McGRAW-HILL COMPUTER SCIENCE SERIES

RICHARD W. HAMMING
Bell Telephone Laboratories

EDWARD A. FEIGENBAUM
Stanford University

Bell and Newell *Computer Structures: Readings and Examples*
Cole *Introduction to Computing*
Donovan *Systems Programming*
Gear *Computer Organization and Programming*
Givone *Introduction to Switching Circuit Theory*
Hamming *Computers and Society*
Hamming *Introduction to Applied Numerical Analysis*
Hellerman *Digital Computer System Principles*
Kain *Automata Theory: Machines and Languages*
Kohavi *Switching and Finite Automata Theory*
Liu *Introduction to Combinatorial Mathematics*
Madnick and Donovan *Operating Systems*
Manna *Mathematical Theory of Computation*
Newman and Sproull *Principles of Interactive Computer Graphics*
Nilsson *Artificial Intelligence*
Ralston *Introduction to Programming and Computer Science*
Rosen *Programming Systems and Languages*
Salton *Automatic Information Organization and Retrieval*
Stone *Introduction to Computer Organization and Data Structures*
Watson *Timesharing System Design Concepts*
Wegner *Programming Languages, Information Structures, and Machine Organization*

ZOHAR MANNA

Applied Mathematics Department
Weizmann Institute of Science
Rehovot, Israel

Mathematical Theory of Computation

McGRAW-HILL BOOK COMPANY
New York St. Louis San Francisco Düsseldorf Johannesburg
Kuala Lumpur London Mexico Montreal New Delhi
Panama Paris São Paulo Singapore Sydney Tokyo Toronto

Prob. 3-22† The following program (in-place permutation) rearranges the elements of an array $S[1], S[2], \ldots, S[n]$, $n \geqq 1$, using a given permutation function $f(i)$ over the integers $1, 2, \ldots, n$; that is, the program rearranges the elements of S without using extra storage in such a way that $S_{\text{output}}[i] = S_{\text{input}}[f(i)]$ for $1 \leqq i \leqq n$. Prove that the program is totally correct wrt

$\varphi(n, f, S_{\text{input}})$: $n \geqq 1 \wedge$ (f is a permutation function over $1, 2, \ldots, n$)

nd

$$\psi(n, f, S_{\text{input}}, S_{\text{output}}): \quad \forall i(1 \leqq i \leqq n \supset S_{\text{output}}[i] = S_{\text{input}}[f(i)]).$$

START
or $k \leftarrow 1$ **step** 1 **until** n **do**
begin $l \leftarrow f(k)$;
while $l < k$ **do** $l \leftarrow f(l)$;
$(S[k], S[l]) \leftarrow (S[l], S[k])$
end
HALT

ee A. J. W. Duijvestijn, "Correctness Proof of an In-place Permutation," *BIT*, **12**(3): 318–324 972).

CHAPTER 4

Flowchart Schemas

Introduction

A *flowchart schema* is a flowchart program in which the domain of the variables is not specified (it is indicated by the symbol D) and the functions and predicates are left unspecified (they are denoted by the function symbols $f_1, f_2, \ldots$, and the predicate symbols $p_1, p_2, \ldots$). Thus a flowchart schema may be thought of as representing a family of flowchart programs. A program of the family is obtained by providing an interpretation for the symbols of the schema, i.e., a specific domain for D and specific functions and predicates for the symbols f_i and p_i. In Section 4-1 we introduce the basic notions and results regarding flowchart schemas.

In Sec. 4-2 we discuss the application of the theory of flowchart schemas for proving properties of programs. It often turns out that properties of a given program can be proved independent of the exact meaning of its functions and predicates; only the control structure of the program is really important in this case. Once the properties are proved for a schema, they apply immediately to all programs that can be obtained by specifying interpretations to the schema.

Flowchart programs are obtained from a given flowchart schema by specifying an interpretation in much the same way as interpreted wffs are obtained from a wff in the predicate calculus. In Sec. 4-3 we discuss first the relation between interpreted wffs of the predicate calculus and

flowchart programs, and then the relation between (uninterpreted) wffs and flowchart schemas.

Essentially, a flowchart schema depicts the control structure of the program, leaving much of the details to be specified in an interpretation. This leads to a better understanding of program features because it allows for the separation of the effect of the specific properties of a given domain and the control mechanism used in the program. In Sec. 4-4 this idea is used to show that *recursion is more powerful than iteration*. For this purpose we define a class of recursive schemas and prove that every flowchart schema can be translated into an equivalent recursive schema but not vice versa.

4-1 BASIC NOTIONS

4-1.1 Syntax

An *alphabet* Σ_S of a flowchart schema S is a finite subset of the following set of symbols.

1. Constants:
   n-ary *functions constants* f_i^n $(i \geq 1, n \geq 0)$; f_i^0 is called an *individual constant* and is denoted by a_i.
   n-ary *predicate constants* p_i^n $(i \geq 1, n \geq 0)$; p_i^0 is called a *propositional constant*.

2. Individual variables:

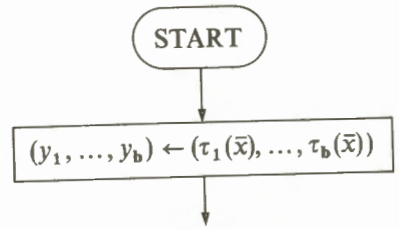
Input variables	$x_i\,(i \geq 1)$
Program variables	$y_i\,(i \geq 1)$
Output variables	$z_i\,(i \geq 1)$

The number of input variables $\bar{x}$, program variables $\bar{y}$, and output variables $\bar{z}$, in Σ_S is denoted by **a**, **b**, and **c**, respectively, where **a**, **b**, **c** ≥ 0. The subscripts of the symbols are used for enumerating the symbols and will be omitted whenever their omission causes no confusion. The superscripts of f_i^n and p_i^n are used only to indicate the number of arguments and therefore will always be omitted.

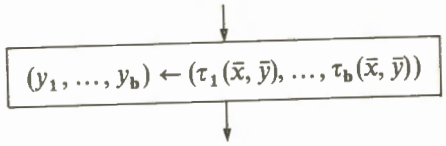
A *term* τ *over* Σ_S is constructed in the normal sense by composing individual variables, individual constants, and function constants of Σ_S. An *atomic formula* A *over* Σ_S is a propositional constant p_i^0 or an expression of the form $p_i^n(t_1, \ldots, t_n)$, $n \geq 1$, where $t_1, \ldots, t_n$ are terms over Σ_S. We shall write $\tau(\bar{x})$ and $A(\bar{x})$ to indicate that the term τ and the atomic formula A contain no individual variables other than members of $\bar{x}$; similarly, we shall write $\tau(\bar{x}, \bar{y})$ and $A(\bar{x}, \bar{y})$ to indicate that the term τ and the atomic formula A contain no individual variables other than members of $\bar{x}$ and $\bar{y}$.

A *statement over* Σ_S is of one of the following five forms.

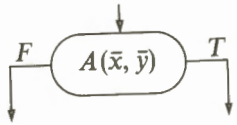
1. START statement:



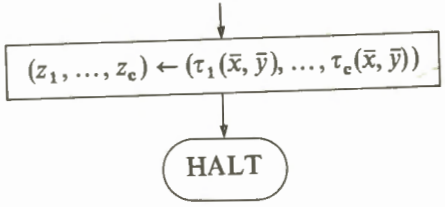
2. ASSIGNMENT statement:



3. TEST statement:



4. HALT statement:



5. LOOP statement:

where all the terms and atomic formulas are over Σ_S.

A *flowchart schema* S over alphabet Σ_S (called *schema*, for short) is a finite flow-diagram constructed from statements over Σ_S, with one START statement, such that every ASSIGNMENT or TEST statement is on a path from the START statement to some HALT or LOOP statement.

EXAMPLE 4-1

In the sequel we shall discuss the schema S_1 described in Fig. 4-1. Σ_{S_1} consists of the individual constant a, the unary function constant f_1, the binary function constant f_2, the unary predicate constant p, the input variable x, the program variables y_1 and y_2, and the output variable z.

□

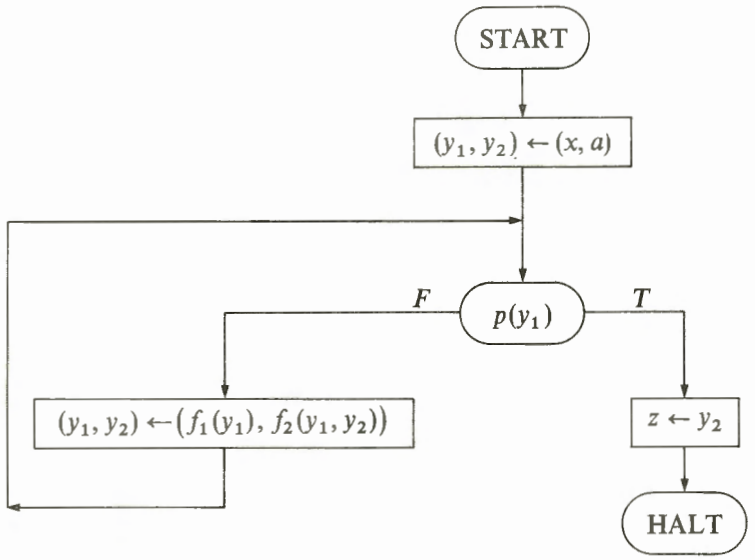


Figure 4-1 Schema S_1.

4-1.2 Semantics (Interpretations)

An *interpretation* $\mathscr{I}$ of a flowchart schema S consists of

1. A nonempty set of elements D (called the *domain* of the interpretation).
2. Assignments to the constants of Σ_S:
   (*a*) To each function constant f_i^n in Σ_S we assign a total function mapping D^n into D (if $n = 0$, the individual constant f_i^0 is assigned some fixed element of D).
   (*b*) To each predicate constant p_i^n in Σ_S we assign a total predicate mapping D^n into $\{true, false\}$ (if $n = 0$, the propositional constant p_i^0 is assigned the truth value *true* or *false*).

The pair $P = \langle S, \mathscr{I} \rangle$, where S is a flowchart schema and $\mathscr{I}$ is an interpretation of S, is called a *flowchart program* (or *program*, for short). Given initial values $\bar{\xi} \in D^{\mathbf{a}}$ for the input variables $\bar{x}$ of S, the program can be executed.

