[Previous Section | Next Section | Chapter Index | Main Index]

Section 8.2

Writing Correct Programs

Subsections

CORRECT PROGRAMS DON'T just happen. It takes planning and attention to detail to avoid errors in programs. There are some techniques that programmers can use to increase the likelihood that their programs are correct.

8.2.1 Provably Correct Programs

In some cases, it is possible to **prove** that a program is correct. That is, it is possible to demonstrate mathematically that the sequence of computations represented by the program will always produce the correct result. Rigorous proof is difficult enough that in practice it can only be applied to fairly small programs. Furthermore, it depends on the fact that the "correct result" has been specified correctly and completely. As I've already pointed out, a program that correctly meets its specification is not useful if its specification was wrong. Nevertheless, even in everyday programming, we can apply some of the ideas and techniques that are used in proving that programs are correct.

The fundamental ideas are process and state. A state consists of all the information relevant to the execution of a program at a given moment during its execution. The state includes, for example, the values of all the variables in the program, the output that has been produced, any input that is waiting to be read, and a record of the position in the program where the computer is working. A process is the sequence of states that the computer goes through as it executes the program. From this point of view, the meaning of a statement in a program can be expressed in terms of the effect that the execution of that statement has on the computer's state. As a simple example, the meaning of the assignment statement "`x = 7;`" is that after this statement is executed, the value of the variable `x` will be 7. We can be absolutely sure of this fact, so it is something upon which we can build part of a mathematical proof.

In fact, it is often possible to look at a program and deduce that some fact must be true at a given point during the execution of a program. For example, consider the `do` loop:

```
do {
   System.out.print("Enter a positive integer: ");
   N = TextIO.getlnInt();
} while (N <= 0);
```

After this loop ends, we can be absolutely sure that the value of the variable `N` is greater than zero. The loop cannot end until this condition is satisfied. This fact is part of the meaning of the `while` loop. More generally, if a `while` loop uses the test "`while (condition)`", then after the loop ends, we can be sure that the `condition` is false. We can then use this fact to draw further deductions about what happens as the execution of the program continues. (With a loop, by the way, we also have to worry about the question of whether the loop will ever end. This is something that has to be verified separately.)

A fact that can be proven to be true after a given program segment has been executed is called a postcondition of that program segment. Postconditions are known facts upon which we can build further deductions about the behavior of the program. A postcondition of a program as a whole is simply a fact that can be proven to be true after the program has finished executing. A program can be

proven to be correct by showing that the postconditions of the program meet the program's specification.

Consider the following program segment, where all the variables are of type double:

```
disc = B*B - 4*A*C;
x = (-B + Math.sqrt(disc)) / (2*A);
```

The quadratic formula (from high-school mathematics) assures us that the value assigned to `x` is a solution of the equation $A*x^2 + B*x + C = 0$, provided that the value of `disc` is greater than or equal to zero and the value of `A` is not zero. **If** we can assume or guarantee that `B*B-4*A*C >= 0` and that `A != 0`, then the fact that `x` is a solution of the equation becomes a postcondition of the program segment. We say that the condition, `B*B-4*A*C >= 0` is a precondition of the program segment. The condition that `A != 0` is another precondition. A precondition is defined to be a condition that must be true at a given point in the execution of a program in order for the program to continue correctly. A precondition is something that you want to be true. It's something that you have to check or force to be true, if you want your program to be correct.

We've encountered preconditions and postconditions once before, in Subsection 4.6.1. That section introduced preconditions and postconditions as a way of specifying the contract of a subroutine. As the terms are being used here, a precondition of a subroutine is just a precondition of the code that makes up the definition of the subroutine, and the postcondition of a subroutine is a postcondition of the same code. In this section, we have generalized these terms to make them more useful in talking about program correctness.

