Chapter 1

Basics

In this chapter, we review some basics of programming, consider what it means for a condition to always hold at a particular program point, and see how to write simple specifications.

1.0.Methods

A *method* is a program declaration that prescribes some behavior. For instance, here is the declaration of a method named `Triple`:

```
method Triple(x: int) returns (r: int) {

var y := 2 * x;

r := x + y;

}
```

This method takes an *in-parameter* `x` of type integer and returns an *out-parameter* `r`, also of type integer. The *body* of a method, given in curly braces after the method signature, is a list of *statements* that give the method's implementation. Here, the body consists of two statements. The first statement declares a local variable `y` to which it assigns the value `2*x`. The second statement assigns the sum of the in-parameter `x` and the local variable `y` to out-parameter `r`.

> **Exercise 1.0.**
>
> If `x` is `10`, what value does the method assign to `r`?

In Dafny, methods can have any number of in-parameters and any number of out-parameters. In the body of the method, the out-parameters act as local variables and can be assigned and read. When the method body ends, whatever values the out-parameters have will be the values returned to the caller. The in-parameters can of course also be read, but they cannot be re-assigned in the method body.

Here is a statement that shows a call to the method:

```
var t := Triple(18); // sets t to 54
```

Next to the statement, I wrote a comment that the effect of this call is to set `t` to `54`, which is `3` times `18`. By the end of the next chapter, we will be able to turn comments like this into conditions that we can prove to be true.

1.1.Assert Statements

By inspecting method `Triple` above, you can see that it returns the value `3*x`. We can state this observation explicitly in the program using an **`assert`** statement (sometimes called an *inline assertion*, because it places an assertion at a particular point in the code):

```
method Triple(x: int) returns (r: int) {

var y := 2 * x;

r := x + y;

assert r == 3 * x;

}
```

It is good software-engineering practice to use assertions (or in some other languages, comments) to write down essential and non-obvious conditions in the code. Writing down the condition helps our thinking and serves as documentation.

Throughout this book, these kinds of assertions are more than just documentation. They are *proof obligations*. For the program to be considered correct, we must *prove* that the asserted condition holds every time the program's control flow reaches the assertion. The proof should apply for all possible values of the enclosing method's parameters. This means that *running* the program is always going to be hopelessly insucient as a form of proof—there are far too many inputs to try. Instead, we will construct a mathematical proof where we use symbolic *variables*, not specific values, to stand for the program's inputs.

When you write an assertion in a Dafny program, the verifier immediately tries to do the proof for you. For the assertion in `Triple`, the verifier easily constructs the proof, which confirms that the asserted condition always holds. In other cases, if the verifier cannot prove the assertion (for example, if we change the condition above to `3*x + 1`), then it produces an error message.

> **Exercise 1.1.**
>
> Write the method `Triple` in the Dafny IDE. If you don't introduce any typos, you should see the indication “verified” at the bottom. Change the asserted condition to `3 * x + 1`. What happens?

1.2.Working with the Verifier

When receiving an error message from the program verifier, we need to figure out the problem (in other words, we need to debug the verification) and take some recourse. Both of these can be dicult.

If the problem isn't immediately obvious, we can try to figure out the problem by writing more assertions at various points in the program. For example, we might add another assertion between the two assignment statements above. Writing down an assertion amounts to asking the verifier, “do you know that this condition (always) holds at this program point?”.

In some cases, our analysis results in detecting an error in the code. In other cases, we realize that the condition we wrote down isn't correct, which sharpens our understanding of the code. In yet other cases, the condition may in fact always hold but there isn't enough information to conclude this. The typical recourse in such cases is to write some preconditions or to strengthen some invariant; we will see many examples of this in the next few chapters. Finally, in some cases, the problem may be that the verifier isn't “clever” enough to do the proof automatically, in which case we'll need to help it along. The verifier tends to do very well on the sorts of programs discussed in the next several chapters, but for more intricate programs, it will frequently be necessary to supply lemmas or proof steps.

Any error message received from the verifier should be understood as a *possible* error (in the code, in a specification, or in some auxiliary declaration required for the proof), rather than as evidence that the program will not work as intended. In general, think of the program verifier as a dedicated, detailed-oriented colleague who is constantly peering over your shoulder. Sometimes, it detects errors quickly. Other times, just like the best of our colleagues, the verifier isn't able to figure it out, and thus we need to supply further information as hints that help explain why we think the program is correct.