The *computation of* $\langle S, \mathscr{I}, \bar{\xi} \rangle$ proceeds in the normal sense, starting from the START statement, with $\bar{x} = \bar{\xi}$. The values of $(y_1, \ldots, y_{\mathbf{b}})$ are initialized in the START statement to $(\tau_1(\bar{\xi}), \ldots, \tau_{\mathbf{b}}(\bar{\xi}))$.† Note that if an ASSIGNMENT statement of the form $(y_1, \ldots, y_{\mathbf{b}}) \leftarrow (\tau_1(\bar{x}, \bar{y}), \ldots, \tau_{\mathbf{b}}(\bar{x}, \bar{y}))$ is reached with $\bar{y} = \bar{\eta}$ for some $\bar{\eta} \in D^{\mathbf{b}}$, the execution of the statement results in $(y_1, \ldots, y_{\mathbf{b}}) = (\tau_1(\bar{\xi}, \bar{\eta}), \ldots, \tau_{\mathbf{b}}(\bar{\xi}, \bar{\eta}))$; in other words, the new values of $y_1, \ldots, y_{\mathbf{b}}$ are computed simultaneously. Thus, for example, to interchange the values of y_1 and y_2, we can write simply $(y_1, y_2) \leftarrow (y_2, y_1)$. The computation terminates as soon as a HALT statement is executed or a LOOP statement is reached: In the first case, if the execution of the HALT statement results in $\bar{z} = \bar{\zeta} \in D^{\mathbf{c}}$, we say that $val\langle S, \mathscr{I}, \bar{\xi} \rangle$ is defined and $val\langle S, \mathscr{I}, \bar{\xi} \rangle = \bar{\zeta}$; in the second case (i.e., if the computation reaches a LOOP statement) or if the computation never terminates, we say that $val\langle S, \mathscr{I}, \bar{\xi} \rangle$ is undefined. Thus a program P represents a partial function mapping $D^{\mathbf{a}}$ into $D^{\mathbf{c}}$.

EXAMPLE 4-2

Consider the schema S_1 (Figure 4-1) with the following interpretations.

1. Interpretation $\mathscr{I}_A$: $D = \{$the natural numbers$\}$; a is 1; $f_1(y_1)$ is $y_1 - 1$; $f_2(y_1, y_2)$ is $y_1 y_2$; and $p(y_1)$ is $y_1 = 0$. The program $P_A = \langle S_1, \mathscr{I}_A \rangle$, represented in Figure 4-2*a* clearly computes the factorial function; i.e., $z = factorial(x)$.

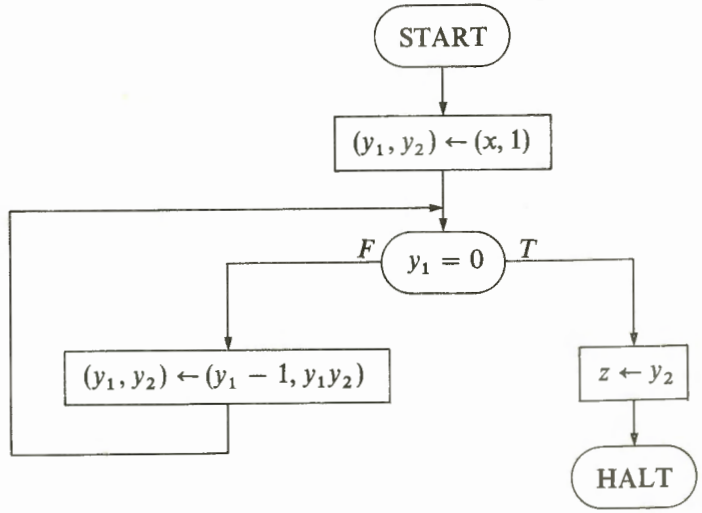


Figure 4-2*a* Program $P_A = \langle S_1, \mathscr{I}_A \rangle$ for computing $z = factorial(x)$.

† In some of the examples and problems in this chapter we do not initialize all the program variables in the START statement; however, the program variables are always assigned initial values before they are first needed.

2. Interpretation $\mathcal{I}_B$: This uses as domain the set Σ^* of all finite strings over the finite alphabet $\Sigma = \{A, B, \ldots, Z\}$, including the empty string Λ, thus $D = \{A, B, \ldots, Z\}^*$; a is Λ (the empty string); $f_1(y_1)$ is $tail(y_1)$; $f_2(y_1, y_2)$ is $head(y_1) \cdot y_2$; and $p(y_1)$ is $y_1 = \Lambda$. The program $P_B = \langle S_1, \mathcal{I}_B \rangle$ represented in Figure 4-2*b* clearly reverses the order of letters in a given string; i.e., $z = reverse(x)$. □

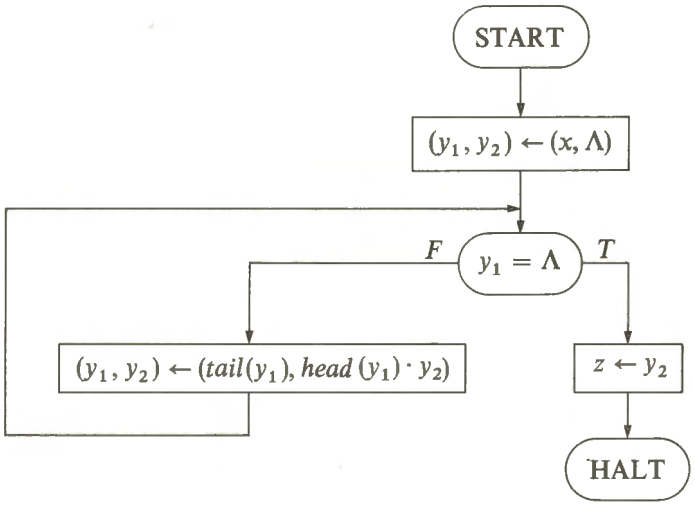


Figure 4-2*b* Program $P_B = \langle S_1, \mathcal{I}_B \rangle$ for computing $z = reverse(x)$.

EXAMPLE 4-3

Consider the schema S_3 (Fig. 4-3), where $y_1 \leftarrow f(y_1)$ abbreviates $(y_1, y_2) \leftarrow (f(y_1), y_2)$. Note that S_3 does not contain any input variable; thus, for a given interpretation $\mathcal{I}$ of S_3, the program $\langle S_3, \mathcal{I} \rangle$ can be executed without indicating any input value. Therefore, in this case we shall discuss the value of $val\langle S_3, \mathcal{I} \rangle$ rather than $val\langle S_3, \mathcal{I}, \bar{\xi} \rangle$. S_3 contains a dummy output variable (it is always assigned the individual constant a) because in this example we would like to know just whether or not $val\langle S_3, \mathcal{I} \rangle$ is defined rather than the value of $val\langle S_3, \mathcal{I} \rangle$.

Let us consider the following interpretation $\mathcal{I}^*$ of S_3:

1. The domain D consists of all strings of the form: $a, f(a), f(f(a)), f(f(f(a))), \ldots$. For clarity we enclose the strings within quotation marks and write

$$\text{“}a\text{”}, \quad \text{“}f(a)\text{”}, \quad \text{“}f(f(a))\text{”}, \quad \text{“}f(f(f(a)))\text{”}, \quad \ldots$$

† Here, $f^{(0)}(a)$ stands for a; $f^{(1)}(a)$ for $f(a)$; $f^{(2)}(a)$ for $f(f(a))$; $f^{(3)}(a)$ for $f(f(f(a)))$; and so forth.

2. The individual constant a of Σ_{S_3} is assigned the element “a” of D.

3. The unary function constant f of Σ_{S_3} is assigned a unary function mapping D into D as follows:† Any element “$f^{(n)}(a)$” of D ($n \geqq 0$) is mapped into “$f^{(n+1)}(a)$”, which is also an element of D.

4. The unary predicate constant p of Σ_{S_3} is assigned a unary predicate mapping D into $\{true, false\}$ as follows: $p(\text{“}f^{(n)}(a)\text{”})$, $n \geqq 0$, takes the values (where t stands for *true* and f for *false*)

n:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$p(\text{“}f^{(n)}(a)\text{”})$:	f	f	t	f	t	t	f	t	t	t	f	t	t	t	t	f ...

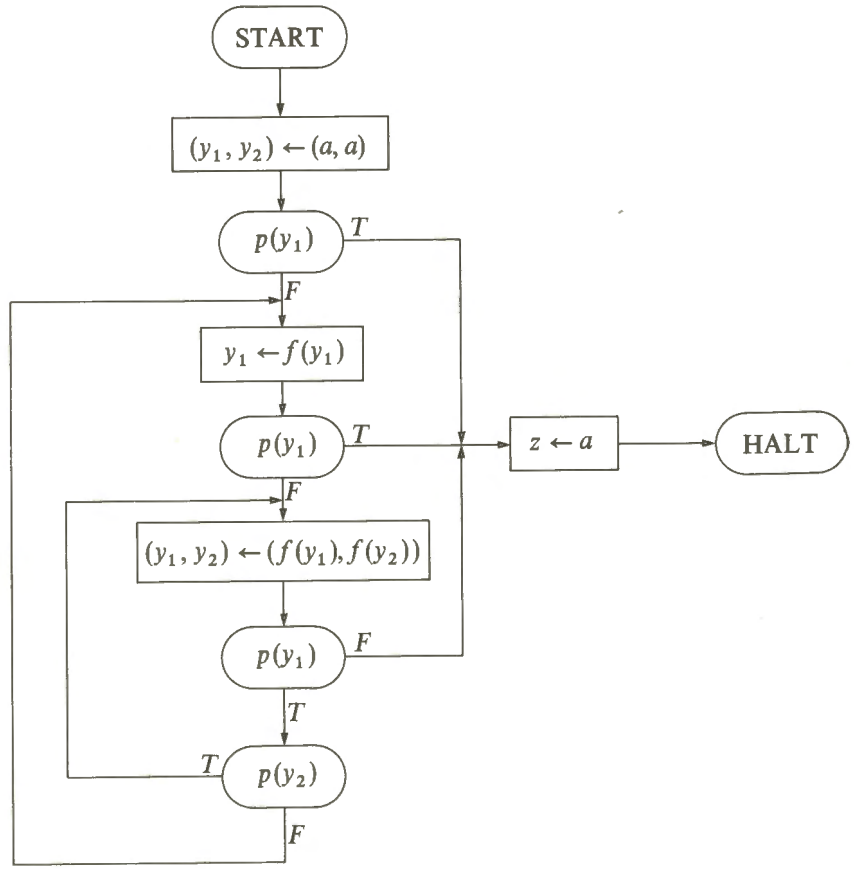


Figure 4-3 Schema S_3.

The computation of $\langle S_3, \mathscr{I}^* \rangle$ is best described by the following sequence of elements of $D \times D$ indicating the successive values of (y_1, y_2):

$$(\text{“}a\text{”}, \text{“}a\text{”}) \to (\text{“}f(a)\text{”}, \text{“}a\text{”}) \to (\text{“}f^{(2)}(a)\text{”}, \text{“}f(a)\text{”}) \to (\text{“}f^{(3)}(a)\text{”}, \text{“}f(a)\text{”})$$
$$\to (\text{“}f^{(4)}(a)\text{”}, \text{“}f^{(2)}(a)\text{”}) \to (\text{“}f^{(5)}(a)\text{”}, \text{“}f^{(3)}(a)\text{”})$$
$$\to (\text{“}f^{(6)}(a)\text{”}, \text{“}f^{(3)}(a)\text{”}) \to \cdots\cdots\cdots\cdots\cdots$$

It can be proved (by induction) that $val\langle S_3, \mathscr{I}^* \rangle$ is undefined. Interpretations similar to $\mathscr{I}^*$ are of special interest and will be discussed in Sec. 4-1.4.