Let's see how this works by considering a longer program segment:

```
do {
   System.out.println("Enter A, B, and C.  B*B-4*A*C must be >= 0.");
   System.out.print("A = ");
   A = TextIO.getlnDouble();
   System.out.print("B = ");
   B = TextIO.getlnDouble();
   System.out.print("C = ");
   C = TextIO.getlnDouble();
   if (A == 0 || B*B - 4*A*C < 0)
      System.out.println("Your input is illegal.  Try again.");
} while (A == 0 || B*B - 4*A*C < 0);

disc = B*B - 4*A*C;
x = (-B + Math.sqrt(disc)) / (2*A);
```

After the loop ends, we can be sure that `B*B-4*A*C >= 0` and that `A != 0`. The preconditions for the last two lines are fulfilled, so the postcondition that `x` is a solution of the equation $A*x^2 + B*x + C = 0$ is also valid. This program segment correctly and provably computes a solution to the equation. (Actually, because of problems with representing real numbers on computers, this is not 100% true. The **algorithm** is correct, but the **program** is not a perfect implementation of the algorithm. See the discussion in Subsection 8.1.3.)

Here is another variation, in which the precondition is checked by an `if` statement. In the first part of the `if` statement, where a solution is computed and printed, we know that the preconditions are fulfilled. In the other parts, we know that one of the preconditions fails to hold. In any case, the program is correct.

```
System.out.println("Enter your values for A, B, and C.");
System.out.print("A = ");
A = TextIO.getlnDouble();
System.out.print("B = ");
```

```
B = TextIO.getlnDouble();
System.out.print("C = ");
C = TextIO.getlnDouble();

if (A != 0 && B*B - 4*A*C >= 0) {
   disc = B*B - 4*A*C;
   x = (-B + Math.sqrt(disc)) / (2*A);
   System.out.println("A solution of A*X*X + B*X + C = 0 is " + x);
}
else if (A == 0) {
   System.out.println("The value of A cannot be zero.");
}
else {
   System.out.println("Since B*B - 4*A*C is less than zero, the");
   System.out.println("equation A*X*X + B*X + C = 0 has no solution.");
}
```

Whenever you write a program, it's a good idea to watch out for preconditions and think about how your program handles them. Often, a precondition can offer a clue about how to write the program.

For example, every array reference, such as `A[i]`, has a precondition: The index must be within the range of legal indices for the array. For `A[i]`, the precondition is that `0 <= i < A.length`. The computer will check this condition when it evaluates `A[i]`, and if the condition is not satisfied, the program will be terminated. In order to avoid this, you need to make sure that the index has a legal value. (There is actually another precondition, namely that `A` is not `null`, but let's leave that aside for the moment.) Consider the following code, which searches for the number `x` in the array `A` and sets the value of `i` to be the index of the array element that contains `x`:

```
i = 0;
while (A[i] != x) {
   i++;
}
```

As this program segment stands, it has a precondition, namely that `x` is actually in the array. If this precondition is satisfied, then the loop will end when `A[i] == x`. That is, the value of `i` when the loop ends will be the position of `x` in the array. However, if `x` is not in the array, then the value of `i` will just keep increasing until it is equal to `A.length`. At that time, the reference to `A[i]` is illegal and the program will be terminated. To avoid this, we can add a test to make sure that the precondition for referring to `A[i]` is satisfied:

```
i = 0;
while (i < A.length && A[i] != x) {
   i++;
}
```

Now, the loop will definitely end. After it ends, `i` will satisfy **either** `i == A.length` or `A[i] == x`. An `if` statement can be used after the loop to test which of these conditions caused the loop to end:

```
i = 0;
while (i < A.length && A[i] != x) {
   i++;
}

if (i == A.length)
   System.out.println("x is not in the array");
else
   System.out.println("x is in position " + i);
```

8.2.2 Robust Handling of Input

One place where correctness and robustness are important -- and especially difficult -- is in the processing of input data, whether that data is typed in by the user, read from a file, or received over a network. Files and networking will be covered in Chapter 11, which will make essential use of material that will be covered in the next section of this chapter. For now, let's look at an example of processing user input.

