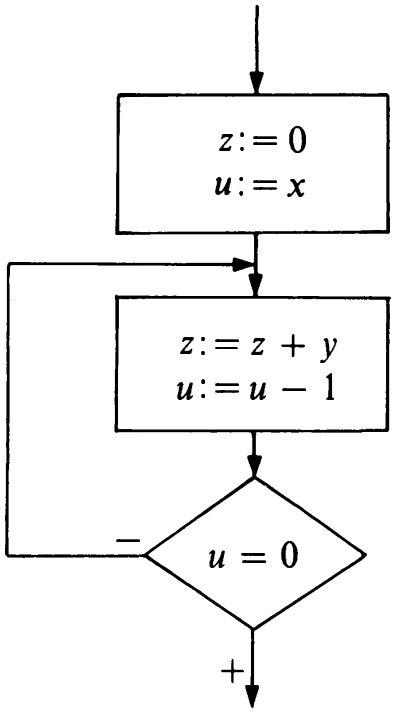
5 SOME SIMPLE PROGRAMS

Chapter 4 showed clearly why a program must consist of statements formulated in a notation that the computer "understands." Although we do not know yet which kinds of statements and formulas a programming language contains, we do know that these statements will precisely specify the intended actions. This unquestionable necessity for precision probably constitutes the main difference between communication among humans and communication with machines. Work with computers requires both precision and clarity. Vagueness and ambiguity are strictly forbidden.

A commonly used and easily comprehended notation to express programs is the so-called *flow-diagram* or flowchart. Program (2.3) is depicted in (5.1) as a flow-diagram.

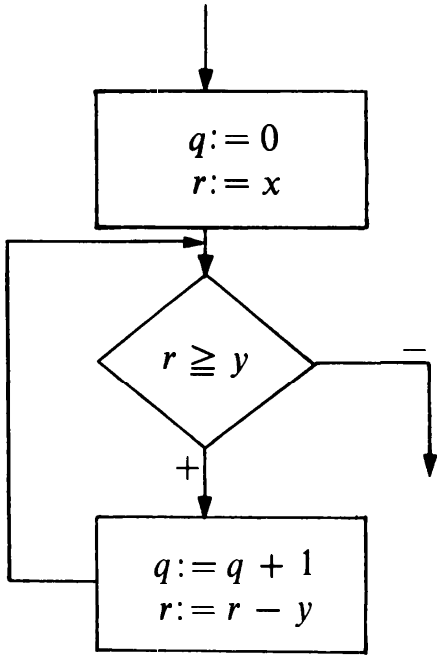


(5.1)

This pictorial representation clearly displays the possible sequence of steps by visually illustrating two kinds of instructions:

(a) the assignment of values to variables, denoted by rectangles, and
(b) the decisions, denoted by diamonds.

A decision step has more than one possible successor. If the specified relation or condition is satisfied, then the path denoted by the + sign is followed; otherwise the one denoted by the − sign is taken. A repetition manifests itself by a *loop*, that is, a closed sequence consisting of statements and at least one decision that will determine the termination of the repetition. In the same fashion, program (2.7) is represented by flow-diagram (5.2).



(5.2)

A program determines a pattern of behavior for an unspecified, often infinite number of possible processes. The individual processes are distinguished by the values of the variables involved at related time intervals and, in particular, by the *initial values* or arguments. But how can we ensure that all processes executable according to a given program will compute the specified results? This question of the *correctness of programs* is one of the most central, crucial, and unavoidable issues of programming.

The correctness of programs (2.3) and (2.7) was demonstrated in tables (2.4) and (2.8) for a fixed pair of values x and y. This method of establishing correctness is called *program testing*; it consists of selecting arguments (x and y), executing the process with these selected arguments, and comparing the computed results with the previously known correct results. This experimental testing is repeated with several arguments, using the computer as the ideal tool. Nevertheless, this conventional method is

expensive, time consuming, and cumbersome. In the end, it is also unsatisfactory, since to remove all doubts about the correctness of a program, it would be necessary to execute all possible computations—not just a selected few. But if the results of all processes have to be known beforehand, there would hardly be any purpose in writing a computer program. Anyway, in practice, such an exhaustive testing of a program is quite impossible. For example: Assume that a given computer takes 1 μsec for the addition of two numbers (according to its program) and that it is capable of representing numbers up to an absolute value of 2^{60}. Then 2^{2*60} different additions are possible, taking

$$2^{2*60} * 10^{-6} \text{ sec} \doteq 3.2 * 10^{22} \text{ yr}$$

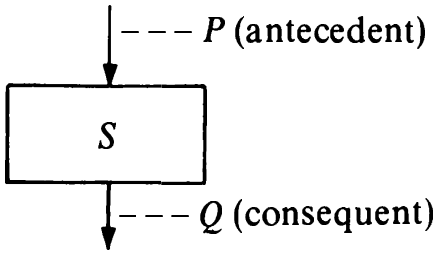
Since this kind of exhaustive experimental testing is both senseless and impossible, we can formulate an important ground rule.

Experimental testing of programs can be used to show the presence of errors but never to prove their absence.