A program trace that reaches a failing assertion (that is, an assertion whose condition evaluates to **`false`**) is erroneous and does not continue past the assertion. In other words, the only control flows that continue after an assertion are ones where the condition holds. For illustration, consider the program

```
method Triple(x: int) returns (r: int) {
var y := 2 * x;
r := x + y;
assert r == 10 * x; // error
assert r < 5;
assert false; // error
}
```

The first assertion does not always hold, so the verifier issues an error message. For any trace where the first assertion does in fact hold (namely, when `3*x == 10*x`, which holds just when `x == 0`), the second assertion holds as well. Therefore, the verifier has no complaint about the second assertion. But even the traces that pass both of the first assertions do not pass the third, so the verifier will issue a complaint about it.

> **Exercise 1.2.**
>
> Try this program with the three **`assert`** statements for yourself in the Dafny IDE. Change the second assertion to make the verifier complain about the first two assertions but not about the third.

1.3.Control Paths

The simple code examples we've seen so far have a single control path containing a sequence of statements. With **`if`** statements and other conditional statements, a piece of code can contain many control paths. A program is correct when the traces along *all* control paths are correct.

For example, here is method `Triple` again, this time using a conditional statement:[0]

```
method Triple(x: int) returns (r: int) {

if x == 0 {

r := 0;

} else {

var y := 2 * x;

r := x + y;

}

assert r == 3 * x;

}
```

The **`if`** statement divides the program traces into those that flow through the “then” branch (when `x == 0`) and those that flow through the “else” branch (when `x != 0`). Program correctness requires the asserted condition to hold regardless of which branch is taken.

When we reason about the control path through the “then” branch, we can confine our attention to the case where `x` is `0`, because only when `x` is `0` can the program take that path. Likewise, when we reason about the “else” branch, we can confine our attention to the case where `x` is non-`0`, because only when x is non-0 can the program take that path.

Control in the ordinary **`if`** statement is *deterministic*. This means that the program inputs uniquely determine the control path taken. Control flow can also be *nondeterministic*, which means that repeatedly running the program, even on the same input, may result in different traces.

An example of nondeterministic control flow is found in Dafny's **`if-case`** statement, where the guards are allowed to be overlapping. This is illustrated by the following, overly convoluted version of `Triple`:

```
method Triple(x: int) returns (r: int) {

if {

case x < 18 =>

var a, b := 2 * x, 4 * x;
```

```
r := (a + b) / 2;

case 0 <= x =>

var y := 2 * x;

r := x + y;

}

assert r == 3 * x;

}
```

When reasoning about the first branch, we can assume `x < 18`, because that branch is taken only if `x` is less than `18`. Similarly, when reasoning about the second branch, we can assume `0 <= x`, because that branch is taken only if `x` is non-negative. If `x` is *both* less than `18` and non-negative, then either branch can be chosen; therefore, in order for the assertion to be correct, both control paths must be checked to be correct for such values of `x`. In this example, both branches compute the same value for `r`, but in general, different branches can have vastly different behaviors.

Two other points are worth making about this last example. One point is that the first branch makes use of a simultaneous assignment, where the right-hand sides are all evaluated before the variables on the left are assigned. The other point is that the scope of local variables declared in a **`case`** branch is limited to that branch, even though the branch is not enclosed in separate curly-brace pairs.

1.4.Method Contracts

A *client* of a method (or function or type or module) is a piece of code that wants to use the method (or function or type or module). Consider a client of the `Triple` method:

```
method Caller() {

var t := Triple(18);

assert t < 100;

}
```

By looking at the body of `Triple` in the previous sections, we can see that `t` will be assigned the value `54`, and thus the caller's assertion will hold. However, by the good software-engineering principle of *information hiding*, the method body is considered to be private to the method and should not be looked at by callers. This allows the method to alter the details of its implementation without affecting callers.

> **Exercise 1.3.**
>
> Using the definition of `Triple` from Section 1.0 and the definition of `Caller` just given, how does the verifier respond?

So, if a caller is not allowed to peek into the method body, how can the caller know anything about what the method will do? The answer is that the method additionally is declared with a *specification*. The specification describes the behavior of the method, but abstracts from the details of the method body. More precisely, a method uses an “agreement”, or *contract*, between the caller and implementation that says what the caller can rely on. The contract thus gives the method author flexibility to describe the interface to the method without divulging the details of the current

implementation. Since the method contract, not the method body, is used at call sites, we say that the methods are *opaque*.

