

Coq 表达式 4. 包含多元函数的表达式. Coq 中的二元函数实质上是接收一个参数后会计算得到一个一元函数的函数。例如，当`F`是一个二元函数时，我们通常将“`F`作用于`X`与`Y`的结果”写作`F X Y`，即`(F X) Y`的简写。这是因为`F X`实质上是一个一元函数，当他再接收一个参数`Y`之后的计算结果就是`(F X) Y`。类似的，Coq 中的三元函数实质上是接收一个参数后会计算得到一个二元函数的函数；Coq 中的 n+1 元函数实质上是接收一个参数后会计算得到一个 n 元函数的函数。

下面 Coq 代码定义了“非负”这一概念。在 Coq 中，可以通过定义类型为`Prop`的函数来定义谓词。以下面定义为例，对于每个整数`x`，`:=`符号右侧表达式`x >= 0`是真还是假决定了`x`是否满足性质`nonneg`（即，非负）。

```
Definition nonneg (x: Z): Prop := x >= 0.
```

```
Fact nonneg_plus_one: forall x: Z,
  nonneg x -> nonneg (plus_one x).
Proof. unfold nonneg, plus_one. lia. Qed.
```

```
Fact nonneg_square: forall x: Z,
  nonneg (square x).
Proof. unfold nonneg, square. nia. Qed.
```

习题 2. 请在 Coq 中证明下面结论。

```
Fact nonneg_smul: forall x y: Z,
  nonneg x -> nonneg y -> nonneg (smul x y).
(* 请在此处填入你的证明，以_[Qed]_结束。 *)
```

程序语言的语法

下面三组 C 程序语句中，每一组内的两句程序语句是相同的程序语句吗？第一组：

```
y=x+1; // 代码中的空格更少
```

```
y = x + 1; // 代码中的空格更多
```

第二组：

```
y = (x) + 1; // 代码中的包含多余的括号
```

```
y = x + 1; // 代码中无多余的括号
```

第三组：

```
y = 1 + x;
```

```
y = x + 1;
```

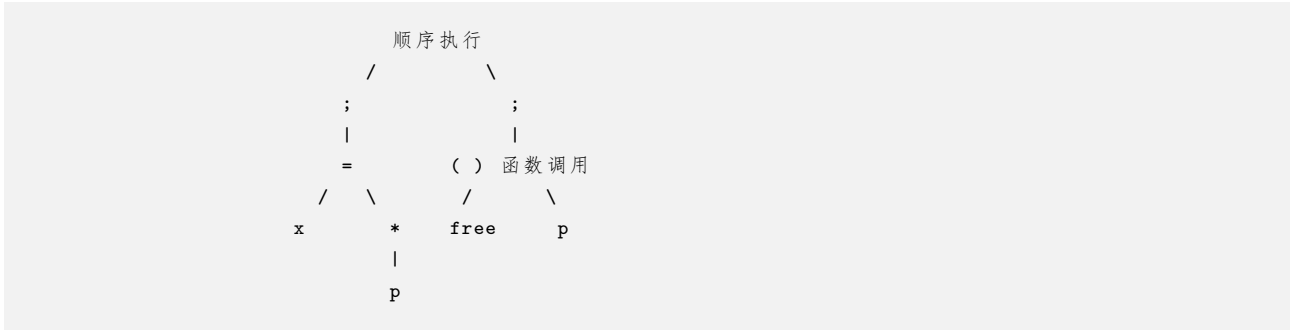
对于描述程序行为与程序正确性的理论而言，像第一组例子中的多余空格与第二组例子中的多余括号并不重要，因此，我们认为第一组与第二组都包含了相同的程序语句。而第三组中的两句程序语句则是不同的程序语句。

C 表达式`* (p + 1) ++`的结构：

```
          ++ (后缀运算符)
           |
           *
           |
          ( )
           |
           +
          / \
         p   1
```

又例如，C 表达式`x = * p; free(p);`可以理解成为下面树结构。

C 表达式`x = * p; free(p);`的结构：

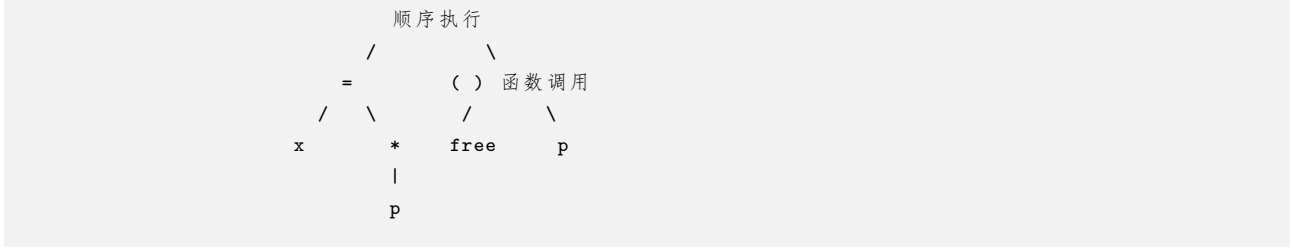


这些树结构中，有些信息是有些多余的。例如，上一个例子中的括号是多余信息。精简后可以得到以下树结构。

C 表达式`* (p + 1) ++`的结构：

类似的，C 表达式`x = * p; free(p);`的结构也可以简化。

C 表达式`x = * p; free(p);`的结构：



我们就把这样子精简之后的树结构称为这个程序的抽象语法树（Abstract Syntax Tree, AST）。

词法分析

1 While 语言

- 常数
  - `N ::= 0 | 1 | ...`
  - While 语言的常数是以非 0 数字开头的一串数字或者 0。
- 保留字
  - While 语言的保留字有：var, if, then, else, while, do
- 变量名
  - `V ::= ...`
  - While 语言变量名的第一个字符为字母或下划线，while 语言的变量名可以包含字母、下划线与数字。
  - 保留字不是变量名。
  - 例如：`a0`, `__x`, `leaf_counter`等等都可以是 while 语言的变量名。
- 表达式
  - `E :: = N | V | -E | E+E | E-E | E*E | E/E | E%E |`
    `E<E | E<=E | E==E | E!=E | E>=E | E>E |`
    `E&&E | E||E | !E`
  - 优先级：`|| < && < ! < Comparisons < +,- < *,/,%`
  - 同优先级运算符之间左结合，可以使用小括号改变优先级。
  - 例如：`a0+1`, `x<=y&&x>0`等等都是 while 语言的表达式。
- 语句

```
C :: = var V |
       V = E |
       C; C |
       if (E) then { C } else { C } |
       while (E) do { C }
```

到这里为止我们就定义了一个最简单的程序语言 while 语言。我们已经可以用这个程序语言写一些比较复杂的程序了。

2 While 语言 + Dereference + Built-in functions

表达式

```
E ::= N | V | -E | E+E | E-E | E*E | E/E | E%E |
      E<E | E<=E | E==E | E!=E | E>=E | E>E |
      E&&E | E||E | !E | *E |
      malloc(E) | read_int() | read_char()
```

语句

```
C ::= var V |
      write_int(E) |
      write_char(E) |
      E = E |
      C; C |
      if (E) then { C } else { C } |
      while (E) do { C }
```

While 程序示例 1

```
var x;
var n;
var flag;
x = read_int();
n = 2;
flag = 1;
while (n * n <= x && flag) do {
  if (x % n == 0)
  then { flag = 0 }
  else { n = n + 1 }
};
write_int(flag)
```

While 程序示例 2

```
var x;
var l;
var h;
var mid;
x = read_int();
l = 0;
h = x + 1;
while (l + 1 < h) do {
  mid = (l + h) / 2;
  if (mid * mid <= x)
  then { l = mid }
  else { h = mid }
};
write_int(l)
```

While 程序示例 3

```
  var n;
  var s;
  s = 0;
  n = 1;
  while (n != 0) do {
    n = read_int();
    s = s + n
  };
  write_int(s)
```

3 词法分析的基本知识

词法分析是编译器前端的第一步，它的主要任务是将源代码的字符串切分为一个一个的标记（token）。例如在我们的 While+DB 语言中有以下几类标记：

- 运算符：`+ - * / % < <= == != >= > ! && ||`
- 赋值符号：`=`
- 间隔符：`( ) { } ;`
- 自然数：`0 1 2 ...`
- 变量名：`a0 __x ...`
- 保留字：`var if then else while do`
- 内置函数名：`malloc read_char read_int write_char write_int`

根据不同标记承担的语法功能是否相同，可以将标记分为若干类。事实上，在上述列举的标记除了所有自然数分为一类，所有变量名分为一类之外，其他每个标记应当单独分为一类。下面是标记分类在 C 语言中的定义（lang.h）。

```
enum token_class {
  // 运算符
    TOK_OR = 1, TOK_AND, TOK_NOT,
    TOK_LT, TOK_LE, TOK_GT, TOK_GE, TOK_EQ, TOK_NE,
    TOK_PLUS, TOK_MINUS, TOK_MUL, TOK_DIV, TOK_MOD,
  // 赋值符号
    TOK_ASGNOP,
  // 间隔符号
    TOK_LEFT_BRACE, TOK_RIGHT_BRACE,
    TOK_LEFT_PAREN, TOK_RIGHT_PAREN,
    TOK_SEMICOL,
  // 自然数
    TOK_NAT,
  // 变量名
    TOK_IDENT,
  // 保留字
    TOK_VAR, TOK_IF, TOK_THEN, TOK_ELSE, TOK_WHILE, TOK_DO,
  // 内置函数名
    TOK_MALLOC, TOK_RI, TOK_RC, TOK_WI, TOK_WC
};
```

其中，`TOK_NAT`与`TOK_IDENT`两类标记需要额外存储他们的值。这个 C 语言的 `union`说的是，如果标记属于`TOK_NAT`，那么就在`n`这个域中存储这个标记表示的无符号整数值，如果标记属于`TOK_IDENT`，那么就在`i`这个域中存储这个标识符对应的字符串的地址，其他情况下，不需要存储额外的信息。

```
union token_value {
  unsigned int n;
  char * i;
  void * none;
};
```

4 正则表达式

正则表达式是用于描述字符串集合的一种语言。正则表达式本身的语法结构定义如下：

- $r ::= c \mid \epsilon \mid r|r \mid rr \mid r^*$
- 优先级：* > 连接 > |

具体而言，正则表达式可以表达五种意思。第一，单个字符；第二，空字符串；第三，两个字符串的连接；第四：两类字符串的并集；第五：重复出现一类字符串 0 次 1 次或多次。例如：

- `(a|b)c` 表达的字符串有 `ac`, `bc`。
- `(a|b)*` 表达的字符串有空串, `a`, `b`, `ab`, `aaaaa`, `babbb` 等。
- `ab*` 表达的字符串有 `a`, `ab`, `abb`, `abbb`, `abbbb` 等。
- `(ab)*` 表达的字符串有空串, `ab`, `abab` 等。

下面是正则表达式中的一些常见简写：

- 字符的集合：例如 `[a-cA-C]` 表示 `a|b|c|A|B|C`。
- 可选的字符串：$r?$ 表示 $r|\epsilon$。例如 `a?b` 表达的字符串有 `b`, `ab`。
- 字符串：例如 `"abc"` 表达的字符串是 `abc`。
- 重复至少一次：例如 `a+` 表达的字符串是 `a`, `aa`, `aaa`, 等等。
- 自然数常量：`"0"|[1-9][0-9]*`。
- 标识符（不排除保留字、内置函数名）：`[_a-zA-Z][_a-zA-Z0-9]*`。

5 Flex 词法分析工具

安装 Flex

- 请预先在电脑上安装 flex，windows 系统上的安装方式请参考：
- `blog.csdn.net/m944256098a/article/details/104992880`
- 输入（.l 文件）：基于正则表达式的词法分析规则
- 输出（.c 文件）：用 C 语言实现的词法分析器

输入文件头

- 说明 Flex 生成词法分析器的基本参数
- 指定所生成的词法分析器的文件名
- 词法分析器所需头文件 (.h 文件) 与辅助函数

- 例如（lang.l）：

```
%option noyywrap yylineno
%option outfile="lexer.c" header-file="lexer.h"
%{
#include "lang.h"
%}
```

词法分析用到的辅助函数和全局变量 lang.h

```
extern union token_value val;
unsigned int build_nat(char * c, int len);
char * new_str(char * str, int len);
void print_token(enum token_class c);
```

词法分析规则

- 所有词法分析规则都放在一组 `%%` 中。
- 每一条词法分析规则由匹配规则和处理程序两部分构成。
- 匹配规则用正则表达式表示。
- 处理程序是一段 C 代码。
- 例如（lang.l）：

```
%%
0|[1-9][0-9]* {
    val.n = build_nat(yytext, yyleng);
    return TOK_NAT;
}
"var" { return TOK_VAR; }
...
%%
```

Flex 专有 C 程序变量

- `yytext`：经过词法分析切分得到的字符串。
- `yyleng`：经过词法分析切分得到的字符串的长度。

Flex 词法分析的附加规则

- 如果有多种可行的切分方案，则选择长度较长的方案。
- 如果同一种切分方案符合多个词法分析规则（正则表达式），则选择先出现的规则。
- `==` 会匹配 `==` 而不是 `=`。
- 标识符的词法分析规则不需要排除保留字，只需在 Flex 中把保留字放在标识符之前即可。

lang.l

```
%%
...
"write_int" { return TOK_WI; }
"write_char" { return TOK_WC;}
[_A-Za-z][_A-Za-z0-9]* {
    val.i = new_str(yytext, yyleng);
    return TOK_IDENT;
}
";" { return TOK_SEMICOL; }
"(" { return TOK_LEFT_PAREN; }
...
%%
```

main.c （只打印词法分析结果）

```
#include "lang.h"
#include "lexer.h"
int main() {
  enum token_class c;
  while (1) {
    c = yylex();
    if (c != 0) {
      print_token(c);
    }
    else {
      break;
    }
  }
}
```

编译 Makefile

```
lang.o: lang.c lang.h
        gcc -c lang.c
lexer.c: lang.l
        flex lang.l
lexer.o: lexer.c lang.h
        gcc -c lexer.c
main.o: main.c lang.h lexer.c
        gcc -c main.c
main: main.o lang.o lexer.o
        gcc main.o lang.o lexer.o -o main
%.c: %.l
```

测试 sample00.jtl

```
  var n;
  var m;
  n = read_int();
  m = n + 1;
  write_int(m + 2)
```

```
Qinxiang$ ./main < sample00.jtl
VAR
IDENT(n)
SEMICOL
VAR
IDENT(m)
...
```

Flex 生成的词法分析器其算法流程如下：先根据输入的字符串找到正则表达式的匹配结果（若有多种结果则选最长、最靠前的），再更新字符串扫描的初始地址，并执行匹配结果对应的处理程序。

如果处理程序中有`return`语句中断了词法分析器`yylex()`的执行，下一次重新启动`yylex()`时会从新的字符串扫描初始地址开始继续进行词法分析。如果处理程序中没有`return`语句中断，那么词法分析器就会自动继续执行后续的词法分析。

因此，处理程序中可以不用每次 return，相应的也不用反复重新启动`yylex()`。下面为代码概要：

lang.l

```
%%
0|[1-9][0-9]* {
    val.n = build_nat(yytext, yyleng);
    print_token(TOK_NAT);
}
"var" { print_token(TOK_VAR); }
...
%%
```

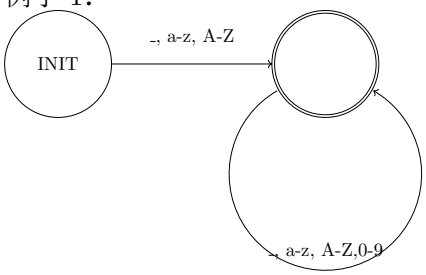
main.c

```
int main() {
  yylex();
}
```

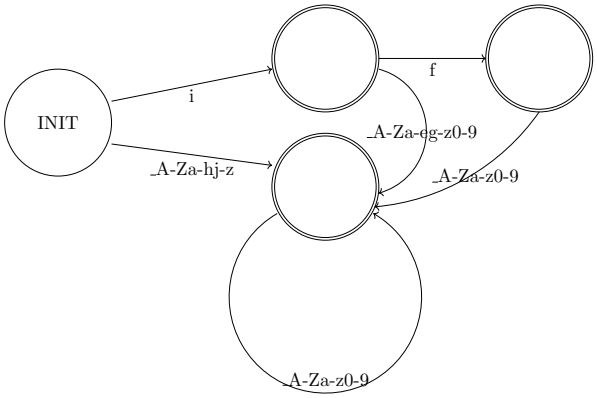
6 有限状态自动机

- 有限状态自动机包含一个状态集（图中的节点）与一组状态转移规则（图中的边）。
- 每一条状态转移规则上有一个符号
- 状态集中包含一个起始状态，和一组终止状态；起始状态也可能是一个终止状态。

例子 1：

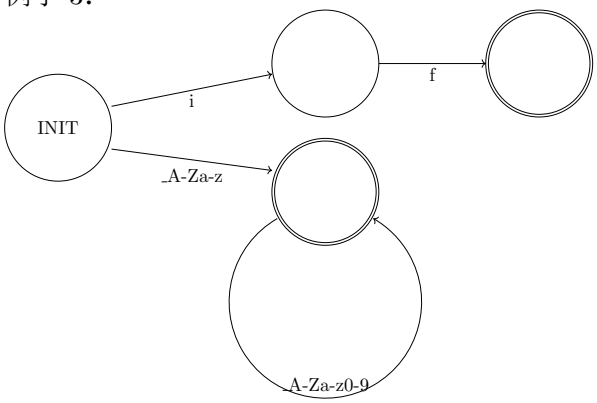


例子 2：



下面这个自动机的例子与前面不同，从起点出发，经过字符 `i` 能到达的节点有两个，这也是自动机。

例子 3：

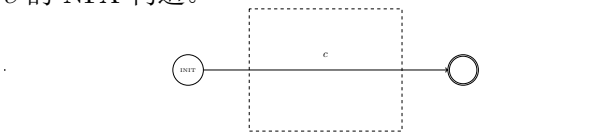


- 确定性有限状态自动机（DFA，Deterministic Finite Automata）
  - 每一条状态转移规则上的符号都是 ascii 码字符。
  - 从任何一个状态出发，每个字符至多对应一条状态转移规则。
- 非确定性有限状态自动机（NFA，Nondeterministic Finite Automata）
  - 每一条状态转移规则上的符号要么是 ascii 码字符，要么是 ϵ。
  - 从一个状态出发，每个符号可能对应多条状态转移规则。
  - 通过 ϵ 转移规则时，不消耗字符串中的字符。
- 定义：一个自动机能接受一个字符串，当且仅当存在一种状态转移的方法，可以从自动机的起始状态出发，依次经过这个字符串的所有字符，到达一个终止状态。

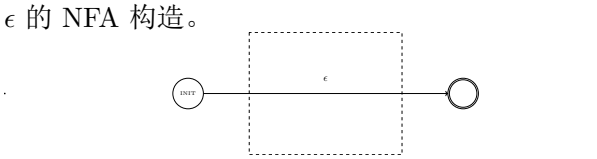
7 将正则表达式转化为 NFA

每个正则表达式构造一个 NFA，使得一个字符串能被这个 NFA 接受当且仅当这个字符串是正则表达式表示的字符串集合的元素。在下面定义的构造中，我们构造出的 NFA 总之只有一个起点、一个终点。但是请特别注意：起点也可能在图中有入度，终点也可能在图中有出度。

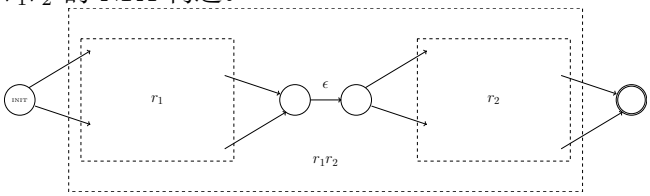
c 的 NFA 构造。



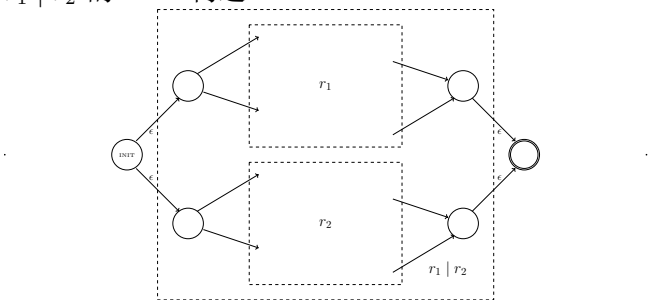
ϵ 的 NFA 构造。



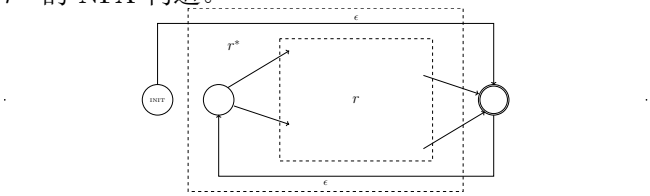
$r_1 r_2$ 的 NFA 构造。



$r_1 \mid r_2$ 的 NFA 构造。



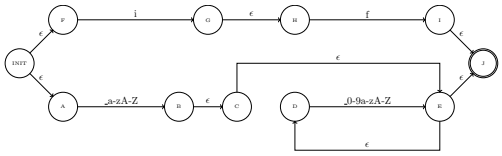
r^* 的 NFA 构造。



8 将 NFA 转化为 DFA

关键方法：考虑在 NFA 中所有当前可能处于的状态的集合。
例子：

- 正则表达式：`if|[_a-zA-Z][_0-9a-zA-Z]*`
- NFA：



- 起始状态：{ INIT, A, F } 空串，可能在INIT，A，F三个节点

  i: { INIT, A, F } → { B, C, D, E, G, H, J }

  f: { B, C, D, E, G, H, J } → { D, E, I, J }

  _0-9a-zA-Z: { D, E, I, J } → { D, E, J }

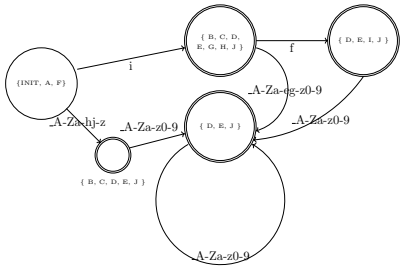
  _0-9a-zA-Z: { D, E, J } → { D, E, J }

  _0-9a-eg-zA-Z: { B, C, D, E, G, H, J } → { D, E, J }

  _0-9a-hj-zA-Z: { INIT, A, F } → { B, C, D, E, J }

  _0-9a-zA-Z: { B, C, D, E, J } → { D, E, J }

- 构造生成的 DFA： 理论上来说是2的N次方，但是实际情况（语法分析）比较简单


{ B, C, D,
E, G, H, J }
{ D, E, I, J }
{INIT, A, F}
i
f
_A-Za-eg-z0-9
_A-Za-hj-z
_A-Za-z0-9
{ D, E, J }
_A-Za-z0-9
{ B, C, D, E, J }
_A-Za-z0-9

语法分析

1 语法分析的任务

语法分析的主要任务是在词法分析结果的基础上，进一步得到解析树（parsing tree）。例如，下面两个算术表达式：

```
1 + x * y
```

```
(1 + x) * y
```

它们经过词法分析结果如下：

```
TOK_NAT(1) TOK_PLUS TOK_IDENT(x) TOK_MUL TOK_IDENT(y)
```

```
TOK_LEFT_PAREN TOK_NAT(1) TOK_PLUS TOK_IDENT(x)
TOK_RIGHT_PAREN TOK_MUL TOK_IDENT(y)
```

```
      *
     / \
   ( )   y
    |
    +
   / \
  1   x
```

```
      *                    *
     / \                  / \
   ( )   y              +   y
    |                  / \
    +                1   x
   / \
  1   x
```

其中，`(1 + x) * y`这个式子我们给出了两个树结构，一个是直接反应字符串内容的树结构，另一个是省去了括号这一冗余结构之后的结果，这个我们之前介绍过了，我们称其为抽象语法树。

2 上下文无关语法与解析树

本课程中将主要介绍基于上下文无关语法（Context-free grammar）的移入规约分析算法（shift-reduce parsing）。上下文无关语法从形式化与抽象语法树的语法规定是类似的，但是它也要处理括号等抽象语法树不关注的源代码信息。一套上下文无关语法包含若干条产生式（production），例如：

顺序执行　　表达式　　表达式列表

```
S -> S ; S          E -> ID          L -> E
S -> ID := E        E -> NAT         L -> L , E
S -> PRINT ( L )    E -> E + E
     打印输出       E -> ( E )
```

```
S -> S; S
  -> ID := E; S
  -> ID := E + E; S
  -> ID := ID + E; S
  -> ID := ID + NAT; S
  -> ID := ID + NAT; PRINT(L)
  -> ID := ID + NAT; PRINT(L, E)
  -> ID := ID + NAT; PRINT(E, E)
  -> ID := ID + NAT; PRINT(ID, E)
  -> ID := ID + NAT; PRINT(ID, ID)
```

在上面例子中，`S`表示语句，`E`表示表达式，`L`表示表达式列表。在这个语言中，`PRINT`指令可以带多个参数，表达式中只允许出现加法运算，多条语句只允许顺序执行，没有条件分支与循环。

上下文无关语法（定义）

- 一个初始符号，例如上面例子中的：`S`；
- 一个终结符（terminal symbols）集合，例如上面例子中的：`ID NAT , ; ( ) + :=`；
- 一个非终结符（nonterminal symbols）集合，例如上面例子中的：`S E L`，当词法分析器和语法分析器结合使用的时候，这个非终结符集合一般就是词法分析中的标记集合；
- 一系列产生式，每个产生式的左边是一个非终结符，每个产生式的右边是一列（可以为空）终结符或非终结符。

解析树

- 根节点为上下文无关语法的初始符号；
- 每个叶子节点是一个终结符，每个内部节点是一个非终结符；
- 每一个父节点和他的子节点构成一条上下文无关语法中的产生式；

对于所有标志串，至多只有一种解析树可以生成。

3　歧义与歧义的消除

ID + ID * ID，我们想要默认先做乘法；但是可能有两种解析树。

可能有歧义：

可能有不止一种解析树

```
E -> ID      E -> E + E      E -> E * E      E -> ( E )
```

下面语法能够消除上面标记串语法分析中的歧义。

```
E -> F       E -> E + E      F -> F * F
F -> ( E )   F -> ID
```

如果考虑 ID + ID + ID, 还是有歧义

E – F, E – E + F, F – F*G, F – G, G – (E), G– ID

```
          E
        / | \                          E
       E  +  E                         -> E + E
       |     |                         -> E + F
       F     F                         -> E + F * F
       |   / | \                       -> E + F * ID
      ID   F  *  F                      -> E + ID * ID
           |     |                      -> F + ID * ID
          ID    ID                      -> ID + ID * ID
```

.