Furthermore, it can be shown (see Prob. 4-1) that $val\langle S_3, \mathscr{I} \rangle$ is defined for every interpretation $\mathscr{I}$ with finite domain, and that it is also defined for every interpretation $\mathscr{I}$ with infinite domain unless $p(f^{(n)}(a))$, $n \geq 0$, takes the following values under $\mathscr{I}$:

$$f,\ f,\ \underbrace{t,}\ f,\ \underbrace{t,\ t,}\ f,\ \underbrace{t,\ t,\ t,}\ f,\ \underbrace{t,\ t,\ t,\ t,}\ f,\ \ldots$$

□

4-1.3 Basic Properties

After defining the syntax and the semantics of schemas, we shall now introduce a few basic properties of schemas which will be discussed in the rest of this chapter.

(A) Halting and divergence For a given program $\langle S, \mathscr{I} \rangle$ we say that

1. $\langle S, \mathscr{I} \rangle$ *halts* if for every input $\bar{\xi} \in D^a$, $val\langle S, \mathscr{I}, \bar{\xi} \rangle$ is defined.
2. $\langle S, \mathscr{I} \rangle$ *diverges* if for every input $\bar{\xi} \in D^a$, $val\langle S, \mathscr{I}, \bar{\xi} \rangle$ is undefined.

Note that a program $\langle S, \mathscr{I} \rangle$ may neither halt nor diverge if for some $\bar{\xi}_1 \in D^a$, $val\langle S, \mathscr{I}, \bar{\xi}_1 \rangle$ is defined and for some other $\bar{\xi}_2 \in D^a$, $val\langle S, \mathscr{I}, \bar{\xi}_2 \rangle$ is undefined. For a given schema S we say that

1. S *halts* if for every interpretation $\mathscr{I}$ of S, $\langle S, \mathscr{I} \rangle$ halts.
2. S *diverges* if for every interpretation $\mathscr{I}$ of S, $\langle S, \mathscr{I} \rangle$ diverges.

EXAMPLE 4-4

The schema S_4 in Fig. 4-4 halts (for every interpretation). It can be observed that (1) the LOOP statement can never be reached because whenever we reach TEST statement 1, $p(a)$ is *false*; and (2) we can never go through the loop more than once because whenever we reach TEST statement 2, either $p(a)$ is *true*, or $p(a)$ is *false* and $p(f(a))$ is *true*.

□

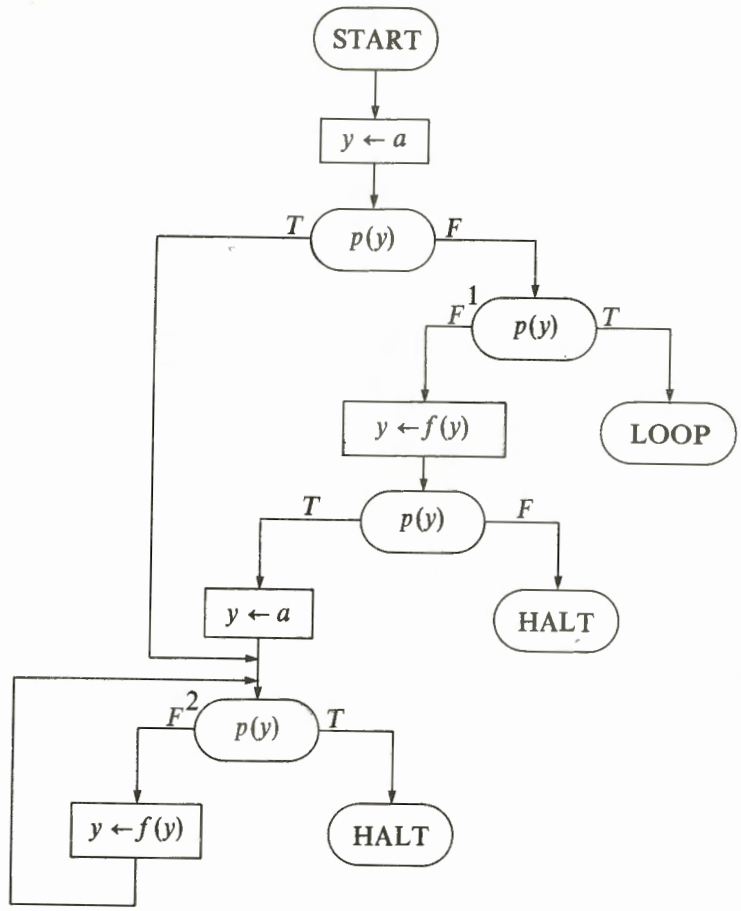


Figure 4-4 Schema S_4 that halts for every interpretation.

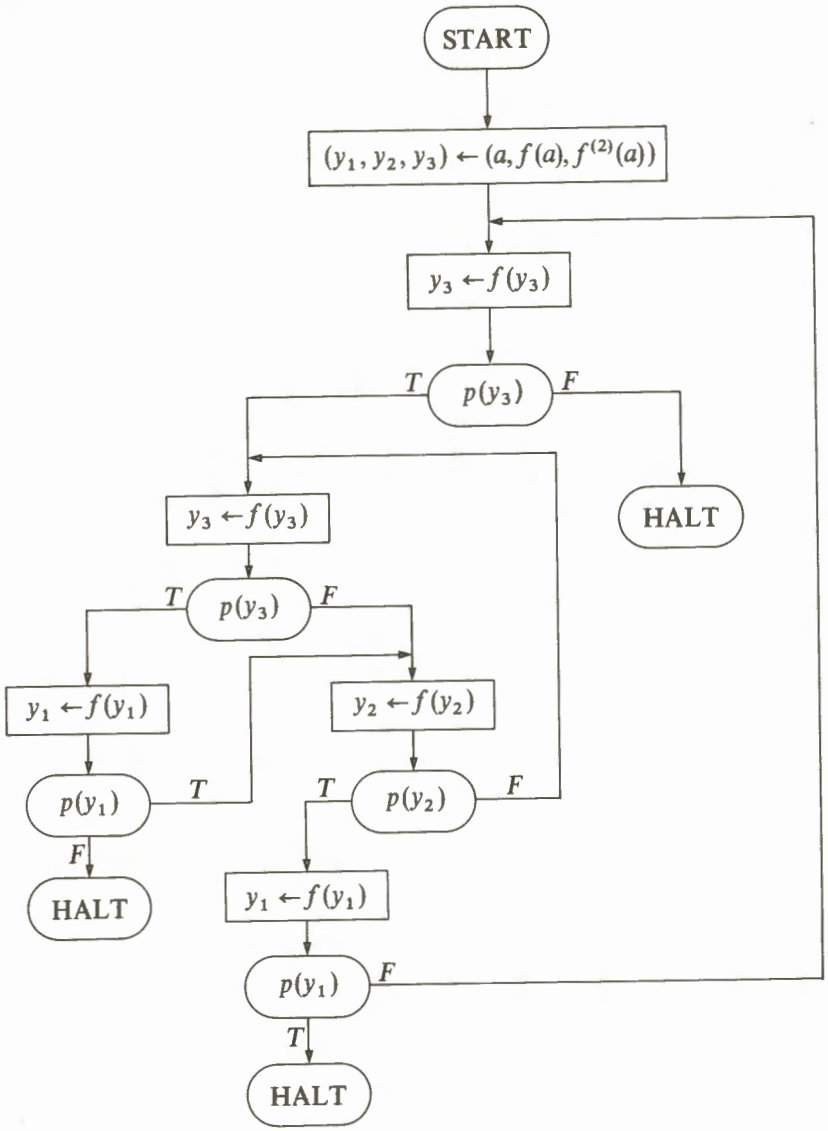


Figure 4-5 Schema S_5 that halts for every interpretation.

EXAMPLE 4-5 (Paterson)

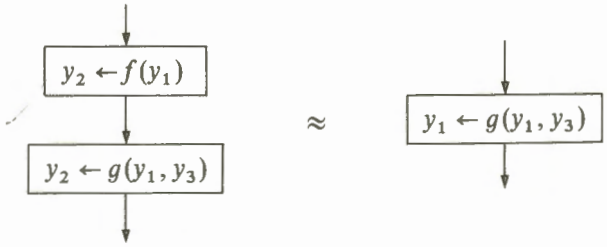
The schema S_5 in Fig. 4-5 also halts (for every interpretation); however, it is very hard to prove it because there are some interpretations for which we must test $p(f^{(140)}(a))$ [but we shall never test $p(f^{(i)}(a))$ where $i > 140$]. □

(B) Equivalence The next property of schemas we shall discuss is the *equivalence of two schemas*, which is certainly the most important relation between schemas. Two schemas S and S' are said to be *compatible* if they have the same vector of input variables $\bar{x}$ and the same vector of output variables $\bar{z}$. Two programs $\langle S, \mathscr{I} \rangle$ and $\langle S', \mathscr{I}' \rangle$ are said to be *compatible* if the schemas S and S' are compatible and the interpretations $\mathscr{I}$ and $\mathscr{I}'$ have the same domain.

For two given compatible programs $\langle S, \mathscr{I} \rangle$ and $\langle S', \mathscr{I}' \rangle$, we say that $\langle S, \mathscr{I} \rangle$ *and* $\langle S', \mathscr{I}' \rangle$ *are equivalent* [notation: $\langle S, \mathscr{I} \rangle \approx \langle S', \mathscr{I}' \rangle$], if for every input $\xi \in D^a$, $val\langle S, \mathscr{I}, \xi \rangle \equiv val\langle S', \mathscr{I}', \xi \rangle$†, i.e., either both $val\langle S, \mathscr{I}, \xi \rangle$ and $val\langle S', \mathscr{I}', \xi \rangle$ are undefined, or both $val\langle S, \mathscr{I}, \xi \rangle$ and $val\langle S', \mathscr{I}', \xi \rangle$ are defined and $val\langle S, \mathscr{I}, \xi \rangle = val\langle S', \mathscr{I}', \xi \rangle$.