A method contract has two fundamental parts: a *precondition* and a *postcondition*. The precondition says when it is legal for a caller to invoke the method. It is a proof obligation at every call site, and in exchange it can be assumed to hold at the start of the method body. The postcondition is a proof obligation at every return point from the method body, and in exchange it can be assumed to hold upon return from the invocation at the call site.

In our example, we can write a postcondition for `Triple` that lets `Caller` prove the assertion in its body. The postcondition is introduced with the **`ensures`** keyword:

```
method Triple(x: int) returns (r: int)
ensures r == 3 * x
{
var y := 2 * x;
r := x + y;
}
```

Method `Caller` now verifies, and so does `Triple`.

Our example does not warrant a precondition, but let us nevertheless declare one for illustration purposes. A precondition is introduced with the **`requires`** keyword:

```
method Triple(x: int) returns (r: int)
requires x % 2 == 0
ensures r == 3 * x
{
var y := x / 2;
r := 6 * y;
}
```

This version of `Triple` requires the given `x` to be even. Indeed, the new implementation body relies on the evenness of `x`—without the precondition, the value assigned to `r` would be less than `3*x` for odd values of `x` (since integer division performs some rounding) and the verifier would issue an error message that `Triple` may fail to establish its postcondition.

Exercise 1.4.

Remove (or comment out) the precondition of `Triple`. What error does the verifier give you?

Exercise 1.5.

Write two stronger alternatives to the precondition `x % 2 == 0` that also make the method `Triple` verify.

Let's take a look at some more examples.

1.4.0.Underspecification

Consider the following method specification:

```
method Index(n: int) returns (i: int)
requires 1 <= n
ensures 0 <= i < n
```

The precondition says that `Index` can be called only on positive integers. The postcondition says that `Index(n)` will return `i` as some number between `0` and less than `n`.

> **Sidebar 1.0**
>
> In this book, I'll be careful to say "`0` *to* `n`" to mean the possible values of `i` in the half-open interval `0 <= i < n`, and I will say "`0` *through* `n`" to mean the inclusive interval `0 <= i <= n`.

So, *which* number from `0` to `n` will be returned? The specification does not say. Therefore, when we reason about calls to `Index`, we consider all of these values. Viewed by the caller, it is as if this method were nondeterministic.

Here is one way to implement the method:

```
method Index(n: int) returns (i: int)
requires 1 <= n
ensures 0 <= i < n
{
i := n / 2;
}
```

This implementation is correct: for every possible input value `n` that satisfies the precondition, the method body establishes the postcondition.

Here is another way to implement `Index`:

```
method Index(n: int) returns (i: int)
requires 1 <= n
ensures 0 <= i < n
{
i := 0;
}
```

This implementation is also correct. Because methods are opaque—that is, callers only get to see the specification of a method, not its body—it is fine to change the body of `Index` from `i := n / 2;` to `i := 0;` without affecting the correctness of any callers.

Each of the two implementations we just considered is deterministic. That is, for a given input, the method body computes the output in the same way. But since methods are opaque, this is not something a caller can rely on. For example, in

```
var x := Index(50);

var y := Index(50);

assert x == y; // error
```

the assertion cannot be proved, because all we know from the specification of `Index` is

```
0 <= x < 50 && 0 <= y < 50
```

but we know of no connection between `x` and `y`.

While the specification allows any value, the implementation can be—and most often is—deterministic. This important idea is called *underspecification*. It precisely specifies the freedom entailed by the caller-implementation contract.

1.4.1.Multiple postconditions

Consider a method that computes the smaller of two given values:

```
method Min(x: int, y: int) returns (m: int)

ensures m <= x && m <= y
```

This specification says that the value returned is no greater than either `x` or `y`. But it allows the value returned to be significantly smaller than both.

> **Exercise 1.6.**
>
> Write an implementation for `Min` that satisfies the postcondition above but is not always the minimum of `x` and `y`.

To make the specification be the expected one for minimum, we also need to say that the value returned is one of the two inputs:

```
method Min(x: int, y: int) returns (m: int)

ensures m <= x && m <= y

ensures m == x || m == y
```

> **Exercise 1.7.**
>
> Here is the type signature of a method to compute `s` to be the sum of `x` and `y` and `m` to be the maximum of `x` and `y`:
>
> ```
> method MaxSum(x: int, y: int) returns (s: int, m: int)
> ```

(a) Specify the intended postcondition of this method. (b) Write a method that calls `MaxSum` with the input arguments `1928` and `1`. Follow the call with an assertion about what you expect the two out-parameters to be. If the verifier complains that the assertion may be violated, go back to (a) to improve the specification. This is a useful way to "test the specification". Note, you're testing the specification by using the verifier, not by running the program. (c) Write an implementation for `MaxSum`.