Consequently, it is necessary to abstract from individual processes and to postulate certain generally valid conditions that can be derived from the pattern of behavior. This analytic method of testing is called *program verification.* In contrast to program testing—where the properties of individual *processes* are investigated—program verification is concerned with the properties of the *program.*

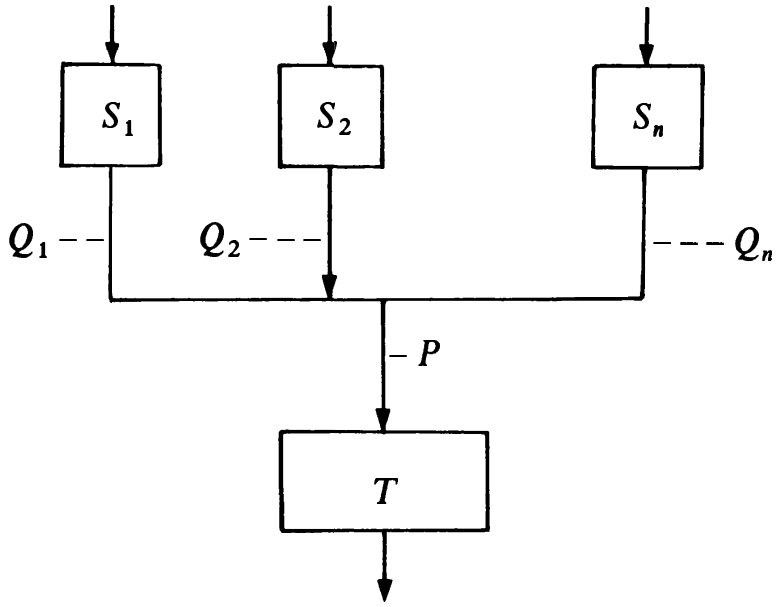
Program verification employs the same principles as empirical testing and might therefore be considered analogous with process verification. But instead of recording the individual values of variables in a trace table, we postulate generally valid ranges of values and relationships among variables after each statement. "Generally valid" should be understood to mean "valid for each process executable according to the given program and valid at the point of annotation irrespective of the statements previously obeyed." We can now postulate four basic *rules of analytic program verification.*

1. Preceding and succeeding every statement, one or several conditions are specified that are satisfied before and after every execution of the statement. The annotated conditions are called *assertions*; the ones preceding a statement S are called its *antecedents* P; and the ones following it are called consequences or *consequents* Q.



(5.3)

2. If several paths of a flow-diagram merge in front of a statement T, then the consequents Q_i of all preceding statements S_i must logically imply the antecedent P of statement T. Thus

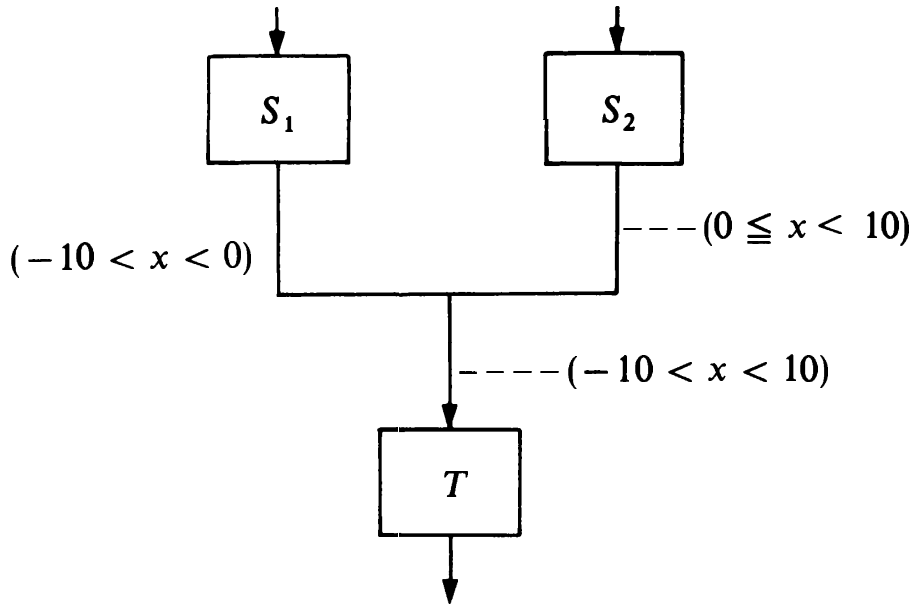


(5.4.1)

$$Q_i \supset P \quad \text{for} \quad i = 1 \ldots n$$

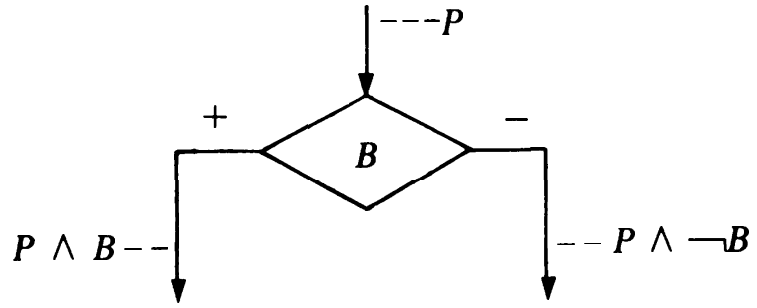
($Q \supset P$ is verbalized as "Q implies P.")