4 派生与规约

- 一个派生中，如果每次都展开最左侧的非终结符，那么这个派生就称为一个最左派生（left-most derivation）；
- 一个派生中，如果每次都展开最右侧的非终结符，那么这个派生就称为一个最右派生（right-most derivation）；
- 同一棵解析树能够唯一确定一种最左派生，同一棵解析树能够唯一确定一种最右派生；
- 如果一串标记串没有歧义，那么只有唯一的最左派生可以生成这一标记串，也只有唯一的最右派生可以生成这一标记串；
- 规约是派生反向过程：

E = ID + (ID + ID) * ID

reduction规约

```
   ID + ID + ID
-> G + ID + ID
-> F + ID + ID
-> E + ID + ID
-> E + G + ID
-> E + F + ID
-> E + ID
-> E + G
-> E + F
-> E
```

派生的顺序不唯一。

左规约对应右派生

```
E -> E + F
  -> E + G
  -> E + ID
  -> E + F + ID
  -> E + G + ID
  -> E + ID + ID
  -> F + ID + ID
  -> G + ID + ID
  -> ID + ID + ID
```

```
E = E + F
E = G + F
E = ID + F
E = ID + F * G
E = ID + G * G
E = ID + (E) * G
E = ID + (E + F) * G
E = ID + (F + F) * G
E = ID + (G + F) * G
.......
```

请大家注意，上图中的派生是最右派生，但是他对应的规约却是每次尽可能地进行左侧规约。这是因为派生与规约的方向是相反的。

5 移入规约分析

下面将要介绍的移入规约分析，是一个计算生成最左规约（或者说最右派生）的过程。移入规约分析的主要思想是：从左向右扫描标记串，从左向右进行规约。

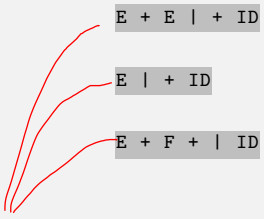
- 共有两类操作：移入、规约；
- 扫描线的右侧全部都是终结符；
- 规约操作都发生在扫描线的左侧紧贴扫描线的区域内。
- 例子

```
   | ID + ID + ID
-> ID | + ID + ID
-> G | + ID + ID
-> F | + ID + ID
-> E | + ID + ID
-> E + | ID + ID
-> E + ID | + ID
-> E + G | + ID
-> E + F | + ID
-> E | + ID
-> E + | ID
-> E + ID |
-> E + G |
-> E + F |
-> E |
```

这里进行规约

- 移入与规约的选择
  - 既能移入，又能规约的情形
  - 有多种规约方案的情形

```
   | ID + ID + ID
-> ID | + ID + ID
-> G | + ID + ID
-> F | + ID + ID
-> E | + ID + ID
-> E + | ID + ID
-> E + ID | + ID
-> E + G | + ID
-> E + F | + ID
```



三种选择

线性时间算法不允许都扫描一遍！

E + E |
E + F + | 无法完成规约！

扫描线右侧一定是Terminal Symbol（ID，可以得到额外信息）
左侧成分复杂一些。

语法分析

1 语法分析的任务

语法分析的主要任务是在词法分析结果的基础上，进一步得到解析树（parsing tree）。例如，下面两个算术表达式：

```
1 + x * y
```

```
(1 + x) * y
```

它们经过词法分析结果如下：

```
TOK_NAT(1) TOK_PLUS TOK_IDENT(x) TOK_MUL TOK_IDENT(y)
```

```
TOK_LEFT_PAREN TOK_NAT(1) TOK_PLUS TOK_IDENT(x)
TOK_RIGHT_PAREN TOK_MUL TOK_IDENT(y)
```

期望的语法分析结果如下：

```
      *                  *
     / \                / \
   ( )   y             +   y
    |                 / \
    +                1   x
   / \
  1   x
```

2 上下文无关语法与解析树

下面是一套上下文无关语法（context-free grammar，CFG）的例子：

```
S -> S ; S          E -> ID          L -> E
S -> ID := E        E -> NAT         L -> L , E
S -> PRINT ( L )    E -> E + E
                    E -> ( E )
```

下面是这套上下文无关语法的一个派生（derivation）：

```
S -> S; S
  -> ID := E; S
  -> ID := E + E; S
  -> ID := ID + E; S
  -> ID := ID + NAT; S
  -> ID := ID + NAT; PRINT(L)
  -> ID := ID + NAT; PRINT(L, E)
  -> ID := ID + NAT; PRINT(E, E)
  -> ID := ID + NAT; PRINT(ID, E)
  -> ID := ID + NAT; PRINT(ID, ID)
```

在上面例子中，`S`表示语句，`E`表示表达式，`L`表示表达式列表。在这个语言中，`PRINT`指令可以带多个参数，表达式中只允许出现加法运算，多条语句只允许顺序执行，没有条件分支与循环。

一套上下文无关语法包含以下几个组成部分：

- 一个初始符号，例如：`S`；
- 一个终结符（terminal symbols）集合，例如：`ID NAT , ; ( ) + :=`，当词法分析器和语法分析器结合使用的时候，这个终结符集合一般就是词法分析中的标记集合；
- 一个非终结符（nonterminal symbols）集合，例如：`S E L`；
- 一系列产生式（production），每个产生式的左边是一个非终结符，每个产生式的右边是一列（可以为空）终结符或非终结符。

将初始符号依据产生式不断展开最后得到一个终结符序列的过程称为派生（derivable）。

下面是这套上下文无关语法的一棵解析树（parsing tree）：

```
          E                          E
        / | \                        -> E * E
       E  *  E                       -> E + E * E
     / | \   |                       -> ID + E * E
    E  +  E  ID                      -> ID + ID * E
    |     |                          -> ID + ID * ID
   ID     ID
```

解析树具有下面性质：

- 根节点为上下文无关语法的初始符号；
- 每个叶子节点是一个终结符，每个内部节点是一个非终结符；
- 每一个父节点和他的子节点构成一条上下文无关语法中的产生式；

3 歧义与歧义的消除

下面上下文无关语法有歧义：

```
E -> ID       E -> E + E      E -> E * E      E -> ( E )
```

同一标记串，有两种解析树：

```
          E                          E
        / | \                      / | \
       E  +  E                    E  *  E
       |     / | \                / | \    |
      ID     E  *  E              E    E    ID
             |     |              |    |
            ID     ID            ID   ID
```

.

修正上下文无关语法，消除加号与乘号优先级有关的歧义

```
E -> F       E -> E + E      F -> F * F
F -> ( E )   F -> ID
```

```
          E                          E
        / | \                        -> E + E
       E  +  E                       -> E + F
       |     |                       -> E + F * F
       F     F                       -> E + F * ID
       |     / | \                   -> E + ID * ID
      ID     F  *  F                 -> F + ID * ID
             |     |                 -> ID + ID * ID
            ID     ID
```

.

4 派生与规约

最左派生与最右派生：

- 一个派生中，如果每次都展开最左侧的非终结符，那么这个派生就称为一个最左派生（left-most derivation）；
- 一个派生中，如果每次都展开最右侧的非终结符，那么这个派生就称为一个最右派生（right-most derivation）；
- 同一棵解析树能够唯一确定一种最左派生，同一棵解析树能够唯一确定一种最右派生；
- 如果一串标记串没有歧义，那么只有唯一的最左派生可以生成这一标记串，也只有唯一的最右派生可以生成这一标记串；

规约是派生反向过程：

```
   ID + ID + ID                  E -> E + F
-> G + ID + ID                     -> E + G
-> F + ID + ID                     -> E + ID
-> E + ID + ID                     -> E + F + ID
-> E + G + ID                      -> E + G + ID
-> E + F + ID                      -> E + ID + ID
-> E + ID                          -> F + ID + ID
-> E + G                           -> G + ID + ID
-> E + F                           -> ID + ID + ID
-> E
```

请大家注意，上图中的派生是最右派生，但是他对应的规约却是每次尽可能地进行左侧规约。这是因为派生与规约的方向是相反的。

5 移入规约分析

下面将要介绍的移入规约分析，是一个计算生成最左规约（或者说最右派生）的过程。移入规约分析的主要思想是：从左向右扫描标记串，从左向右进行规约。

- 共有两类操作：移入、规约；
- 扫描线的右侧全部都是终结符；
- 规约操作都发生在扫描线的左侧紧贴扫描线的区域内。

例子

```
   | ID + ID + ID
-> ID | + ID + ID
-> G | + ID + ID
-> F | + ID + ID
-> E | + ID + ID
-> E + | ID + ID
-> E + ID | + ID
-> E + G | + ID
-> E + F | + ID
-> E | + ID
-> E + | ID
-> E + ID |
-> E + G |
-> E + F |
-> E |
```

移入与规约的选择问题：

- 既能移入，又能规约的情形，应当选择移入还是规约？
- 有多种规约方案的情形，应当如何选择？

例子：

```
   | ID + ID + ID
-> ID | + ID + ID
-> G | + ID + ID          E + E | + ID
-> F | + ID + ID
-> E | + ID + ID          E | + ID
-> E + | ID + ID
-> E + ID | + ID          E + F + | ID
-> E + G | + ID
-> E + F | + ID
```

6 已移入部分的结构

已移入部分的结构

- 已移入规约的部分可以分为 n 段；
- 每段对应一条产生式；
- 第 i 段拼接上第 $i+1$ 条产生式的左侧非终结符是第 i 条产生式右侧符号串的一个前缀。

例子

- `F * (E) |` 中已移入规约的部分可以分为：`F *` `( E )`
- 分别对应产生式：`F -> F * . G` `G -> ( E ) .`
- `F * (E + | ID )` 中已移入规约的部分可以分为：`F *` `(` `E +`
- 分别对应产生式：`F -> F * . G` `G -> ( . E )` `E -> E + . F`

移入与规约对应的扫描线左侧结构变化：
原操作：移入

```
| ID + ID + ID     ->     ID | + ID + ID
```

带结构的操作：

```
(START -> . E)     ->     (START -> . E)
                          (E -> . E + F)
                          (E -> . E + F)
                          (E -> . F)
                          (F -> . G)
                          (G -> ID .)
                                                    .
```

原操作：规约

```
ID | + ID + ID    ->    G | + ID + ID
```

带结构的操作：

```
(START -> . E)     ->     (START -> . E)
(E -> . E + F)            (E -> . E + F)
(E -> . E + F)            (E -> . E + F)
(E -> . F)                (E -> . F)
(F -> . G)                (F -> G .)
(G -> ID .)
                                                    .
```

原操作：规约

```
G | + ID + ID      ->     F | + ID + ID
```

带结构的操作：

```
(START -> . E)     ->     (START -> . E)
(E -> . E + F)            (E -> . E + F)
(E -> . E + F)            (E -> . E + F)
(E -> . F)                (E -> F .)
(F -> G .)


                                                    .
```

原操作：规约

```
F | + ID + ID      ->     E | + ID + ID
```

增加结构：

```
(START -> . E)      ->      (START -> . E)
(E -> . E + F)              (E -> . E + F)
(E -> . E + F)              (E -> E . + F)
(E -> F .)
```

.

原操作：移入

```
E | + ID + ID       ->      E + | ID + ID
```

增加结构：

```
(START -> . E)      ->      (START -> . E)
(E -> . E + F)              (E -> . E + F)
(E -> E . + F)              (E -> E + . F)
```

.

原操作：移入

```
E + | ID + ID       ->      E + ID | + ID
```

增加结构：

```
(START -> . E)      ->      (START -> . E)
(E -> . E + F)              (E -> . E + F)
(E -> E + . F)              (E -> E + . F)
                            (F -> .G)
                            (G -> ID .)
```

.

7 已移入部分的结构判定

以扫描线左侧的最右段为状态，构建 NFA。新增一段带来的变化对应一条 ϵ 边，其他加入符号带来的变化对应一条普通边。该 NFA 中的所有状态都是终止状态，要判断一串符号串是否是可行的扫描线左侧结构，只需判断这个符号串能否被这个 NFA 接受。

ϵ: `START -> . E` 变为 `E -> . F`

ϵ: `START -> . E` 变为 `E -> . E + F`

E: `START -> . E` 变为 `START -> E .`

ϵ: `E -> . F` 变为 `F -> . F * G`

ϵ: `E -> . F` 变为 `F -> . G`

F: `E -> . F` 变为 `E -> F .`

ϵ: `E -> . E + F` 变为 `E -> . F`

ϵ: `E -> . E + F` 变为 `E -> . E + F`

E: `E -> . E + F` 变为 `E -> E . + F`

+: `E -> E . + F` 变为 `E -> E + . F`

ϵ: `E -> E + . F` 变为 `F -> . F * G`

ϵ: `E -> E + . F` 变为 `F -> . G`

...

判断`E + F + | ...`是否是可行的扫描线左侧结构：

```
E + F + | ...

START -> . E
    E -> . F
    E -> . E + F
    F -> . G
    F -> . F * G
    G -> . ( E )
    G -> . ID                                   .
```

```
E + F + | ...

START -> E .
    E -> E . + F
                                                .
```

```
E + F + | ...

    E -> E + . F
    F -> . G
    F -> . F * G
    G -> . ID
    G -> . ( E )
                                                .
```

```
E + F + | ...

    E -> E + F .
    F -> F . * G
                                                .
```

```
E + F + | ...

    ( No possible state )
                                                .
```

8 基于 follow 集合的判定法

定义

- `X` 是 Follow(`Y`) 的元素当且仅当 `... Y X ...` 是可能被完全规约的。
- `X` 是 First(`Y`) 的元素当且仅当 `X ...` 是可能被规约为 `Y` 的。
- 上述计算只考虑 `X` 是终结符的情形。

First 集合的计算

- 对任意终结符 `X`，`X` 都是 First(`X`) 的元素。
- 对任意产生式 `Y -> Z ...`，First(`Z`) 的元素都是 First(`Y`) 的元素。

Follow 集合的计算

- 对任意产生式 `U -> ... Y Z ...`，First(`Z`) 是 Follow(`Y`) 的子集。
- 对任意产生式 `Z -> ... Y`，Follow(`Z`) 是 Follow(`Y`) 的子集。

9 移入规约分析完整过程实例

`ID * (ID + ID)` 的移入规约分析：

- 初始：`| ID * ( ID + ID )`
- 移入：`ID | * ( ID + ID )`
- 规约：`G | * ( ID + ID )`，因为 `ID * | ...` 不可行
- 规约：`F | * ( ID + ID )`，因为 `G * | ...` 不可行
- 移入：`F * | ( ID + ID )`，因为 `*` 不是 Follow(`E`) 中的元素
- 移入：`F * ( | ID + ID )`
- 移入：`F * ( ID | + ID )`
- 规约：`F * ( G | + ID )`，因为 `F * ( ID + | ...` 不可行
- 规约：`F * ( F | + ID )`，因为 `F * ( G + | ...` 不可行
- 规约：`F * ( E | + ID )`，因为 `F * ( F + | ...` 不可行
- 移入：`F * ( E + | ID )`
- 移入：`F * ( E + ID | )`
- 规约：`F * ( E + G | )`，因为 `F * ( E + ID ) | ...` 不可行
- 规约：`F * ( E + F | )`，因为 `F * ( E + G ) | ...` 不可行
- 规约：`F * ( E | )`，因为另外两种方案扫描线左侧的结构都不可行
- 移入：`F * ( E ) |`
- 规约：`F * G |`

- 规约：`F |`
- 规约：`E |`，亦可看做`E | EOF`
- 移入：`E EOF |`
- 规约：`START |`
- 语法分析结束

Coq 中的程序语法树

1 一个极简的指令式程序语言

以下考虑一种极简的程序语言。它的程序表达式分为整数类型表达式与布尔类型表达式，其中整数类型表达式只包含加减乘运算与变量、常数。布尔表达式中只包含整数类型表达式之间的大小比较或者这些比较结果之间的布尔运算。而它的程序语句也只有对变量赋值、顺序执行、if 语句与 while 语句。

整数类型表达式

```
EI :: = N | V | EI + EI | EI - EI | EI * EI
```

布尔类型表达式

```
EB :: = TRUE | FALSE | EI < EI | EB && EB | ! EB
```

语句

```
C :: = SKIP |
       V = EI |
       C; C |
       if (EB) then { C } else { C } |
       while (EB) do { C }
```

下面依次在 Coq 中定义该语言变量名、表达式与语句。

```
Definition var_name: Type := string.
```

```
Inductive expr_int : Type :=
  | EConst (n: Z): expr_int
  | EVar (x: var_name): expr_int
  | EAdd (e1 e2: expr_int): expr_int
  | ESub (e1 e2: expr_int): expr_int
  | EMul (e1 e2: expr_int): expr_int.
```

在 Coq 中，可以利用`Notation`使得这些表达式更加易读。`Notation`的具体定义详见 Coq 源代码。使用`Notation`的效果如下：

```
Check [[1 + "x"]].
Check [["x" * ("a" + "b" + 1)]].
```

```
Inductive expr_bool: Type :=
  | ETrue: expr_bool
  | EFalse: expr_bool
  | ELt (e1 e2: expr_int): expr_bool
  | EAnd (e1 e2: expr_bool): expr_bool
  | ENot (e: expr_bool): expr_bool.
```

```
Inductive com : Type :=
  | CSkip: com
  | CAsgn (x: var_name) (e: expr_int): com
  | CSeq (c1 c2: com): com
  | CIf (e: expr_bool) (c1 c2: com): com
  | CWhile (e: expr_bool) (c: com): com.
```

2 While 语言

在许多以 C 语言为代表的常用程序语言中，往往不区分整数类型表达式与布尔类型表达式，同时表达式中也包含更多运算符。例如，我们可以如下规定一种程序语言的语法。

表达式

```
E :: = N | V | E+E | E-E | E*E | E/E | E%E |
       E<E | E<=E | E==E | E!=E | E>=E | E>E |
       E&&E | E||E | !E
```

语句

```
C :: = SKIP |
       V = E |
       C; C |
       if (E) then { C } else { C } |
       while (E) do { C }
```

下面依次在 Coq 中定义该语言的变量名、表达式与语句。

```
Definition var_name: Type := string.
```

再定义二元运算符和一元运算符。

```
Inductive binop : Type :=
  | OOr | OAnd
  | OLt | OLe | OGt | OGe | OEq | ONe
  | OPlus | OMinus | OMul | ODiv | OMod.

Inductive unop : Type :=
  | ONot | ONeg.
```

下面是表达式的抽象语法树。

```
Inductive expr : Type :=
  | EConst (n: Z): expr
  | EVar (x: var_name): expr
  | EBinop (op: binop) (e1 e2: expr): expr
  | EUnop (op: unop) (e: expr): expr.
```

最后程序语句的定义是类似的。

```
Inductive com : Type :=
  | CSkip: com
  | CAsgn (x: var_name) (e: expr): com
  | CSeq (c1 c2: com): com
  | CIf (e: expr) (c1 c2: com): com
  | CWhile (e: expr) (c: com): com.
```

3 更多的程序语言：WhileDeref

下面在 While 程序语言中增加取地址上的值`EDeref`操作。

```
Inductive expr : Type :=
  | EConst (n: Z): expr
  | EVar (x: var_name): expr
  | EBinop (op: binop) (e1 e2: expr): expr
  | EUnop (op: unop) (e: expr): expr
  | EDeref (e: expr): expr.
```

相应的，赋值语句也可以分为两种情况。

```
Inductive com : Type :=
  | CSkip: com
  | CAsgnVar (x: var_name) (e: expr): com
  | CAsgnDeref (e1 e2: expr): com
  | CSeq (c1 c2: com): com
  | CIf (e: expr) (c1 c2: com): com
  | CWhile (e: expr) (c: com): com.
```

4 更多的程序语言：WhileD

在大多数程序语言中，会同时支持或不支持取地址`EAddrOf`与取地址上的值 `EDeref`两类操作，我们也可以在 WhileDeref 语言中再加入取地址操作。

```
Inductive expr : Type :=
  | EConst (n: Z): expr
  | EVar (x: var_name): expr
  | EBinop (op: binop) (e1 e2: expr): expr
  | EUnop (op: unop) (e: expr): expr
  | EDeref (e: expr): expr
  | EAddrOf (e: expr): expr.
```

程序语句的语法树不变。

```
Inductive com : Type :=
  | CSkip: com
  | CAsgnVar (x: var_name) (e: expr): com
  | CAsgnDeref (e1 e2: expr): com
  | CSeq (c1 c2: com): com
  | CIf (e: expr) (c1 c2: com): com
  | CWhile (e: expr) (c: com): com.
```

5 更多的程序语言：WhileDC

下面在程序语句中增加控制流语句 continue 与 break，并增加多种循环语句。

```
Inductive expr : Type :=
  | EConst (n: Z): expr
  | EVar (x: var_name): expr
  | EBinop (op: binop) (e1 e2: expr): expr
  | EUnop (op: unop) (e: expr): expr
  | EDeref (e: expr): expr
  | EAddrOf (e: expr): expr.
```

```
Inductive com : Type :=
  | CSkip: com
  | CAsgnVar (x: var_name) (e: expr): com
  | CAsgnDeref (e1 e2: expr): com
  | CSeq (c1 c2: com): com
  | CIf (e: expr) (c1 c2: com): com
  | CWhile (e: expr) (c: com): com
  | CFor (c1: com) (e: expr) (c2: com) (c3: com): com
  | CDoWhile (c: com) (e: expr): com
  | CContinue: com
  | CBreak: com.
```

6 更多的程序语言：WhileDCL

下面在程序语句中增加局部变量声明。

```
C :: = skip | continue | break |
       V = E | * E = E |
       C; C |
       if (E) then { C } else { C } |
       while (E) do { C } |
       for (C; E; E) { C } |
       do { C } while (E)
       local X in { C }
```

```
Inductive com : Type :=
  | CSkip: com
  | CLocalVar (x: var_name) (c1: com): com
  | CAsgnVar (x: var_name) (e: expr): com
  | CAsgnDeref (e1 e2: expr): com
  | CSeq (c1 c2: com): com
  | CIf (e: expr) (c1 c2: com): com
  | CWhile (e: expr) (c: com): com
  | CFor (c1: com) (e: expr) (c2: com) (c3: com): com
  | CDoWhile (c: com) (e: expr): com
  | CContinue: com
  | CBreak: com.
```