Exercise 1.8.

Consider a method that attempts to reconstruct the arguments `x` and `y` from the return values of `MaxSum` in Exercise 1.7. In other words, consider a method with the following type signature and postcondition:

```
method ReconstructFromMaxSum(s: int, m: int) returns (x: int, y: int)
ensures s == x + y
ensures (m == x || m == y) && x <= m && y <= m
```

(a) Try to write the body for this method. You will find you cannot. Write an appropriate precondition for the method that allows you to implement the method. (b) Write the following test harness to test the method's specification:

```
method TestMaxSum(x: int, y: int) {
var s, m := MaxSum(x, y);
var xx, yy := ReconstructFromMaxSum(s, m);
assert (xx == x && yy == y) || (xx == y && yy == x);
}
```

How can you change the specification of `ReconstructFromMaxSum` to allow the assertion in the test harness to succeed?

1.5.Functions

There's one more central declaration construct we'll see often: functions. As you know from mathematics, a *function* denotes a value computed from given arguments. The key property of a function is that it is *deterministic*, that is, any two invocations of the function with the same arguments result in the same value.

Here is an example function declaration in Dafny:

```
function Average(a: int, b: int): int {
(a + b) / 2
}
```

Note that, whereas a method is declared to have some number of out-parameters, a function instead declares a result type, and whereas a method body is a statement, the body of a function is an expression.

Functions can be used in expressions, so we can write a specification like this:

```
method Triple’(x: int) returns (r: int)
ensures Average(r, 3 * x) == 3 * x
```

> **Sidebar 1.1**
>
> Identifiers in Dafny can contain the `’` character. Here, I declared a method named `Triple’`, which you can pronounce as “triple prime”.

> **Exercise 1.9.**
>
> The specification of `Triple’` is not the same as that of `Triple`. (a) Write a correct body for `Triple’` that does not meet the specification of `Triple`. (b) How can you strengthen the specification of `Triple’` with minimal changes to make it equivalent to the specification of `Triple` ? (c) How can you use Dafny to prove that the specifications in (b) are indeed equivalent?

The example above highlights another important difference between methods and functions in Dafny: whereas methods are opaque, functions are *transparent*. This is necessary, because if callers had to understand a function only from its specification, then the specification of functions could never make use of functions, which would severely limit what can be said about a function.

Nevertheless, a function can have specifications. Of special importance is the function's precondition, which says under which circumstances the function is allowed to be invoked. For example, we can restrict uses of `Average` to non-negative arguments:

```
function Average(a: int, b: int): int
requires 0 <= a && 0 <= b
{
(a + b) / 2
}


method Triple(x: int) returns (r: int)
ensures r == 3 * x
{
if 0 <= x {
r := Average(2 * x, 4 * x);
} else {
r := -Average(-2 * x, -4 * x);
}
}
```

Function preconditions are enforced at call sites, just as for method preconditions.

Since assertions, preconditions, and postconditions use boolean conditions, it frequently happens that we declare boolean functions (that is, a function with result type **bool**) for use in specifications. A boolean function is also known as a *predicate* and Dafny reserves a keyword for this purpose: the meaning of

```
predicate IsEven(x: int) {

x % 2 == 0

}
```

is identical to

```
function IsEven(x: int): bool {

x % 2 == 0

}
```

1.6.Compiled versus Ghost

I'll end our introduction of basics with one more important concept. When reasoning about a program, it is frequently necessary to make use of more information than is needed by the compiler or at run time. The Dafny language integrates many features for this purpose. A declaration, variable, statement, etc., that is used only for specification purposes is called a *ghost*. The verifier takes all ghosts into account, whereas the compiler erases all ghosts when it emits executable code. Program constructs that make it into the executable code are referred to as *compiled* or *non-ghost*.

We have seen several ghost constructs already. Pre- and postconditions (declared by **requires** and **ensures** clauses) are ghost. They are used to specify the behavior of the program and establish a contract between callers and implementations. We have also seen a ghost statement: **assert**.

Pre- and postconditions and assertions are checked by the verifier when you write the program. Since they are ghost, they are erased by the compiler and thus they bear no run-time cost. Even if we were not worried about run-time cost, there would be no point in checking assertions at run-time, because once they pass the verifier, the assertions are known to hold in every trace of the program.