Example:



(5.4.2)

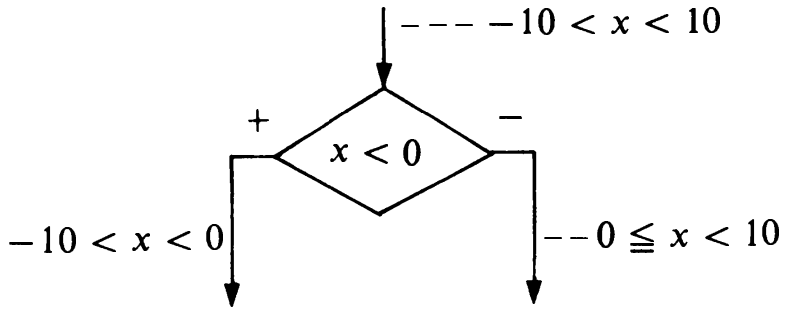
3. If an assertion P holds preceding a decision with condition B, then the two consequents are



(5.5.1)

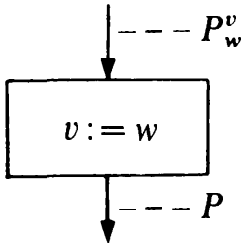
($P \wedge B$ is verbalized as "P and B," $P \wedge \neg B$ as "P and not B.")

Example:



(5.5.2)

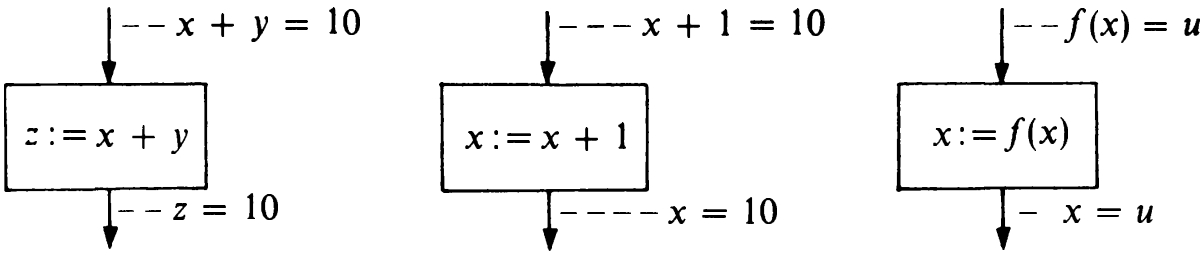
4. If an assertion P holds after the assignment of the value w to the variable v, the antecedent of the assignment is obtained by substituting w for all free occurrences of v in P.



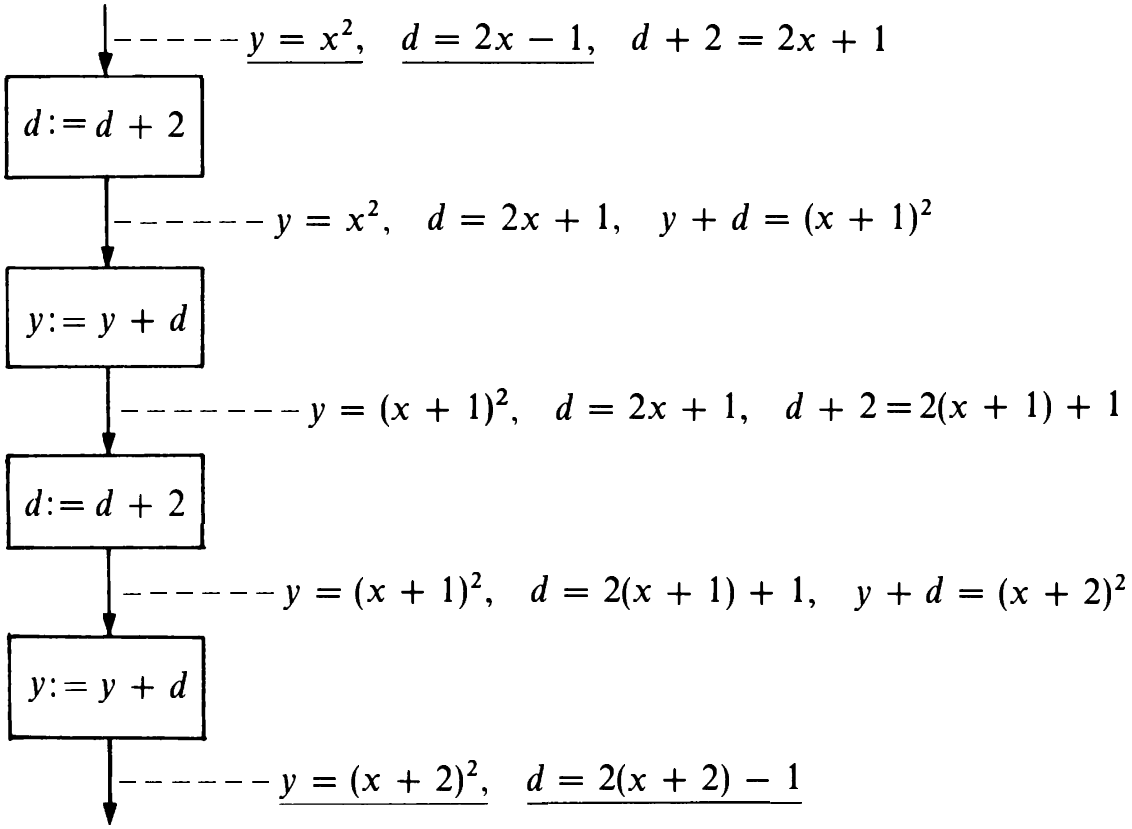
(5.6.1)

(This rule may well be regarded as the definition of the *effect of assignment*.)