7 更多的程序语言：WhileF

下面在程序表达式中增加函数调用，在程序语句中增加过程调用。

```
E ::= N | V | E+E | E-E | E*E | E/E | E%E |
      E<E | E<=E | E==E | E!=E | E>=E | E>E |
      E&&E | E||E | !E |
      * E | & E | F(E, E, ..., E)
```

```
C ::= skip | continue | break | return
      V = E | * E = E |
      P(E, E, ..., E) |
      C; C |
      if (E) then { C } else { C } |
      while (E) do { C } |
      for (C; E; E) { C } |
      do { C } while (E)
      local X in { C }
```

除了在程序表达式和程序语句中增加新的语法结构之外，程序中还要新增一类新对象：函数与过程。

- 函数 f
  - 函数名：`f.(name_of_func)`
  - 函数参数变量列表：`f.(args_of_func)`
  - 函数体：`f.(body_of_func)`
- 过程 p
  - 过程名：`p.(name_of_proc)`
  - 过程参数变量列表：`p.(args_of_proc)`
  - 过程体：`p.(body_of_proc)`

```
Definition func_name: Type := string.
Definition proc_name: Type := string.
```

下面是新的表达式定义，这是一个嵌套递归定义。

```
Inductive expr : Type :=
  | EConst (n: Z): expr
  | EVar (x: var_name): expr
  | EBinop (op: binop) (e1 e2: expr): expr
  | EUnop (op: unop) (e: expr): expr
  | EDeref (e: expr): expr
  | EAddrOf (e: expr): expr
  | EFuncCall (f: func_name) (es: list expr).
```

下面是新的程序语句定义，这也是一个嵌套递归定义。这里约定`return_var`是一个特定的表示返回值的变量，而返回指令本身没有参数。

```
Definition return_var: var_name := "__return".
```

```
Inductive com : Type :=
  | CSkip: com
  | CLocalVar (x: var_name) (c1: com): com
  | CAsgnVar (x: var_name) (e: expr): com
  | CAsgnDeref (e1 e2: expr): com
  | CProcCall (p: proc_name) (es: list expr): com
  | CSeq (c1 c2: com): com
  | CIf (e: expr) (c1 c2: com): com
  | CWhile (e: expr) (c: com): com
  | CFor (c1: com) (e: expr) (c2: com) (c3: com): com
  | CDoWhile (c: com) (e: expr): com
  | CContinue: com
  | CBreak: com
  | CReturn: com.
```

下面定义程序中的函数与过程。

```
Record func: Type := {
  name_of_func: func_name;
  body_of_func: com;
  args_of_func: list var_name;
}.
```

```
Record proc: Type := {
  name_of_proc: proc_name;
  body_of_proc: com;
  args_of_proc: list var_name;
}.
```

最后，一段完整的程序由全局变量列表、函数列表、过程列表与入口函数组成。

```
Record prog: Type := {
  gvars: list var_name;
  funcs: list func;
  procs: list proc;
  entry: func_name
}.
```

8 简单语法变换与证明

习题 1. 下面的递归函数`remove_skip`定义了在程序语句中删除多余空语句的操作。

```
Fixpoint remove_skip (c: com): com :=
  match c with
  | CSeq c1 c2 =>
      match remove_skip c1, remove_skip c2 with
      | CSkip, _ => remove_skip c2
      | _, CSkip => remove_skip c1
      | _, _ => CSeq (remove_skip c1) (remove_skip c2)
      end
  | CIf e c1 c2 =>
      CIf e (remove_skip c1) (remove_skip c2)
  | CWhile e c1 =>
      CWhile e (remove_skip c1)
  | _ =>
      c
  end.
```

下面请证明：`remove_skip`之后，确实不再有多余的空语句了。所谓没有空语句，是指不出现`c; SKIP`或`SKIP; c`这样的语句。首先定义：局部不存在多余的空语句。

```
Definition not_sequencing_skip (c: com): Prop :=
  match c with
  | CSeq CSkip _ => False
  | CSeq _ CSkip => False
  | _ => True
  end.
```

其次定义语法树的所有子树中都不存在多余的空语句。

```
Fixpoint no_sequenced_skip (c: com): Prop :=
  match c with
  | CSeq c1 c2 =>
      not_sequencing_skip c /\
      no_sequenced_skip c1 /\ no_sequenced_skip c2
  | CIf e c1 c2 =>
      no_sequenced_skip c1 /\ no_sequenced_skip c2
  | CWhile e c1 =>
      no_sequenced_skip c1
  | _ =>
      True
  end.
```

下面是需要证明的结论。

```
Theorem remove_skip_no_sequenced_skip: forall c,
  no_sequenced_skip (remove_skip c).
(* 请在此处填入你的证明，以_[Qed]_结束。 *)
```

课前阅读：Coq 中的归纳类型

1 关于等式的证明

在先前证明单调函数性质的过程中，我们经常会要通过参数之间的大小关系推导出函数值之间的大小关系。例如，当 f 是一个单调函数时，可以由 $x \leqslant y$ 推出 $f(x) \leqslant f(y)$。对于一般的函数，我们也可以由参数相等，推出函数值相等，即由 $x = y$ 推出 $f(x) = f(y)$。下面的证明中就要用到这一性质。

```
Definition is_fixpoint (f: Z -> Z) (x: Z): Prop :=
  f x = x.
```

```
Theorem fixpoint_self_comp: forall f x,
  is_fixpoint f x ->
  is_fixpoint (Zcomp f f) x.
Proof.
  unfold is_fixpoint, Zcomp.
  intros.
  rewrite H.
  rewrite H.
  reflexivity.
Qed.
```

在数学上，如果`f x = x`，那么我们就称`x`是函数`f`的一个不动点。上面的定理证明了，如果`x`是`f`的不动点，那么`x`也是`f`与自身复合结果的不动点。在这一证明中，前提`H`是命题`f x = x`。证明指令`rewrite H`的效果是将结论中的`f x`替换为`x`，因此，第一次使用该指令后，原先需要证明的`f (f x) = x`规约为了`f x = x`。这一步背后的证明实际就用到了“只要函数`f`的参数不变，那么函数值也不变”这条性质。

```
```

Coq 证明脚本 1. 利用等式做证明的 rewrite 指令. 如果`H`是具有形式`a = b`定理或证明前提，`rewrite H`可以将待证明结论中的`a`替换成`b`，`rewrite H in H0`也可以将前提`H0`中的`a`替换成`b`。有时，这个`a`可能出现了多次，但是我们只希望将其中的若干个`a`而不是全部的`a`都替换成`b`，此时可以在 rewrite 指令中增加`at`，即使用`rewrite H at ...`或`rewrite H at ... in ...`指明需要进行替换的位置。例如，当前提`H`为`x = f x`，待证明结论为`x = f (f x)`时，

`rewrite H` 与 `rewrite H at 1, 2`

都会将结论变为`f x = f (f (f x))`；

`rewrite H at 1`

会将结论变为`f x = f (f x)`；而

`rewrite H at 2`

会将结论变为`x = f (f (f x))`。

Coq 允许 rewrite 使用的定理或前提中有`forall`概称量词，用户可以手动地填入这些`forall`约束的变量或由 Coq 自动填入这些变量。例如，当前提`H1`与待证明结论分别为：

```
forall x y: Z, g x y = g y x
g 3 5 = 6
```

时，`rewrite H1`、`rewrite (H1 3)`或`rewrite (H1 3 5)`都会将待证明结论变换为`g 5 3 = 6`。Coq 还允许 rewrite 使用的定理或前提中带有附加条件，例如当前提 `H2`与待证明结论分别为：

```
forall x: Z, x <= 0 -> h x = 0
h (h (-5)) = 0
```

时，Coq 中的`rewrite H2`指令会将原证明目标规约为两个新的证明目标：第一个证明目标中结论变为`h 0 = 0`，第二个证明目标为需要补充证明的附加条件`-5 <= 0`。如果产生的附加条件可以用一条证明脚本完成证明，那么可以在 rewrite 指令中加入`by`。上面例子中，`rewrite H2 by lia`可以直接将结论变为`h 0 = 0`并不再额外产生附加的证明条件。

最后，Coq 中不仅允许将等式左侧的内容替换为等式右侧的内容，也允许使用`<-`进行反向操作。例如，当`H`具有形式`a = b`时，`rewrite <- H`会将结论中的`b`替换为`a`。同时，Coq 也允许在一条 rewrite 指令中，按指定顺序连续进行多次替换，例如`rewrite H1, H2`表示先`rewrite H1`再`rewrite H2`。

下面关于不动点简单性质的证明需要我们灵活使用 rewrite 指令。

```
Example fixpoint_self_comp23: forall f x,
  is_fixpoint (Zcomp f f) x ->
  is_fixpoint (Zcomp f (Zcomp f f)) x ->
  is_fixpoint f x.
Proof.
  unfold is_fixpoint, Zcomp.
  intros.
  rewrite H in H0.
  rewrite H0.
  reflexivity.
Qed.
```

2 用 Coq 归纳类型定义二叉树

```
Inductive tree: Type :=
| Leaf: tree
| Node (l: tree) (v: Z) (r: tree): tree.
```

这个定义说的是，一棵二叉树要么是一棵空树`Leaf`，要么有一棵左子树、有一棵右子树外加有一个根节点整数标号。我们可以在 Coq 中写出一些具体的二叉树的例子。

```
Definition tree_example0: tree :=
  Node Leaf 1 Leaf.
```

```
Definition tree_example1: tree :=
  Node (Node Leaf 0 Leaf) 2 Leaf.
```

```
Definition tree_example2a: tree :=
  Node (Node Leaf 8 Leaf) 100 (Node Leaf 9 Leaf).
```

```
Definition tree_example2b: tree :=
  Node (Node Leaf 9 Leaf) 100 (Node Leaf 8 Leaf).
```

```
Definition tree_example3a: tree :=
  Node (Node Leaf 3 Leaf) 5 tree_example2a.
```

```
Definition tree_example3b: tree :=
  Node tree_example2b 5 (Node Leaf 3 Leaf).
```

它们分别表示下面这些树结构

```
          2       100         100
    1    /       /   \       /   \
        0       8     9     9     8
```

```
        5                   5
       / \                 / \
      3  100             100  3
        / \               / \
       8   9             9   8
```

Coq 中，我们往往可以使用递归函数定义归纳类型元素的性质。Coq 中定义递归函数时使用的关键字是`Fixpoint`。下面两个定义通过递归定义了二叉树的高度和节点个数。

```
Fixpoint tree_height (t: tree): Z :=
  match t with
  | Leaf => 0
  | Node l v r => Z.max (tree_height l) (tree_height r) + 1
  end.
```

```
Fixpoint tree_size (t: tree): Z :=
  match t with
  | Leaf => 0
  | Node l v r => tree_size l + tree_size r + 1
  end.
```

Coq 系统“知道”每一棵特定树的高度和节点个数是多少。下面是一些 Coq 代码的例子。

```
Example Leaf_height:
  tree_height Leaf = 0.
Proof. reflexivity. Qed.
```

```
Example tree_example2a_height:
  tree_height tree_example2a = 2.
Proof. reflexivity. Qed.
```

```
Example treeexample3b_size:
  tree_size tree_example3b = 5.
Proof. reflexivity. Qed.
```

Coq 中也可以定义树到树的函数。下面的`tree_reverse`函数把二叉树进行了左右翻转。

```
Fixpoint tree_reverse (t: tree): tree :=
  match t with
  | Leaf => Leaf
  | Node l v r => Node (tree_reverse r) v (tree_reverse l)
  end.
```

下面是三个二叉树左右翻转的例子：

```
Example Leaf_tree_reverse:
  tree_reverse Leaf = Leaf.
Proof. reflexivity. Qed.
```

```
Example tree_example0_tree_reverse:
  tree_reverse tree_example0 = tree_example0.
Proof. reflexivity. Qed.
```

```
Example tree_example3_tree_reverse:
  tree_reverse tree_example3a = tree_example3b.
Proof. reflexivity. Qed.
```

归纳类型有几条基本性质。(1) 归纳定义规定了一种分类方法，以`tree`类型为例，一棵二叉树`t`要么是`Leaf`，要么具有形式`Node l v r`；(2) 以上的分类之间是互斥的，即无论`l`、`v`与`r`取什么值，`Leaf`与`Node l v r`都不会相等；(3) `Node`这样的构造子是函数也是单射。这三条性质对应了 Coq 中的三条证明指令：`destruct`、`discriminate`与`injection`。利用它们就可以证明几条最简单的性质：

```
Lemma Node_inj_left: forall l1 v1 r1 l2 v2 r2,
  Node l1 v1 r1 = Node l2 v2 r2 ->
  l1 = l2.
Proof.
  intros.
  injection H as H_l H_v H_r.
```

上面的`injection`指令使用了`Node`是单射这一性质。

```
  rewrite H_l.
  reflexivity.
Qed.
```

有时，手动为`injection`生成的命题进行命名显得很啰嗦，Coq 允许用户使用问号占位，从而让 Coq 进行自动命名。

```
Lemma Node_inj_right: forall l1 v1 r1 l2 v2 r2,
  Node l1 v1 r1 = Node l2 v2 r2 ->
  r1 = r2.
Proof.
  intros.
  injection H as ? ? ?.
```

这里，Coq 自动命名的结果是使用了`H`、`H0`与`H1`。下面也使用`apply`指令取代`rewrite`简化后续证明。

```
  apply H1.
Qed.
```

如果不需要用到`injection`生成的左右命题，可以将不需要用到的部分用下划线占位。

```
Lemma Node_inj_value: forall l1 v1 r1 l2 v2 r2,
  Node l1 v1 r1 = Node l2 v2 r2 ->
  v1 = v2.
Proof.
  intros.
  injection H as _ ? _.
  apply H.
Qed.
```

下面引理说：若`Leaf`与`Node l v r`相等，那么`1 = 2`。换言之，`Leaf`与`Node l v r`始终不相等，否则就形成了一个矛盾的前提。

```
Lemma Leaf_Node_conflict: forall l v r,
  Leaf = Node l v r -> 1 = 2.
Proof.
  intros.
  discriminate.
Qed.
```

下面这个简单性质与`tree_reverse`有关。

```
Lemma reverse_result_Leaf: forall t,
  tree_reverse t = Leaf ->
  t = Leaf.
Proof.
  intros.
```

下面的`destruct`指令根据`t`是否为空树进行分类讨论。

```
  destruct t.
```

执行这一条指令之后，Coq 中待证明的证明目标由一条变成了两条，对应两种情况。为了增加 Coq 证明的可读性，我们推荐大家使用 bullet 记号把各个子证明过程分割开来，就像一个一个抽屉或者一个一个文件夹一样。Coq 中可以使用的 bullet 标记有：`+ - * ++ -- **`等等

```
  + reflexivity.
```

第一种情况是`t`是空树的情况。这时，待证明的结论是显然的。

```
  + discriminate H.
```

第二种情况下，其实前提`H`就可以推出矛盾。可以看出，`discriminate`指令也会先根据定义化简，再试图推出矛盾。

```
Qed.
```

3 结构归纳法

我们接下去将证明一些关于`tree_height`，`tree_size`与`tree_reverse`的基本性质。我们在证明中将会使用的主要方法是归纳法。

相信大家都很熟悉自然数集上的数学归纳法。数学归纳法说的是：如果我们要证明某性质`P`对于任意自然数`n`都成立，那么我可以将证明分为如下两步：

- 奠基步骤：证明`P 0`成立；
- 归纳步骤：证明对于任意自然数`n`，如果`P n`成立，那么`P (n + 1)`也成立。

对二叉树的归纳证明与上面的数学归纳法稍有不同。具体而言，如果我们要证明某性质`P`对于一切二叉树`t`都成立，那么我们只需要证明以下两个结论：

- 奠基步骤：证明`P Leaf`成立；
- 归纳步骤:证明对于任意二叉树`l` `r`以及任意整数标签`n`,如果`P l`与`P r`都成立,那么`P (Node l n r)`也成立。

这样的证明方法就成为结构归纳法。在 Coq 中，`induction`指令表示：使用结构归纳法。下面是几个证明的例子。

第一个例子是证明`tree_size`与`tree_reverse`之间的关系。

```
Lemma reverse_size: forall t,
  tree_size (tree_reverse t) = tree_size t.
Proof.
  intros.
  induction t.
```

上面这个指令说的是：对`t`结构归纳。Coq 会自动将原命题规约为两个证明目标，即奠基步骤和归纳步骤。

```
  + simpl.
```

第一个分支是奠基步骤。这个`simpl`指令表示将结论中用到的递归函数根据定义化简。

```
    reflexivity.
  + simpl.
```

第二个分支是归纳步骤。我们看到证明目标中有两个前提`IHt1`以及`IHt2`。在英文中`IH`表示 induction hypothesis 的缩写，也就是归纳假设。在这个证明中`IHt1`与`IHt2`分别是左子树`t1`与右子树`t2`的归纳假设。

```
    rewrite IHt1.
    rewrite IHt2.
    lia.
Qed.
```

第二个例子很类似，是证明`tree_height`与`tree_reverse`之间的关系。

```
Lemma reverse_height: forall t,
  tree_height (tree_reverse t) = tree_height t.
Proof.
  intros.
  induction t.
  + simpl.
    reflexivity.
  + simpl.
    rewrite IHt1.
    rewrite IHt2.
    lia.
```

注意：`lia`指令也是能够处理`Z.max`与`Z.min`的。

```
Qed.
```

下面我们将通过重写上面这一段证明，介绍 Coq 证明语言的一些其他功能。

```
Lemma reverse_height_attempt2: forall t,
  tree_height (tree_reverse t) = tree_height t.
Proof.
  intros.
  induction t; simpl.
```

在 Coq 证明语言中可以用分号将小的证明指令连接起来形成大的证明指令，其中`tac1 ; tac2`这个证明指令表示先执行指令`tac1`，再对于`tac1`生成的每一个证明目标执行`tac2`。分号是右结合的。

```
  + reflexivity.
  + simpl.
    lia.
Qed.
```

习题 1.

```
Lemma reverse_involutive: forall t,
  tree_reverse (tree_reverse t) = t.
(* 请在此处填入你的证明，以_[Qed]_结束。 *)
```

4 加强归纳

下面证明`tree_reverse`是一个单射。

```
Lemma tree_reverse_inj: forall t1 t2,
  tree_reverse t1 = tree_reverse t2 ->
  t1 = t2.
Proof.
```

这个引理的 Coq 证明需要我们特别关注：真正需要归纳证明的结论是什么？如果选择对`t1`做结构归纳，那么究竟是归纳证明对于任意`t2`上述性质成立，还是归纳证明对于某“特定”的`t2`上述性质成立？如果我们按照之前的 Coq 证明习惯，用`intros`与`induction t1`两条指令开始证明，那就表示用归纳法证明一条关于“特定”`t2`的性质。

```
  intros.
  induction t1.
  + destruct t2.
```

奠基步骤的证明可以通过对`t2`的分类讨论完成。

```
    - reflexivity.
```

如果`t2`是空树，那么结论是显然的。

```
    - simpl in H.
      discriminate H.
```

如果`t2`是非空树，那么前提`H`就能导出矛盾。如上面指令展示的那样，`simpl`指令也可以对前提中的递归定义化简。当然，在这个证明中，由于之后的`discriminate`指令也会完成自动化简，这条`simpl`指令其实是可以省略的。

```
Abort.
```

进入归纳步骤的证明时，不难发现，证明已经无法继续进行。因为需要使用的归纳假设并非关于原`t2`值的性质。正确的证明方法是用归纳法证明一条对于一切`t2`的结论。

```
Lemma tree_reverse_inj: forall t1 t2,
  tree_reverse t1 = tree_reverse t2 ->
  t1 = t2.
Proof.
  intros t1.
```

上面这条`intros 1t`指令可以精确地将`t1`放到证明目标的前提中，同时却将`t2`保留在待证明目标的结论中。

```
  induction t1; simpl; intros.
  + destruct t2.
    - reflexivity.
    - discriminate H.
  + destruct t2.
    - discriminate H.
    - injection H as ? ? ?.
      rewrite (IHt1_1 _ H1).
      rewrite (IHt1_2 _ H).
      rewrite H0.
      reflexivity.
Qed.
```

当然，上面这条引理其实可以不用归纳法证明。下面的证明中使用了前面证明的结论：`reverse_involutive`。

```
Lemma tree_reverse_inj_again: forall t1 t2,
  tree_reverse t1 = tree_reverse t2 ->
  t1 = t2.
Proof.
  intros.
  rewrite <- (reverse_involutive t1), <- (reverse_involutive t2).
  rewrite H.
  reflexivity.
Qed.
```

语法分析（结尾部分）

1 冲突

移入/规约冲突

- 如果在同一时刻既能进行移入操作，又能进行规约操作，这就称为一个移入/规约冲突（shift/reduce conflict）。

规约/规约冲突

- 如果在同一时刻能进行两种不同的规约操作，这就称为一个规约/规约冲突（reduce/reduce conflict）。

歧义与冲突的关系

- 给定一套上下文无关语法，如果移入规约分析中，一定不会出现移入/规约冲突，也不会出现规约/规约冲突，那么任何一串标记串都不会有歧义；
- 反之不一定。

```
A -> B X Y       A -> C X Z
B -> U V W       C -> U V W
```

2 Bison 语法分析器

- 输入（.y 文件）：基于上下文无关语法与优先级、结合性的语法分析规则
- 输出（.c 文件）：用 C 语言实现的基于移入规约分析算法的语法分析器
- C 程序中不构造解析树
- 解析树的构造只与语法分析的过程相对应
- 语法分析中直接构造抽象语法树

定义 While 语言的抽象语法树
lang.h

```
enum BinOpType {
  T_PLUS, T_MINUS, T_MUL, T_DIV, T_MOD,
  T_LT, T_GT, T_LE, T_GE, T_EQ, T_NE,
  T_AND, T_OR
};

enum UnOpType {
  T_UMINUS, T_NOT
};

enum ExprType {
  T_CONST, T_VAR,
  T_BINOP, T_UNOP,
  T_DEREF,
  T_MALLOC, T_RI, T_RC
};
```

```
struct expr {
  enum ExprType t;
  union {
    struct {unsigned int value; } CONST;
    struct {char * name; } VAR;
    struct {enum BinOpType op;
            struct expr * left;
            struct expr * right; } BINOP;
    struct {enum UnOpType op; struct expr * arg; } UNOP;
    struct {struct expr * arg; } DEREF;
    struct {struct expr * arg; } MALLOC;
    struct {void * none; } RI;
    struct {void * none; } RC;
  } d;
};
```

```
enum CmdType {
  T_DECL, T_ASGN, T_SEQ, T_IF, T_WHILE, T_WI, T_WC
};

struct cmd {
  enum CmdType t;
  union {
    struct {char * name; } DECL;
    struct {struct expr * left; struct expr * right; } ASGN;
    struct {struct cmd * left; struct cmd * right; } SEQ;
    struct {struct expr * cond;
            struct cmd * left; struct cmd * right; } IF;
    struct {struct expr * cond; struct cmd * body; } WHILE;
    struct {struct expr * arg; } WI;
    struct {struct expr * arg; } WC;
  } d;
};
```

构造抽象语法树的辅助函数 lang.h

```
struct expr * TConst(unsigned int value);
struct expr * TVar(char * name);
struct expr * TBinOp(enum BinOpType op,
                     struct expr * left,
                     struct expr * right);
struct expr * TUnOp(enum UnOpType op, struct expr * arg);
struct expr * TDeref(struct expr * arg);
struct expr * TMalloc(struct expr * arg);
struct expr * TReadInt();
struct expr * TReadChar();
```

```
struct cmd * TDecl(char * name);
struct cmd * TAsgn(struct expr * left, struct expr * right);
struct cmd * TSeq(struct cmd * left, struct cmd * right);
struct cmd * TIf(struct expr * cond,
                 struct cmd * left, struct cmd * right);
struct cmd * TWhile(struct expr * cond, struct cmd * body);
struct cmd * TWriteInt(struct expr * arg);
struct cmd * TWriteChar(struct expr * arg);
```

输出调试函数 lang.h

```
void print_binop(enum BinOpType op);
void print_unop(enum UnOpType op);
void print_expr(struct expr * e);
void print_cmd(struct cmd * c);
```

文件头 lang.y

```
%{
  #include <stdio.h>
  #include "lang.h"
  #include "lexer.h"
  int yyerror(char * str);
  int yylex();
  struct cmd * root;
%}
```

- 描述语法分析器所需的头文件、全局变量以及函数；
- 其中`yylex()`是 Flex 词法分析器提供的函数；
- `yyerror(str)`是处理语法错误的必要设定；
- `root`是语法分析最后生成的语法树根节点。

语法分析结果在 C 中的存储 lang.y

```
%union {
  unsigned int n;
  char * i;
  struct expr * e;
  struct cmd * c;
  void * none;
}
```