Several kinds of declarations in Dafny exist in both a ghost form and a compiled form. They look mostly the same and are treated the same by the verifier, but the purpose for using them is often different. To declare a variable, (in- or out-)parameter, method, function, or predicate as a ghost, simply precede its declaration with the keyword **ghost**. To make the erasure of ghosts possible, Dafny checks that compiled code does not rely on ghost constructs. For example, as illustrated here:

```
method IllegalAssignment() returns (y: int) {

ghost var x := 10;

y := 2 * x; // error: cannot assign to compiled

 // variable using a ghost

}
```

a program is not allowed to use a ghost variable in the right-hand side of an assignment to a compiled

variable.

> **Exercise 1.10.**
>
> The function and method in the following code snippet are both declared as compiled.
>
> ```
> function Average(a: int, b: int): int {
> (a + b) / 2
> }
>
>
> method Triple(x: int) returns (r: int)
> ensures r == 3 * x
> {
> r := Average(2 * x, 4 * x);
> }
> ```
>
> (a) Change this example to declare `Average` as ghost. What error message do you get? (b) Declare both `Average` and `Triple` as ghost. Is that legal? (c) Declare `Average` as compiled and `Triple` as ghost. Is that legal?

1.6.0.Ghost-method example

Let me illustrate a (silly) use of ghost constructs by considering another version of our running example:

```
method Triple(x: int) returns (r: int)
ensures r == 3 * x
{
var y := 2 * x;
r := x + y;
ghost var a, b := DoubleQuadruple(x);
assert a <= r <= b || b <= r <= a;
}


ghost method DoubleQuadruple(x: int) returns (a: int, b: int)
ensures a == 2 * x && b == 4 * x
{
a := 2 * x;
b := 2 * a;
```

```
}
```

This ghost method computes `2*x` and `4*x` into the method's two out-parameters. As you can see, other than the **`ghost`** keyword in front of the method declaration, `Double-Quadruple` looks like an ordinary, compiled method.

This version of `Triple` calls the ghost method, storing its out-parameter in two local ghost variables, `a` and `b`. The subsequent assertion mentions both ghost and compiled variables to express a condition that is expected to hold. When this code is compiled, the postcondition of `Triple`, `Triple`'s local variables `a` and `b`, the call to `DoubleQuadruple`, the assertion, and the entire `DoubleQuadruple` method are erased.

Sidebar 1.2

The example above (and also the `Index` example of Section 1.4.0) illustrates a notational feature that we will make frequent use of: the *chaining* of comparison operators. For example, the operators <, <=, and == can be chained together without repeating the shared operands. The meaning is that of conjoining the pairwise comparisons. For example, the expression

```
0 <= i < j < a.Length == N
```

has the same meaning as

```
0 <= i && i < j && j < a.Length && a.Length == N
```

but is easier to read. The **`assert`** statement in our last version of `Triple` above could also have been written

```
assert (a <= r && r <= b) || (b <= r && r <= a);
```

1.7.Summary

In this chapter, I have introduced methods, functions, pre- and postconditions, simple statements and control flow, and the distinction between ghost and compiled.

A method has any number of in- and out-parameters. It is specified with pre- and postconditions, and its body is a list of statements. The body is opaque, which says that callers reason about method invocations in terms of the specification alone. The behavior of a method can be nondeterministic.

A function has any number of in-parameters and exactly one result value. It can have a specification, and its body is an expression. The body is transparent, which says that callers can look into the function body when reasoning about function invocations. The behavior of a function is deterministic. A function with a boolean result type is called a predicate.

An **`assert`** statement points out a condition that a programmer expects to hold. It gives rise to a proof obligation that must be proved before you can compile and run your program. Thus, in a verified program, you can rely on every asserted condition to hold.

Pre- and postconditions describe the behavior of methods. The precondition limits the uses of a

method by declaring a condition that must be proved at every call site. The postcondition limits the behaviors of the method implementation by declaring a condition that must be proved for every control path through the method body.

A ghost is a construct that serves to explain the intended behavior of a program. It is used by the verifier, but erased by the compiler. Ghost constructs include pre- and postconditions and **`assert`** statements.