Examples:

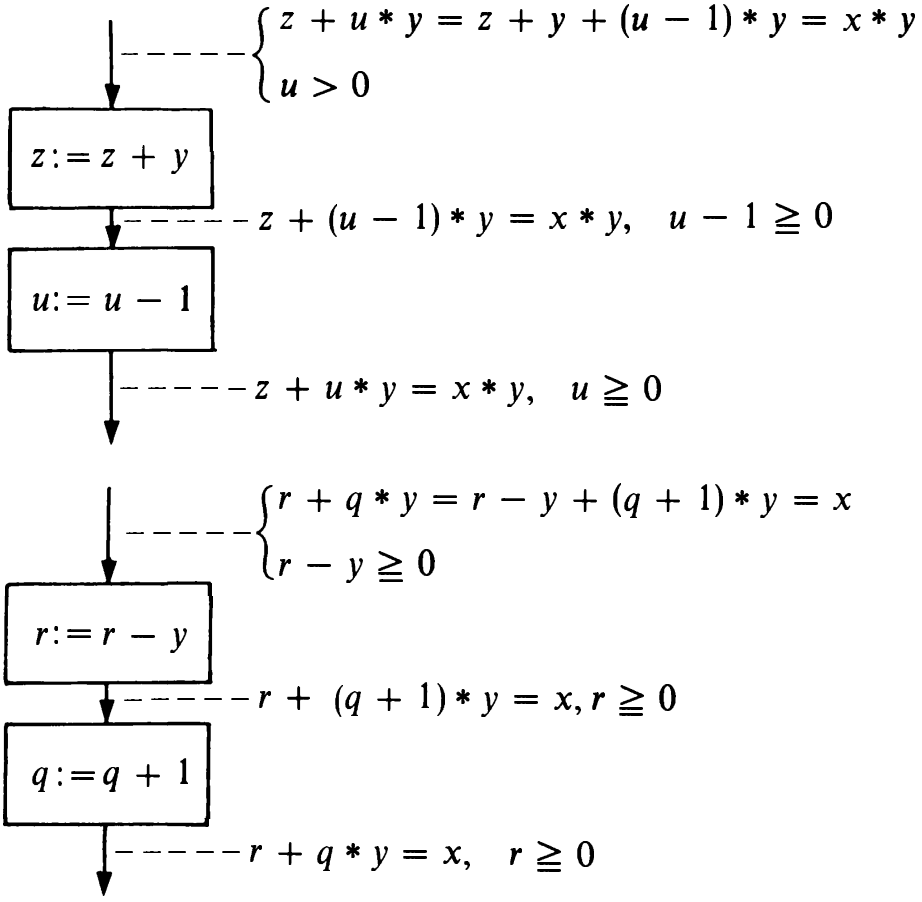


(5.6.2)



(5.7)

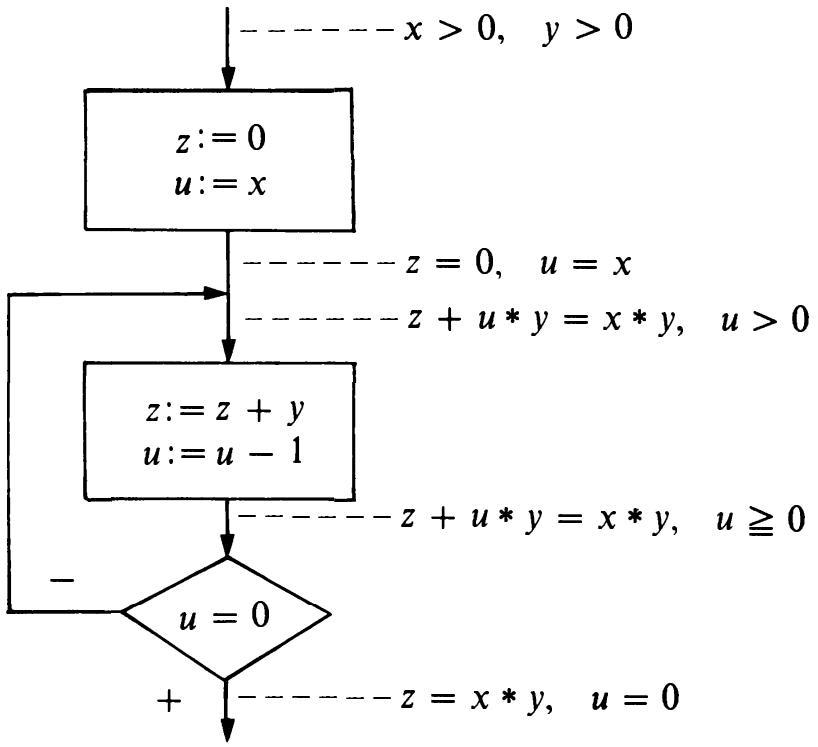
We will now apply these basic rules in verifying the correctness of programs (5.1) and (5.2), which compute the product and quotient of two natural numbers, respectively. First, we establish the following intermediate results by applying the derivation rule for the assignment statement.