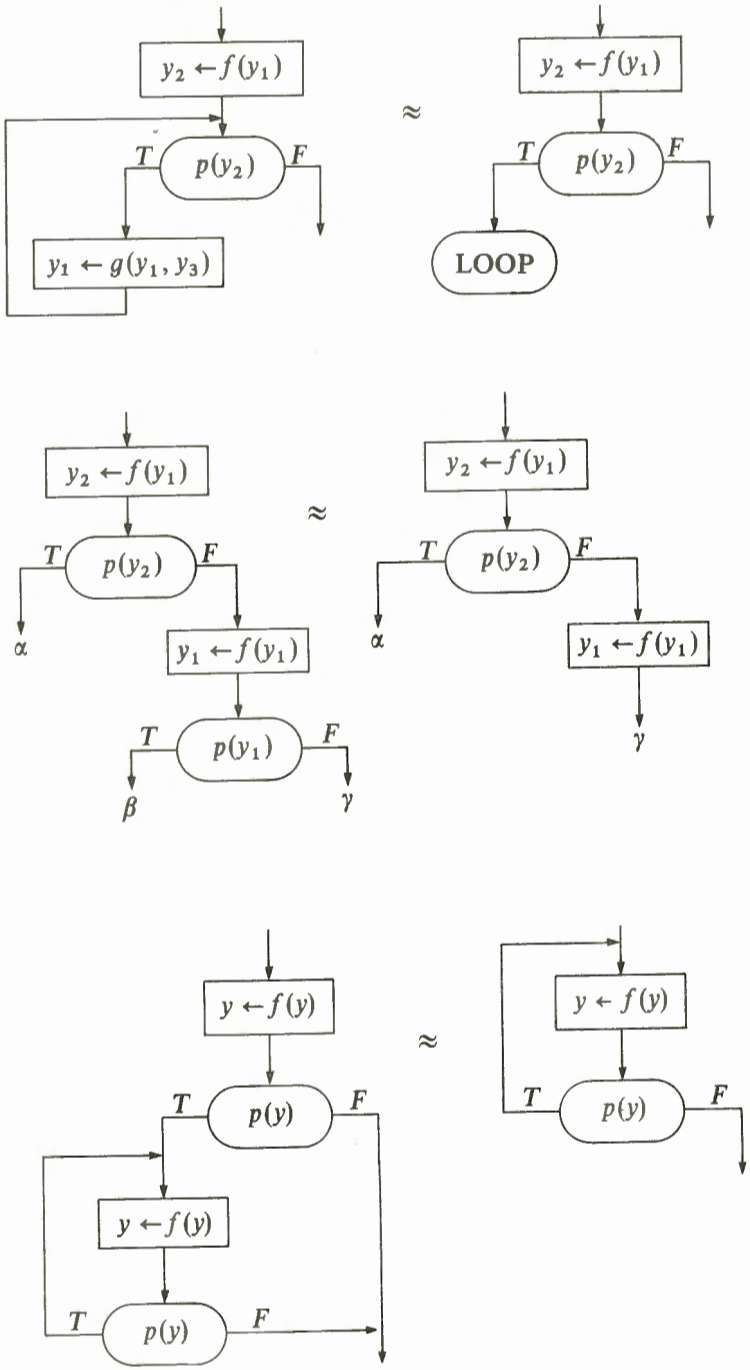
For two given compatible schemas S and S', we say that S *and* S' *are equivalent* [notation: $S \approx S'$] if for every interpretation‡ $\mathscr{I}$ of S and S', $\langle S, \mathscr{I} \rangle$ and $\langle S', \mathscr{I} \rangle$ are equivalent.

Note that the notion of equivalence is not only reflexive and symmetric but also transitive; i.e., it is really an equivalence relation. Thus one way to prove the equivalence of two schemas is by passing from one to the other by a chain of simple transformations, each of which obviously preserves equivalence. Some examples of such transformations are shown below. In each case it should be clear that equivalence is preserved.



† ≡ is the extension of the = relation for handling undefined values: It is *true* if either both arguments are undefined or both are defined and equal; otherwise it is *false*. (Thus, the value is *false* if only one of the arguments is undefined.)

‡ That is, $\mathscr{I}$ includes assignments to all constant symbols occurring in $\Sigma_S \cup \Sigma_{S'}$.



EXAMPLE 4-6 (Paterson)

We shall apply some of these transformations to the schema S_{6A} (Fig. 4-6*a*) to show that it is equivalent to the schema S_{6E} (Fig. 4-6*e*). We proceed in 4 steps (see Figs 4-6*b* to *e*).

Step 1: $S_{6A} \approx S_{6B}$. Consider the schema S_{6A}. Note first that if at some stage we take the F branch of statement 7 (that is, $p(y_4) = \mathit{false}$), the schema gets into an infinite loop because the value of y_4 is not changed in statements 3, 4, and 6; note also that after execution of statements 2 and 3, we have $y_1 = y_4$. Thus, the first time we take the F branch of statement 5, either we get into an infinite loop through statements 3 and 4 or, if we get out of the loop, we get into an infinite loop when $p(y_4)$ is tested. Hence, let us replace the F branch of statement 5 by a LOOP statement; now, whenever we reach statement 5, we have $y_1 = y_4$. Thus whenever we reach statement 7, we must take its T branch; therefore statement 7 can be removed. Since at statement 4 we always have $y_1 = y_4$, we can replace y_4 by y_1 in this statement. Now, since y_4 is not used any longer, we can remove statement 2. Finally, we introduce the extra test statement 8′ just by unwinding once the loop through statements 8 and 9.

Step 2: $S_{6B} \approx S_{6C}$. Consider the schema S_{6B}. Since y_2 and y_3 are assigned the same values at statement 4, we have $y_2 = y_3$ when we first test $p(y_2)$ in statement 8. Therefore, if we take the T branch of $p(y_2)$, we can go directly to statement 11; however, if we take the F branch of $p(y_2)$ and later take the T branch of statement 8′ we return to statement 3 [since $p(y_3)$ is *false* (y_3 has not been modified)]. Thus, statement 10 can be removed (as shown in Fig. 4-6*c*), and y_3 can be replaced by y_2 in statement 6. Finally, since y_3 is now redundant, it can be eliminated.

Step 3: $S_{6C} \approx S_{6D}$. Consider the schema S_{6C}. Leaving the inner loop (8′–9), the value of y_2 is not used in statement 3 while we reset y_2 in statement 4; thus, the value of y_2 created in statement 9 serves no purpose. This suggests that we can remove the inner loop; however, there is the chance that we could loop indefinitely through statements 9 and 8′. In this case, if we reach statement 8 with some value $y_2 = \eta$, then $p(f^{(i)}(\eta)) = \mathit{false}$ for all i and especially $p(f^{(2)}(\eta)) = \mathit{false}$. Now, if we remove the inner loop, in statement 6 we set $y_1 = f(\eta)$ and then in statement 3 we set $y_1 = f^{(2)}(\eta)$. Thus $p(y_1)$ in statement 5 will be *false* leading to the LOOP statement. Thus, since the only possible use of statements 9 and 8′ is covered by the LOOP statement, the inner loop can be removed. Finally, since the value

of y_2 is not used in statement 5, we can execute statement 4 after testing $p(y_1)$; similarly, since the value of y_1 is not used in statement 8, we can execute statement 6 after testing $p(y_2)$.

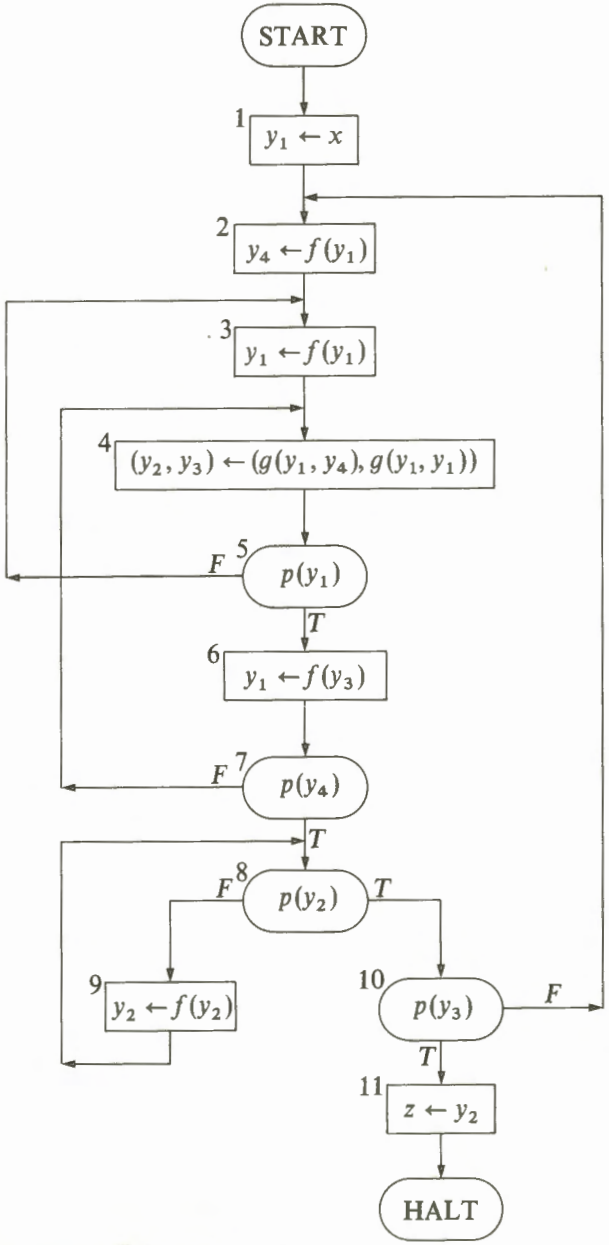


Figure 4-6*a* Schema S_{6A}.

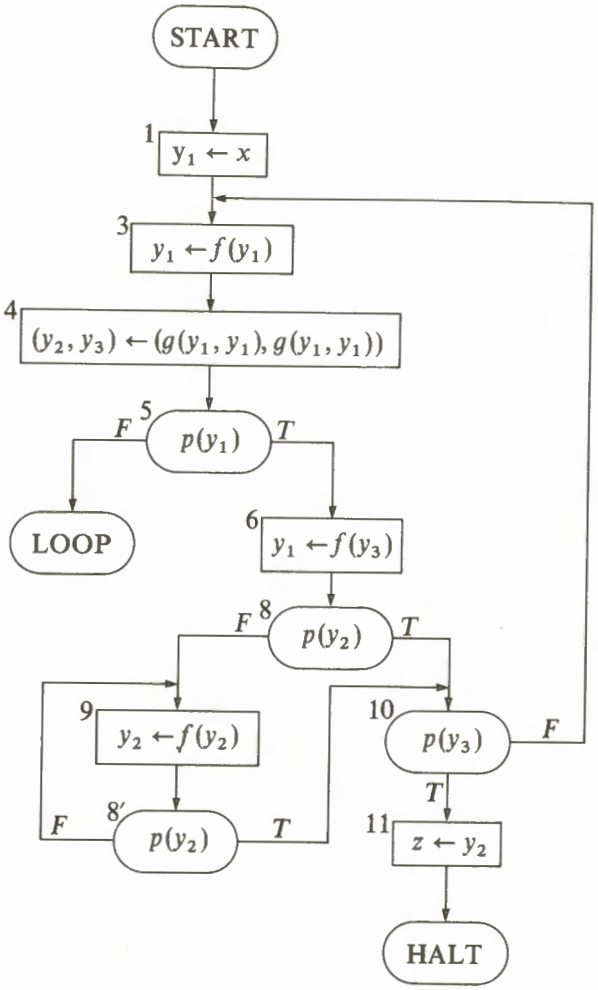


Figure 4-6*b* Schema S_{6B}.