> **Exercise 1.11.**
>
> (a) What does the verifier have to say about the two assertions in this program?
>
> ```
> function F(): int {
> 29
> }
>
>
> method M() returns (r: int) {
> r := 29;
> }
>
>
> method Caller() {
> var a := F();
> var b := M();
> assert a == 29;
> assert b == 29;
> }
> ```
>
> (b) Explain why. (c) How can you change the program to make both assertions verify?

Notes

Programming languages do not agree on what to call procedural routines. For example, C and Python refer to a body of code as a “function”, and Rust and Go use the word “method” for a “function” that takes a receiver parameter (like **`this`**). In this book, a “method” has code for its body, and a “function” has an expression for its body. Importantly, a “function” in this book (and, indeed, a **`function`** in Dafny) behaves like a mathematical function. Whether or not a method or function has a receiver parameter depends on whether or not it's declared as a member of a type (we'll have to wait until Chapter 16 before we see a receiver parameter).

A method with side effects is inappropriate for use in specifications. Since Java only has methods, the Java Modeling Language (JML) [31] allows a method to be marked as “pure”, which makes it usable in specifications, approximately like a mathematical function. The OpenJML tool also supports a “function” mark-up for a method, which declares it to be like a function [32].

In WhyML [20] and F* [53], there is no distinction between statements and expressions. Instead, these languages rely on other mechanisms (regions and effect types, respectively) to keep track of the state mutations that are possible in their program functions.

The idea of writing pre- and postconditions and proof assertions as part of a program text is old. For example, they are featured as comments in early program proofs (e.g., [51, 88]). The standard documentation style of the influential language CLU [87] includes pre- and postconditions (and a form of the **`modifies`** clauses we'll see in Part 2). The Euclid language for verifiable programs supports pre- and postconditions as part of its syntax [74]. The role of specifications in the design of programs in emphasized by VDM [67] and Larch [57]. For additional historical notes, I refer you to a survey paper on behavioral specification languages [59].

It was Bertrand Meyer's Eiffel language and the accompanying *Design by Contract* methodology that popularized the idea of writing pre- and postconditions in the program text [89]. These contracts between callers and implementations are written using the ordinary language syntax for expressions, an idea that influenced the designs of JML [66] and Spec# [13].

The use of ghost variables and ghost code has been part of specification folklore for decades. They sometimes go by different names. For example, Owicki and Gries called them “auxiliary” variables [106]. In SPARK [43] and WhyML [20], ghost functions are known as “logical functions”. For a wider perspective on ghost declarations, see [48, 59].

Lastly, an important remark about integers. Dafny's type **`int`**, which I use throughout the book, is *unbounded*. This means that the numbers behave as in mathematics, without any maximum or minimum value. So, if your Dafny program computes a large number (perhaps using the Ackermann function in Section 3.3.1), adds up numbers (like in Section 12.3), or does something as innocent as tripling the value of a given parameter (like method `Triple` in this chapter), you don't have to worry that the magnitude of the result will be too large. (Well, if numbers become *so* large that your computer doesn't have enough memory to represent them , your program will halt. Dafny does not check that you have enough resources to run your programs.) Dafny additionally supports *bounded integers*, like those you can represent using 32 bits, but I won't use them in this book. Some programming languages support only unbounded integers (e.g., Python), some support only bounded integers (e.g., C and Java), and some verification languages support a mix of the two (e.g., SPARK and JML).

---

[0] Note that, unlike some other popular languages, Dafny does not require the guard condition of **`if`** to be surrounded in parentheses. The program text looks cleaner without them.

Chapter 2

Making It Formal

This chapter gives the formal underpinnings of what the previous chapter described informally. The meaning of programs is given precisely with mathematical formulas. These are said to give the *semantics* of the program statements.

Throughout this book, the reasoning is based on what is called *Floyd logic*. It uses boolean formulas to describe what is known before and after each statement, which allows us to break down the reasoning about programs into reasoning about individual statements of the program. Along the way, I will define *Hoare triples*, which give a good guide to understanding semantics, and *weakest preconditions*, which give convenient ways to automate program reasoning.

The study of program semantics may seem overly concerned with picky details. Think of it as long multiplication—it prescribes steps for computing something in detail. Once you've applied long multiplication on many problems, you develop a deeper understanding of multiplication. You then realize that some steps can be replaced by shortcuts. More importantly, you can then put multiplication to use in daily problems. It is not the tiny steps of long multiplication that let you solve problems, but it is your familiarity with multiplication that does. Because you can carry out the steps, you know of a way to check something in detail, but in practice you use a calculator or spreadsheet to perform multiplication for you.