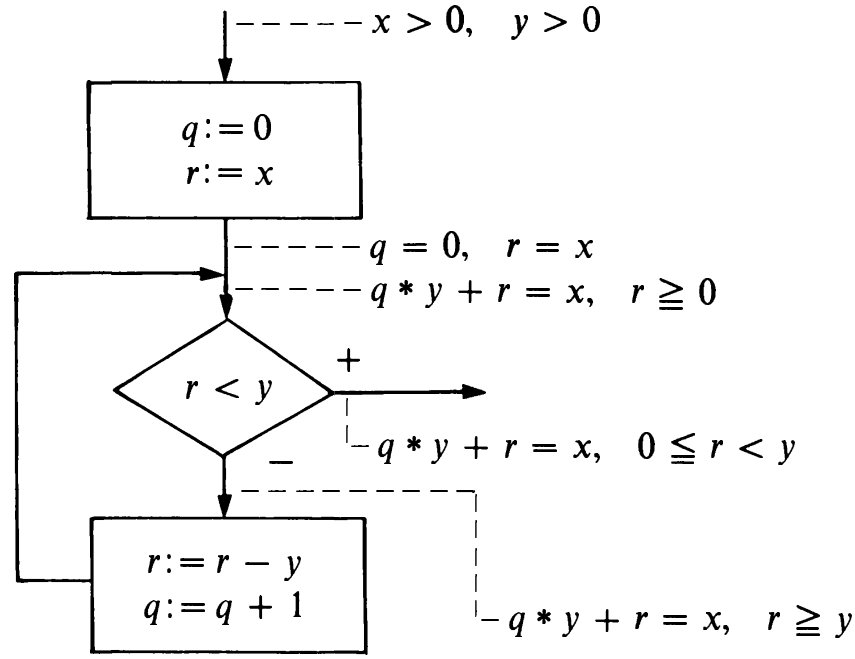
(5.8)

(5.9)

The two complete programs with the relevant assertions are shown in flow-diagrams (5.10) and (5.11).



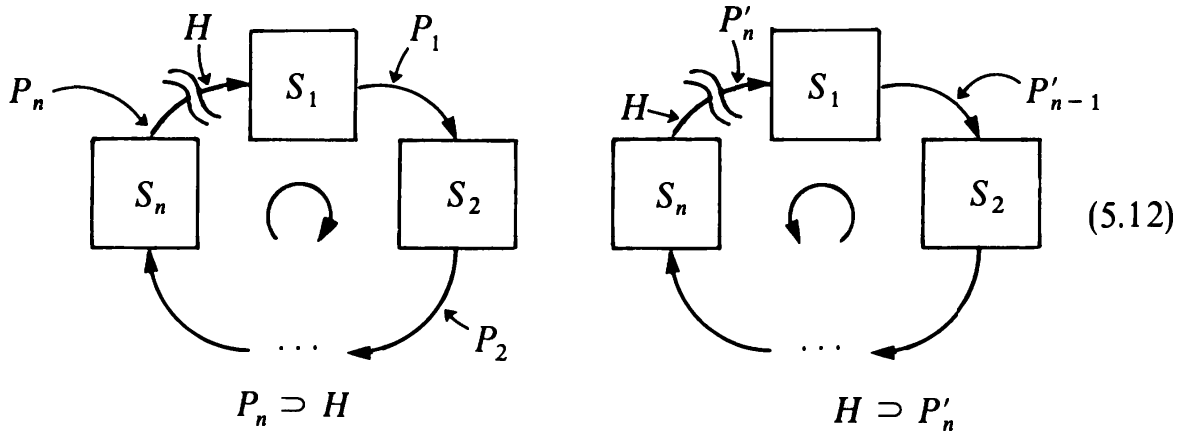
(5.10)



(5.11)

The determination of assertions in a sequence of statements, based on rules (5.3)–(5.7), is generally a straightforward matter, when either the antecedent of the first statement or the consequent of the last statement is given. A serious difficulty arises, however, as soon as the flow-diagram contains closed circular paths, that is, if the program contains repetitions. In this case, the best approach is to cut the loop at some place and then postulate a *hypothesis H* at the cut. Starting with the hypothesis, assertions can be

derived through the now linearized sequence, going either forward or backward. The resulting assertion P_n at the end (i.e., at the cut) must then logically imply (or be implied by) H so that the cut may be closed. If this is not satisfied, another hypothesis must be assumed, and the process must be repeated, as shown in (5.12).



(5.12)

It is advantageous to place the cut preceding the condition B according to which the repetition is terminated. The logical combination of conditions H and B will then constitute the consequent of the entire repetitive statement group.