- 描述多种语法结构产生对应的语法分析结果；

- `n`表示自然数常数的词法语法分析结果；
- `i`表示变量的词法语法分析结果；
- `e`表示表达式的语法分析结果；
- `c`表示程序语句的语法分析结果；

终结符与非终结符 lang.y

```
%token <n> TM_NAT
%token <i> TM_IDENT
%token <none> TM_LEFT_BRACE TM_RIGHT_BRACE
%token <none> TM_LEFT_PAREN TM_RIGHT_PAREN
%token <none> TM_MALLOC TM_RI TM_RC TM_WI TM_WC
%token <none> TM_VAR TM_IF TM_THEN TM_ELSE TM_WHILE TM_DO
%token <none> TM_SEMICOL TM_ASGNOP TM_OR TM_AND TM_NOT
%token <none> TM_LT TM_LE TM_GT TM_GE TM_EQ TM_NE
%token <none> TM_PLUS TM_MINUS TM_MUL TM_DIV TM_MOD
%type <c> NT_WHOLE NT_CMD
%type <e> NT_EXPR
```

- `%token`表示终结符，`%type`表示非终结符；
- 尖括号内表示语义值的存储方式。

优先级与结合性 lang.y

```
%nonassoc TM_ASGNOP
%left TM_OR
%left TM_AND
%left TM_LT TM_LE TM_GT TM_GE TM_EQ TM_NE
%left TM_PLUS TM_MINUS
%left TM_MUL TM_DIV TM_MOD
%left TM_NOT
%left TM_LEFT_PAREN TM_RIGHT_PAREN
%right TM_SEMICOL
```

- 越先出现优先级越低；
- 同一行声明内的优先级相同。

上下文无关语法 lang.y

```
%%

NT_WHOLE:
  NT_CMD {
    $$ = ($1);
    root = $$;
  }
;
...
```

- 所有词法分析规则都放在一组`%%`中；
- 第一条语法分析规则描述初始符号对应的产生式
- 用`$$`表示产生式左侧符号的语义值；

- 用 `$1`、`$2` 等表示产生式右侧符号的语义值；

上下文无关语法（续）

```
NT_EXPR:
  TM_NAT {
    $$ = (TConst($1));
  }
| TM_LEFT_PAREN NT_EXPR TM_RIGHT_PAREN {
    $$ = ($2);
  }
| TM_MINUS NT_EXPR {
    $$ = (TUnOp(T_UMINUS,$2));
  }
| NT_EXPR TM_PLUS NT_EXPR {
    $$ = (TBinOp(T_PLUS,$1,$3));
  }
| NT_EXPR TM_MINUS NT_EXPR {
    $$ = (TBinOp(T_MINUS,$1,$3));
  }
...
```

Flex 与 Bison 的协同使用

```
%option noyywrap yylineno
%option outfile="lexer.c" header-file="lexer.h"
%{
#include "lang.h"
#include "parser.h"
%}

%%
0|[1-9][0-9]* {
    yylval.n = build_nat(yytext, yyleng);
    return TM_NAT;
}
"var" {
    return TM_VAR;
}
...
```

main.c （只打印结果）

```
#include <stdio.h>
#include "lang.h"
#include "lexer.h"
#include "parser.h"

extern struct cmd * root;
void yyparse();

int main(int argc, char **argv) {
    yyin = stdin;
    yyparse();
    fclose(stdin);
    print_cmd(root);
}
```

编译 Makefile

```
lexer.c: lang.l
        flex lang.l
parser.c: lang.y
        bison -o parser.c -d -v lang.y
lang.o: lang.c lang.h
        gcc -c lang.c
parser.o: parser.c parser.h lexer.h lang.h
        gcc -c parser.c
lexer.o: lexer.c lexer.h parser.h lang.h
        gcc -c lexer.c
main.o: main.c lexer.h parser.h lang.h
        gcc -c main.c
main: lang.o parser.o lexer.o main.o
        gcc lang.o parser.o lexer.o main.o -o main
%.c: %.y
%.c: %.l
```

语法分析之后

- 可以基于 AST 进行进一步的合法性检查；
- 例如：判定 while 语言程序是否有变量重名，如果有则判定为非法程序；
- 例如：判定 while 语言程序是否所有使用过的变量名都有实现声明，如果没有则判定为非法程序；
- 等等

Bison 生成语法分析器的调试

- 编译时`bison -v`指令会将移入规约分析中的 DFA 信息输出到 parser.output 文件中。

指称语义

1 简单表达式的指称语义

指称语义是一种定义程序行为的方式。在极简的 SimpleWhile 语言中，整数类型表达式中只有整数常量、变量、加法、减法与乘法运算。

```
EI :: = N | V | EI + EI | EI - EI | EI * EI
```

我们约定其中整数变量的值、整数运算的结果都是没有范围限制的。基于这一约定，我们可以如下定义表达式`e`在程序状态`st`上的值。

首先定义程序状态集合：

$$\text{state} \triangleq \text{var\_name} \to \mathbb{Z}$$

```
Definition state: Type := var_name -> Z.
```

- $[\![n]\!](s) = n$
- $[\![x]\!](s) = s(x)$
- $[\![e_1 + e_2]\!](s) = [\![e_1]\!](s) + [\![e_2]\!](s)$
- $[\![e_1 - e_2]\!](s) = [\![e_1]\!](s) - [\![e_2]\!](s)$
- $[\![e_1 * e_2]\!](s) = [\![e_1]\!](s) * [\![e_2]\!](s)$

其中 $s \in \text{state}$。

下面使用 Coq 递归函数定义整数类型表达式的行为。

```
Fixpoint eval_expr_int (e: expr_int) (s: state) : Z :=
  match e with
  | EConst n => n
  | EVar X => s X
  | EAdd e1 e2 => eval_expr_int e1 s + eval_expr_int e2 s
  | ESub e1 e2 => eval_expr_int e1 s - eval_expr_int e2 s
  | EMul e1 e2 => eval_expr_int e1 s * eval_expr_int e2 s
  end.
```

下面是两个具体的例子。

```
Example eval_example1: forall (s: state),
  s "x" = 1 ->
  s "y" = 2 ->
  eval_expr_int [["x" + "y"]] s = 3.
Proof. intros. simpl. rewrite H, H0. reflexivity. Qed.
```

```
Example eval_example2: forall (s: state),
  s "x" = 1 ->
  s "y" = 2 ->
  eval_expr_int [["x" * "y" + 1]] s = 3.
Proof. intros. simpl. rewrite H, H0. reflexivity. Qed.
```

2 行为等价

基于整数类型表达式的语义定义`eval_expr_int`，我们可以定义整数类型表达式之间的行为等价（亦称语义等价）：两个表达式`e1`与`e2`是等价的当且仅当它们在任何程序状态上的求值结果都相同。

$$e_1 \equiv e_2 \quad \text{iff.} \quad \forall s.\ [\![e_1]\!]\,(s) = [\![e_2]\!]\,(s)$$

```
Definition expr_int_equiv (e1 e2: expr_int): Prop :=
  forall st, eval_expr_int e1 st = eval_expr_int e2 st.
```

```
Notation "e1 '~=~' e2" := (expr_int_equiv e1 e2)
  (at level 69, no associativity).
```

下面是一些表达式语义等价的例子。

Example 1. $x * 2 \equiv x + x$

证明. 对于任意程序状态 s, $[\![x * 2]\!]\,(s) = s(x) * 2 = s(x) + s(x) = [\![x + x]\!]\,(s)$。 □

```
Example expr_int_equiv_sample:
  [["x" + "x"]] ~=~ [["x" * 2]].
```

```
Proof.
  intros.
  unfold expr_int_equiv.
```

上面的`unfold`指令表示展开一项定义，一般用于非递归的定义。

```
  intros.
  simpl.
  lia.
Qed.

Lemma zero_plus_equiv: forall (a: expr_int),
  [[0 + a]] ~=~ a.
(* 证明详见Coq源代码。 *)

Lemma plus_zero_equiv: forall (a: expr_int),
  [[a + 0]] ~=~ a.
(* 证明详见Coq源代码。 *)

Lemma minus_zero_equiv: forall (a: expr_int),
  [[a - 0]] ~=~ a.
(* 证明详见Coq源代码。 *)

Lemma zero_mult_equiv: forall (a: expr_int),
  [[0 * a]] ~=~ 0.
(* 证明详见Coq源代码。 *)

Lemma mult_zero_equiv: forall (a: expr_int),
  [[a * 0]] ~=~ 0.
(* 证明详见Coq源代码。 *)

Lemma const_plus_const: forall n m: Z,
  [[EConst n + EConst m]] ~=~ EConst (n + m).
(* 证明详见Coq源代码。 *)

Lemma const_minus_const: forall n m: Z,
  [[EConst n - EConst m]] ~=~ EConst (n - m).
(* 证明详见Coq源代码。 *)

Lemma const_mult_const: forall n m: Z,
  [[EConst n * EConst m]] ~=~ EConst (n * m).
(* 证明详见Coq源代码。 *)
```

下面定义一种简单的语法变换—常量折叠—并证明其保持语义等价性。所谓常量折叠指的是将只包含常量而不包含变量的表达式替换成为这个表达式的值。

```
Fixpoint fold_constants (e : expr_int) : expr_int :=
  match e with
  | EConst n    => EConst n
  | EVar x      => EVar x
  | EAdd e1 e2  =>
      match fold_constants e1, fold_constants e2 with
      | EConst n1, EConst n2 => EConst (n1 + n2)
      | _, _ => EAdd (fold_constants e1) (fold_constants e2)
      end
  | ESub e1 e2 =>
      match fold_constants e1, fold_constants e2 with
      | EConst n1, EConst n2 => EConst (n1 - n2)
      | _, _ => ESub (fold_constants e1) (fold_constants e2)
    end
  | EMul e1 e2 =>
      match fold_constants e1, fold_constants e2 with
      | EConst n1, EConst n2 => EConst (n1 * n2)
      | _, _ => EMul (fold_constants e1) (fold_constants e2)
    end
  end.
```

这里我们可以看到，Coq 中`match`的使用是非常灵活的。(1) 我们不仅可以对一个变量的值做分类讨论，还可以对一个复杂的 Coq 式子的取值做分类讨论；(2) 我们可以对多个值同时做分类讨论；(3) 我们可以用下划线表示`match`的缺省情况。下面是两个例子：

```
Example fold_constants_ex1:
    fold_constants [[(1 + 2) * "k"]] = [[3 * "k"]].
Proof. intros. reflexivity. Qed.
```

注意，根据我们的定义，`fold_constants`并不会将`0 + "y"`中的`0`消去。

```
Example fold_expr_int_ex2 :
  fold_constants [["x" - ((0 * 6) + "y")]] = [["x" - (0 + "y")]].
Proof. intros. reflexivity. Qed.
```

下面我们在 Coq 中证明，`fold_constants`保持表达式行为不变。

```
Theorem fold_constants_sound : forall a,
  fold_constants a ~=~ a.
Proof.
  unfold expr_int_equiv. intros.
  induction a.
```

常量的情况

```
  + simpl.
    reflexivity.
```

变量的情况

```
  + simpl.
    reflexivity.
```

加号的情况

```
  + simpl.
    destruct (fold_constants a1), (fold_constants a2);
    rewrite <- IHa1, <- IHa2;
    reflexivity.
```

减号的情况

```
  + simpl.
    destruct (fold_constants a1), (fold_constants a2);
    rewrite <- IHa1, <- IHa2;
    reflexivity.
```

乘号的情况

```
  + simpl.
    destruct (fold_constants a1), (fold_constants a2);
    rewrite <- IHa1, <- IHa2;
    reflexivity.
Qed.
```

3 利用高阶函数定义指称语义

```
Definition add_sem (D1 D2: state -> Z) (s: state): Z :=
  D1 s + D2 s.
```

```
Definition sub_sem (D1 D2: state -> Z) (s: state): Z :=
  D1 s - D2 s.
```

```
Definition mul_sem (D1 D2: state -> Z) (s: state): Z :=
  D1 s * D2 s.
```

下面是用于类型查询的`Check`指令。

```
Check add_sem.
```

可以看到`add_sem`的类型是`(state -> Z) -> (state -> Z) -> state -> Z`，这既可以被看做一个三元函数，也可以被看做一个二元函数，即函数之间的二元函数。

基于上面高阶函数，可以重新定义表达式的指称语义。

```
Definition const_sem (n: Z) (s: state): Z := n.
Definition var_sem (X: var_name) (s: state): Z := s X.
```

```
Fixpoint eval_expr_int (e: expr_int): state -> Z :=
  match e with
  | EConst n =>
      const_sem n
  | EVar X =>
      var_sem X
  | EAdd e1 e2 =>
      add_sem (eval_expr_int e1) (eval_expr_int e2)
  | ESub e1 e2 =>
      sub_sem (eval_expr_int e1) (eval_expr_int e2)
  | EMul e1 e2 =>
      mul_sem (eval_expr_int e1) (eval_expr_int e2)
  end.
```

4 布尔表达式语义

对于任意布尔表达式 e，我们规定它的语义 $[\![e]\!]$ 是一个程序状态到真值的函数，表示表达式 e 在各个程序状态上的求值结果。

- $[\![\mathrm{TRUE}]\!](s) = \mathbf{T}$
- $[\![\mathrm{FALSE}]\!](s) = \mathbf{F}$
- $[\![e_1 < e_2]\!](s)$ 为真当且仅当 $[\![e_1]\!](s) < [\![e_2]\!](s)$
- $[\![e_1 \&\& e_2]\!](s) = [\![e_1]\!](s)$ and $[\![e_2]\!](s)$
- $[\![!e_1]\!](s) = \mathrm{not}\ [\![e_1]\!](s)$

在 Coq 中可以如下定义：

```
Definition true_sem (s: state): bool := true.
```

```
Definition false_sem (s: state): bool := false.
```

```
Definition lt_sem (D1 D2: state -> Z) s: bool :=
  Z.ltb (D1 s) (D2 s).
```

```
Definition and_sem (D1 D2: state -> bool) s: bool :=
  andb (D1 s) (D2 s).
```

```
Definition not_sem (D: state -> bool) s: bool :=
  negb (D s).
```

```
Fixpoint eval_expr_bool (e: expr_bool): state -> bool :=
  match e with
  | ETrue =>
      true_sem
  | EFalse =>
      false_sem
  | ELt e1 e2 =>
      lt_sem (eval_expr_int e1) (eval_expr_int e2)
  | EAnd e1 e2 =>
      and_sem (eval_expr_bool e1) (eval_expr_bool e2)
  | ENot e1 =>
      not_sem (eval_expr_bool e1)
  end.
```

5 程序语句的语义

$(s_1, s_2) \in [\![c]\!]$ 当且仅当从 s_1 状态开始执行程序 c 会以程序状态 s_2 终止。

5.1 赋值语句的语义

$$[\![x = e]\!] = \{(s_1, s_2) \mid s_2(x) = [\![e]\!](s_1), \text{for any } y \in \text{var\_name},\ \text{if } x \neq y,\ s_1(y) = s_2(y)\}$$

5.2 空语句的语义

$$[\![\text{SKIP}]\!] = \{(s_1, s_2) \mid s_1 = s_2\}$$

5.3 顺序执行语句的语义

$$[\![c_1; c_2]\!] = [\![c_1]\!] \circ [\![c_2]\!] = \{(s_1, s_3) \mid (s_1, s_2) \in [\![c_1]\!], (s_2, s_3) \in [\![c_2]\!]\}$$

5.4 条件分支语句的语义

$$\begin{aligned}[\![\text{if } (e) \text{ then}\{c_1\} \text{ else } \{c_2\}]\!] = \quad & \left(\{(s_1, s_2) \mid [\![e]\!](s_1) = \mathbf{T}\} \cap [\![c_1]\!]\right) \cup \\ & \left(\{(s_1, s_2) \mid [\![e]\!](s_1) = \mathbf{F}\} \cap [\![c_2]\!]\right)\end{aligned}$$

这又可以改写为：

$[\![\text{if } (e) \text{ then}\{c_1\} \text{ else } \{c_2\}]\!] = \text{test\_true}([\![e]\!]) \circ [\![c_1]\!] \cup \text{test\_false}([\![e]\!]) \circ [\![c_2]\!]$

其中，

$$\text{test\_true}(F) = \{(s_1, s_2) \mid F(s_1) = \mathbf{T}, s_1 = s_2\}$$

$$\text{test\_false}(F) = \{(s_1, s_2) \mid F(s_1) = \mathbf{F}, s_1 = s_2\}$$

5.5 循环语句的语义

定义方式一：

$\text{iterLB}_0(X, R) = \text{test\_false}(X)$

$\text{iterLB}_{n+1}(X, R) = \text{test\_true}(X) \circ R \circ \text{iterLB}_n(X, R)$

$[\![\text{while } (e) \text{ do } \{c\}]\!] = \bigcup_{n\in\mathbb{N}} \text{iterLB}_n([\![e]\!], [\![c]\!])$

定义方式二：

$\text{boundedLB}_0(X, R) = \varnothing$

$\text{boundedLB}_{n+1}(X, R) = \text{test\_true}(X) \circ R \circ \text{boundedLB}_n(X, R) \cup \text{test\_false}(X)$

$[\![\text{while } (e) \text{ do } \{c\}]\!] = \bigcup_{n\in\mathbb{N}} \text{boundedLB}_n([\![e]\!], [\![c]\!])$

6 Coq 中的集合与关系

在 Coq 中往往使用 `X: A -> Prop` 来表示某类型 `A` 中元素构成的集合 `X`。字面上看，这里的 `A -> Prop` 表示 `X` 是一个从 `A` 中元素到命题的映射，这也相当于说 `X` 是一个关于 `A` 中元素性质。对于每个 `A` 中元素 `a` 而言，`a` 符合该性质 `X` 等价于 `a` 对应的命题 `X a` 为真，又等价于 `a` 是集合 `X` 的元素，在 SetsClass 库中也直接写作 `a ∈ X`。

类似的，`R: A -> B -> Prop` 也用来表示 `A` 与 `B` 中元素之间的二元关系。

本课程提供的 SetsClass 库中提供了有关集合的一系列定义。例如：

- 空集：用 `∅` 或者一堆方括号表示，定义为 `Sets.empty`；
- 单元集：用一对方括号表示，定义为 `Sets.singleton`；
- 并集：用 `∪` 表示，定义为 `Sets.union`；
- 交集：用 `∩` 表示，定义为 `Sets.intersect`；
- 一列集合的并：用 `⋃` 表示，定义为 `Sets.indexed_union`；
- 一列集合的交：用 `⋂` 表示，定义为 `Sets.indexed_intersect`；
- 集合相等：用 `==` 表示，定义为 `Sets.equiv`；
- 元素与集合关系：用 `∈` 表示，定义为 `Sets.In`；
- 子集关系：用 `⊆` 表示，定义为 `Sets.included`；
- 二元关系的连接：用 `∘` 表示，定义为 `Rels.concat`；
- 等同关系：定义为 `Rels.id`；
- 测试关系：定义为 `Rels.test`。

在 CoqIDE 中，你可以利用 CoqIDE 对于 unicode 的支持打出特殊字符：

- 首先，在打出特殊字符的 latex 表示法；
- 再按 shift+ 空格键；
- latex 表示法就自动转化为了相应的特殊字符。

例如，如果你需要打出符号 `∈`，请先在输入框中输入 `\in`，当光标紧跟在 `n` 这个字符之后的时候，按 shift+ 空格键即可。例如，下面是两个关于集合的命题：

```
Check forall A (X: A -> Prop), X ∪ ∅ == X.

Check forall A B (X Y: A -> B -> Prop), X ∪ Y ⊆ X.
```

由于集合以及集合间的运算是基于 Coq 中的命题进行定义的，集合相关性质的证明也可以规约为与命题有关的逻辑证明。例如，我们想要证明，交集运算具有交换律：

```
Lemma Sets_intersect_comm: forall A (X Y: A -> Prop),
  X ∩ Y == Y ∩ X.
Proof.
  intros.
```

下面一条命令`Sets_unfold`是 SetsClass 库提供的自动证明指令，它可以将有关集合的性质转化为有关命题的性质。

```
  Sets_unfold.
```

原本要证明的关于交集的性质现在就转化为了：`forall a : A, a ∈ X /\ a ∈ Y <-> a ∈ Y /\ a ∈ X` 这个关于逻辑的命题在 Coq 中是容易证明的。

```
  intros.
  tauto.
Qed.
```

下面是一条关于并集运算的性质。

```
Lemma Sets_included_union1: forall A (X Y: A -> Prop),
  X ⊆ X ∪ Y.
Proof.
  intros.
  Sets_unfold.
```

经过转化，要证明的结论是：`forall a : A, a ∈ X -> a ∈ X \/ a ∈ Y`。

```
  intros.
  tauto.
Qed.
```

习题 1.

下面是一条关于二元关系复合的性质。转化后得到的命题要复杂一些，请在 Coq 中证明这个关于逻辑的命题。

```
Lemma Rels_concat_assoc: forall A (X Y Z: A -> A -> Prop),
  (X ∘ Y) ∘ Z == X ∘ (Y ∘ Z).
Proof.
  intros.
  Sets_unfold.
(* 请在此处填入你的证明，以_[Qed]_结束。 *)
```

SetsClass 库中提供了一系列有关集合运算的性质的证明。未来大家在证明中既可以使用`Sets_unfold`将关于集合运算的命题转化为关于逻辑的命题，也可以直接使用下面这些性质完成证明。

```
Sets_equiv_Sets_included:
  forall x y, x == y <-> x ⊆ y /\ y ⊆ x;
Sets_empty_included:
  forall x, ∅ ⊆ x;
Sets_included_full:
  forall x, x ⊆ Sets.full;
Sets_intersect_included1:
  forall x y, x ∩ y ⊆ x;
Sets_intersect_included2:
  forall x y, x ∩ y ⊆ y;
Sets_included_intersect:
  forall x y z, x ⊆ y -> x ⊆ z -> x ⊆ y ∩ z;
Sets_included_union1:
  forall x y, x ⊆ x ∪ y;
Sets_included_union2:
  forall x y, y ⊆ x ∪ y;
Sets_union_included_strong2:
  forall x y z u,
     x ∩ u ⊆ z -> y ∩ u ⊆ z -> (x ∪ y) ∩ u ⊆ z;
```

```
Sets_included_indexed_union:
  forall xs n, xs n ⊆ ⋃ xs;
Sets_indexed_union_included:
  forall xs y, (forall n, xs n ⊆ y) -> ⋃ xs ⊆ y;
Sets_indexed_intersect_included:
  forall xs n, ⋂ xs ⊆ xs n;
Sets_included_indexed_intersect:
  forall xs y, (forall n : nat, y ⊆ xs n) -> y ⊆ ⋂ xs;
```

```
Rels_concat_union_distr_r:
  forall x1 x2 y,
    (x1 ∪ x2) ∘ y == (x1 ∘ y) ∪ (x2 ∘ y);
Rels_concat_union_distr_l:
  forall x y1 y2,
    x ∘ (y1 ∪ y2) == (x ∘ y1) ∪ (x ∘ y2);
Rels_concat_mono:
  forall x1 x2,
    x1 ⊆ x2 ->
    forall y1 y2,
      y1 ⊆ y2 ->
      x1 ∘ y1 ⊆ x2 ∘ y2;
Rels_concat_assoc:
  forall x y z,
    (x ∘ y) ∘ z == x ∘ (y ∘ z);
Rels_concat_id_l:
  forall x, Rels.id ∘ x == x;
Rels_concat_id_r:
  forall x, x ∘ Rels.id == x;
```

由于上面提到的集合与关系运算都能保持集合相等，也都能保持集合包含关系，因此 SetsClass 库支持其用户使用`rewrite`指令处理集合之间的相等关系与包含关系。下面是两个典型的例子

```
Fact sets_ex1:
  forall (A: Type) (R1 R2 R3 R4: A -> A -> Prop),
    R1 ∘ (R2 ∘ (R3 ∘ R4)) == ((R1 ∘ R2) ∘ R3) ∘ R4.
Proof.
  intros.
  rewrite Rels_concat_assoc.
  rewrite Rels_concat_assoc.
  reflexivity.
Qed.
```

```
Fact sets_ex2:
  forall (A: Type) (R1 R2 R3 R4: A -> A -> Prop),
    R1 ⊆ R2 ->
    R1 ∘ R3 ∪ R4 ⊆ R2 ∘ R3 ∪ R4.
Proof.
  intros.
  rewrite H.
  reflexivity.
Qed.
```