Examples in this textbook use my TextIO class for reading input from the user. This class has built-in error handling. For example, the function `TextIO.getDouble()` is guaranteed to return a legal value of type double. If the user types an illegal value, then TextIO will ask the user to re-enter their response; your program never sees the illegal value. However, this approach can be clumsy and unsatisfactory, especially when the user is entering complex data. In the following example, I'll do my own error-checking.

Sometimes, it's useful to be able to look ahead at what's coming up in the input without actually reading it. For example, a program might need to know whether the next item in the input is a number or a word. For this purpose, the TextIO class includes the function `TextIO.peek()`. This function returns a char which is the next character in the user's input, but it does not actually read that character. If the next thing in the input is an end-of-line, then `TextIO.peek()` returns the new-line character, `'\n'`.

Often, what we really need to know is the next **non-blank** character in the user's input. Before we can test this, we need to skip past any spaces (and tabs). Here is a function that does this. It uses `TextIO.peek()` to look ahead, and it reads characters until the next character in the input is either an end-of-line or some non-blank character. (The function `TextIO.getAnyChar()` reads and returns the next character in the user's input, even if that character is a space. By contrast, the more common `TextIO.getChar()` would skip any blanks and then read and return the next non-blank character. We can't use `TextIO.getChar()` here since the object is to skip the blanks **without** reading the next non-blank character.)

```
/**
 * Reads past any blanks and tabs in the input.
 * Postcondition:  The next character in the input is an
 *                 end-of-line or a non-blank character.
 */
static void skipBlanks() {
   char ch;
   ch = TextIO.peek();
   while (ch == ' ' || ch == '\t') {
         // Next character is a space or tab; read it
         // and look at the character that follows it.
      ch = TextIO.getAnyChar();
      ch = TextIO.peek();
   }
} // end skipBlanks()
```

(In fact, this operation is so common that it is built into `TextIO`. The method `TextIO.skipBlanks()` does essentially the same thing as the `skipBlanks()` method presented here.)

An example in Subsection 3.5.3 allowed the user to enter length measurements such as "3 miles" or "1 foot". It would then convert the measurement into inches, feet, yards, and miles. But people commonly use combined measurements such as "3 feet 7 inches". Let's improve the program so that it allows inputs of this form.

More specifically, the user will input lines containing one or more measurements such as "1 foot" or "3 miles 20 yards 2 feet". The legal units of measure are inch, foot, yard, and mile. The program will also recognize plurals (inches, feet, yards, miles) and abbreviations (in, ft, yd, mi). Let's write a subroutine that will read one line of input of this form and compute the equivalent number of inches.

The main program uses the number of inches to compute the equivalent number of feet, yards, and miles. If there is any error in the input, the subroutine will print an error message and return the value `-1`. The subroutine assumes that the input line is not empty. The main program tests for this before calling the subroutine and uses an empty line as a signal for ending the program.

Ignoring the possibility of illegal inputs, a pseudocode algorithm for the subroutine is

```
inches = 0    // This will be the total number of inches
while there is more input on the line:
    read the numerical measurement
    read the units of measure
    add the measurement to inches
return inches
```

We can test whether there is more input on the line by checking whether the next non-blank character is the end-of-line character. But this test has a precondition: Before we can test the next non-blank character, we have to skip over any blanks. So, the algorithm becomes

```
inches = 0
skipBlanks()
while TextIO.peek() is not '\n':
    read the numerical measurement
    read the unit of measure
    add the measurement to inches
    skipBlanks()
return inches
```

Note the call to `skipBlanks()` at the end of the `while` loop. This subroutine must be executed before the computer returns to the test at the beginning of the loop. More generally, if the test in a `while` loop has a precondition, then you have to make sure that this precondition holds at the **end** of the while loop, before the computer jumps back to re-evaluate the test, as well as before the start of the loop.