Such an assertion, holding independently of the number of previously executed repetitions, is called a loop-invariant or simply an *invariant*, since it represents a condition that does not change while the process progresses. In programs (5.10) and (5.11), the two invariants are, respectively,

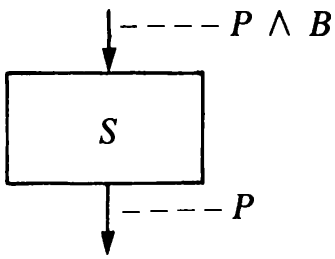
$$(z + u * y = x * y) \wedge (u \geqq 0)$$

and

$$(q * y + r = x) \wedge (r \geqq 0)$$

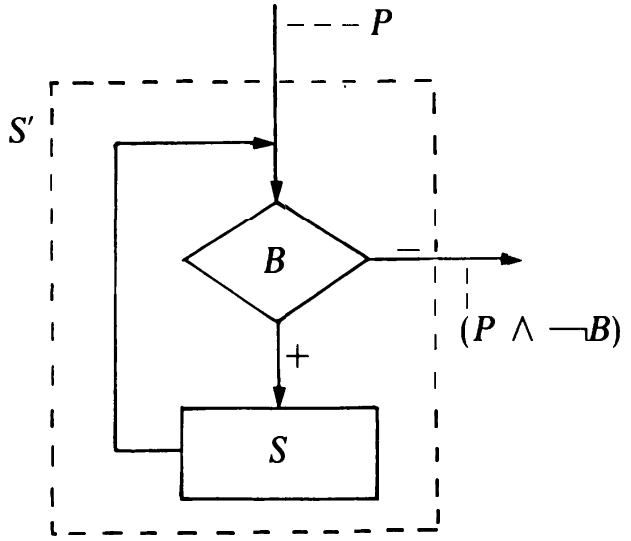
Since repetitions or loops are fundamental constituents in all computational processes and computer programs, we can formulate these instructions as *rules of derivation.* These rules specify the consequent of a repetitive statement, given its antecedent and given the assertions for the statement that is repeated.

1. Given an assertion P that is invariant over statement S—that is, if given as its antecedent, it is also its consequent—and is represented by



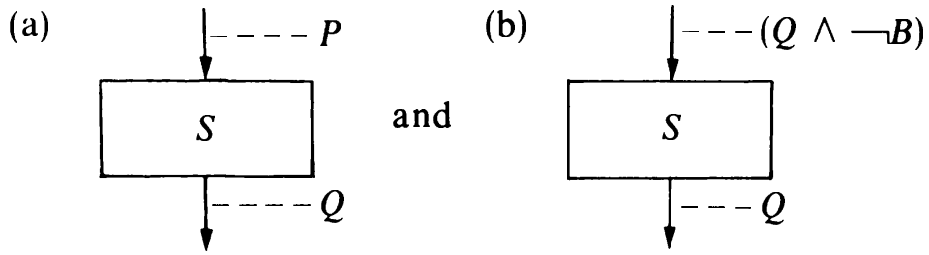
(5.13.1)

we can make the following assertion about the repetitive statement S':



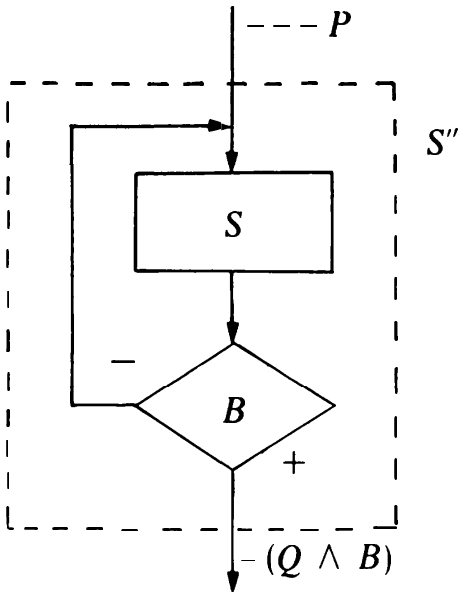
(5.13.2)

2. Given the two preconditions about statement S,



(5.14.1)

we can make the following assertion about the repetitive statement S'':



(5.14.2)

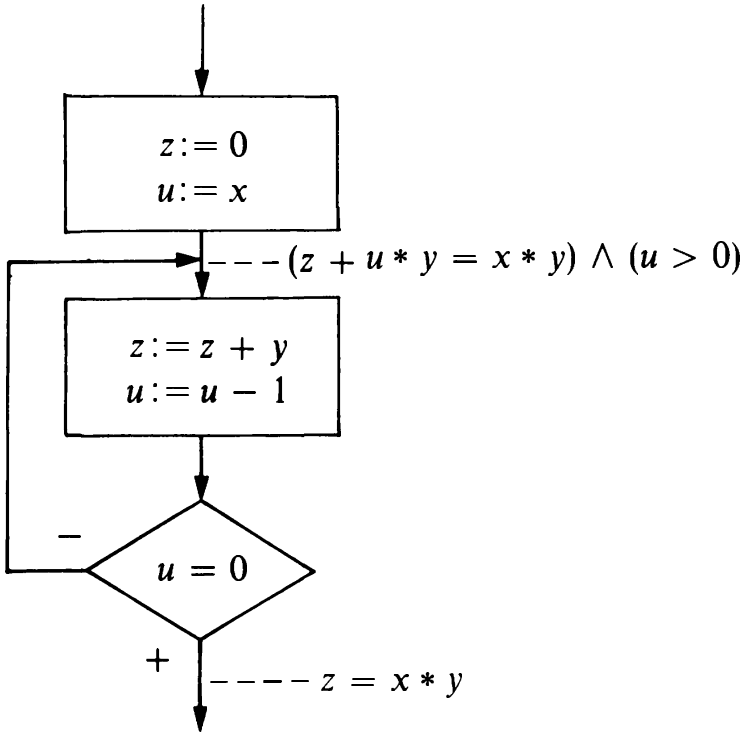
Note that for this second form of repetition, *two* preconditions must be satisfied in order to apply the derivation rule. This form may justly be called the more "dangerous," since programming errors may frequently be traced directly back to the programmer's oversight of one of the two preconditions [usually (a)]. In cases of doubt, the first form, where the termination condition B precedes the statement S, is recommended.