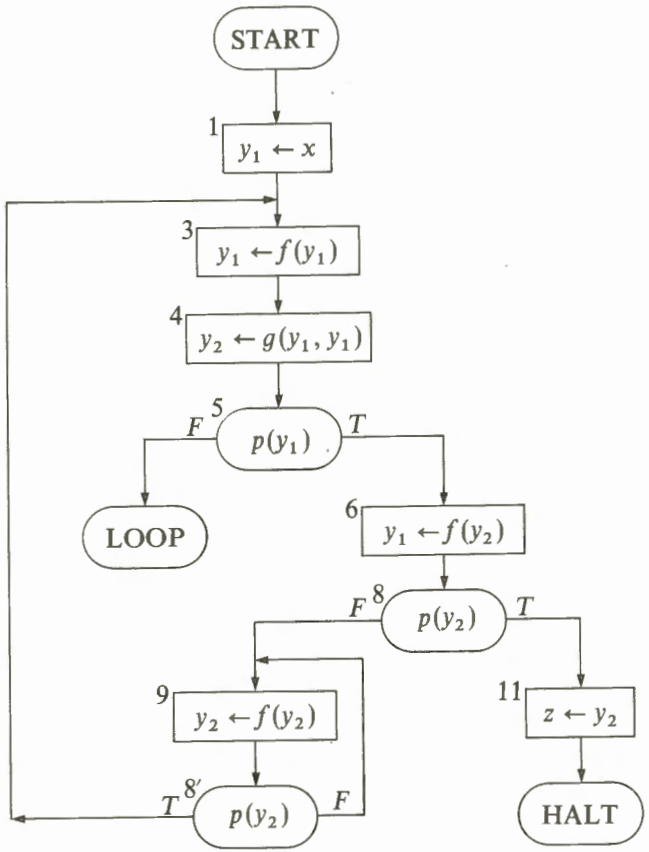


Figure 4-6*c* Schema S_{6C}.

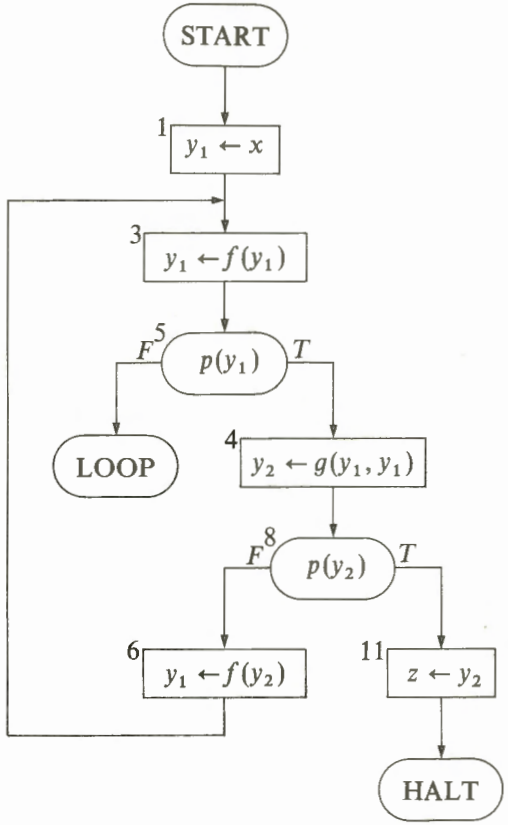


Figure 4-6*d* Schema S_{6D}.

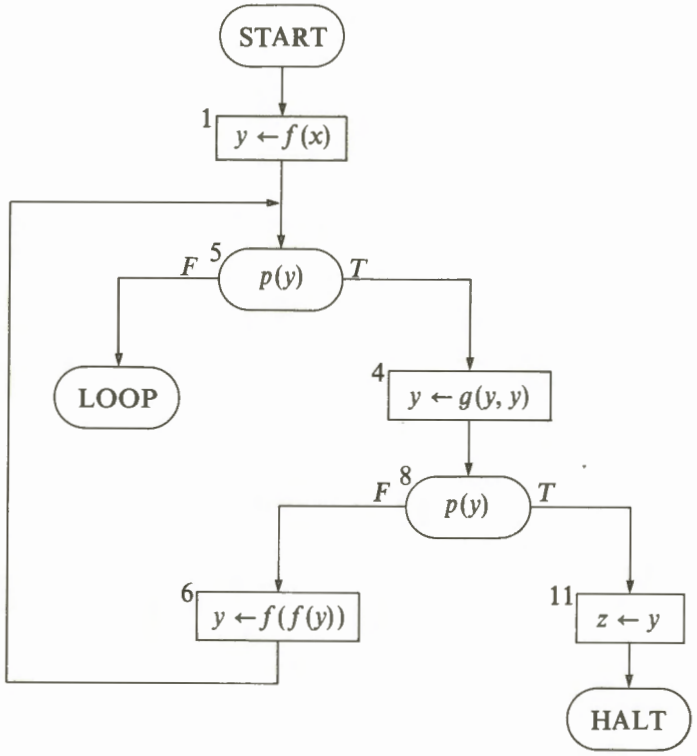


Figure 4-6e Schema S_{6E}

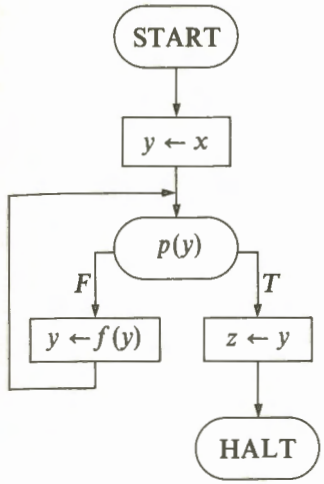
Step 4: $S_{6D} \approx S_{6E}$. Considering statements 4 and 6 in S_{6D}, we realize that y_2 is merely a dummy variable and can be replaced by y_1. Therefore, dropping the subscript and modifying statements 1, 3, and 6, we obtain S_{6E}. □

(C) Isomorphism Sometimes we would like to know, not only whether or not two schemas S and S' yield the same final value for the same interpretation, but also whether or not both schemas compute this value in the same manner. Therefore we introduce the stronger notion of equivalence called *isomorphism*. Two compatible schemas S and S' are said to be *isomorphic* (notation: $S \approx S'$) if for every interpretation $\mathcal{I}$ of S and S' and for every input $\bar{\xi} \in D^{\mathbf{a}}$, the (finite or infinite) sequence of statements executed in the computation of $\langle S, \mathcal{I}, \bar{\xi} \rangle$ is identical to the sequence of statements executed in the computation of $\langle S', \mathcal{I}, \bar{\xi} \rangle$.

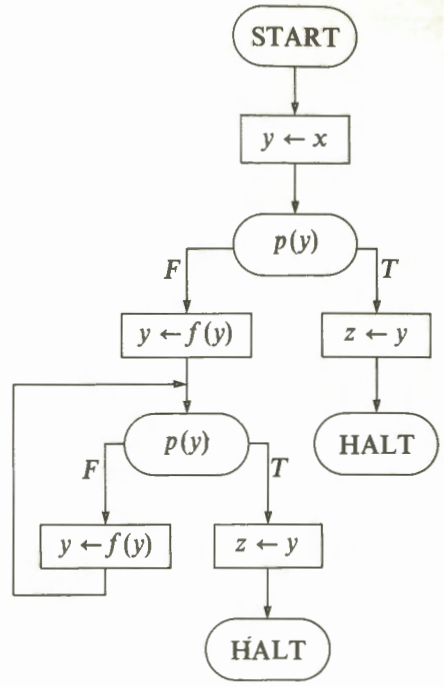
EXAMPLE 4-7

The three schemas S_{7A}, S_{7B}, and S_{7C} (Fig. 4-7) are compatible. It is clear that $S_{7A} \approx S_{7B} \approx S_{7C}$, but $S_{7A} \not\approx S_{7B} \not\approx S_{7C}$.

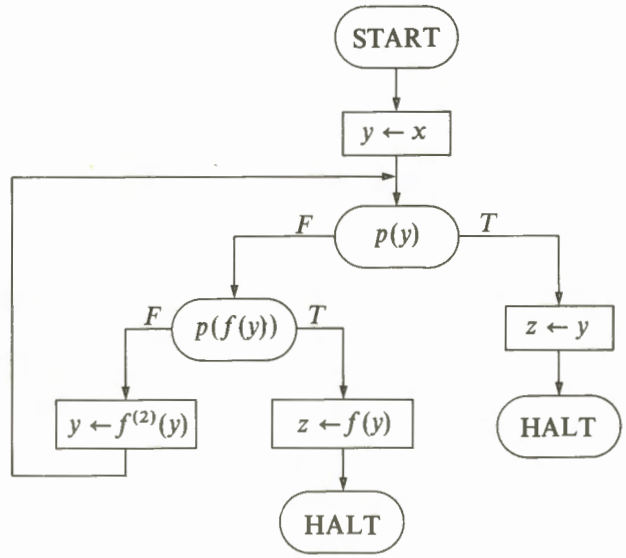
□



Schema S_{7A}



Schema S_{7B}



Schema S_{7C}

Figure 4-7

4-1.4 Herbrand Interpretations

The basic properties of schemas, such as halting, divergence, equivalence, or isomorphism, depend by definition on their behavior for all interpretations (over all domains). Clearly it would be of great help in proving properties of schemas if we could fix on one special domain such that the behavior of the schemas for all interpretations over this domain characterize their behavior for all interpretations over any domain. Fortunately, for any schema S, there does exist such a domain: It is called the *Herbrand universe of* S and is denoted by H_S. H_S consists of all strings of the following form: If x_i is an input variable and a_i is an individual constant occurring in S, then "x_i" and "a_i" are in H_S; if f_i^n is an n-ary function constant occurring in S and "t_1", "t_2", . . . , "t_n" are elements of H_S, then so is "$f_i^n(t_1, \ldots, t_n)$".

EXAMPLE 4-8

For the schema S_1 (Fig. 4-1), H_{S_1} consists of the strings

"a", "x", "$f_1(a)$", "$f_1(x)$", "$f_2(a, a)$", "$f_2(a, x)$", "$f_2(x, a)$", "$f_2(x, x)$",

"$f_1(f_1(a))$", . . .

