反馈与讨论

5.4. (a) Modify Figure 2.4, so that if an employee has several direct managers then the employee's salary is added to the accumulated sum if it is higher than that of at least one of those direct managers. Call the new algorithm CA.
(b) Prove the total correctness of algorithm CA with respect to this new specification.
(c) Assuming that each employee has at most one manager, is algorithm CA correct with respect to the original specification given for Figure 2.4?
(d) Assuming that each employee has at most one manager, is the algorithm of Figure 2.4 correct with respect to the new specification for algorithm CA?

To facilitate precise treatment of the correctness problem for algorithms, researchers distinguish between two kinds of correctness, depending upon whether termination is or is not included. In one case it is assumed *a priori* that the program terminates and in the other it is not. More precisely, it is said that an algorithm A is **partially correct** (with respect to its definition of legal inputs and desired relationship with outputs) if, for every legal input X, if A terminates when run on X then the specified relationship holds between X and the resulting output set. Thus, a partially correct sorting algorithm might not terminate on all legal lists, but whenever it does, a correctly sorted list is the result. We say that A **terminates** if it halts when run on any one of the legal inputs. Both these notions taken together—partial correctness and termination—yield a **totally correct** algorithm, which correctly solves the algorithmic problem for every legal input: the process of running A on any such input X indeed terminates and produces outputs satisfying the desired relationship (see Figure 5.3).

Figure 5.3

Partial and total correctness.