These guidelines should be considered essentially as an informal approach to the problem of analytic verification of the properties of programs. It is particularly important to realize the difficulties involved in the problem of finding invariants. The lesson that every programmer should learn is that the *explicit indication of the relevant invariant for each repetition represents the most valuable element in every program documentation.* But even if a program is intended for the exclusive use of its author, the explicit determination of invariants may help in preventing many errors, which otherwise would be discovered only by extensive testing, and often even remain a permanent "feature" of the supposedly correct program. Just as important, however, is the explicit indication of the variables' ranges of values, particularly those for the initial values for which the stated program properties hold.

Finally, figures (5.15) and (5.16) demonstrate what constitutes both a necessary and an adequate program documentation. It is important to remember that a program may be *overdocumented.* A program containing so many comments that the actual statements are difficult to spot is useless!

Multiplication of natural numbers

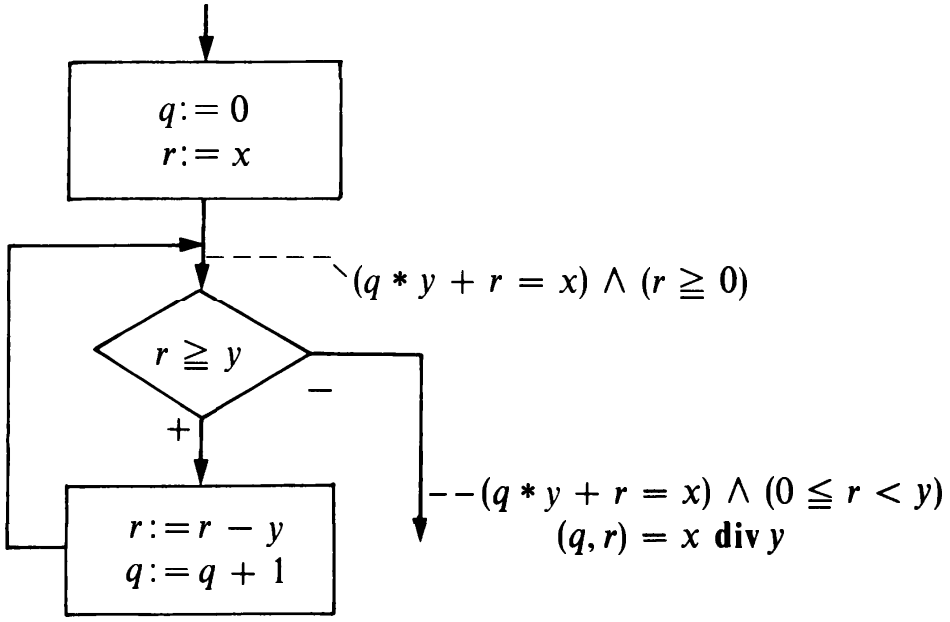
Arguments: x, y
Result: z
Auxiliary variable: u



(5.15)