7 在 Coq 中定义程序语句的语义

下面在 Coq 中写出程序语句的指称语义。

```
Definition skip_sem: state -> state -> Prop :=
  Rels.id.
```

```
Definition asgn_sem
             (X: var_name)
             (D: state -> Z)
             (st1 st2: state): Prop :=
  st2 X = D st1 /\
  forall Y, X <> Y -> st2 Y = st1 Y.
```

```
Definition seq_sem
             (D1 D2: state -> state -> Prop):
  state -> state -> Prop :=
  D1 ∘ D2.
```

```
Definition test_true
             (D: state -> bool):
  state -> state -> Prop :=
  Rels.test (fun st => D st = true).
```

```
Definition test_false
             (D: state -> bool):
  state -> state -> Prop :=
  Rels.test (fun st => D st = false).
```

```
Definition if_sem
             (D0: state -> bool)
             (D1 D2: state -> state -> Prop):
  state -> state -> Prop :=
  (test_true D0 ∘ D1) ∪ (test_false D0 ∘ D2).
```

```
Fixpoint iterLB
           (D0: state -> bool)
           (D1: state -> state -> Prop)
           (n: nat):
  state -> state -> Prop :=
  match n with
  | O => test_false D0
  | S n0 => test_true D0 ∘ D1 ∘ iterLB D0 D1 n0
  end.
```

```
Definition while_sem
             (D0: state -> bool)
             (D1: state -> state -> Prop):
  state -> state -> Prop :=
  ⋃ (iterLB D0 D1).
```

```
Fixpoint boundedLB
           (D0: state -> bool)
           (D1: state -> state -> Prop)
           (n: nat):
  state -> state -> Prop :=
  match n with
  | O => ∅
  | S n0 =>
      (test_true D0 ∘ D1 ∘ boundedLB D0 D1 n0) ∪
      (test_false D0)
  end.
```

```
Definition while_sem
             (D0: state -> bool)
             (D1: state -> state -> Prop):
  state -> state -> Prop :=
  ⋃ (boundedLB D0 D1).
```

下面是程序语句指称语义的递归定义。

```
Fixpoint eval_com (c: com): state -> state -> Prop :=
  match c with
  | CSkip =>
      skip_sem
  | CAsgn X e =>
      asgn_sem X (eval_expr_int e)
  | CSeq c1 c2 =>
      seq_sem (eval_com c1) (eval_com c2)
  | CIf e c1 c2 =>
      if_sem (eval_expr_bool e) (eval_com c1) (eval_com c2)
  | CWhile e c1 =>
      while_sem (eval_expr_bool e) (eval_com c1)
  end.
```

基于上面定义，可以证明一些简单的程序性质。

```
Example inc_x_fact: forall s1 s2 n,
  (s1, s2) ∈ eval_com (CAsgn "x" [["x" + 1]]) ->
  s1 "x" = n ->
  s2 "x" = n + 1.
Proof.
  intros.
  simpl in H.
  unfold asgn_sem, add_sem, var_sem, const_sem in H.
  lia.
Qed.
```

更多关于程序行为的有用性质可以使用集合与关系的运算性质完成证明，`seq_skip`与`skip_seq`表明了删除顺序执行中多余的空语句不改变程序行为。

```
Lemma seq_skip: forall c,
  eval_com (CSeq c CSkip) == eval_com c.
Proof.
  intros.
  simpl.
  unfold seq_sem, skip_sem.
  apply Rels_concat_id_r.
Qed.
```

```
Lemma skip_seq: forall c,
  eval_com (CSeq CSkip c) == eval_com c.
Proof.
  intros.
  simpl.
  unfold seq_sem, skip_sem.
  apply Rels_concat_id_l.
Qed.
```

类似的，`seq_assoc`表明顺序执行的结合顺序是不影响程序行为的，因此，所有实际的编程中都不需要在程序开发的过程中额外标明顺序执行的结合方式。

```
Lemma seq_assoc: forall c1 c2 c3,
  eval_com (CSeq (CSeq c1 c2) c3) ==
  eval_com (CSeq c1 (CSeq c2 c3)).
Proof.
  intros.
  simpl.
  unfold seq_sem.
  apply Rels_concat_assoc.
Qed.
```

下面的`while_sem_congr2`说的则是：如果对循环体做行为等价变换，那么整个循环的行为也不变。

```
Lemma while_sem_congr2: forall D1 D2 D2',
  D2 == D2' ->
  while_sem D1 D2 == while_sem D1 D2'.
Proof.
  intros.
  unfold while_sem.
  apply Sets_indexed_union_congr.
  intros n.
  induction n; simpl.
  + reflexivity.
  + rewrite IHn.
    rewrite H.
    reflexivity.
Qed.
```

下面我们证明，我们先前定义的`remove_skip`变换保持程序行为不变。

```
Theorem remove_skip_sound: forall c,
  eval_com (remove_skip c) == eval_com c.
(* 证明详见Coq源代码。 *)
```

前面提到，while 循环语句的行为也可以描述为：只要循环条件成立，就先执行循环体再重新执行循环。我们可以证明，我们目前定义的程序语义符合这一性质。

```
Lemma while_sem_unroll1: forall D0 D1,
  while_sem D0 D1 ==
  test_true D0 ∘ D1 ∘ while_sem D0 D1 ∪ test_false D0.
Proof.
  intros.
  simpl.
  unfold while_sem.
  apply Sets_equiv_Sets_included; split.
  + apply Sets_indexed_union_included.
    intros.
    destruct n as [| n0]; simpl boundedLB.
    - apply Sets_empty_included.
    - rewrite <- (Sets_included_indexed_union _ _ n0).
      reflexivity.
  + apply Sets_union_included.
    - rewrite Rels_concat_indexed_union_distr_l.
      rewrite Rels_concat_indexed_union_distr_l.
      apply Sets_indexed_union_included; intros.
      rewrite <- (Sets_included_indexed_union _ _ (S n)).
      simpl.
      apply Sets_included_union1.
    - rewrite <- (Sets_included_indexed_union _ _ (S 0)).
      simpl boundedLB.
      apply Sets_included_union2.
Qed.
```

从这一结论可以看出，`while_sem D0 D1`是下面方程的一个解：

`X == test_true D0 ∘ D1 ∘ X ∪ test_false D0`;

数学上，如果一个函数 f 与某个取值 x 满足 $f(x) = x$，那么就称 x 是 f 的一个不动点。因此，我们也可以说`while_sem D0 D1`是下面函数的一个不动点：

课后阅读：Coq 中自然数相关的定义与证明

在 Coq 中，许多数学上的集合可以用归纳类型定义。例如，Coq 中自然数的定义就是最简单的归纳类型之一。

下面 Coq 代码可以用于查看`nat`在 Coq 中的定义。

```
Print nat.
```

查询结果如下。

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

可以看到，自然数集合的归纳定义可以看做`list`进一步退化的结果。下面我们在 Coq 中定义自然数的加法，并且也试着证明一条基本性质：加法交换律。

由于 Coq 的标准库中已经定义了自然数以及自然数的加法。我们开辟一个`NatDemo`来开发我们自己的定义与证明。以免与 Coq 标准库的定义相混淆。

```
Module NatDemo.
```

先定义自然数`nat`。

```
Inductive nat :=
| O: nat
| S (n: nat): nat.
```

再定义自然数加法。

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

下面证明加法交换律。

```
Theorem add_comm: forall n m,
  add n m = add m n.
Proof.
  intros.
  induction n.
```

证明到此处，我们发现我们需要首先证明`n + 0 = n`这条性质，我们先终止交换律的证明，而先证明这条引理。

```
Abort.
```

```
Lemma add_0_r: forall n, add n 0 = n.
Proof.
  intros.
  induction n; simpl.
  + reflexivity.
  + rewrite IHn.
    reflexivity.
Qed.
```

```
Theorem add_comm: forall n m,
  add n m = add m n.
Proof.
  intros.
  induction n; simpl.
  + rewrite add_0_r.
    reflexivity.
  +
```

证明到此处，我们发现我们需要还需要证明关于`m + (S n)`相关的性质。

```
Abort.
```

```
Lemma add_succ_r: forall n m,
  add n (S m) = S (add n m).
Proof.
  intros.
  induction n; simpl.
  + reflexivity.
  + rewrite IHn.
    reflexivity.
Qed.
```

现在已经可以在 Coq 中完成加法交换律的证明了。

```
Theorem add_comm: forall n m,
  add n m = add m n.
Proof.
  intros.
  induction n; simpl.
  + rewrite add_0_r.
    reflexivity.
  + rewrite add_succ_r.
    rewrite IHn.
    reflexivity.
Qed.
```

由于自然数范围内，数学意义上的减法是一个部分函数，因此，相关定义在 Coq 中并不常用。相对而言，自然数的加法与乘法在 Coq 中更常用。

```
Fixpoint mul (n m: nat): nat :=
  match n with
  | O => O
  | S p => add m (mul p m)
  end.
```

下面列举加法与乘法的其它重要性质。

```
Theorem add_assoc:
  forall n m p, add n (add m p) = add (add n m) p.
(* 证明详见Coq源代码。 *)
```

```
Theorem add_cancel_l:
  forall n m p, add p n = add p m <-> n = m.
(* 证明详见Coq源代码。 *)
```

```
Theorem add_cancel_r:
  forall n m p, add n p = add m p <-> n = m.
(* 证明详见Coq源代码。 *)
```

```
Lemma mul_0_r: forall n, mul n O = O.
(* 证明详见Coq源代码。 *)
```

```
Lemma mul_succ_r:
  forall n m, mul n (S m) = add (mul n m) n.
(* 证明详见Coq源代码。 *)
```

```
Theorem mul_comm:
  forall n m, mul n m = mul m n.
(* 证明详见Coq源代码。 *)
```

```
Theorem mul_add_distr_r:
  forall n m p, mul (add n m) p = add (mul n p) (mul m p).
(* 证明详见Coq源代码。 *)
```

```
Theorem mul_add_distr_l:
  forall n m p, mul n (add m p) = add (mul n m) (mul n p).
(* 证明详见Coq源代码。 *)
```

```
Theorem mul_assoc:
  forall n m p, mul n (mul m p) = mul (mul n m) p.
(* 证明详见Coq源代码。 *)
```

```
Theorem mul_1_l : forall n, mul (S O) n = n.
(* 证明详见Coq源代码。 *)
```

```
Theorem mul_1_r : forall n, mul n (S O) = n.
(* 证明详见Coq源代码。 *)
```

```
End NatDemo.
```

上面介绍的加法与乘法运算性质在 Coq 标准库中已有证明，其定理名称如下。

```
Check Nat.add_comm.
Check Nat.add_assoc.
Check Nat.add_cancel_l.
Check Nat.add_cancel_r.
Check Nat.mul_comm.
Check Nat.mul_add_distr_r.
Check Nat.mul_add_distr_l.
Check Nat.mul_assoc.
Check Nat.mul_1_l.
Check Nat.mul_1_r.
```

前面已经提到，Coq 在自然数集合上不便于表达减法等运算，因此，Coq 用户有些时候可以选用 `Z` 而非 `nat`。然而，由于其便于表示计数概念以及表述数学归纳法，`nat` 依然有许多用途。例如，Coq 标准库中的 `Nat.iter` 就表示函数多次迭代，具体而言，`Nat.iter n f` 表示将函数 `f` 迭代 `n` 次的结果。其 Coq 定义如下：

```
Fixpoint iter {A: Type} (n: nat) (f: A -> A) (x: A): A :=
  match n with
  | O => x
  | S n' => f (iter n' f x)
  end.
```

它符合许多重要性质，例如：

```
Theorem iter_S: forall {A: Type} (n: nat) (f: A -> A) (x: A),
  Nat.iter n f (f x) = Nat.iter (S n) f x.
```

注意，哪怕是如此简单的性质，我们还是需要在 Coq 中使用归纳法证明。

```
Proof.
  intros.
  induction n; simpl.
  + reflexivity.
  + rewrite IHn; simpl.
    reflexivity.
Qed.
```

习题 1. 请证明下面关于 `Nat.iter` 的性质。

```
Theorem iter_add: forall {A: Type} (n m: nat) (f: A -> A) (x: A),
  Nat.iter (n + m) f x = Nat.iter n f (Nat.iter m f x).
(* 请在此处填入你的证明，以_[Qed]_结束。 *)
```

习题 2. 请证明下面关于 `Nat.iter` 的性质。

```
Theorem iter_mul: forall {A: Type} (n m: nat) (f: A -> A) (x: A),
  Nat.iter (n * m) f x = Nat.iter n (Nat.iter m f) x.
(* 请在此处填入你的证明，以_[Qed]_结束。 *)
```

课后阅读：Coq 中关于逻辑的证明

【写在开始的注】课前阅读与课后阅读中的习题仅供参考，不是作业内容。

数学中可以用“并且”、“或”、“非”、“如果-那么”、“存在”以及“任意”把简单性质组合起来构成复杂性质或复杂命题，例如“单调且连续”与“无限且不循环”这两个常用数学概念的定义中就用到了逻辑连接词“并且”，又例如“有零点”这一数学概念的定义就要用到“存在”这个逻辑中的量词（quantifier）。Coq 中也允许用户使用这些常用的逻辑符号。

Coq 表达式 1. 逻辑表达式. Coq 标准库中定义的逻辑符号有：

- “并且”：`/\`
- “或”：`\/`
- “非”：`~`
- “如果-那么”：`->`
- “当且仅当”：`<->`
- “真”：`True`
- “假”：`False`
- “存在”：`exists`
- “任意”：`forall`。

这些符号中，“存在”与“任意”的优先级最低，之后优先级从低到高依次是“当且仅当”、“如果-那么”、“或”、“并且”与“非”。值得一提的是，Coq 的二元逻辑连接词中“并且”、“或”以及“如果-那么”都是右结合的，换言之，`P /\ Q /\ R`是`P /\ (Q /\ R)`的简写。Coq 中只有“当且仅当”这个逻辑连接词是左结合的。

下面是一个使用逻辑符号定义复合命题的例子。当我们如下定义“下凸函数”时，就可以用`mono f /\ convex f`表示函数`f`是一个单调下凸函数。

```
Definition convex (f: Z -> Z): Prop :=
  forall x: Z, f (x - 1) + f (x + 1) >= 2 * f x.
```

下面性质说的是，如果一个函数变换`T`能保持单调性，也能保持凸性，那么它也能保持“单调下凸”这个性质。

```
Fact logic_ex1: forall T: (Z -> Z) -> (Z -> Z),
  (forall f, mono f -> mono (T f)) ->
  (forall f, convex f -> convex (T f)) ->
  (forall f, mono f /\ convex f -> mono (T f) /\ convex (T f)).
```

不难发现，这一性质的证明与单调性的定义无关，也和凸性的定义无关，这一性质的证明只需用到其中各个逻辑符号的性质。下面是 Coq 证明

```
Proof.
  intros.
  pose proof H f.
  pose proof H0 f.
  tauto.
Qed.
```

最后一条证明指令`tauto`是英文单词“tautology”的缩写，表示当前证明目标是一个命题逻辑永真式，可以自动证明。在上面证明中，如果把命题`mono f`与`convex f`记作命题`P1`与`Q1`，将命题`mono (T f)`看作一个整体记作`P2`，将命题`convex (T f)`也看作一个整体记为`Q2`，那么证明指令`tauto`在此处证明的结论就可以概括为：如果

- `P1`成立并且`Q1`成立（前提`H1`）
- `P1`能推出`P2`（前提`H2`）
- `Q1`能推出`Q2`（前提`H3`）

那么`P2`成立并且`Q2`成立。不难看出，无论`P1`、`Q1`、`P2`与`Q2`这四个命题中的每一个是真是假，上述推导都成立。因此，这一推导过程可以用一个命题逻辑永真式刻画，`tauto`能够自动完成它的证明。

Coq 中也可以把这一原理单独地表述出来：

```
Fact logic_ex2: forall P1 Q1 P2 Q2: Prop,
  P1 /\ Q1 ->
  (P1 -> P2) ->
  (Q1 -> Q2) ->
  P2 /\ Q2.
Proof.
  intros P1 Q1 P2 Q2 H1 H2 H3.
  tauto.
Qed.
```

仅仅利用`tauto`指令就已经能够证明不少关于逻辑的结论了。下面两个例子刻画了一个命题与其逆否命题的关系。

```
Fact logic_ex3: forall {A: Type} (P Q: A -> Prop),
  (forall a: A, P a -> Q a) ->
  (forall a: A, ~ Q a -> ~ P a).
Proof. intros A P Q H a. pose proof H a. tauto. Qed.
```

```
Fact logic_ex4: forall {A: Type} (P Q: A -> Prop),
  (forall a: A, ~ Q a -> ~ P a) ->
  (forall a: A, P a -> Q a).
Proof. intros A P Q H a. pose proof H a. tauto. Qed.
```

Coq 证明脚本 1. tauto 指令. 如果当前证明目标可以完全通过命题逻辑永真式完成证明，那么`tauto`就可以自动构造这样的证明。这里，所谓完全通过命题逻辑永真式完成证明，指的是通过对于“并且”、“或”、“非”、“如果-那么”、“当且仅当”、“真”与“假”的推理完成证明。另外，`tauto`也能支持关于等式的简单证明，具体而言，`tauto`会将形如`a = a`的命题看做“真”，将形如`a <> a`的命题看作“假”。

习题 1. 请在 Coq 中证明下面结论。

```
Fact logic_ex5: forall {A: Type} (P Q: A -> Prop),
  (forall a: A, P a -> Q a) ->
  (forall a: A, P a) ->
  (forall a: A, Q a).
Proof.
(* 请在此处填入你的证明，以_[Qed]_结束。 *)
```

习题 2. 请在 Coq 中证明下面结论。

```
Fact logic_ex6: forall {A: Type} (P Q: A -> Prop) (a0: A),
  P a0 ->
  (forall a: A, P a -> Q a) ->
  Q a0.
Proof.
(* 请在此处填入你的证明，以_[Qed]_结束。 *)
```

习题 3. 请在 Coq 中证明下面结论。

```
Fact logic_ex7: forall {A: Type} (P Q: A -> Prop) (a0: A),
  (forall a: A, P a -> Q a -> False) ->
  Q a0 ->
  ~ P a0.
Proof.
(* 请在此处填入你的证明，以_[Qed]_结束。 *)
```

习题 4. 请在 Coq 中证明下面结论。

```
Fact logic_ex8: forall {A B: Type} (P Q: A -> B -> Prop),
  (forall (a: A) (b: B), P a b -> Q a b) ->
  (forall (a: A) (b: B), ~ P a b \/ Q a b).
Proof.
(* 请在此处填入你的证明，以_[Qed]_结束。 *)
```

习题 5. 请在 Coq 中证明下面结论。

```
Fact logic_ex9: forall {A B: Type} (P Q: A -> B -> Prop),
  (forall (a: A) (b: B), ~ P a b \/ Q a b) ->
  (forall (a: A) (b: B), P a b -> Q a b).
Proof.
(* 请在此处填入你的证明，以_[Qed]_结束。 *)
```

尽管`tauto`指令利用命题逻辑永真式已经能够证明不少逻辑性质，但有些时候，我们需要对包含逻辑符号的命题进行更细粒度的操作才能完成 Coq 证明。本章的后续各节将展开介绍这些 Coq 中的证明方法。

1 关于“并且”的证明

要证明“某命题甲并且某命题乙”成立，可以在 Coq 中使用`split`证明指令进行证明。该指令会将当前的证明目标拆成两个子目标。

```
Lemma and_intro: forall A B: Prop, A -> B -> A /\ B.
Proof.
  intros A B HA HB.
  split.
```

下面的`apply`指令表示在证明中使用一条前提，或者使用一条已经经过证明的定理或引理。

```
  + apply HA.
  + apply HB.
Qed.
```

如果当前一条前提假设具有“某命题并且某命题”的形式，我们可以在 Coq 中使用`destruct`指令将其拆分成为两个前提。

```
Lemma proj1: forall P Q: Prop,
  P /\ Q -> P.
Proof.
  intros.
  destruct H as [HP HQ].
  apply HP.
Qed.
```

另外，`destruct`指令也可以不指名拆分后的前提的名字，Coq 会自动命名。

```
Lemma proj2: forall P Q: Prop,
  P /\ Q -> Q.
Proof.
  intros.
  destruct H.
  apply H0.
Qed.
```

当前提与结论中，都有`/\`的时候，我们就既需要使用`split`指令，又需要使用`destruct`指令。

```
Theorem and_comm: forall P Q: Prop,
  P /\ Q -> Q /\ P.
Proof.
  intros.
  destruct H as [HP HQ].
  split.
  + apply HQ.
  + apply HP.
Qed.
```

习题 6. 请在不使用`tauto`指令的情况下证明下面结论。

```
Theorem and_assoc1: forall P Q R: Prop,
  P /\ (Q /\ R) -> (P /\ Q) /\ R.
(* 请在此处填入你的证明，以_[Qed]_结束。 *)
```

```
Theorem and_assoc2: forall P Q R: Prop,
  (P /\ Q) /\ R -> P /\ (Q /\ R).
(* 请在此处填入你的证明，以_[Qed]_结束。 *)
```

2 关于“或”的证明

“或”是另一个重要的逻辑连接词。如果“或”出现在前提中，我们可以用 Coq 中的`destruct`指令进行分类讨论。在下面的例子中，我们对于前提`P \/ Q`进行分类讨论。要证明`P \/ Q`能推出原结论，就需要证明`P`与`Q`中的任意一个都可以推出原结论。

```
Fact or_example:
  forall P Q R: Prop, (P -> R) -> (Q -> R) -> (P \/ Q -> R).
Proof.
  intros.
  destruct H1 as [HP | HQ].
  + pose proof H HP.
    apply H1.
  + pose proof H0 HQ.
    apply H1.
Qed.
```

相反的，如果要证明一条形如`A \/ B`的结论整理，我们就只需要证明`A`与`B`两者之一成立就可以了。在 Coq 中的指令是：`left`与`right`。例如，下面是选择左侧命题的例子。

```
Lemma or_introl: forall A B: Prop, A -> A \/ B.
Proof.
  intros.
  left.
  apply H.
Qed.
```

下面是选择右侧命题的例子。

```
Lemma or_intror: forall A B: Prop, B -> A \/ B.
Proof.
  intros.
  right.
  apply H.
Qed.
```

习题 7. 请在不使用`tauto`指令的情况下证明下面结论。

```
Theorem or_comm: forall P Q: Prop,
  P \/ Q  -> Q \/ P.
(* 请在此处填入你的证明，以_[Qed]_结束。 *)
```

习题 8. 请在不使用`tauto`指令的情况下证明下面结论。

```
Theorem or_assoc1: forall P Q R: Prop,
  P \/ (Q \/ R)  -> (P \/ Q) \/ R.
(* 请在此处填入你的证明，以_[Qed]_结束。 *)
```

```
Theorem or_assoc2: forall P Q R: Prop,
  (P \/ Q) \/ R -> P \/ (Q \/ R).
(* 请在此处填入你的证明，以_[Qed]_结束。 *)
```

3 关于“当且仅当”的证明

在 Coq 中，`<->`符号对应的定义是`iff`，其将`P <-> Q`定义为`(P -> Q) /\ (Q -> P)`因此，要证明关于“当且仅当”的性质，首先可以使用其定义进行证明。

```
Theorem iff_refl: forall P: Prop, P <-> P.
Proof.
  intros.
  unfold iff.
  split.
  + intros.
    apply H.
  + intros.
    apply H.
Qed.
```

Coq 也允许在不展开“当且仅当”的定义时就是用`split`或`destruct`指令进行证明。

```
Theorem and_dup: forall P: Prop, P /\ P <-> P.
Proof.
  intros.
  split.
  + intros.
    destruct H.
    apply H.
  + intros.
    split.
    - apply H.
    - apply H.
Qed.
```

```
Theorem iff_imply: forall P Q: Prop, (P <-> Q) -> (P -> Q).
Proof.
  intros P Q H.
  destruct H.
  apply H.
Qed.
```

习题 9. 请在不使用`tauto`指令的情况下证明下面结论。

```
Theorem or_dup: forall P: Prop, P \/ P <-> P.
(* 请在此处填入你的证明，以_[Qed]_结束。 *)
```

4 命题逻辑综合应用

下面是证明“并且”、“或”与“当且仅当”时常用的证明指令汇总与拓展。

Coq 证明脚本 2. left 指令与 right 指令. 如果待证明结论具有`P \/ Q`的形式，那么`left`可以将该结论规约为`P`，`right`可以将该结论规约为`Q`。

Coq 证明脚本 3. 命题逻辑证明中的 split 指令. 如果待证明结论具有`P /\ Q`的形式，那么`split`可以将当前证明目标规约为两个更简单的证明目标，它们的前提与原证明目标的前提相同，它们的结论分别为`P`与`Q`。