For the schema S_3 (Fig. 4-3), H_{S_3} consists of the strings

"a", "$f(a)$", "$f(f(a))$", "$f(f(f(a)))$", . . . □

Now, for any given schema S, an interpretation over the Herbrand universe H_S of S consists of assignments to the constants of S as follows: To each function constant f_i^n which occurs in S, we assign an n-ary function over H_S (in particular, each individual constant a_i is assigned some element of H_S); and to each predicate constant p_i^n which occurs in S, we assign an n-ary predicate over H_S (in particular, each propositional constant is assigned the truth value *true* or *false*). Among all these interpretations over H_S, we are interested in a special subclass of interpretations called *Herbrand interpretations of* S which satisfy the following conditions: Each individual constant a_i in S is assigned the string "a_i" of H_S; and each constant function f_i^n occurring in S is assigned the n-ary function over H_S which maps the strings "t_1", "t_2", . . . , and "t_n" of H_S into the string "$f_i^n(t_1, t_2, \ldots, t_n)$" of H_S. (Note that there is no restriction on the assignments to the predicate constants of S.)

EXAMPLE 4-9

The interpretation $\mathcal{I}^*$ described in Example 4-3 is a Herbrand interpretation of the schema S_3. □

Note that H_S contains the strings "x_i" for all input variables x_i occurring in S. We let "$\bar{x}$" denote the vector of strings ("x_1", "x_2", . . . , "x_n"). In general, among all possible computations of a schema S with Herbrand interpretation $\mathcal{I}^*$, the most interesting computation is $\langle S, \mathcal{I}^*, \text{"}\bar{x}\text{"}\rangle$, that is, the one in which the strings "x_i" of H_S are assigned to the input variables x_i. The reason is that for any interpretation $\mathcal{I}$ of a schema S and input $\bar{\xi} \in D^n$, there exists a Herbrand interpretation $\mathcal{I}^*$ such that the (finite or infinite) sequence of statements executed in the computation of $\langle S, \mathcal{I}^*, \text{"}\bar{x}\text{"}\rangle$ is identical to the sequence of statements executed in the computation of $\langle S, \mathcal{I}, \bar{\xi}\rangle$.

The appropriate Herbrand interpretation $\mathcal{I}^*$ of S is obtained by defining the truth value of p_i^n("τ_1", . . . , "τ_n") as follows: Suppose that under interpretation $\mathcal{I}$ and input $\bar{\xi}$, $(\tau_1, \ldots, \tau_n) = (d_1, \ldots, d_n)$ where $d_i \in D$. Then, if $p(d_1, \ldots, d_n) = \textit{true}$, we let $p(\text{"}\tau_1\text{"}, \ldots, \text{"}\tau_n\text{"}) = \textit{true}$ under interpretation $\mathcal{I}^*$; and if $p(d_1, \ldots, d_n) = \textit{false}$, we let $p(\text{"}\tau_1\text{"}, \ldots, \text{"}\tau_n\text{"}) = \textit{false}$. This implies that many properties of schemas can be described and proved by considering just Herbrand interpretations rather than all interpretations, as suggested by the following theorem.

THEOREM 4-1 (Luckham-Park-Paterson).

(1) *For every schema S, S halts/diverges if and only if $val\langle S, \mathcal{I}^*, \text{"}\bar{x}\text{"}\rangle$ is defined/undefined for every Herbrand interpretation $\mathcal{I}^*$ of S.*

(2) *For every pair of compatible schemas S and S', S and S' are equivalent if and only if $val\langle S, \mathcal{I}^*, \text{"}\bar{x}\text{"}\rangle \equiv val\langle S', \mathcal{I}^*, \text{"}\bar{x}\text{"}\rangle$ for every Herbrand interpretation $\mathcal{I}^*$ of S and S'.*

(3) *For every pair of compatible schemas S and S', S and S' are isomorphic if and only if for every Herbrand interpretation $\mathcal{I}^*$ of S and S' the sequence of statements executed in the computation of $\langle S, \mathcal{I}^*, \text{"}\bar{x}\text{"}\rangle$ is identical to the sequence of statements executed in the computation of $\langle S', \mathcal{I}^*, \text{"}\bar{x}\text{"}\rangle$.*

Proof. Let us sketch the proof of part (2). (Parts 1 and 3 can be proved similarly.) It is clear that if S and S' are equivalent, then $val\langle S, \mathcal{I}^*, \text{"}\bar{x}\text{"}\rangle \equiv val\langle S', \mathcal{I}^*, \text{"}\bar{x}\text{"}\rangle$ for every Herbrand interpretation $\mathcal{I}^*$. To prove the other direction, we assume that S and S' are not equivalent and show the existence of a Herbrand interpretation $\mathcal{I}^*$ such that $val\langle S, \mathcal{I}^*, \text{"}\bar{x}\text{"}\rangle \not\equiv val\langle S', \mathcal{I}^*, \text{"}\bar{x}\text{"}\rangle$. If S and S' are not equivalent, there must exist an interpretation $\mathcal{I}$ of S and S' and an input $\bar{\xi} \in D^n$ such that $val\langle S, \mathcal{I}, \bar{\xi}\rangle \not\equiv val\langle S', \mathcal{I}, \bar{\xi}\rangle$. For this interpretation $\mathcal{I}$ and input $\bar{\xi}$, there exists a Herbrand interpretation $\mathcal{I}^*$ of S and S' such that the computations of $\langle S, \mathcal{I}^*, \text{"}\bar{x}\text{"}\rangle$ and $\langle S', \mathcal{I}^*, \text{"}\bar{x}\text{"}\rangle$ follow the same traces as the computations of $\langle S, \mathcal{I}, \bar{\xi}\rangle$ and $\langle S', \mathcal{I}, \bar{\xi}\rangle$, respectively. Now, suppose $val\langle S, \mathcal{I}^*, \text{"}\bar{x}\text{"}\rangle \equiv$

$val\langle S', \mathscr{I}^*, \text{“}\bar{x}\text{”}\rangle$, this means that both final values are either undefined or are an identical string over $H_S \cup H_{S'}$. This string is actually the term obtained by combining all the assignments along the computations of $\langle S, \mathscr{I}^*, \text{“}\bar{x}\text{”}\rangle$ or $\langle S', \mathscr{I}^*, \text{“}\bar{x}\text{”}\rangle$. Because of the correspondence between these computations and those of $\langle S, \mathscr{I}, \bar{\xi}\rangle$ and $\langle S', \mathscr{I}, \bar{\xi}\rangle$, it follows that† $val\langle S, \mathscr{I}, \bar{\xi}\rangle \equiv val\langle S', \mathscr{I}, \bar{\xi}\rangle$: a contradiction.

Q.E.D.

4-2 DECISION PROBLEMS

The notions of *solvable* and *partially solvable* were introduced in Chap. 1. Sec. 1-5. We say that a class of *yes/no problems* (that is, a class of problems, the answer to each one of which is either “yes” or “no”) is *solvable*, if there exists an algorithm (Turing machine) that takes any problem in the class as input and always halts with the correct yes or no answer. We say that a class of yes/no problems is *partially solvable* if there exists an algorithm (Turing machine) that takes any problem in the class as input and: if it is a yes problem, the algorithm will eventually stop with a yes answer; otherwise, that is, if it is a no problem, the algorithm either stops with a no answer or loops forever. It is clear that if a class of yes/no problems is solvable then it is also partially solvable.

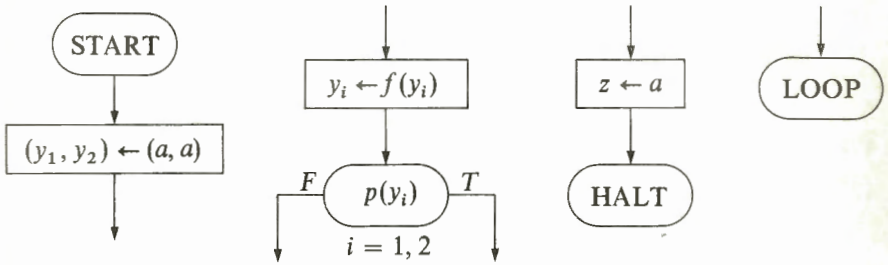
In this section we shall discuss four classes of yes/no problems.

1. *The halting problem for schemas*: Does there exist an algorithm that takes any schema S as input and always halts with a correct yes (S halts for every interpretation) or no (S does not halt for every interpretation) answer?
2. *The divergence problem for schemas*: Does there exist an algorithm that takes any schema S as input and always halts with a correct yes (S diverges for every interpretation) or no (S does not diverge for every interpretation) answer?
3. *The equivalence problem for schemas*: Does there exist an algorithm that takes any two compatible schemas S and S' as input and always halts with a correct yes (S and S' are equivalent) or no (S and S' are not equivalent) answer?
4. *The isomorphism problem for schemas*: Does there exist an algorithm that takes any two compatible schemas S and S' as input and always halts with a correct yes (S and S' are isomorphic) or no (S and S' are not isomorphic) answer?

† Note that $val\langle S, \mathscr{I}^*, \text{“}\bar{x}\text{”}\rangle \equiv val\langle S, \mathscr{I}^*, \text{“}\bar{x}\text{”}\rangle$ implies $val\langle S, \mathscr{I}, \xi\rangle \equiv val\langle S', \mathscr{I}, \xi\rangle$ but not necessarily vice versa!

We shall show that the halting problem for schemas is unsolvable but partially solvable while the divergence, equivalence, and isomorphism problems for schemas are not even partially solvable. It is quite surprising that all these unsolvability results can actually be shown for a very restricted class of schemas. For this purpose let us consider the class of schemas $\mathscr{S}_1$. We say that a schema S is in the class $\mathscr{S}_1$ if

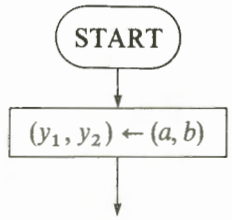
1. Σ_S consists of a single individual constant a, a single unary function f, a single unary predicate p, two program variables y_1 and y_2, a single output variable z, but no input variables.
2. All statements in S are of one of the following forms:



In Section 4-2.1 we shall show that

1. The halting problem for schemas in $\mathscr{S}_1$ is unsolvable.
2. The divergence problem for schemas in $\mathscr{S}_1$ is not partially solvable.
3. The equivalence problem for schemas in $\mathscr{S}_1$ is not partially solvable.
4. The isomorphism problem for schemas in $\mathscr{S}_1$ is not partially solvable.