Division of natural numbers

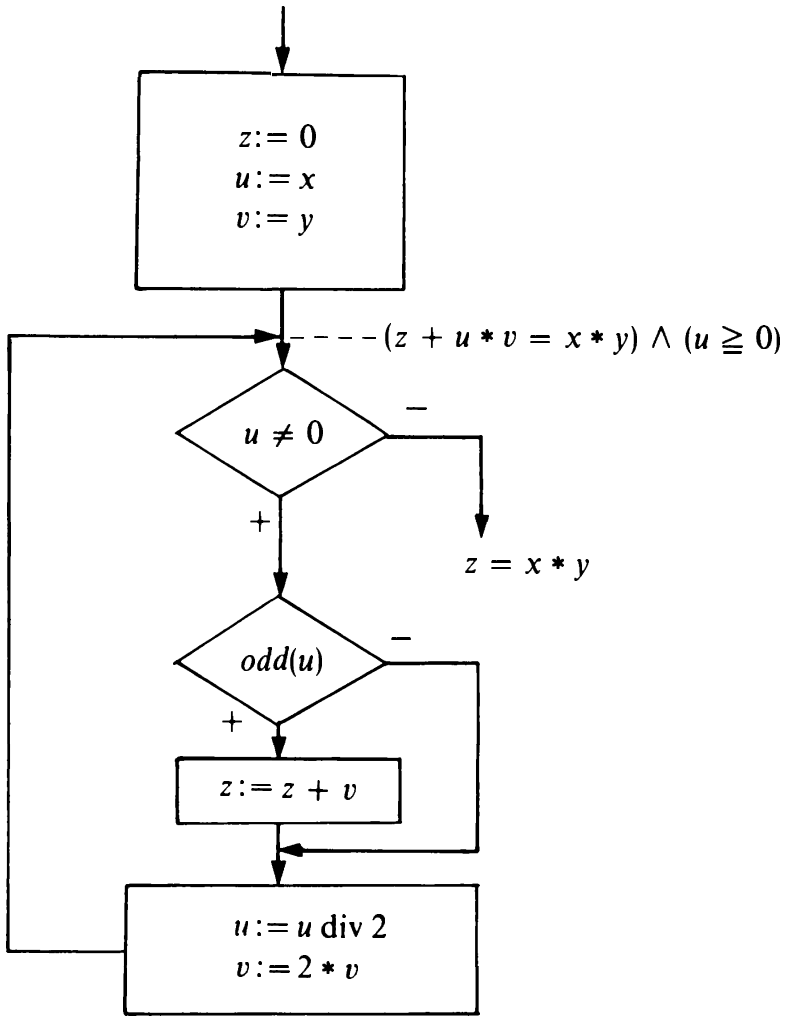
Arguments: x, y
Results: q (integer quotient), r (remainder)



(5.16)

EXERCISES

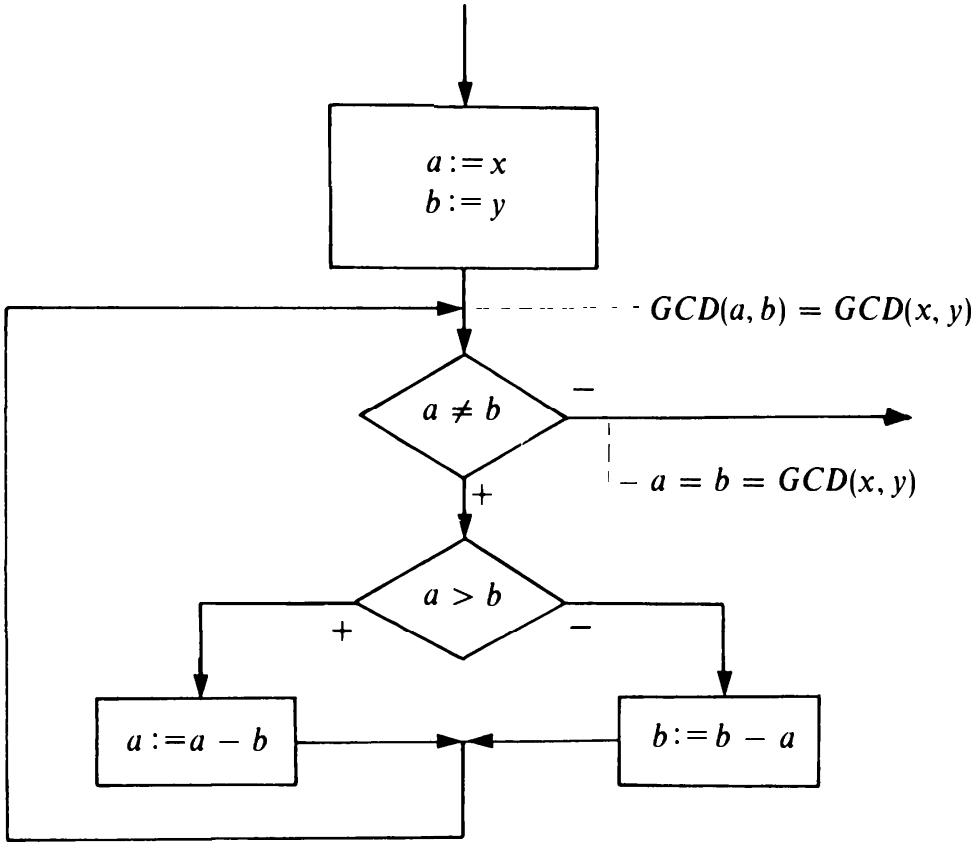
5.1 The following program computes the product $z := x * y$, where only the operations of addition, doubling, and halving are employed. The arguments x and y are natural numbers; u and v are (auxiliary) integer variables. The predicate *odd*(*u*)



(5.17)

is satisfied if u is an odd number. Determine the relevant antecedents and consequents of each statement that can be derived by applying rules (5.3)–(5.6) from the given invariant.

5.2 The following program computes the greatest common divisor (GCD) of two natural numbers x and y; a and b are integer variables whose final value represents the desired result.



(5.18)

As in Exercise 5.1, determine the necessary assertions by using these known relationships of the function GCD:

(a) $u > v : GCD(u, v) = GCD(u - v, v)$
(b) $GCD(u, v) = GCD(v, u)$
(c) $GCD(u, u) = u$

5.3 Following the pattern in program (5.17), design a program to compute $z = x^y$ for the given natural numbers x and y. Include the necessary assertions to verify the correctness of your program.