Coq 证明脚本 4. 命题逻辑证明中的 destruct 指令. 如果证明时前提`H`具有形式`P /\ Q`或具有形式`P \/ Q`，则可以使用`destruct H`指令。当`H`具有形式`P /\ Q`时，该指令会将此前提分解为两个前提`P`与`Q`。当`H`具有形式`P \/ Q`时，该指令会将当前证明目标规约为两个证明目标，其中一个将前提`H`被改为`P`，另一个将前提`H`改为`Q`。

Coq 允许用户使用`destruct ... as ...`指令对 destruct 得到的新前提手动重命名。例如当`H`具有形式`P /\ Q`时，`destruct H as [H1 H2]`指令将生成下面两个证明前提：

```
H1: P
H2: Q
```

又例如当`H`具有形式`P \/ Q`时，`destruct H as [H1 | H2]`指令也可以用于手动命名。与 intros 指令中的规定一样，当需要对 destruct 结果中的一部分手动命名而对另一部分自动命名时，可以使用问号`?`表示那些需要由 Coq 自动命名的名字。

Coq 允许用户对多个前提同时执行 destruct 指令，例如`destruct H, H0`就表示先`destruct H`再`destruct H0`。Coq 也允许用户在一条 destruct 指令中对 destruct 的结果再进一步分解或进一步分类讨论，但具体需要分解多少层，需要使用`destruct ... as ...`指令做具体说明。例如，当`H`具有形式

```
(P /\ Q) /\ (R /\ S)
```

时，`destruct H as [? [? ?]]`指令会将`H`分解为`P /\ Q`、`R`与`S`；而`destruct H as [[? ?] ?]`指令会将`H`分解为`P`、`Q`与`R /\ S`。另外，对“并且”与“或”的分解与分类讨论也可以相互嵌套，例如，当`H`具有形式

```
(P /\ Q) \/ R
```

时，可以使用`destruct H as [[HP HQ] | HR]`指令在分类讨论的同时对其中一个分类讨论的分支对前提做进一步分解。

Coq 证明脚本 5. 引入模式. 前面已经介绍，`destruct ... as ...`指令可以一次性完成若干次的命题拆解或分类讨论。在这一指令中，`as`之后的结构成为“引入模式”（intro-pattern）。以下是与目前所学相关的几种引入模式：

- 针对“并且”的命题拆解`intro_pattern1 intro_pattern2`；
- 针对“或”的分类讨论`intro_pattern1 | intro_pattern2`；
- 新引入的名字，例如`H`；
- 表示由 Coq 系统自动命名的问号`?`；
- 表示直接丢弃拆分结果的下划线`_`；

除了 destruct 指令之外，intros 指令与 pose proof 指令也可以使用引入模式，在完成原有功能的基础上，再进行一次 destruct。例如，`intros [H1 H2] [H3 | H3]`相当于依次执行：

```
intros H1 H2
destruct H1 as [H1 H2]
destruct H3 as [H3 | H3];
```

而`pose proof H x y as [H1 H2]`相当于依次执行：

```
pose proof H x y as H1
destruct H1 as [H1 H2]。
```

下面罗列了一些命题逻辑中的常见性质。请有兴趣的读者综合前几节所学内容，在不使用`tauto`证明指令的限制下完成下面证明。

习题 10. 请在 Coq 中证明“假言推理”规则。

```
Theorem modus_ponens: forall P Q: Prop,
  P /\ (P -> Q) -> Q.
(* 请在此处填入你的证明，以_[Qed]_结束。 *)
```

习题 11. 请在 Coq 中证明“并且”对“或”的分配律。

```
Theorem and_or_distr_l: forall P Q R: Prop,
  P /\ (Q \/ R) <-> P /\ Q \/ P /\ R.
(* 请在此处填入你的证明，以_[Qed]_结束。 *)
```

习题 12. 请在 Coq 中证明“或”对“并且”的分配律。

```
Theorem or_and_distr_l: forall P Q R: Prop,
  P \/ (Q /\ R) <-> (P \/ Q) /\ (P \/ R).
(* 请在此处填入你的证明，以_[Qed]_结束。 *)
```

习题 13. 请在 Coq 中证明“并且”对“或”的吸收律。

```
Theorem and_or_absorb: forall P Q: Prop,
  P /\ (P \/ Q) <-> P.
(* 请在此处填入你的证明，以_[Qed]_结束。 *)
```

习题 14. 请在 Coq 中证明“或”对“并且”的吸收律。

```
Theorem or_and_absorb: forall P Q: Prop,
  P \/ (P /\ Q) <-> P.
(* 请在此处填入你的证明，以_[Qed]_结束。 *)
```

习题 15. 请在 Coq 中证明“并且”能保持逻辑等价性。

```
Theorem and_congr: forall P1 Q1 P2 Q2: Prop,
  (P1 <-> P2) ->
  (Q1 <-> Q2) ->
  (P1 /\ Q1 <-> P2 /\ Q2).
(* 请在此处填入你的证明，以_[Qed]_结束。 *)
```

习题 16. 请在 Coq 中证明“或”能保持逻辑等价性。

```
Theorem or_congr: forall P1 Q1 P2 Q2: Prop,
  (P1 <-> P2) ->
  (Q1 <-> Q2) ->
  (P1 \/ Q1 <-> P2 \/ Q2).
(* 请在此处填入你的证明，以_[Qed]_结束。 *)
```

习题 17. 请在 Coq 中证明“或”能保持逻辑等价性。

```
Theorem imply_congr: forall P1 Q1 P2 Q2: Prop,
  (P1 <-> P2) ->
  (Q1 <-> Q2) ->
  ((P1 -> Q1) <-> (P2 -> Q2)).
(* 请在此处填入你的证明，以_[Qed]_结束。 *)
```

习题 18. 请在 Coq 中证明“并且”与“如果-那么”之间的关系。提示：必要时可以使用`assert`指令证明辅助性质。

```
Theorem and_imply: forall P Q R: Prop,
  (P /\ Q -> R) <-> (P -> Q -> R).
(* 请在此处填入你的证明，以_[Qed]_结束。 *)
```

习题 19. 请在 Coq 中证明“或”与“如果-那么”之间的关系。提示：必要时可以使用`assert`指令证明辅助性质。

```
Theorem or_imply: forall P Q R: Prop,
  (P \/ Q -> R) <-> (P -> R) /\ (Q -> R).
(* 请在此处填入你的证明，以_[Qed]_结束。 *)
```

本章之后的内容中将介绍关于“任意”、“存在”与“非”的证明方式，在相关逻辑命题的证明过程中，涉及命题逻辑的部分将会灵活使用上面介绍的各种证明方式，包括`tauto`指令。习题中也不再限制使用`tauto`指令。

5 关于“存在”的证明

当待证明结论形为：“存在一个`x`使得...”，那么可以用`exists`指明究竟哪个`x`使得该性质成立。

```
Lemma four_is_even : exists n, 4 = n + n.
Proof.
  exists 2.
  lia.
Qed.
```

习题 20.

```
Lemma six_is_not_prime: exists n, 2 <= n < 6 /\ exists q, n * q = 6.
(* 请在此处填入你的证明，以_[Qed]_结束。 *)
```

当某前提形为：存在一个`x`使得...，那么可以使用 Coq 中的`destruct`指令进行证明。这一证明指令相当于数学证明中的：任意给定一个这样的`x`。

```
Theorem dist_exists_and : forall (X: Type) (P Q: X -> Prop),
  (exists x, P x /\ Q x) -> (exists x, P x) /\ (exists x, Q x).
Proof.
  intros.
  destruct H as [x [HP HQ]].
  split.
  + exists x.
    apply HP.
  + exists x.
    apply HQ.
Qed.
```

从上面证明可以看出，Coq 中可以使用引入模式将一个存在性的命题拆分，并且与其他引入模式嵌套使用。

习题 21.

请在 Coq 中证明下面性质：

```
Theorem exists_exists : forall (X Y: Type) (P: X -> Y -> Prop),
  (exists x y, P x y) <-> (exists y x, P x y).
(* 请在此处填入你的证明，以_[Qed]_结束。 *)
```

6 关于“任意”的证明

在逻辑中，与“存在”相对偶的量词是“任意”，即 Coq 中的`forall`。其实我们已经在 Coq 中证明了许多关于`forall`的命题，最常见的证明方法就是使用`pose proof`指令。下面是一个简单的例子。

```
Example forall_ex1: forall (X: Type) (P Q R: X -> Prop),
  (forall x: X, P x -> Q x -> R x) ->
  (forall x: X, P x /\ Q x -> R x).
Proof.
  intros X P Q R H x [HP HQ].
  pose proof H x HP HQ.
  apply H0.
Qed.
```

有时在证明中，我们不需要反复使用一个概称的前提，而只需要使用它的一个特例，此时如果能在 pose proof 指令后删去原命题，能够使得证明目标更加简洁。这可以使用 Coq 中的 specialize 指令实现。

```
Example forall_ex2: forall (X: Type) (P Q R: X -> Prop),
  (forall x: X, P x /\ Q x -> R x) ->
  (forall x: X, P x -> Q x -> R x).
Proof.
  intros.
  specialize (H x ltac:(tauto)).
  apply H.
Qed.
```

在上面的证明中，specialize 指令并没有生成新的前提，而是把原有的前提`H`改为了特化后的`R x`，换言之，原有的概称命题被删去了。在 Coq 证明中，我们可以灵活使用 Coq 提供的自动化证明指令，例如，在下面证明中，我们使用`tauto`指令大大简化了证明。

```
Theorem forall_and: forall (A: Type) (P Q: A -> Prop),
  (forall a: A, P a /\ Q a) <-> (forall a: A, P a) /\ (forall a: A, Q a).
Proof.
  intros.
```

注意，此处不能使用`tauto`直接完成证明。

```
  split.
  + intros.
    split.
    - intros a.
      specialize (H a).
```

此时可以用`tauto`完成剩余证明了，另外两个分支也是类似。

```
      tauto.
    - intros a.
      specialize (H a).
      tauto.
  + intros.
    destruct H.
    specialize (H a).
    specialize (H0 a).
    tauto.
Qed.
```

习题 22. 请在 Coq 中证明下面性质：

```
Theorem forall_forall : forall (X Y: Type) (P: X -> Y -> Prop),
  (forall x y, P x y) -> (forall y x, P x y).
(* 请在此处填入你的证明，以_[Qed]_结束。 *)
```

习题 23. 请在 Coq 中证明下面性质：

```
Theorem forall_iff : forall (X: Type) (P Q: X -> Prop),
  (forall x: X, P x <-> Q x) ->
  ((forall x: X, P x) <-> (forall x: X, Q x)).
(* 请在此处填入你的证明，以_[Qed]_结束。 *)
```

7 关于“非”的证明

“非”是一个命题逻辑连接词，它符合两条重要性质：排中律与矛盾律。排中律说的是，对于任意一个命题 P，P 与非 P 中必有至少有一个为真。在 Coq 标准库中排中律称为`classic`，下面例子展示了在 Coq 应用排中律的方法。

```
Example not_ex1: forall n m: Z, n < m \/ ~ n < m.
Proof.
  intros.
  pose proof classic (n < m).
  apply H.
Qed.
```

矛盾律说的是，对于任意命题 P，P 与非 P 不能都为真。在 Coq 中，如果能从前提中同时推导出 P 与非 P 就意味着导出了矛盾。这一般可以用`tauto`完成证明。

```
Example not_ex2: forall P Q: Prop,
  P -> ~ P -> Q.
Proof.
  intros.
  tauto.
Qed.
```

下面是一些关于“非”的重要性质，它们中的一部分可以直接使用`tauto`证明。

```
Theorem not_and_iff: forall P Q: Prop,
  ~ (P /\ Q) <-> ~ P \/ ~ Q.
Proof. intros. tauto. Qed.
```

```
Theorem not_or_iff: forall P Q: Prop,
  ~ (P \/ Q) <-> ~ P /\ ~ Q.
Proof. intros. tauto. Qed.
```

```
Theorem not_imply_iff: forall P Q: Prop,
  ~ (P -> Q) <-> P /\ ~ Q.
Proof. intros. tauto. Qed.
```

```
Theorem double_negation_iff: forall P: Prop,
  ~ ~ P <-> P.
Proof. intros. tauto. Qed.
```

在证明“非”与量词（`exists`，`forall`）的关系时，有时可能需要使用排中律。下面证明的是，如果不存在一个`x`使得`P x`成立，那么对于每一个`x`而言，都有`~ P x`成立。证明中，我们根据排中律，对于每一个特定的`x`进行分类讨论，究竟是`P x`成立还是`~ P x`成立。如果是后者，那么我们已经完成了证明。如果是前者，则可以推出与前提（不存在一个`x`使得`P x`成立）的矛盾。

```
Theorem not_exists: forall (X: Type) (P: X -> Prop),
  ~ (exists x: X, P x) -> (forall x: X, ~ P x).
Proof.
  intros.
  pose proof classic (P x) as [? | ?].
  + assert (exists x: X, P x). {
      exists x.
      apply H0.
    }
    tauto.
  + apply H0.
Qed.
```

下面再证明，如果并非每一个`x`都满足`P x`，那么就存在一个`x`使得`~ P x`成立。我们还是选择利用排中律进行证明。

```
Theorem not_forall: forall (X: Type) (P: X -> Prop),
  ~ (forall x: X, P x) -> (exists x: X, ~ P x).
Proof.
  intros.
  pose proof classic (exists x: X, ~ P x) as [? | ?].
  + tauto.
  + pose proof not_exists _ _ H0.
    assert (forall x: X, P x <-> ~ ~ P x). {
      intros.
      tauto.
    }
    pose proof forall_iff _ P (fun x => ~ ~ P x) H2.
    tauto.
Qed.
```

利用前面的结论，我们还可以证明`not_all`的带约束版本。

```
Corollary not_forall_imply: forall (X: Type) (P Q: X -> Prop),
  ~ (forall x: X, P x -> Q x) -> (exists x: X, P x /\ ~ Q x).
Proof.
  intros.
  pose proof not_forall _ _ H.
  destruct H0 as [x H0].
  exists x.
  pose proof not_imply_iff (P x) (Q x).
  tauto.
Qed.
```

指称语义

1 简单表达式的指称语义

指称语义是一种定义程序行为的方式。在极简的 SimpleWhile 语言中，整数类型表达式中只有整数常量、变量、加法、减法与乘法运算。

```
EI :: = N | V | EI + EI | EI - EI | EI * EI
```

我们约定其中整数变量的值、整数运算的结果都是没有范围限制的。基于这一约定，我们可以如下定义表达式`e`在程序状态`st`上的值。

首先定义程序状态集合：

$$\text{state} \triangleq \text{var\_name} \to \mathbb{Z}$$

```
Definition state: Type := var_name -> Z.
```

- $[\![n]\!](s) = n$
- $[\![x]\!](s) = s(x)$
- $[\![e_1 + e_2]\!](s) = [\![e_1]\!](s) + [\![e_2]\!](s)$
- $[\![e_1 - e_2]\!](s) = [\![e_1]\!](s) - [\![e_2]\!](s)$
- $[\![e_1 * e_2]\!](s) = [\![e_1]\!](s) * [\![e_2]\!](s)$

其中 $s \in \text{state}$。

下面使用 Coq 递归函数定义整数类型表达式的行为。

```
Fixpoint eval_expr_int (e: expr_int) (s: state) : Z :=
  match e with
  | EConst n => n
  | EVar X => s X
  | EAdd e1 e2 => eval_expr_int e1 s + eval_expr_int e2 s
  | ESub e1 e2 => eval_expr_int e1 s - eval_expr_int e2 s
  | EMul e1 e2 => eval_expr_int e1 s * eval_expr_int e2 s
  end.
```

下面是两个具体的例子。

```
Example eval_example1: forall (s: state),
  s "x" = 1 ->
  s "y" = 2 ->
  eval_expr_int [["x" + "y"]] s = 3.
Proof. intros. simpl. rewrite H, H0. reflexivity. Qed.
```

```
Example eval_example2: forall (s: state),
  s "x" = 1 ->
  s "y" = 2 ->
  eval_expr_int [["x" * "y" + 1]] s = 3.
Proof. intros. simpl. rewrite H, H0. reflexivity. Qed.
```

2 行为等价

基于整数类型表达式的语义定义`eval_expr_int`，我们可以定义整数类型表达式之间的行为等价（亦称语义等价）：两个表达式`e1`与`e2`是等价的当且仅当它们在任何程序状态上的求值结果都相同。

$$e_1 \equiv e_2 \quad \text{iff.} \quad \forall s.\ [\![e_1]\!]\,(s) = [\![e_2]\!]\,(s)$$

```
Definition expr_int_equiv (e1 e2: expr_int): Prop :=
  forall st, eval_expr_int e1 st = eval_expr_int e2 st.
```

```
Notation "e1 '~=~' e2" := (expr_int_equiv e1 e2)
  (at level 69, no associativity).
```

下面是一些表达式语义等价的例子。

Example 1. $x * 2 \equiv x + x$

证明. 对于任意程序状态 s, $[\![x * 2]\!]\,(s) = s(x) * 2 = s(x) + s(x) = [\![x + x]\!]\,(s)$。 □

```
Example expr_int_equiv_sample:
  [["x" + "x"]] ~=~ [["x" * 2]].
```

```
Proof.
  intros.
  unfold expr_int_equiv.
```

上面的`unfold`指令表示展开一项定义，一般用于非递归的定义。

```
  intros.
  simpl.
  lia.
Qed.

Lemma zero_plus_equiv: forall (a: expr_int),
  [[0 + a]] ~=~ a.
(* 证明详见Coq源代码。 *)

Lemma plus_zero_equiv: forall (a: expr_int),
  [[a + 0]] ~=~ a.
(* 证明详见Coq源代码。 *)

Lemma minus_zero_equiv: forall (a: expr_int),
  [[a - 0]] ~=~ a.
(* 证明详见Coq源代码。 *)

Lemma zero_mult_equiv: forall (a: expr_int),
  [[0 * a]] ~=~ 0.
(* 证明详见Coq源代码。 *)

Lemma mult_zero_equiv: forall (a: expr_int),
  [[a * 0]] ~=~ 0.
(* 证明详见Coq源代码。 *)

Lemma const_plus_const: forall n m: Z,
  [[EConst n + EConst m]] ~=~ EConst (n + m).
(* 证明详见Coq源代码。 *)

Lemma const_minus_const: forall n m: Z,
  [[EConst n - EConst m]] ~=~ EConst (n - m).
(* 证明详见Coq源代码。 *)

Lemma const_mult_const: forall n m: Z,
  [[EConst n * EConst m]] ~=~ EConst (n * m).
(* 证明详见Coq源代码。 *)
```

下面定义一种简单的语法变换—常量折叠—并证明其保持语义等价性。所谓常量折叠指的是将只包含常量而不包含变量的表达式替换成为这个表达式的值。

```
Fixpoint fold_constants (e : expr_int) : expr_int :=
  match e with
  | EConst n    => EConst n
  | EVar x      => EVar x
  | EAdd e1 e2  =>
      match fold_constants e1, fold_constants e2 with
      | EConst n1, EConst n2 => EConst (n1 + n2)
      | _, _ => EAdd (fold_constants e1) (fold_constants e2)
      end
  | ESub e1 e2 =>
      match fold_constants e1, fold_constants e2 with
      | EConst n1, EConst n2 => EConst (n1 - n2)
      | _, _ => ESub (fold_constants e1) (fold_constants e2)
    end
  | EMul e1 e2 =>
      match fold_constants e1, fold_constants e2 with
      | EConst n1, EConst n2 => EConst (n1 * n2)
      | _, _ => EMul (fold_constants e1) (fold_constants e2)
    end
  end.
```

这里我们可以看到，Coq 中`match`的使用是非常灵活的。(1) 我们不仅可以对一个变量的值做分类讨论，还可以对一个复杂的 Coq 式子的取值做分类讨论；(2) 我们可以对多个值同时做分类讨论；(3) 我们可以用下划线表示`match`的缺省情况。下面是两个例子：

```
Example fold_constants_ex1:
    fold_constants [[(1 + 2) * "k"]] = [[3 * "k"]].
Proof. intros. reflexivity. Qed.
```

注意，根据我们的定义，`fold_constants`并不会将`0 + "y"`中的`0`消去。

```
Example fold_expr_int_ex2 :
  fold_constants [["x" - ((0 * 6) + "y")]] = [["x" - (0 + "y")]].
Proof. intros. reflexivity. Qed.
```

下面我们在 Coq 中证明，`fold_constants`保持表达式行为不变。

```
Theorem fold_constants_sound : forall a,
  fold_constants a ~=~ a.
Proof.
  unfold expr_int_equiv. intros.
  induction a.
```

常量的情况

```
  + simpl.
    reflexivity.
```

变量的情况

```
  + simpl.
    reflexivity.
```

加号的情况

```
  + simpl.
    destruct (fold_constants a1), (fold_constants a2);
    rewrite <- IHa1, <- IHa2;
    reflexivity.
```

减号的情况

```
  + simpl.
    destruct (fold_constants a1), (fold_constants a2);
    rewrite <- IHa1, <- IHa2;
    reflexivity.
```

乘号的情况

```
  + simpl.
    destruct (fold_constants a1), (fold_constants a2);
    rewrite <- IHa1, <- IHa2;
    reflexivity.
Qed.
```

3 利用高阶函数定义指称语义

```
Definition add_sem (D1 D2: state -> Z) (s: state): Z :=
  D1 s + D2 s.
```

```
Definition sub_sem (D1 D2: state -> Z) (s: state): Z :=
  D1 s - D2 s.
```

```
Definition mul_sem (D1 D2: state -> Z) (s: state): Z :=
  D1 s * D2 s.
```

下面是用于类型查询的`Check`指令。

```
Check add_sem.
```

可以看到`add_sem`的类型是`(state -> Z) -> (state -> Z) -> state -> Z`，这既可以被看做一个三元函数，也可以被看做一个二元函数，即函数之间的二元函数。

基于上面高阶函数，可以重新定义表达式的指称语义。

```
Definition const_sem (n: Z) (s: state): Z := n.
Definition var_sem (X: var_name) (s: state): Z := s X.
```

```
Fixpoint eval_expr_int (e: expr_int): state -> Z :=
  match e with
  | EConst n =>
      const_sem n
  | EVar X =>
      var_sem X
  | EAdd e1 e2 =>
      add_sem (eval_expr_int e1) (eval_expr_int e2)
  | ESub e1 e2 =>
      sub_sem (eval_expr_int e1) (eval_expr_int e2)
  | EMul e1 e2 =>
      mul_sem (eval_expr_int e1) (eval_expr_int e2)
  end.
```

4 布尔表达式语义

对于任意布尔表达式 e，我们规定它的语义 $[\![e]\!]$ 是一个程序状态到真值的函数，表示表达式 e 在各个程序状态上的求值结果。

- $[\![\mathrm{TRUE}]\!](s) = \mathbf{T}$
- $[\![\mathrm{FALSE}]\!](s) = \mathbf{F}$
- $[\![e_1 < e_2]\!](s)$ 为真当且仅当 $[\![e_1]\!](s) < [\![e_2]\!](s)$
- $[\![e_1 \&\& e_2]\!](s) = [\![e_1]\!](s)$ and $[\![e_2]\!](s)$
- $[\![!e_1]\!](s) = \mathrm{not}\ [\![e_1]\!](s)$

在 Coq 中可以如下定义：

```
Definition true_sem (s: state): bool := true.
```

```
Definition false_sem (s: state): bool := false.
```

```
Definition lt_sem (D1 D2: state -> Z) s: bool :=
  Z.ltb (D1 s) (D2 s).
```

```
Definition and_sem (D1 D2: state -> bool) s: bool :=
  andb (D1 s) (D2 s).
```

```
Definition not_sem (D: state -> bool) s: bool :=
  negb (D s).
```

```
Fixpoint eval_expr_bool (e: expr_bool): state -> bool :=
  match e with
  | ETrue =>
      true_sem
  | EFalse =>
      false_sem
  | ELt e1 e2 =>
      lt_sem (eval_expr_int e1) (eval_expr_int e2)
  | EAnd e1 e2 =>
      and_sem (eval_expr_bool e1) (eval_expr_bool e2)
  | ENot e1 =>
      not_sem (eval_expr_bool e1)
  end.
```

5 程序语句的语义

$(s_1, s_2) \in [\![c]\!]$ 当且仅当从 s_1 状态开始执行程序 c 会以程序状态 s_2 终止。

5.1 赋值语句的语义

$$[\![x = e]\!] = \{(s_1, s_2) \mid s_2(x) = [\![e]\!](s_1), \text{for any } y \in \text{var\_name},\ \text{if } x \neq y,\ s_1(y) = s_2(y)\}$$