What about error checking? Before reading the numerical measurement, we have to make sure that there is really a number there to read. Before reading the unit of measure, we have to test that there is something there to read. (The number might have been the last thing on the line. An input such as "3", without a unit of measure, is not acceptable.) Also, we have to check that the unit of measure is one of the valid units: inches, feet, yards, or miles. Here is an algorithm that includes error-checking:

```
inches = 0
skipBlanks()

while TextIO.peek() is not '\n':

    if the next character is not a digit:
       report an error and return -1
    Let measurement = TextIO.getDouble();

    skipBlanks()    // Precondition for the next test!!
    if the next character is end-of-line:
       report an error and return -1
    Let units = TextIO.getWord()

    if the units are inches:
        add measurement to inches
    else if the units are feet:
        add 12*measurement to inches
    else if the units are yards:
        add 36*measurement to inches
    else if the units are miles:
        add 12*5280*measurement to inches
    else
```

```
            report an error and return -1

        skipBlanks()

    return inches
```

As you can see, error-testing adds significantly to the complexity of the algorithm. Yet this is still a fairly simple example, and it doesn't even handle all the possible errors. For example, if the user enters a numerical measurement such as `1e400` that is outside the legal range of values of type double, then the program will fall back on the default error-handling in TextIO. Something even more interesting happens if the measurement is "1e308 miles". The number `1e308` is legal, but the corresponding number of inches is outside the legal range of values for type double. As mentioned in the previous section, the computer will get the value `Double.POSITIVE_INFINITY` when it does the computation. You might want to run the program and try this out.

Here is the subroutine written out in Java:

```
/**
 * Reads the user's input measurement from one line of input.
 * Precondition:   The input line is not empty.
 * Postcondition:  If the user's input is legal, the measurement
 *                 is converted to inches and returned.  If the
 *                 input is not legal, the value -1 is returned.
 *                 The end-of-line is NOT read by this routine.
 */
static double readMeasurement() {

   double inches;  // Total number of inches in user's measurement.

   double measurement;  // One measurement,
                        //   such as the 12 in "12 miles"
   String units;        // The units specified for the measurement,
                        //   such as "miles"

   char ch;  // Used to peek at next character in the user's input.

   inches = 0;  // No inches have yet been read.

   skipBlanks();
   ch = TextIO.peek();

   /* As long as there is more input on the line, read a measurement and
      add the equivalent number of inches to the variable, inches.  If an
      error is detected during the loop, end the subroutine immediately
      by returning -1. */

   while (ch != '\n') {

       /* Get the next measurement and the units.  Before reading
          anything, make sure that a legal value is there to read. */

       if ( ! Character.isDigit(ch) ) {
           System.out.println(
                 "Error:  Expected to find a number, but found " + ch);
           return -1;
       }
       measurement = TextIO.getDouble();

       skipBlanks();
       if (TextIO.peek() == '\n') {
           System.out.println(
                 "Error:  Missing unit of measure at end of line.");
           return -1;
```

```
                }
                units = TextIO.getWord();
                units = units.toLowerCase();

                /* Convert the measurement to inches and add it to the total. */

                if (units.equals("inch")
                        || units.equals("inches") || units.equals("in")) {
                    inches += measurement;
                }
                else if (units.equals("foot")
                           || units.equals("feet") || units.equals("ft")) {
                    inches += measurement * 12;
                }
                else if (units.equals("yard")
                           || units.equals("yards") || units.equals("yd")) {
                    inches += measurement * 36;
                }
                else if (units.equals("mile")
                           || units.equals("miles") || units.equals("mi")) {
                    inches += measurement * 12 * 5280;
                }
                else {
                    System.out.println("Error: \"" + units
                                       + "\" is not a legal unit of measure.");
                    return -1;
                }

                /* Look ahead to see whether the next thing on the line is
                   the end-of-line. */

                skipBlanks();
                ch = TextIO.peek();

            }  // end while

            return inches;

        } // end readMeasurement()
```

The source code for the complete program can be found in the file *LengthConverter2.java*.

[Previous Section | Next Section | Chapter Index | Main Index]