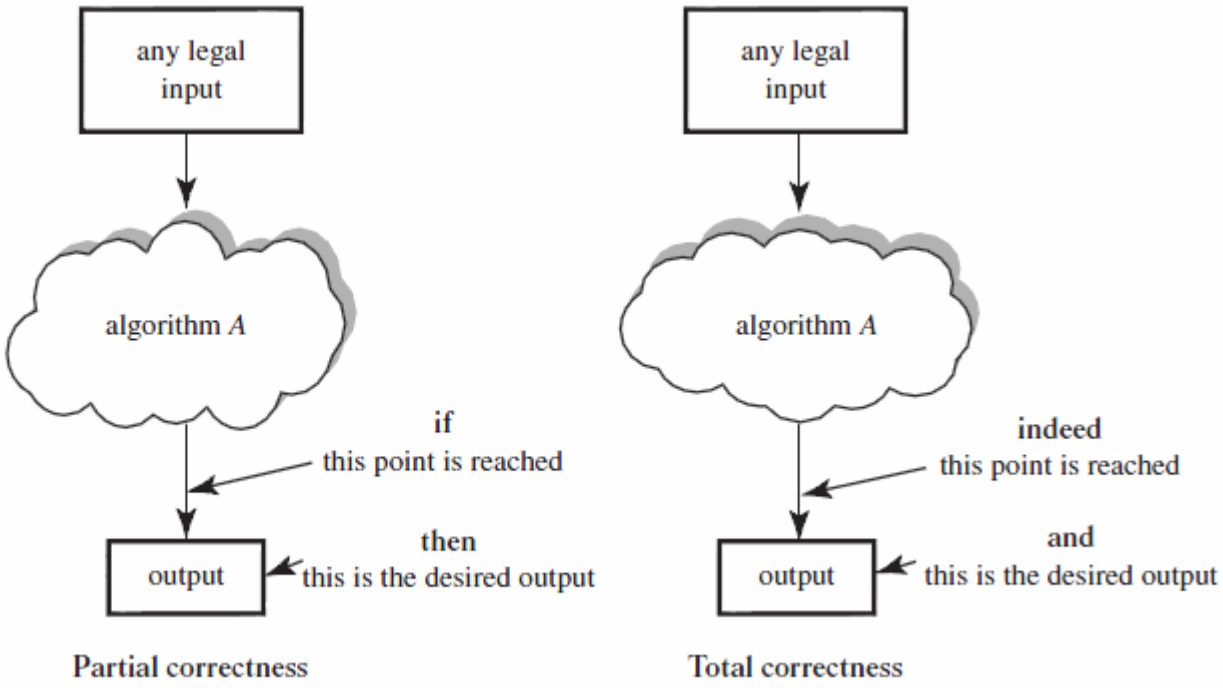
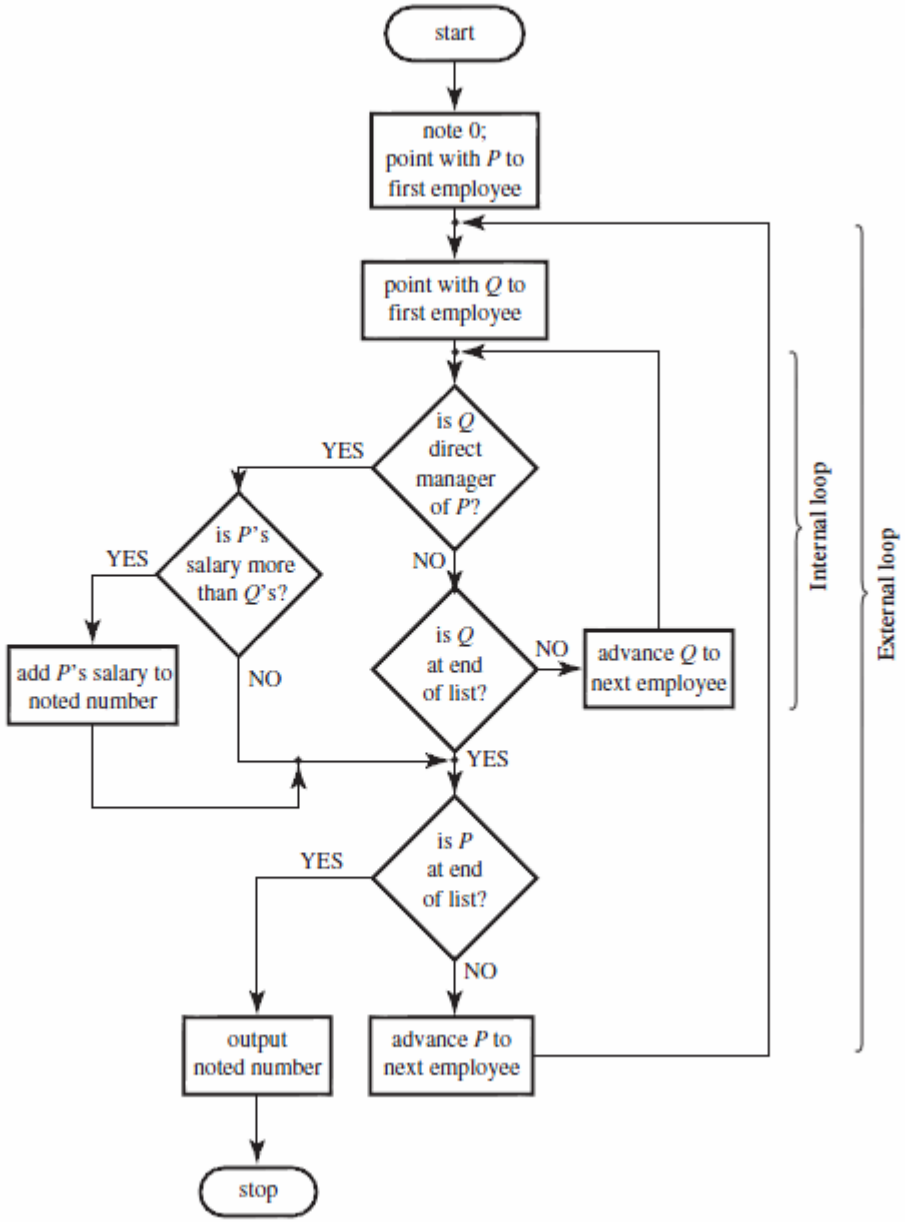


Figure 2.4

Flowchart for sophisticated salary summation.



5.6. We have seen in the correctness demonstration for the **reverse** algorithm that only three (well-placed) invariants are sufficient for the proof.

(a) How would you generalize the above claim to any algorithm whose structure (i.e., the structure of its flowchart) is similar to **reverse**?

(b) What kind of flowchart enables a partial correctness proof of the corresponding algorithm with only two invariants, attached to the **start** and **stop** points?

(c) How many well-attached invariants are sufficient for proving the partial correctness of an algorithm whose flowchart contains two loops?

(d) For any flowchart with two loops, how many invariants are necessary for proving partial correctness in the method given in the text? How would you classify the sufficient number of assertions according to the structure of a two-loop flowchart? (Hint: consider connectedness, nesting.)

5.8. Construct a function **rev**(X) that reverses the string X, using only the operations **last**, **all-but-last**, **eq**, and "$\cdot$" (concatenation), in addition to testing the emptiness of a string. Prove the total correctness of your algorithm.