5.2 空语句的语义

$$[\![\text{SKIP}]\!] = \{(s_1, s_2) \mid s_1 = s_2\}$$

5.3 顺序执行语句的语义

$$[\![c_1; c_2]\!] = [\![c_1]\!] \circ [\![c_2]\!] = \{(s_1, s_3) \mid (s_1, s_2) \in [\![c_1]\!], (s_2, s_3) \in [\![c_2]\!]\}$$

5.4 条件分支语句的语义

$$\begin{aligned}[\![\text{if } (e) \text{ then}\{c_1\} \text{ else } \{c_2\}]\!] = \quad & \left(\{(s_1, s_2) \mid [\![e]\!](s_1) = \mathbf{T}\} \cap [\![c_1]\!]\right) \cup \\ & \left(\{(s_1, s_2) \mid [\![e]\!](s_1) = \mathbf{F}\} \cap [\![c_2]\!]\right)\end{aligned}$$

这又可以改写为：

$[\![\text{if } (e) \text{ then}\{c_1\} \text{ else } \{c_2\}]\!] = \text{test\_true}([\![e]\!]) \circ [\![c_1]\!] \cup \text{test\_false}([\![e]\!]) \circ [\![c_2]\!]$

其中，

$$\text{test\_true}(F) = \{(s_1, s_2) \mid F(s_1) = \mathbf{T}, s_1 = s_2\}$$

$$\text{test\_false}(F) = \{(s_1, s_2) \mid F(s_1) = \mathbf{F}, s_1 = s_2\}$$

5.5 循环语句的语义

定义方式一：

$\text{iterLB}_0(X, R) = \text{test\_false}(X)$

$\text{iterLB}_{n+1}(X, R) = \text{test\_true}(X) \circ R \circ \text{iterLB}_n(X, R)$

$[\![\text{while } (e) \text{ do } \{c\}]\!] = \bigcup_{n\in\mathbb{N}} \text{iterLB}_n([\![e]\!], [\![c]\!])$

定义方式二：

$\text{boundedLB}_0(X, R) = \varnothing$

$\text{boundedLB}_{n+1}(X, R) = \text{test\_true}(X) \circ R \circ \text{boundedLB}_n(X, R) \cup \text{test\_false}(X)$

$[\![\text{while } (e) \text{ do } \{c\}]\!] = \bigcup_{n\in\mathbb{N}} \text{boundedLB}_n([\![e]\!], [\![c]\!])$

6 Coq 中的集合与关系

在 Coq 中往往使用 `X: A -> Prop` 来表示某类型 `A` 中元素构成的集合 `X`。字面上看，这里的 `A -> Prop` 表示 `X` 是一个从 `A` 中元素到命题的映射，这也相当于说 `X` 是一个关于 `A` 中元素性质。对于每个 `A` 中元素 `a` 而言，`a` 符合该性质 `X` 等价于 `a` 对应的命题 `X a` 为真，又等价于 `a` 是集合 `X` 的元素，在 SetsClass 库中也直接写作 `a ∈ X`。

类似的，`R: A -> B -> Prop` 也用来表示 `A` 与 `B` 中元素之间的二元关系。

本课程提供的 SetsClass 库中提供了有关集合的一系列定义。例如：

- 空集：用 `∅` 或者一堆方括号表示，定义为 `Sets.empty`；
- 单元集：用一对方括号表示，定义为 `Sets.singleton`；
- 并集：用 `∪` 表示，定义为 `Sets.union`；
- 交集：用 `∩` 表示，定义为 `Sets.intersect`；
- 一列集合的并：用 `⋃` 表示，定义为 `Sets.indexed_union`；
- 一列集合的交：用 `⋂` 表示，定义为 `Sets.indexed_intersect`；
- 集合相等：用 `==` 表示，定义为 `Sets.equiv`；
- 元素与集合关系：用 `∈` 表示，定义为 `Sets.In`；
- 子集关系：用 `⊆` 表示，定义为 `Sets.included`；
- 二元关系的连接：用 `∘` 表示，定义为 `Rels.concat`；
- 等同关系：定义为 `Rels.id`；
- 测试关系：定义为 `Rels.test`。

在 CoqIDE 中，你可以利用 CoqIDE 对于 unicode 的支持打出特殊字符：

- 首先，在打出特殊字符的 latex 表示法；
- 再按 shift+ 空格键；
- latex 表示法就自动转化为了相应的特殊字符。

例如，如果你需要打出符号 `∈`，请先在输入框中输入 `\in`，当光标紧跟在 `n` 这个字符之后的时候，按 shift+ 空格键即可。例如，下面是两个关于集合的命题：

```
Check forall A (X: A -> Prop), X ∪ ∅ == X.

Check forall A B (X Y: A -> B -> Prop), X ∪ Y ⊆ X.
```

由于集合以及集合间的运算是基于 Coq 中的命题进行定义的，集合相关性质的证明也可以规约为与命题有关的逻辑证明。例如，我们想要证明，交集运算具有交换律：

```
Lemma Sets_intersect_comm: forall A (X Y: A -> Prop),
  X ∩ Y == Y ∩ X.
Proof.
  intros.
```

下面一条命令`Sets_unfold`是 SetsClass 库提供的自动证明指令，它可以将有关集合的性质转化为有关命题的性质。

```
  Sets_unfold.
```

原本要证明的关于交集的性质现在就转化为了：`forall a : A, a ∈ X /\ a ∈ Y <-> a ∈ Y /\ a ∈ X` 这个关于逻辑的命题在 Coq 中是容易证明的。

```
  intros.
  tauto.
Qed.
```

下面是一条关于并集运算的性质。

```
Lemma Sets_included_union1: forall A (X Y: A -> Prop),
  X ⊆ X ∪ Y.
Proof.
  intros.
  Sets_unfold.
```

经过转化，要证明的结论是：`forall a : A, a ∈ X -> a ∈ X \/ a ∈ Y`。

```
  intros.
  tauto.
Qed.
```

习题 1.

下面是一条关于二元关系复合的性质。转化后得到的命题要复杂一些，请在 Coq 中证明这个关于逻辑的命题。

```
Lemma Rels_concat_assoc: forall A (X Y Z: A -> A -> Prop),
  (X ∘ Y) ∘ Z == X ∘ (Y ∘ Z).
Proof.
  intros.
  Sets_unfold.
(* 请在此处填入你的证明，以_[Qed]_结束。 *)
```

SetsClass 库中提供了一系列有关集合运算的性质的证明。未来大家在证明中既可以使用`Sets_unfold`将关于集合运算的命题转化为关于逻辑的命题，也可以直接使用下面这些性质完成证明。

```
Sets_equiv_Sets_included:
  forall x y, x == y <-> x ⊆ y /\ y ⊆ x;
Sets_empty_included:
  forall x, ∅ ⊆ x;
Sets_included_full:
  forall x, x ⊆ Sets.full;
Sets_intersect_included1:
  forall x y, x ∩ y ⊆ x;
Sets_intersect_included2:
  forall x y, x ∩ y ⊆ y;
Sets_included_intersect:
  forall x y z, x ⊆ y -> x ⊆ z -> x ⊆ y ∩ z;
Sets_included_union1:
  forall x y, x ⊆ x ∪ y;
Sets_included_union2:
  forall x y, y ⊆ x ∪ y;
Sets_union_included_strong2:
  forall x y z u,
     x ∩ u ⊆ z -> y ∩ u ⊆ z -> (x ∪ y) ∩ u ⊆ z;
```

```
Sets_included_indexed_union:
  forall xs n, xs n ⊆ ⋃ xs;
Sets_indexed_union_included:
  forall xs y, (forall n, xs n ⊆ y) -> ⋃ xs ⊆ y;
Sets_indexed_intersect_included:
  forall xs n, ⋂ xs ⊆ xs n;
Sets_included_indexed_intersect:
  forall xs y, (forall n : nat, y ⊆ xs n) -> y ⊆ ⋂ xs;
```

```
Rels_concat_union_distr_r:
  forall x1 x2 y,
    (x1 ∪ x2) ∘ y == (x1 ∘ y) ∪ (x2 ∘ y);
Rels_concat_union_distr_l:
  forall x y1 y2,
    x ∘ (y1 ∪ y2) == (x ∘ y1) ∪ (x ∘ y2);
Rels_concat_mono:
  forall x1 x2,
    x1 ⊆ x2 ->
    forall y1 y2,
      y1 ⊆ y2 ->
      x1 ∘ y1 ⊆ x2 ∘ y2;
Rels_concat_assoc:
  forall x y z,
    (x ∘ y) ∘ z == x ∘ (y ∘ z);
Rels_concat_id_l:
  forall x, Rels.id ∘ x == x;
Rels_concat_id_r:
  forall x, x ∘ Rels.id == x;
```

由于上面提到的集合与关系运算都能保持集合相等，也都能保持集合包含关系，因此 SetsClass 库支持其用户使用`rewrite`指令处理集合之间的相等关系与包含关系。下面是两个典型的例子

```
Fact sets_ex1:
  forall (A: Type) (R1 R2 R3 R4: A -> A -> Prop),
    R1 ∘ (R2 ∘ (R3 ∘ R4)) == ((R1 ∘ R2) ∘ R3) ∘ R4.
Proof.
  intros.
  rewrite Rels_concat_assoc.
  rewrite Rels_concat_assoc.
  reflexivity.
Qed.
```

```
Fact sets_ex2:
  forall (A: Type) (R1 R2 R3 R4: A -> A -> Prop),
    R1 ⊆ R2 ->
    R1 ∘ R3 ∪ R4 ⊆ R2 ∘ R3 ∪ R4.
Proof.
  intros.
  rewrite H.
  reflexivity.
Qed.
```

7 在 Coq 中定义程序语句的语义

下面在 Coq 中写出程序语句的指称语义。

```
Definition skip_sem: state -> state -> Prop :=
  Rels.id.
```

```
Definition asgn_sem
             (X: var_name)
             (D: state -> Z)
             (st1 st2: state): Prop :=
  st2 X = D st1 /\
  forall Y, X <> Y -> st2 Y = st1 Y.
```

```
Definition seq_sem
             (D1 D2: state -> state -> Prop):
  state -> state -> Prop :=
  D1 ∘ D2.
```

```
Definition test_true
             (D: state -> bool):
  state -> state -> Prop :=
  Rels.test (fun st => D st = true).
```

```
Definition test_false
             (D: state -> bool):
  state -> state -> Prop :=
  Rels.test (fun st => D st = false).
```

```
Definition if_sem
             (D0: state -> bool)
             (D1 D2: state -> state -> Prop):
  state -> state -> Prop :=
  (test_true D0 ∘ D1) ∪ (test_false D0 ∘ D2).
```

```
Fixpoint iterLB
           (D0: state -> bool)
           (D1: state -> state -> Prop)
           (n: nat):
  state -> state -> Prop :=
  match n with
  | O => test_false D0
  | S n0 => test_true D0 ∘ D1 ∘ iterLB D0 D1 n0
  end.
```

```
Definition while_sem
             (D0: state -> bool)
             (D1: state -> state -> Prop):
  state -> state -> Prop :=
  ⋃ (iterLB D0 D1).
```

```
Fixpoint boundedLB
           (D0: state -> bool)
           (D1: state -> state -> Prop)
           (n: nat):
  state -> state -> Prop :=
  match n with
  | O => ∅
  | S n0 =>
      (test_true D0 ∘ D1 ∘ boundedLB D0 D1 n0) ∪
      (test_false D0)
  end.
```

```
Definition while_sem
             (D0: state -> bool)
             (D1: state -> state -> Prop):
  state -> state -> Prop :=
  ⋃ (boundedLB D0 D1).
```

下面是程序语句指称语义的递归定义。

```
Fixpoint eval_com (c: com): state -> state -> Prop :=
  match c with
  | CSkip =>
      skip_sem
  | CAsgn X e =>
      asgn_sem X (eval_expr_int e)
  | CSeq c1 c2 =>
      seq_sem (eval_com c1) (eval_com c2)
  | CIf e c1 c2 =>
      if_sem (eval_expr_bool e) (eval_com c1) (eval_com c2)
  | CWhile e c1 =>
      while_sem (eval_expr_bool e) (eval_com c1)
  end.
```

基于上面定义，可以证明一些简单的程序性质。

```
Example inc_x_fact: forall s1 s2 n,
  (s1, s2) ∈ eval_com (CAsgn "x" [["x" + 1]]) ->
  s1 "x" = n ->
  s2 "x" = n + 1.
Proof.
  intros.
  simpl in H.
  unfold asgn_sem, add_sem, var_sem, const_sem in H.
  lia.
Qed.
```

更多关于程序行为的有用性质可以使用集合与关系的运算性质完成证明，`seq_skip`与`skip_seq`表明了删除顺序执行中多余的空语句不改变程序行为。

```
Lemma seq_skip: forall c,
  eval_com (CSeq c CSkip) == eval_com c.
Proof.
  intros.
  simpl.
  unfold seq_sem, skip_sem.
  apply Rels_concat_id_r.
Qed.
```

```
Lemma skip_seq: forall c,
  eval_com (CSeq CSkip c) == eval_com c.
Proof.
  intros.
  simpl.
  unfold seq_sem, skip_sem.
  apply Rels_concat_id_l.
Qed.
```

类似的，`seq_assoc`表明顺序执行的结合顺序是不影响程序行为的，因此，所有实际的编程中都不需要在程序开发的过程中额外标明顺序执行的结合方式。

```
Lemma seq_assoc: forall c1 c2 c3,
  eval_com (CSeq (CSeq c1 c2) c3) ==
  eval_com (CSeq c1 (CSeq c2 c3)).
Proof.
  intros.
  simpl.
  unfold seq_sem.
  apply Rels_concat_assoc.
Qed.
```

下面的`while_sem_congr2`说的则是：如果对循环体做行为等价变换，那么整个循环的行为也不变。

```
Lemma while_sem_congr2: forall D1 D2 D2',
  D2 == D2' ->
  while_sem D1 D2 == while_sem D1 D2'.
Proof.
  intros.
  unfold while_sem.
  apply Sets_indexed_union_congr.
  intros n.
  induction n; simpl.
  + reflexivity.
  + rewrite IHn.
    rewrite H.
    reflexivity.
Qed.
```

下面我们证明，我们先前定义的`remove_skip`变换保持程序行为不变。

```
Theorem remove_skip_sound: forall c,
  eval_com (remove_skip c) == eval_com c.
(* 证明详见Coq源代码。 *)
```

8 不动点分析

前面提到，while 循环语句的行为也可以描述为：只要循环条件成立，就先执行循环体再重新执行循环。我们可以证明，我们目前定义的程序语义符合这一性质。

- 定理：如果 $[\![\text{while } (e) \text{ do } \{c\}]\!] = \text{test\_true}([\![e]\!]) \circ [\![c]\!] \circ [\![\text{while } (e) \text{ do } \{c\}]\!] \cup \text{test\_false}([\![e]\!])$
- 证明：

$$
\begin{aligned}
& [\![\text{while } (e) \text{ do } \{c\}]\!] \\
= & \bigcup_{n\in\mathbb{N}} \text{boundedLB}_n([\![e]\!], [\![c]\!]) \\
= & \bigcup_{n\in\mathbb{N}} \text{boundedLB}_{n+1}([\![e]\!], [\![c]\!]) \\
= & \bigcup_{n\in\mathbb{N}} \Big(\text{test\_true}([\![e]\!]) \circ [\![c]\!] \circ \text{boundedLB}_n([\![e]\!], [\![c]\!]) \cup \text{test\_false}([\![e]\!])\Big) \\
= & \ \text{test\_true}([\![e]\!]) \circ [\![c]\!] \circ \bigcup_{n\in\mathbb{N}} \text{boundedLB}_n([\![e]\!], [\![c]\!]) \cup \text{test\_false}([\![e]\!]) \\
= & \ \text{test\_true}([\![e]\!]) \circ [\![c]\!] \circ [\![\text{while } (e) \text{ do } \{c\}]\!] \cup \text{test\_false}([\![e]\!])
\end{aligned}
$$

Coq 证明可以在 Coq 源代码中找到。

从这一结论可以看出，$[\![\text{while } (e) \text{ do } \{c\}]\!]$ 是下面方程的一个解：

$$X = \text{test\_true}([\![e]\!]) \circ [\![c]\!] \circ X \cup \text{test\_false}([\![e]\!])$$

数学上，如果一个函数 f 与某个取值 x 满足 $f(x) = x$，那么就称 x 是 f 的一个不动点。因此，我们也可以说 $[\![\text{while } (e) \text{ do } \{c\}]\!]$ 是下面函数的一个不动点：

$$F(X) \triangleq \text{test\_true}([\![e]\!]) \circ [\![c]\!] \circ X \cup \text{test\_false}([\![e]\!])$$

下面分析该函数不动点的唯一性。

- 第一种情况，假设 $[\![e]\!](s_0) = \mathbf{F}$，那么

$$\text{test\_true}(F) = \{(s_1, s_2) \mid F(s_1) = \mathbf{T}, s_1 = s_2\}$$

$$\text{test\_false}(F) = \{(s_1, s_2) \mid F(s_1) = \mathbf{F}, s_1 = s_2\}$$

– 对于任意程序状态 s, $(s_0, s) \in \text{test\_false}(\llbracket e \rrbracket)$ 当且仅当 $s = s_0$；

– 不存在这样的程序状态 s 满足 $(s_0, s) \in \text{test\_true}(\llbracket e \rrbracket)$；

因此，对于任意一个 F 的不动点 X 以及任意程序状态 s 都有：

$(s_0, s) \in X$ 当且仅当 $(s_0, s) \in F(X)$
当且仅当 $(s_0, s) \in \text{test\_true}(\llbracket e \rrbracket) \circ \llbracket c \rrbracket \circ X \cup \text{test\_false}(\llbracket e \rrbracket)$
当且仅当 $s = s_0$。

- 第二种情况，假设 $\llbracket e \rrbracket(s_0) = \mathbf{T}$，$(s_0, s_1) \in \llbracket c \rrbracket$ 并且 $\llbracket e \rrbracket(s_1) = \mathbf{F}$，那么

  – 不存在这样的程序状态 s 满足 $(s_0, s) \in \text{test\_false}(\llbracket e \rrbracket)$；

  – 对于任意程序状态 s, $(s_0, s) \in \text{test\_true}(\llbracket e \rrbracket)$ 当且仅当 $s = s_0$；

  – 对于任意程序状态 s, $(s_0, s) \in \llbracket c \rrbracket$ 当且仅当 $s = s_1$；

  因此，对于任意一个 F 的不动点 X 以及任意程序状态 s 都有：

  $(s_0, s) \in X$ 当且仅当 $(s_0, s) \in F(X)$
  当且仅当 $(s_0, s) \in \text{test\_true}(\llbracket e \rrbracket) \circ \llbracket c \rrbracket \circ X \cup \text{test\_false}(\llbracket e \rrbracket)$
  当且仅当 $(s_1, s) \in X$
  当且仅当 $s = s_1$

  上面的最后一步用到了前面情形一的结论。

- 依此类推，不难发现，如果从程序状态 s_0 出发执行循环，并且在执行完 n 次循环体后到达 s_n 状态并离开循环，那么对于 F 的任意一个不动点 X 以及任意一个程序状态 s 都有：$(s_0, s) \in X$ 当且仅当 $s = s_n$。

- 不动点不唯一的例子

  – `while (true) do { skip }`

  – 循环体的语义：$\llbracket c \rrbracket = \text{id}$；

  – 循环条件恒为真：$\text{test\_true}(\llbracket e \rrbracket) = \text{id}$；

  – 相应的：$\text{test\_false}(\llbracket e \rrbracket) = \varnothing$；

  – 因此：

  $$F(X) \triangleq \text{test\_true}(\llbracket e \rrbracket) \circ \llbracket c \rrbracket \circ X \cup \text{test\_false}(\llbracket e \rrbracket) = X$$

  – 故，任意程序状态间的二元关系 X 都是 F 的不动点。

- 假设 s_0, s_1, s_2... 是一列无穷长的程序状态，满足：$\llbracket e \rrbracket(s_i) = \mathbf{T}$ 并且 $(s_i, s_{i+1}) \in \llbracket c \rrbracket$，那么一个不动点 X 可能不包含任何一个形如 (s_i, s) 的程序状态有序对，也可能存在一个程序状态 s，使得某一个不动点 X 包含所有的 (s_0, s), (s_1, s), ... 这是因为：

  $(s_i, s) \in X$ 当且仅当 $(s_i, s) \in F(X)$，
  当且仅当 $(s_i, s) \in \text{test\_true}(\llbracket e \rrbracket) \circ \llbracket c \rrbracket \circ X \cup \text{test\_false}(\llbracket e \rrbracket)$
  当且仅当 $(s_{i+1}, s) \in X$。

因此，不动点是不唯一的，而我们需要找的是在集合包含意义下，最小的不动点，也就是说：

- $\llbracket \text{while } (e) \text{ do } \{c\} \rrbracket$ 可以被定义为下述函数的最小不动点：

$$F(X) \triangleq \text{test\_true}(\llbracket e \rrbracket) \circ \llbracket c \rrbracket \circ X \cup \text{test\_false}(\llbracket e \rrbracket)$$

9 Bourbaki-Witt 不动点定理

下面介绍 Bourbaki-Witt 不动点定理。这将统一的回答为什么有不动点，如何构造最小不动点。

- 定义：偏序集。满足下面三个条件的 $(A, \leqslant_A)$ 成为一个偏序集（partial ordering）：
  - 自反性：对于任意 $a \in A$， $a \leqslant_A a$
  - 传递性：对于任意 $a, b, c \in A$，如果 $a \leqslant_A b$、$b \leqslant_A c$，那么 $a \leqslant_A c$
  - 反对称性：对于任意 $a, b \in A$，如果 $a \leqslant_A b$、$b \leqslant_A a$，那么 $a = b$
- 例子：
  - $(\mathbb{R}, \leqslant)$ 是一个偏序集。
  - 如果 D 表示自然数之间的整除关系，即 $(a, b) \in D$ 当且仅当 $a \mid b$，那么 $(\mathbb{N}, D)$ 是一个偏序集。值得一提的是，在这个偏序关系下，两个自然数之间不一定可以相互比较；例如 $2 \nmid 3$ 并且 $3 \nmid 2$。
  - 如果 X 是一个集合，$\mathcal{P}(X)$ 表示 X 的幂集，那么 $(\mathcal{P}(X), \subseteq)$ 构成一个偏序集。
- 定义：完备偏序集。如果偏序集 $(A, \leqslant_A)$ 还满足下面性质，那么它是一个完备偏序集（complete partial ordering，CPO）：
  - 完备性：对于任意 $S \subseteq A$，如果 S 中任意两个元素之间都可以大小比较，那么 S 有上确界（least upper bound, lub），记做 $\mathrm{lub}(S)$，即：(1) 对于任意 $a \in S$，$a \leqslant_A \mathrm{lub}(S)$；(2) 如果某个 $b \in A$ 使得每一个 $a \in S$ 都有 $a \leqslant_A b$，那么 $\mathrm{lub}(S) \leqslant_A b$。
  - 注：符合上述性质的 S 称为偏序集 A 上的一条链。
- 例子：
  - $(\mathbb{R}, \leqslant)$ 是偏序集但是不是完备偏序集，因为 $\mathbb{Z} \subseteq \mathbb{R}$ 是一条链，但是它没有上确界。
  - 如果 X 是一个集合，$\mathcal{P}(X)$ 表示 X 的幂集，那么 $(\mathcal{P}(X), \subseteq)$ 是一个完备偏序集。其中，对于任意 $U \subseteq \mathcal{P}(X)$，都有 $\mathrm{lub}(U) = \bigcup_{V \in U} V$ 是 U 的上确界。
  - 如果 D 表示自然数之间的整除关系，即 $(a, b) \in D$ 当且仅当 $a \mid b$，那么 $(\mathbb{N}, D)$ 是一个完备偏序集。特别的，如果 $U \subseteq \mathbb{N}$ 是一条链还是一个有穷集，那么 $\mathrm{lub}(U)$ 就是 U 的最小公倍数；如果 $U \subseteq \mathbb{N}$ 是一条链还是一个无穷集，那么 $\mathrm{lub}(U)$ 就是 0。
  - 如果 D^+ 表示正整数之间的整除关系，那么 $(\mathbb{Z}^+, D^+)$ 是一个偏序集，但不是一个完备偏序集。例如，$\{1, 2, 4, 8, ..., 2^n, ...\}$ 是整除关系上的一条链，但是它没有整除关系意义下的上确界。
- 定义：单调函数。如果 $(A, \leqslant_A)$ 是一个偏序集，那么 $F : A \to A$ 是一个单调函数当且仅当：对于任意 $a, b \in A$，如果 $a \leqslant_A b$，那么 $F(a) \leqslant_A F(b)$。
- 引理：如果 S 是偏序集 $(A, \leqslant_A)$ 上的一条链，$F : A \to A$ 是一个单调函数，那么 $F(S) \triangleq \{F(a) \mid a \in S\}$ 也是一条链。
- 证明：任给 $a, b \in S$，要么 $a \leqslant_A b$，要么 $b \leqslant_A a$。若前者成立，那么 $F(a) \leqslant_A F(b)$；若后者成立，那么 $F(b) \leqslant_A F(a)$，因此 $F(S)$ 中的元素间两两可以比较大小。
- 定义：单调连续函数。如果 $(A, \leqslant_A)$ 是一个完备偏序集，那么单调函数 $F : A \to A$ 是连续的当且仅当：对于任意一条非空链 S，$F(\mathrm{lub}(S)) = \mathrm{lub}(F(S))$。
- 引理：如果 $(A, \leqslant_A)$ 是一个完备偏序集，那么这个集合上有最小元，记做 $\bot$。

