讨论文档：基本方法介绍

对于一个函数，程序员会为它标注precondition和postcondition。我们需要证明的是，在precondition成立时，程序的行为保证postcondition一定被满足，并在无法证明时报错。代码中可能有assert语句，也需要证明assertion成立。此外，还需要隐含assertion来保证程序不会发生runtime error，例如除以零，数组下标越界等，它们被称为runtime assertion，也需要被证明。

如果仅证明函数能够结束执行时符合postcondition，而不证明函数一定能够结束执行，则称为partial correctness。如果还证明函数能结束，则称为total correctness。

程序执行中的一个状态包含执行的位置以及在该位置有效的各个变量的值。precondition和postcondition分别是关于进入函数时的状态以及离开函数时的状态的两条一阶逻辑公式。我们希望生成一个一阶逻辑公式，称为verification condition，它（在合适的theory里）永真，当且仅当程序的行为符合我们的验证要求。

这里说到"合适的theory"，是因为程序中会涉及到整数运算、数组操作等一阶逻辑里没有的要素。拿两个整数的加法来说，如果放到纯粹的一阶逻辑里，就只是一个function，而不保证它会被解释成什么样，这样整数运算的性质就必然无法体现。因此需要引入对应的公理系统（也就是一阶逻辑的某个theory）来保证它被解释成我们想要的样子。

为了生成VC，首先考虑一个简单的情况：以一个precondition F开头，接下来是若干条顺序执行的语句$S_1; S_2; \ldots; S_n$，最后是postcondition G。把G看成一个对结束状态的assertion，直观来说，如果能把它逐条语句"前移"，最终变成对开始状态的assertion，记为$\mathrm{wp}(G, S_1; \ldots; S_n)$，那么VC就是

$$F \to \mathrm{wp}(G, S_1; \ldots; S_n)$$

wp是weakest precondition，是一个尽量弱的条件，使得它在前面的状态成立时，能够保证在后面的状态postcondition成立。它可以通过逐条前移从G生成，过程中进行的操作与语句本身有关。

例如考虑一条赋值：`a = some_expression;`，执行前的状态是s，执行后的状态是s'，那么要把一条关于s'的assertion变成关于s的，只要把其中的`a`替换成`some_expression`就行，其中`some_expression`可能包含一些s中的值。

例子：

```
@ x >= 0
rv = x + 1;
@ rv >= 1
```

那么$\mathrm{wp}(rv \geq 1, rv = x + 1)$就得到$x + 1 \geq 1$，于是VC就是：

$$x \geq 0 \to x + 1 \geq 1$$

它在整数的公理系统$T_{\mathbb{Z}}$中是永真的。

下面来考虑分支、循环和函数调用的情形。我们需要将函数的执行分成若干条基本路径，对于每条基本路径得到一系列顺序执行的语句和pre/post condition，然后用上面的方法来验证。

遇到一条`if(cond)`时，枚举`cond`的真假，分别将控制流向下延伸。在对应的基本路径中，跨过`if(cond)`的位置加入一条`assume cond`或`assume !cond`语句。`assume`是一种特别的语句，表示提供关于执行到它时的状态的额外信息。循环则利用标注的循环不变式断开成基本路径，注意循环开始时判断循环条件也是一个分支，需要枚举循环结束和没结束两种控制流转移并添加对应的`assume`语句。函数调用则利用被调用者的标注把控制流断开成基本路径，前面要assert被调用者的precondition被满足，后面则以它的postcondition作为已知条件。

如果要证明total correctness而不仅仅是partial correctness，就要证明函数的执行是一定终止的。这时就要为循环和函数调用标注ranking function。

以循环为例，考虑一个定义了某个well-founded relation的集合S，我们需要在每个循环入口处标注ranking function，它是一个关于当时的状态的函数，得到S中的值。需要验证ranking function的值确实在S中，并且每次控制流经过标注时，ranking function是严格递减的。由于S上不存在无限长的严格下降序列，这样就可以证明不会死循环了。函数调用也类似。

下面考虑控制流两次相邻的经过ranking function标注的时机，它们会分别出现在一条基本路径的开头和结尾。

```
↓ f(s)
...
↓ g(s)
```

注意f和g可能不是同一个函数，因为这可能是不同的两处标注在控制流上相邻（比如循环嵌套）。下面考虑怎样为证明$f(s_{start}) > g(s_{end})$生成VC：

由于f和g中使用了同样的变量，但实际上对应它不同时刻的值，先把$f(s)$中的所有变量替换为别的名字，用$f(s_0)$表示，然后将$f(s_0) > g(s)$用上面定义的wp倒推回起始时刻，得到了一个关于起始时刻的s与占位符s_0的一阶逻辑公式，然后再将s_0换回s，此时涉及的所有变量都代表了它应该代表的值（即起始状态的）。

例如考虑下面的例子，ranking function取在非负整数集。

```
while
@ i >= 0
↓ i
    (i > 0)
{
    i = i - 1;
}
```

假设它前面的语句可以证明出进入循环时满足$i \geq 0$，现在考虑循环里一条比较重要的基本路径是：

```
@ i >= 0
↓ i
assume i > 0
i = i - 1;
@ i >= 0
↓ i
```

循环不变式的保持容易证明，下面来看ranking function。首先由于循环不变式，可以确定$i \in \mathbb{N}$，然后写出$i_0 > i$，经过赋值语句得到$i_0 > i - 1$，经过`assume`语句得到$(i > 0) \to (i_0 > i - 1)$，再把$i_0$换回$i$，得到$(i > 0) \to (i > i - 1)$，是永真的。