5.9. Construct a function **equal**(X, Y) that tests whether the strings X and Y are equal. It should return true or false accordingly. You may use the operations mentioned in the text and those defined above. (Note, however, that there is no way of comparing arbitrarily long strings in a single operation.) Prove the total correctness of your algorithm.

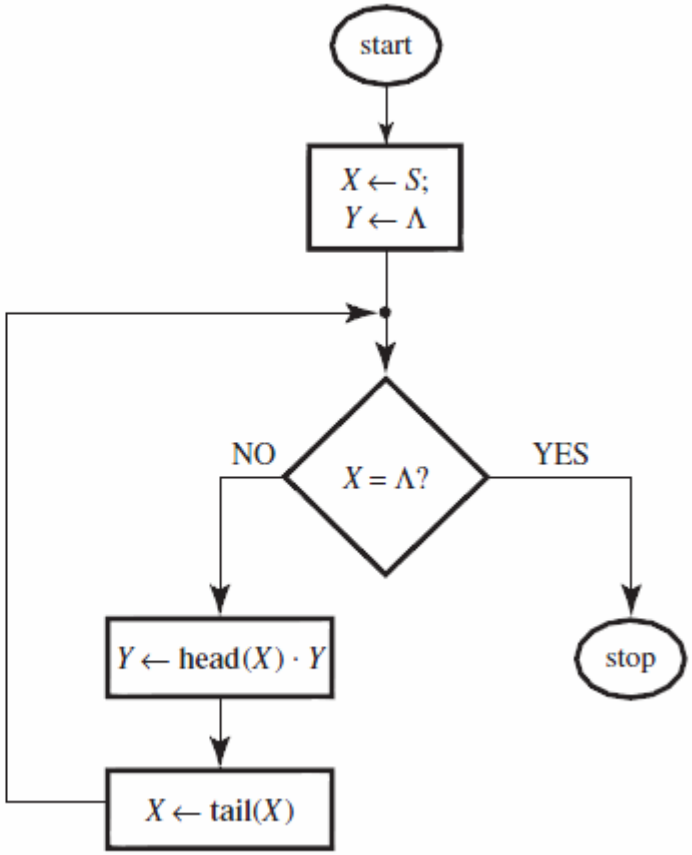
5.10. A palindrome is a string that is the same when read forwards and backwards. Consider the following algorithm *Pal1* for checking whether a string is a palindrome. The algorithm returns true or false, according to whether the input string S is a palindrome or not.

$Y \leftarrow$ **rev**(S);
return **equal**(S, Y).

(a) Prove the total correctness of *Pal1*.
(b) The termination of *Pal1* can be easily proved by relying on the termination of the functions **rev** and **equal**. Can you generalize this type of reasoning to a similar composition of any two programs?

Figure 5.6

A flowchart for reversing a symbol string.



5.11. Algorithm *Pal1* is not very efficient, as it always reverses entire strings. Often, a string can be shown not to be a palindrome with far fewer applications of the allowed basic operations.

(a) Give an example of such a string.
(b) How many operations would suffice to determine whether your string is a palindrome?

5.12. Here is another algorithm, *Pal2*, designed to perform the same task as *Pal1*, but more efficiently:

$X \leftarrow S$;

5.13. The following algorithm, *Pal3*, is another attempt to improve *Pal1*:

$X \leftarrow S$;
$E \leftarrow$ true;
while $X \neq \Lambda$ and E is true do the following:
  $Y \leftarrow$ **tail**(X);
  if **eq**(**head**(X), **last**(Y)) then $X \leftarrow$ **all-but-last**(Y);
  otherwise $E \leftarrow$ false.
return E.

Pal3 is not totally correct either. Prove or disprove:

(a) *Pal3* is partially correct.
(b) *Pal3* terminates on every input string S.

5.14. (a) Construct a correct solution to the problem of checking efficiently whether a string is a palindrome, following the general ideas of *Pal2* and *Pal3*. Call your algorithm *Pal4*.
(b) Prove the total correctness of *Pal4*.
(c) Explain why the string you gave in Exercise 5.11 to exhibit the inefficiency of *Pal1* is no longer a "bad" example for the efficiency of *Pal4*.

For any two positive integer numbers m and n, denote by m^n the number m raised to the power of n. For example, $2^3 = 2 \times 2 \times 2 = 8$, and $3^2 = 3 \times 3 = 9$. Also, $m^1 = m$ and $1^n = 1$, for every positive m and n. In addition, for $n = 0$, we define $m^0 = 1$ for every positive integer m.