In the rest of this chapter (Sec. 4-2.2 to 4-2.4) we shall discuss subclasses of schemas for which these problems are solvable. For example, it is very interesting to compare the four decision problems for $\mathscr{S}_1$ with those of a very “similar” class of schemas, namely, $\mathscr{S}_2$. Classes $\mathscr{S}_1$ and $\mathscr{S}_2$ differ only in that every schema in $\mathscr{S}_2$ contains two individual constants a and b and the START statement is of the form



3-3.1 While Programs

First we introduce the class of while programs. A *while program* consists of a finite sequence of statements B_i separated by semicolons:

$$B_0; B_1; B_2; \ldots ; B_n$$

B_0 is the unique START statement

START
$\bar{y} \leftarrow f(\bar{x})$

and each B_i $(1 \leqq i \leqq n)$ is one of the following statements.

1. *ASSIGNMENT statement*:

$$\bar{y} \leftarrow g(\bar{x}, \bar{y})$$

2. *CONDITIONAL statement*:

$$\textbf{if } t(\bar{x}, \bar{y}) \textbf{ then } B \textbf{ else } B'$$

or

$$\textbf{if } t(\bar{x}, \bar{y}) \textbf{ do } B$$

where B and B' are any statements

3. *WHILE statement*:

$$\textbf{while } t(\bar{x}, \bar{y}) \textbf{ do } B$$

where B is any statement

4. *HALT statement*:

$$\bar{z} \leftarrow h(\bar{x}, \bar{y})$$

HALT

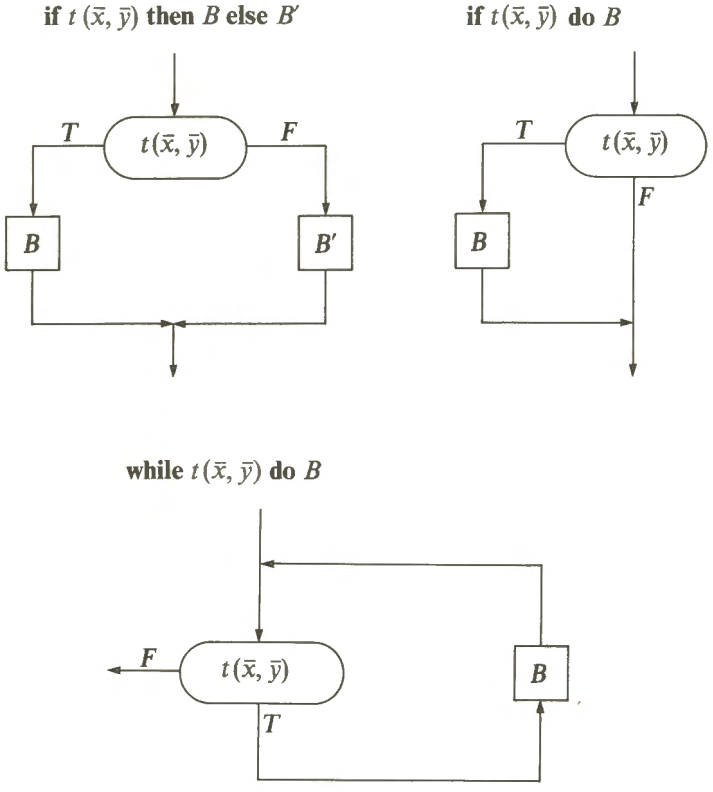
5. *COMPOUND statement*:

$$\textbf{begin } B'_1; B'_2; \ldots ; B'_k \textbf{ end}$$

where B'_j $(1 \leqq j \leqq k)$ are any statements

Given such a while program P and an input value $\bar{\xi} \in D_{\bar{x}}$ for the input vector $\bar{x}$, the program can be executed. Execution always begins at the START statement, initializing the value of $\bar{y}$ to be $f(\bar{\xi})$, and it then proceeds in the normal way following the sequence of statements. Whenever an ASSIGNMENT statement is reached, $\bar{y}$ is replaced by the current

value of $g(\bar{x}, \bar{y})$. The execution of CONDITIONAL and WHILE statements can be described by the following pieces of flowcharts:



If execution terminates (i.e., reaches a HALT statement), $\bar{z}$ is assigned the current value $\bar{\zeta}$ of $h(\bar{x}, \bar{y})$ and we say that $P(\bar{\xi})$ *is defined and* $P(\bar{\xi}) = \bar{\zeta}$; otherwise, i.e., if the execution never terminates, we say that $P(\bar{\xi})$ *is undefined.*

EXAMPLE 3-10

1. The *square-root program* P_1 of Example 3-1 can be expressed by

START
$(y_1, y_2, y_3) \leftarrow (0, 1, 1)$;
while $y_2 \leqq x$ **do** $(y_1, y_2, y_3) \leftarrow (y_1 + 1, y_2 + y_3 + 2, y_3 + 2)$;
$z \leftarrow y_1$
HALT

2. The *gcd program* P_3 of Example 3-3 can be expressed by

START
$(y_1, y_2) \leftarrow (x_1, x_2)$;
while $y_1 \neq y_2$ **do if** $y_1 > y_2$ **then** $y_1 \leftarrow y_1 - y_2$ **else** $y_2 \leftarrow y_2 - y_1$;
$z \leftarrow y_1$
HALT

3. The *gcd program* P_4 of Example 3-5 can be expressed by

START
$(y_1, y_2, y_3) \leftarrow (x_1, x_2, 1)$;
while $even(y_1)$ **do**
    begin
    if $even(y_2)$ **do** $(y_2, y_3) \leftarrow (y_2/2, 2y_3)$;
    $y_1 \leftarrow y_1/2$
    end;
while $even(y_2) \vee y_1 \neq y_2$ **do**
    begin
    if $odd(y_2)$ **do** $(y_1, y_2) \leftarrow (y_2, |y_1 - y_2|)$;
    $y_2 \leftarrow y_2/2$
    end;
$z \leftarrow y_1 y_3$
HALT

□

3-3.2 Partial Correctness

In order to prove partial correctness of while programs, Hoare (1969) introduced the notation (called *inductive expression*)

$$\{p(\bar{x}, \bar{y})\}\, B\, \{q(\bar{x}, \bar{y})\}$$

where p, q are predicates and B is a program segment, meaning that if $p(\bar{x}, \bar{y})$ holds for the values of $\bar{x}$ and $\bar{y}$ immediately prior to execution of B and if execution of B terminates, then $q(\bar{x}, \bar{y})$ will hold for the values of $\bar{x}$ and $\bar{y}$ after execution of B.

Now, suppose we wish to prove the partial correctness of a given while program P with respect to an input predicate φ and an output predicate ψ; in other words, we would like to deduce

$$\{\varphi(\bar{x})\}\, P\, \{\psi(\bar{x}, \bar{z})\}$$

For this purpose we have *verification rules*, which consist of an *assignment axiom*, describing the transformation on program variables effected by ASSIGNMENT statements, and *inference rules*, by which expressions for small segments can be combined into one expression for a larger segment. To apply the rules, first we deduce inductive expressions of the form $\{p\}\,B\,\{q\}$ for each ASSIGNMENT statement B of the program, using the assignment axiom. Then we compose segments of the program, using the conditional rules, the while rule, the concatenation rule, and the consequence rules until we obtain the desired expression $\{\varphi\}\,P\,\{\psi\}$.

The inference rules will be described as

$$\frac{\alpha_1}{\beta} \quad \text{or} \quad \frac{\alpha_1 \text{ and } \alpha_2}{\beta\ —}$$

where α_1 and α_2 are the *antecedents* (the conditions under which the rule is applicable) and β is the *consequent* (the inductive expression to be deduced). Each of the antecedents is either an inductive expression, which should have been previously established, or a logical expression, which should be proved separately as a *lemma*.

The verification rules are

1. *Assignment axiom*:

$$\{p(\bar{x}, g(\bar{x}, \bar{y}))\}\ \bar{y} \leftarrow g(\bar{x}, \bar{y})\ \{p(\bar{x}, \bar{y})\}$$

2. *Conditional rules*:

$$\frac{\{p \wedge t\}\,B_1\,\{q\} \qquad \text{and} \qquad \{p \wedge \sim t\}\,B_2\,\{q\}}{\{p\}\ \textbf{if}\ t\ \textbf{then}\ B_1\ \textbf{else}\ B_2\,\{q\}}$$

and

$$\frac{\{p \wedge t\}\,B\,\{q\} \qquad \text{and} \qquad (p \wedge \sim t) \supset q}{\{p\}\ \textbf{if}\ t\ \textbf{do}\ B\,\{q\}}$$

Note that $(p \wedge \sim t) \supset q$ is a logical expression. It should be considered as a closed wff with all free variables universally quantified. That is,

$$\forall \bar{x} \forall \bar{y}[(p(\bar{x}, \bar{y}) \wedge \sim t(\bar{x}, \bar{y})) \supset q(\bar{x}, \bar{y})]$$

3. *While rule*: The rule for WHILE statement is based on the simple fact that if the execution of the body B of the WHILE statement leaves the assertion p invariant (that is, p is always *true* on completion of B provided it is also *true* on initiation), p is *true* after any number of iterations of B.

$$\frac{\{p \wedge t\}\,B\,\{p\}}{\{p\}\ \textbf{while}\ t\ \textbf{do}\ B\,\{p \wedge \sim t\}}$$

4. *Concatenation rule*:

$$\frac{\{p\}\,B_1\,\{q\} \qquad \text{and} \qquad \{q\}\,B_2\,\{r\}}{\{p\}\,B_1; B_2\,\{r\}}$$

5. *Consequence rules*:

$$\frac{p \supset q \qquad \text{and} \qquad \{q\}\,B\,\{r\}}{\{p\}\,B\,\{r\}} \qquad \text{and} \qquad \frac{\{p\}\,B\,\{q\} \qquad \text{and} \qquad q \supset r}{\{p\}\,B\,\{r\}}$$

Note again that $p \supset q$ and $q \supset r$ are logical expressions and should be proved separately as lemmas.

To summarize, we have the following theorem.

THEOREM 3-3 [VERIFICATION-RULES METHOD (Hoare)]. *Given a while program P, an input predicate $\varphi(\bar{x})$, and an output predicate $\psi(\bar{x}, \bar{z})$. If, by applying successively the verification rules just described, we can deduce*

$$\{\varphi(\bar{x})\}\,P\,\{\psi(\bar{x}, \bar{z})\}$$

then P is partially correct wrt φ and ψ.

Some of the verification rules can be combined to form new rules that are more convenient to use; such rules are usually called *derived verification rules*, four of which are the following.