- 证明：空集 $\varnothing$ 是 $(A, \leqslant_A)$ 上的一条链。而任何一个 A 中元素，都是空集的上界。因此，对于任意 $a \in A$ 都有，$\text{lub}(\varnothing) \leqslant_A a$。

- 引理：如果 F 是完备偏序集 $(A, \leqslant_A)$ 上的单调连续函数，那么 $\{\bot, F(\bot), F(F(\bot)), ...\}$ 是 $(A, \leqslant_A)$ 上的一条链。

- 证明：

  由于 $\bot$ 是最小元，所以 $\bot \leqslant_A F(\bot)$。由于 F 是单调函数，所以 $F(\bot) \leqslant_A F(F(\bot))$。依次类推，$\bot \leqslant_A F(\bot) \leqslant_A F(F(\bot)) \leqslant_A ...$

- 定理：如果 F 是完备偏序集 $(A, \leqslant_A)$ 上的单调连续函数，$\text{lub}(\bot, F(\bot), F(F(\bot)), ...)$ 是 F 的一个不动点。

- 证明：

$$\begin{aligned} & F(\text{lub}(\bot, F(\bot), F(F(\bot)), ...)) \\ = \quad & \text{lub}(F(\bot), F(F(\bot)), F(F(F(\bot))), ...) \\ = \quad & \text{lub}(\bot, F(\bot), F(F(\bot)), F(F(F(\bot))), ...) \end{aligned}$$

- 定理：如果 F 是完备偏序集 $(A, \leqslant_A)$ 上的单调连续函数且 $F(a) = a$，那么 $\text{lub}(\bot, F(\bot), F(F(\bot)), ...) \leqslant_A a$。

- 证明：

$$\begin{aligned} & \bot \leqslant_A a \\ & F(\bot) \leqslant_A F(a) = a \\ & F(F(\bot)) \leqslant_A F(a) = a \\ & ... \end{aligned}$$

用 Bourbaki-Witt 不动点定义 while 语句语义

- $(\mathcal{P}(\text{state} \times \text{state}), \subseteq)$ 是一个完备偏序集；

- $F(X) \triangleq \text{test\_true}([\![e]\!]) \circ [\![c]\!] \circ X \cup \text{test\_false}([\![e]\!])$ 是一个单调连续函数；

  - $G(X) = Y \circ X$ 是单调连续函数；
  - $H(X) = X \cup Y$ 是单调连续函数；
  - 如果 $G(X)$ 与 $H(X)$ 都是单调连续函数，那么 $G(H(X))$ 也是单调连续函数；

- F 的最小不动点是：

$$\bigcup_{n \in \mathbb{N}} \left(F^{(n)}(\varnothing)\right)$$

两种定义的对应关系 $F^{(n)}(\varnothing) = \text{boundedLB}([\![e]\!], [\![c]\!])$。

10 Coq 中证明并应用 Bourbaki-Witt 不动点定理

下面我们将在 Coq 中证明 Bourbaki-Witt 不动点定理。在 Bourbaki-Witt 不动点定理中，我们需要证明满足某些特定条件（例如偏序、完备偏序等）的二元关系的一些性质。在 Coq 中，我们当然可以通过 `R: A -> A -> Prop` 来探讨二元关系 `R` 的性质。然而 Coq 中并不能给这样的变量设定 Notation 符号，例如，我们希望用 `a <= b` 来表示 `R a b`，因此我们选择使用 Coq 的 `Class` 来帮助我们完成定义。

下面这一定义说的是：`Order`是一类数学对象，任给一个类型`A`，`Order A`也是一个类型，这个类型的每个元素都有一个域，这个域的名称是`order_rel`，它的类型是`A -> A -> Prop`，即`A`上的二元关系。

```
Class Order (A: Type): Type :=
  order_rel: A -> A -> Prop.
```
A上的二元谓词。

Coq 中`Class`与`Record`有些像，但是有两点区别。第一：`Class`如果只有一个域，它的中可以不使用大括号将这个域的定义围起来；第二：在定义或证明中，Coq 系统会试图自动搜索并填充类型为`Class`的参数，搜索范围为之前注册过可以使用的`Instance`以及当前环境中的参数。例如，我们先前在证明等价关系、congruence 性质时就使用过`Instance`。例如，下面例子中，不需要指明`order_rel`是哪个`Order A`的`order_rel`域，Coq 会自动默认这是指`RA`的`order_rel`域。

```
Check forall {A: Type} {RA: Order A} (x: A),
        exists (y: A), order_rel x y.
```

这样，我们就可以为`order_rel`定义 Notation 符号。

```
Declare Scope order_scope.
Notation "a <= b" := (order_rel a b): order_scope.
Local Open Scope order_scope.
```
定义

```
Check forall {A: Type} {RA: Order A} (x y: A),
        x <= y \/ y <= x.
```

基于序关系，我们就可以定义上界与下界的概念。由于 Bourbaki-Witt 不动点定理中主要需要探讨无穷长元素序列的上界与上确界，下面既定义了元素集合的上下界也定义了元素序列的上下界。

```
Definition is_lb
             {A: Type} {RA: Order A}
             (X: A -> Prop) (a: A): Prop :=
  forall a', X a' -> a <= a'.
```
A’是X中的元素，那么。。。。

```
Definition is_ub
             {A: Type} {RA: Order A}
             (X: A -> Prop) (a: A): Prop :=
  forall a', X a' -> a' <= a.
```

```
Definition is_omega_lb
             {A: Type} {RA: Order A}
             (l: nat -> A) (a: A): Prop :=
  forall n, a <= l n.
```
A是下界

```
Definition is_omega_ub
             {A: Type} {RA: Order A}
             (l: nat -> A) (a: A): Prop :=
  forall n, l n <= a.
```

下面定义序列的上确界，所谓上确界就是上界中最小的一个，因此它的定义包含两个子句。而在后续证明中，使用上确界性质的时候，有时需要用其第一条性质，有时需要用其第二条性质。为了后续证明的方便，这里在定义之外提供了使用这两条性质的方法：`is_omega_lub_sound`与`is_omega_lub_tight`。比起在证明

中使用`destruct`指令拆解上确界的定义，使用这两条引理，可以使得 Coq 证明更自然地表达我们的证明思路。之后我们将在证明偏序集上上确界唯一性的时候会看到相关的用法。

```
Definition is_omega_lub
             {A: Type} {RA: Order A}
             (l: nat -> A) (a: A): Prop :=
  is_omega_ub l a /\ is_lb (is_omega_ub l) a.
```

```
Lemma is_omega_lub_sound:
  forall {A: Type} {RA: Order A} {l: nat -> A} {a: A},
    is_omega_lub l a -> is_omega_ub l a.
Proof. unfold is_omega_lub; intros; tauto. Qed.
```

```
Lemma is_omega_lub_tight:
  forall {A: Type} {RA: Order A} {l: nat -> A} {a: A},
    is_omega_lub l a -> is_lb (is_omega_ub l) a.
Proof. unfold is_omega_lub; intros; tauto. Qed.
```

在编写 Coq 定义时，另有一个问题需要专门考虑，有些数学上的相等关系在 Coq 中只是一种等价关系。例如，我们之前在 Coq 中用过集合相等的定义。因此，我们描述 Bourbaki-Witt 不动点定理的前提条件时，也需要假设有一个与序关系相关的等价关系，我们用`Equiv`表示，并用`==`这个符号描述这个等价关系。

```
Class Equiv (A: Type): Type :=
  equiv: A -> A -> Prop.
```

```
Notation "a == b" := (equiv a b): order_scope.
```

基于此，我们可以定义基于等价关系的自反性与反对称性。注意，传递性的定义与这个等价关系无关。这里我们也用`Class`定义，这与 Coq 标准库中的自反、对称、传递的定义是类似的，但是也有不同：(1) 我们的定义需要探讨一个二元关系与一个等价关系之间的联系，而 Coq 标准库中只考虑了这个等价关系是普通等号的情况；(2) Coq 标准库中直接使用二元关系`R: A -> A -> Prop`作为参数，而我们的参数使用了`Order`与`Equiv`这两个`Class`。

自反性：

```
Class Reflexive_Setoid
       (A: Type) {RA: Order A} {EA: Equiv A}: Prop :=
  reflexivity_setoid:
    forall a b, a == b -> a <= b.
```

一个集合A，A上有一个序关系和等价关系

Class一般与结构的定义有关。

反对称性：

```
Class AntiSymmetric_Setoid
       (A: Type) {RA: Order A} {EA: Equiv A}: Prop :=
  antisymmetricity_setoid:
    forall a b, a <= b -> b <= a -> a == b.
```

现在，我们就可以如下定义偏序关系。

偏序关系

```
Class PartialOrder_Setoid
        (A: Type) {RA: Order A} {EA: Equiv A}: Prop :=                Class的话是会去看注册列表中是否能推导出来一些参数
{
  PO_Reflexive_Setoid:> Reflexive_Setoid A;
  PO_Transitive:> Transitive order_rel; 提示Coq，有传递性
  PO_AntiSymmetric_Setoid:> AntiSymmetric_Setoid A
}.
```

下面证明两条偏序集的基本性质。在 Coq 中，我们使用前引号 `` ` `` 让 Coq 自动填充 `Class` 类型元素的参数。例如，`` `{POA: PartialOrder_Setoid A} `` 会指引 Coq 额外填上 `RA: Order A` 和 `EA: Equiv A`。

序关系两侧做等价变换不改变序关系：

```
Instance PartialOrder_Setoid_Proper
          {A: Type} `{POA: PartialOrder_Setoid A} {EquivA: Equivalence equiv}:
  Proper (equiv ==> equiv ==> iff) order_rel.
(* 证明详见Coq源代码。 *)
```

第一个参数做equiv变换

等价关系

如果两个序列的所有上界都相同，那么他们的上确界也相同（如果有的话）：

```
Lemma same_omega_ub_same_omega_lub:
  forall
    {A: Type}
    `{POA: PartialOrder_Setoid A}
    (l1 l2: nat -> A)
    (a1 a2: A),
  (forall a, is_omega_ub l1 a <-> is_omega_ub l2 a) ->
  is_omega_lub l1 a1 ->
  is_omega_lub l2 a2 ->
  a1 == a2.
(* 证明详见Coq源代码。 *)
```

证明 Bourbaki-Witt 不动点定理时还需要定义完备偏序集，由于在其证明中实际只用到了完备偏序集有最小元和任意单调不减的元素序列有上确界，我们在 Coq 定义时也只考虑符合这两个条件的偏序集，我们称为 OmegaCPO，Omega 表示可数无穷多项的意思。另外，尽管数学上仅仅要求完备偏序集上的所有链有上确界，但是为了 Coq 证明的方便，我们将 `omega_lub` 定义为所有元素序列的上确界计算函数，只不过我们仅仅要求该函数在其参数为单调不减序列时能确实计算出上确界，见 `oCPO_completeness`。

```
Class OmegaLub (A: Type): Type :=
  omega_lub: (nat -> A) -> A.
```

```
Class Bot (A: Type): Type :=
  bot: A.
```

```
Definition increasing
            {A: Type} {RA: Order A} (l: nat -> A): Prop :=
  forall n, l n <= l (S n).  n+1
```

```
Definition is_least {A: Type} {RA: Order A} (a: A): Prop :=
  forall a', a <= a'.
```

```
Class OmegaCompletePartialOrder_Setoid
        (A: Type)
        {RA: Order A} {EA: Equiv A}
        {oLubA: OmegaLub A} {BotA: Bot A}: Prop :=
{
  oCPO_PartialOrder:> PartialOrder_Setoid A;
  oCPO_completeness: forall T,          这个函数计算上确界
    increasing T -> is_omega_lub T (omega_lub T);
  bot_is_least: is_least bot  bot确实是最小元素
}.
```

利用这里定义中的`omega_lub`函数，可以重述先前证明过的性质：两个单调不减序列如果拥有完全相同的上界，那么他们也有同样的上确界。

```
Lemma same_omega_ub_same_omega_lub':
  forall
    {A: Type}
    `{oCPOA: OmegaCompletePartialOrder_Setoid A}
    (l1 l2: nat -> A),
  (forall a, is_omega_ub l1 a <-> is_omega_ub l2 a) ->
  increasing l1 ->
  increasing l2 ->
  omega_lub l1 == omega_lub l2.
(* 证明详见Coq源代码。 *)
```

下面定义单调连续函数。

```
Definition mono
             {A B: Type}
             `{POA: PartialOrder_Setoid A}
             `{POB: PartialOrder_Setoid B}
             (f: A -> B): Prop :=
  forall a1 a2, a1 <= a2 -> f a1 <= f a2.
```

```
Definition continuous
             {A B: Type}
             `{oCPOA: OmegaCompletePartialOrder_Setoid A}
             `{oCPOB: OmegaCompletePartialOrder_Setoid B}
             (f: A -> B): Prop :=
  forall l: nat -> A,
    increasing l ->
    f (omega_lub l) == omega_lub (fun n => f (l n)).
```

下面我们可以证明：自反函数是单调连续的、复合函数能保持单调连续性。

自反函数的单调性：

```
Lemma id_mono:
  forall {A: Type}
         `{POA: PartialOrder_Setoid A},
  mono (fun x => x).
(* 证明详见Coq源代码。 *)
```

复合函数保持单调性：

```
Lemma compose_mono:
  forall {A B C: Type}
         `{POA: PartialOrder_Setoid A}
         `{POB: PartialOrder_Setoid B}
         `{POC: PartialOrder_Setoid C}
         (f: A -> B)
         (g: B -> C),
  mono f -> mono g -> mono (fun x => g (f x)).
(* 证明详见Coq源代码。 *)
```

自反函数的连续性：

```
Lemma id_continuous:
  forall {A: Type}
         `{oCPOA: OmegaCompletePartialOrder_Setoid A}
         {EquivA: Equivalence equiv},
  continuous (fun x => x).
(* 证明详见Coq源代码。 *)
```

这里，要证明单调连续函数的复合结果也是连续的要复杂一些。显然，这其中需要证明一个单调函数作用在一个单调不减序列的每一项后还会得到一个单调不减序列。下面的引理`increasing_mono_increasing`描述了这一性质。

```
Lemma increasing_mono_increasing:
  forall {A B: Type}
         `{POA: PartialOrder_Setoid A}
         `{POB: PartialOrder_Setoid B}
         (f: A -> B)
         (l: nat -> A),
  increasing l -> mono f -> increasing (fun n => f (l n)).
(* 证明详见Coq源代码。 *)
```

除此之外，我们还需要证明单调函数能保持相等关系，即，如果`f`是一个单调函数，那么`x == y`能推出`f x == f y`。当然，如果这里的等价关系就是等号描述的相等关系，那么这个性质是显然的。但是，对于一般的等价关系，这就并不显然了。这一引理的正确性依赖于偏序关系中的自反性和反对称性。

```
Lemma mono_equiv_congr:
  forall {A B: Type}
         `{POA: PartialOrder_Setoid A}
         `{POB: PartialOrder_Setoid B}
          {EquivA: Equivalence (equiv: A -> A -> Prop)}
         (f: A -> B),
  mono f -> Proper (equiv ==> equiv) f.
(* 证明详见Coq源代码。 *)
```

现在，可以利用上面两条引理证明复合函数的连续性了。

```
Lemma compose_continuous:
  forall {A B C: Type}
         `{oCPOA: OmegaCompletePartialOrder_Setoid A}
         `{oCPOB: OmegaCompletePartialOrder_Setoid B}
         `{oCPOC: OmegaCompletePartialOrder_Setoid C}
          {EquivB: Equivalence (equiv: B -> B -> Prop)}
          {EquivC: Equivalence (equiv: C -> C -> Prop)}
         (f: A -> B)
         (g: B -> C),
  mono f ->
  mono g ->
  continuous f ->
  continuous g ->
  continuous (fun x => g (f x)).
(* 证明详见Coq源代码。 *)
```

到目前为止，我们已经定义了 Omega 完备偏序集与单调连续函数。在证明 Bourbaki-Witt 不动点定理之前还需要最后一项准备工作：定理描述本身的合法性。即，我们需要证明 `bot`, `f bot`, `f (f bot)`... 这个序列的单调性。我们利用 Coq 标准库中的 `Nat.iter` 来定义这个序列，`Nat.iter n f bot` 表示将 `f` 连续 `n` 次作用在 `bot` 上。

```
Lemma iter_bot_increasing:
  forall
    {A: Type}
    `{oCPOA: OmegaCompletePartialOrder_Setoid A}
    (f: A -> A),
    mono f ->
    increasing (fun n => Nat.iter n f bot).
(* 证明详见Coq源代码。 *)
```

当然，`f bot`, `f (f bot)`, `f (f (f bot))`... 这个序列也是单调不减的。

```
Lemma iter_S_bot_increasing:
  forall
    {A: Type}
    `{oCPOA: OmegaCompletePartialOrder_Setoid A}
    (f: A -> A),
    mono f ->
    increasing (fun n => f (Nat.iter n f bot)).
(* 证明详见Coq源代码。 *)
```

`BW_LFix` 定义了 Bourbaki-Witt 最小不动点。

```
Definition BW_LFix
             {A: Type}
             `{CPOA: OmegaCompletePartialOrder_Setoid A}
             (f: A -> A): A :=
  omega_lub (fun n => Nat.iter n f bot).
```

先证明，`BW_LFix` 的计算结果确实是一个不动点。

```
Lemma BW_LFix_is_fix:
  forall
    {A: Type}
    `{CPOA: OmegaCompletePartialOrder_Setoid A}
    {EquivA: Equivalence equiv}
    (f: A -> A),
    mono f ->
    continuous f ->
    f (BW_LFix f) == BW_LFix f.
(* 证明详见Coq源代码。 *)
```

再证明，`BW_LFix`的计算结果是最小不动点。

```
Lemma BW_LFix_is_least_fix:
  forall
    {A: Type}
    `{CPOA: OmegaCompletePartialOrder_Setoid A}
    {EquivA: Equivalence equiv}
    (f: A -> A)
    (a: A),
    mono f ->
    continuous f ->
    f a == a ->
    BW_LFix f <= a.
(* 证明详见Coq源代码。 *)
```

接下去我们将利用 Bourbaki-Witt 最小不动点定义 While 语句的程序语义中运行终止的情况。

首先需要定义我们所需的 OmegaCPO。在定义`Class`类型的值时，可以使用`Intance`关键字。如果`Class`中只有一个域并且`Class`的定义没有使用大括号包围所有域，那么这个域的定义就是整个`Class`类型的值的定义；否则`Class`类型的值应当像`Record`类型的值一样定义。

```
Instance R_while_fin {A B: Type}: Order (A -> B -> Prop) :=
  Sets.included.
```

```
Instance Equiv_while_fin {A B: Type}: Equiv (A -> B -> Prop) :=
  Sets.equiv.
```

下面证明这是一个偏序关系。证明的时候需要展开上面两个二元关系（一个表示序关系另一个表示等价关系）。以序关系为例，此时需要将`R_while_fin`与`order_rel`全部展开，前者表示将上面的定义展开，后者表示将从`Class Order`取出`order_rel`域这一操作展开。其余的证明则只需用`sets_unfold`证明集合相关的性质。

```
Instance PO_while_fin {A B: Type}: PartialOrder_Setoid (A -> B -> Prop).
(* 证明详见Coq源代码。 *)
```

下面再定义上确界计算函数与完备偏序集中的最小值。

```
Instance oLub_while_fin {A B: Type}: OmegaLub (A -> B -> Prop) :=
  Sets.indexed_union.
```

```
Instance Bot_while_fin {A B: Type}: Bot (A -> B -> Prop) :=
  ∅: A -> B -> Prop.
```

下面证明这构成一个 Omega 完备偏序集。

```
Instance oCPO_while_fin {A B: Type}:
  OmegaCompletePartialOrder_Setoid (A -> B -> Prop).
(* 证明详见Coq源代码。 *)
```

额外需要补充一点：`Equiv_while_fin` 确实是一个等价关系。先前 Bourbaki-Witt 不动点定理的证明中用到了这一前提。

```
Instance Equiv_equiv_while_fin {A B: Type}:
  Equivalence (@equiv (A -> B -> Prop) _).
Proof.
  apply Sets_equiv_equiv.
Qed.
```

下面开始证明 `F(X) = (test1(D0) ∘ D ∘ while_denote D0 D) ∪ test0(D0)` 这个函数的单调性与连续性。整体证明思路是：(1) `F(X) = X` 是单调连续的；(2) 如果 `F` 是单调连续的，那么 `G(X) = Y ∘ F(X)` 也是单调连续的；(3) 如果 `F` 是单调连续的，那么 `G(X) = F(X) ∪ Y` 也是单调连续的；其中 `Y` 是给定的二元关系。

下面证明前面提到的步骤 (2)：如果 `F` 是单调连续的，那么 `G(X) = Y ∘ F(X)` 也是单调连续的。主结论为 `BinRel_concat_left_mono_and_continuous`，其用到了两条下面的辅助引理以及前面证明过的复合函数单调连续性定理。

```
Lemma BinRel_concat_left_mono:
  forall (A B C: Type) (Y: A -> B -> Prop),
    mono (fun X: B -> C -> Prop => Y ∘ X).
(* 证明详见Coq源代码。 *)
```

```
Lemma BinRel_concat_left_continuous:
  forall (A B C: Type) (Y: A -> B -> Prop),
    continuous (fun X: B -> C -> Prop => Y ∘ X).
(* 证明详见Coq源代码。 *)
```

```
Lemma BinRel_concat_left_mono_and_continuous:
  forall
    (A B C: Type)
    (Y: A -> B -> Prop)
    (f: (B -> C -> Prop) -> (B -> C -> Prop)),
  mono f /\ continuous f ->
  mono (fun X => Y ∘ f X) /\ continuous (fun X => Y ∘ f X).
(* 证明详见Coq源代码。 *)
```

下面证明前面提到的步骤 (3)：如果 `F` 是单调连续的，那么 `G(X) = Y ∘ F(X)` 也是单调连续的。主结论为 `union_right2_mono_and_continuous`，其用到了两条下面的辅助引理以及前面证明过的复合函数单调连续性定理。

```
Lemma union_right2_mono:
  forall (A B: Type) (Y: A -> B -> Prop),
    mono (fun X => X ∪ Y).
(* 证明详见Coq源代码。 *)
```

```
Lemma union_right2_continuous:
  forall (A B: Type) (Y: A -> B -> Prop),
    continuous (fun X => X ∪ Y).
(* 证明详见Coq源代码。 *)
```

```
Lemma union_right2_mono_and_continuous:
  forall
    (A B: Type)
    (Y: A -> B -> Prop)
    (f: (A -> B -> Prop) -> (A -> B -> Prop)),
  mono f /\ continuous f ->
  mono (fun X => f X ∪ Y) /\ continuous (fun X => f X ∪ Y).
(* 证明详见Coq源代码。 *)
```

最终我们可以用 Bourbaki-Witt 不动点定义 while 语句运行终止的情况。
首先给出语义定义。

```
Definition while_sem
             (D0: state -> bool)
             (D: state -> state -> Prop):
  state -> state -> Prop :=
  BW_LFix (fun X => (test_true(D0) ∘ D ∘ X) ∪ test_false(D0)).
```

下面可以直接使用 Bourbaki-Witt 不动点定理证明上述定义就是我们要的最小不动点。

```
Theorem while_sem_is_least_fix: forall D0 D,
  (test_true(D0) ∘ D ∘ while_sem D0 D) ∪ test_false(D0) == while_sem D0 D /\
  (forall X,
    (test_true(D0) ∘ D ∘ X) ∪ test_false(D0) == X -> while_sem D0 D ⊆ X).
(* 证明详见Coq源代码。 *)
```

:>, 前提，Instance都是class会去找的参数