1. *Assignment rule*: From the assignment axiom and the consequence rule, we obtain

$$\frac{p(\bar{x}, \bar{y}) \supset q(\bar{x}, g(\bar{x}, \bar{y}))}{\{p(\bar{x}, \bar{y})\}\ \bar{y} \leftarrow g(\bar{x}, \bar{y})\ \{q(\bar{x}, \bar{y})\}}$$

2. *Repeated assignment rule*: From the assignment and concatenation rules we obtain

$$\frac{p(\bar{x}, \bar{y}) \supset q(\bar{x}, g_n(\bar{x}, g_{n-1}(\bar{x}, \ldots, g_2(\bar{x}, g_1(\bar{x}, \bar{y}))\ \ldots\ )))}{\{p(\bar{x}, \bar{y})\}\ \bar{y} \leftarrow g_1(\bar{x}, \bar{y}); \bar{y} \leftarrow g_2(\bar{x}, \bar{y}); \ldots; \bar{y} \leftarrow g_n(\bar{x}, \bar{y})\ \{q(\bar{x}, \bar{y})\}}$$

3. *Modified while rule*: From the while and consequence rules we obtain

$$\frac{\{p \wedge t\}\,B\,\{p\} \qquad \text{and} \qquad (p \wedge \sim t) \supset q}{\{p\}\ \textbf{while}\ t\ \textbf{do}\ B\,\{q\}}$$

4. *Modified concatenation rule*: From the concatenation and consequence rules we obtain

$$\frac{\{p\}\,B_1\,\{q\} \quad \text{and} \quad \{r\}\,B_2\,\{s\} \quad \text{and} \quad q \supset r}{\{p\}\,B_1; B_2\,\{s\}}$$

Note that for the START statement

START
$\bar{y} \leftarrow f(\bar{x})$

by the assignment axiom we obtain

$$\{p(\bar{x}, f(\bar{x}))\}\,\bar{y} \leftarrow f(\bar{x})\,\{p(\bar{x}, \bar{y})\}$$

and for the HALT statement

$\bar{z} \leftarrow h(\bar{x}, \bar{y})$
HALT

we obtain

$$\{p(\bar{x}, h(\bar{x}, \bar{y}))\}\,\bar{z} \leftarrow h(\bar{x}, \bar{y})\,\{p(\bar{x}, \bar{z})\}$$

EXAMPLE 3-11

Let us prove again the partial correctness of program P_1 of Example 3-1 wrt the input predicate $x \geqq 0$ and the output predicate $z^2 \leqq x < (z+1)^2$. The program is

START
$(y_1, y_2, y_3) \leftarrow (0, 1, 1)$;
while $y_2 \leqq x$ **do** $(y_1, y_2, y_3) \leftarrow (y_1 + 1, y_2 + y_3 + 2, y_3 + 2)$;
$z \leftarrow y_1$
HALT

In our proof we shall use the following assertion:

$$R(x, y_1, y_2, y_3): \quad (y_1^2 \leqq x) \wedge (y_2 = (y_1+1)^2) \wedge (y_3 = 2y_1 + 1)$$

The proof can be stated formally as follows:

1. $x \geqq 0 \supset R(x, 0, 1, 1)$ — Lemma 1
2. $\{x \geqq 0\}$
   $(y_1, y_2, y_3) \leftarrow (0, 1, 1)$
   $\{R(x, y_1, y_2, y_3)\}$ — Assignment rule (line 1)
3. $R(x, y_1, y_2, y_3) \wedge y_2 \leqq x \supset R(x, y_1 + 1, y_2 + y_3 + 2, y_3 + 2)$ — Lemma 2

4. $\{R(x, y_1, y_2, y_3) \wedge y_2 \leqq x\}$
   $(y_1, y_2, y_3) \leftarrow (y_1 + 1, y_2 + y_3 + 2, y_3 + 2)$
   $\{R(x, y_1, y_2, y_3)\}$ — Assignment rule (line 3)
5. $\{R(x, y_1, y_2, y_3)\}$
   while $y_2 \leqq x$ **do** $(y_1, y_2, y_3) \leftarrow (y_1 + 1, y_2 + y_3 + 2, y_3 + 2)$
   $\{R(x, y_1, y_2, y_3) \wedge y_2 > x\}$ — While rule (line 4)
6. $\{x \geqq 0\}$
   $(y_1, y_2, y_3) \leftarrow (0, 1, 1)$;
   while $y_2 \leqq x$ **do** $(y_1, y_2, y_3) \leftarrow (y_1 + 1, y_2 + y_3 + 2, y_3 + 2)$
   $\{R(x, y_1, y_2, y_3) \wedge y_2 > x\}$ — Concatenation rule (lines 2 and 5)
7. $R(x, y_1, y_2, y_3) \wedge y_2 > x \supset y_1^2 \leqq x < (y_1 + 1)^2$ — Lemma 3
8. $\{R(x, y_1, y_2, y_3) \wedge y_2 > x\}$
   $z \leftarrow y_1$
   $\{z^2 \leqq x < (z+1)^2\}$ — Assignment rule (line 7)
9. $\{x \geqq 0\}$
   $(y_1, y_2, y_3) \leftarrow (0, 1, 1)$;
   while $y_2 \leqq x$ **do** $(y_1, y_2, y_3) \leftarrow (y_1 + 1, y_2 + y_3 + 2, y_3 + 2)$;
   $z \leftarrow y_1$
   $\{z^2 \leqq x < (z+1)^2\}$ — Concatenation rule (lines 6 and 8).

Thus, since Lemmas 1, 2, and 3 are *true*, the proof implies that P_1 is partially correct wrt $x \geqq 0$ and $z^2 \leqq x < (z+1)^2$.

As in the inductive-assertion method (described in Sec. 3-1.2), the choice of the assertion $R(x, y_1, y_2, y_3)$ is the most crucial part of the proof. In general, the process of finding such assertions requires a deep understanding of the program's performance. It is our hope that these assertions will be supplied by the programmer as comments in every program he writes; for example, if we enclose comments within braces, an appropriate way to express the square-root program would be

START
$\{x \geqq 0\}$
$(y_1, y_2, y_3) \leftarrow (0, 1, 1)$;
while $y_2 \leqq x$ **do** $\{(y_1^2 \leqq x) \wedge (y_2 = (y_1+1)^2) \wedge (y_3 = 2y_1 + 1)\}$
$\quad (y_1, y_2, y_3) \leftarrow (y_1 + 1, y_2 + y_3 + 2, y_3 + 2)$;
$z \leftarrow y_1$
$\{z^2 \leqq x < (z+1)^2\}$
HALT

□

EXAMPLE 3-12 [Hoare (1961), (1971)]

The purpose of the program P_8 in this example is to rearrange the elements of an array S of $n+1$ ($n > 0$) real numbers $S[0], \ldots, S[n]$ and to find two integers i and j such that

$$0 \leqq j < i \leqq n$$

and

$$\forall a \forall b [(0 \leqq a < i \wedge j < b \leqq n) \supset S[a] \leqq S[b]]$$

In other words, we would like to rearrange the elements of S into two nonempty partitions such that those in the lower partition $S[0], \ldots, S[i-1]$ are less than or equal to those in the upper partition $S[j+1], \ldots, S[n]$, where $0 \leqq j < i \leqq n$. In the program, n is an input variable, r is a program variable, and i and j are output variables. For simplicity, S is used as an input and program array as well as an output array:

START
$\{n > 0\}$
$r \leftarrow S[div(n, 2)]$; $(i, j) \leftarrow (0, n)$;
while $i \leqq j$ **do**
  begin while $S[i] < r$ **do** $i \leftarrow i + 1$;
    while $r < S[j]$ **do** $j \leftarrow j - 1$;
    if $i \leqq j$ **then begin** $(S[i], S[j]) \leftarrow (S[j], S[i])$;
      end $(i, j) \leftarrow (i + 1, j - 1)$
  end
$\{0 \leqq j < i \leqq n \wedge \forall a \forall b [(0 \leqq a < i \wedge j < b \leqq n) \supset S[a] \leqq S[b]]\}$
HALT

The division into smaller and larger elements is done by selecting the element $r = S[div(n, 2)]$† and placing elements smaller than it in the lower partition and elements larger than it in the upper partition. Initially i is set to 0 and j to n, and i is stepped up for as long as $S[i] < r$ since these elements belong to the lower partition and may be left in position. When an $S[i]$ is encountered which is not less than r (and hence out of place), the stepping up of i is interrupted. The value of j is then stepped down while $r < S[j]$, and this stepping down is interrupted when an $S[j]$ not greater than r is met. The current $S[i]$ and $S[j]$ are now both in the wrong positions; the situation is corrected by exchanging them. If $i \leqq j$, i is stepped up by one and j is stepped down by one and the "i search" and "j search"

†Note that $div(n, 2)$ denotes the quotient of the integer division of n by 2.

are continued until the next out-of-place pair is found. If $j < i$, the lower and upper parts meet and the partition is complete.

We wish to prove that this program is partially correct; that is, when it terminates (if at all), (1) the elements of the array S are a permutation of the elements of the initial array† and (2) the following relation holds

$$(j < i) \wedge \forall a, \forall b [(0 \leqq a < i \wedge j < b \leqq n) \supset S[a] \leqq S[b]]$$

We leave it as an exercise to the reader (Prob. 3-13) to prove that the stronger relation $0 \leqq j < i \leqq n$ holds. It is clear that relation 1 holds because $(S[i], S[j]) \leftarrow (S[j], S[i])$ is the only operation applied to S and it leaves the elements of the array S unchanged except for order.

Relation 2 can be verified in steps using the verification rules. Instead of giving a complete proof, we shall just describe the assertions that should be used in such a proof. The annotated program is

START
$\{n > 0\}$
$r \leftarrow S[div(n, 2)]$; $(i, j) \leftarrow (0, n)$;
$\{0 \leqq i \wedge \forall a (0 \leqq a < i \supset S[a] \leqq r) \wedge$ (i invariant)
$j \leqq n \wedge \forall b (j < b \leqq n \supset r \leqq S[b])\}$ (j invariant)
while $i \leqq j$ **do**
  begin $\{i$ invariant $\wedge\ j$ invariant$\}$
    while $S[i] < r$ **do** $i \leftarrow i + 1$;
    $\{i$ invariant $\wedge\ j$ invariant $\wedge\ r \leqq S[i]\}$
    while $r < S[j]$ **do** $j \leftarrow j - 1$;
    $\{i$ invariant $\wedge\ j$ invariant $\wedge\ S[j] \leqq r \leqq S[i]\}$
    if $i \leqq j$ **then begin** $(S[i], S[j]) \leftarrow (S[j], S[i])$;
      $(i, j) \leftarrow (i + 1, j - 1)$
    end
  end $\{i$ invariant $\wedge\ j$ invariant$\}$

$\{j < i \wedge \forall a \forall b [(0 \leqq a < i \wedge j < b \leqq n) \supset S[a] \leqq S[b]]\}$
HALT

□

3-3.3 Total Correctness

The verification-rules method (Theorem 3-3) enables us to prove only the partial correctness of while programs. In order to extend the method to prove the total correctness (including termination), Manna and Pnueli

†That is, $perm(S_{input}, S_{output}, 0, n)$ holds, where S_{input} indicates the value of S at the START statement and S_{output} indicates the value of S at the HALT statement.