A Method of Programming

Edsger W. Dijkstra
University of Texas at Austin

W. H. J. Feijen
Technical University of Eindhoven

Translated by Joke Sterringa

ADDISON-WESLEY
PUBLISHING
COMPANY

Wokingham, England · Reading, Massachusetts · Menlo Park, California
New York · Don Mills, Ontario · Amsterdam · Bonn
Sydney · Singapore · Tokyo · Madrid · San Juan

Cover designed by Crayon Design of Henley-on-Thames
and printed by The Riverside Printing Co. (Reading) Ltd.
Typeset by Columns of Reading
Printed in Great Britain by T.J. Press (Padstow), Cornwall

The original Dutch edition of this book was published in The Netherlands:
Een methode van programmeren. Edsger W. Dijkstra and W.H.J. Feijen.
The Hague: Academic Service, 1984.

First English edition printed 1988.

British Library Cataloguing in Publication Data
Dijkstra, Edsger W.
A method of programming.
1. Computer systems. Programming.
Applications of mathematical logic
I. Title II. Feijen, W.H.J. III. Een
methode van programmeren. *English*
005.13'1

ISBN 0–201–17536–3

Library of Congress Cataloguing in Publication Data
Dijkstra, Edsger Wybe.
[Methode van programmeren. English]
A method of programming / Edsger W. Dijkstra. W.H.J. Feijen. –
1st English ed.
p. cm.
Translation of: Een methode van programmeren.
ISBN 0–201–17536–3
1. Electronic digital computers—Programming I. Feijen, W.H.J.
II. Title.
QA76.6.D5513 1988
005. 1—dc19 88–1580 CIP

Preface

This text was originally written under relatively high pressure, when a computer science curriculum was thrust upon us rather suddenly. The reason for writing it was the same as the reason why we later agreed to have a second version of it published as a book: the text fulfils what we have come to regard as a serious need.

Programming started out as a craft which was practised intuitively. By 1968 it began to be generally acknowledged that the methods of program development followed so far were inadequate to face what then appeared as the so-called 'software crisis'. After the methods then employed had been identified as fundamentally inadequate, a style of design was developed in which the program and its correctness argument were designed hand in hand. This was a dramatic step forward.

Although it appears in monographs and textbooks, this achievement has not yet reached the introductory programming curriculum. What, for lack of a better term, we shall call 'the computer science boom' induced us to end this situation, for two reasons. Firstly, the growing number of students makes it increasingly unjustifiable to organize the introductory curriculum according to an obsolete model. Secondly, with the growing popularity of computers, the traditionally intuitive introduction contributes less and less to the further education of the student.

It is for these reasons that, in this text, programming is presented as what it has since become – a formal branch of mathematics, in which mathematical logic has become an indispensable tool.

The book consists of two parts, originally corresponding to the lectures and their instruction, respectively. The lectures unfold the subject matter that is specific to programming, while the instruction describes the logical apparatus used for this and contains the exercises. How the reader may best divide his or her attention between the two parts is optional, since the optimum balance will depend on the reader's background.

We owe thanks to all our colleagues in the Department of Computer Science at the Technical University of Eindhoven who have

taught the subject described here with so much enthusiasm and success over the last few years. In particular, we would like to mention A. J. M. van Gasteren, A. Kaldewaij, M. Rem, J. L. A. van de Snepscheut and J. T. Udding. Their experience and stimulation have been a great support.

Eindhoven

Edsger W. Dijkstra
W. H. J. Feijen

Contents

PART 1

PART 0

A method of programming

'Informatics' is the name used since 1968 in non-Anglo-Saxon countries for the subject called 'computer science' in the USA and Great Britain. For the Anglo-Saxon term 'computer' Dutch uses 'automatic calculating machine' or the shorter 'calculating automaton'; both terms are adequate, provided that – as we shall see later – we do not assign too narrow a meaning to the concept of 'calculating'.

We use the term 'automaton' for a mechanism which, if so designed, can do something for us autonomously, that is, without any further interference on our part. A familiar automaton (at least in some countries) is, for example, the cistern of a toilet. After the starting signal – pulling the chain or pushing the button – the rest takes care of itself: the toilet is flushed clean, the cistern fills up and, at the right moment, the feed tap is closed so that the cistern does not overflow.

On this basis one may think that a cigarette machine would not deserve the name of automaton, which it has in Dutch, since the customer must interact in all kinds of ways: for example, he or she must insert coins and pull out a drawer. These actions, however, may be regarded as an elaborate starting signal: the machine is an automaton for the tobacconist, who is not disturbed at all during the transaction.

Other classic examples of automata are the clock and the music box, which if wound up plays 'O, du lieber Augustin'. (It was often the same craftsman who manufactured both music boxes and clocks, whether or not the clocks were provided with a cuckoo.)

The above mechanisms are a bit dull because, in some sense, they do the same each time: the cistern takes care of one flushing after another, the clock repeats its pattern every 12 hours, and the music box lets us have 'O, du lieber Augustin' *ad nauseam*. (Since Watt's steam engine also belongs to this group of dull mechanisms, we should not speak disrespectfully of this dullness.)

These mechanisms were succeeded by a more flexible type, for example, the type of music box with a changeable cylinder: this meant that 'O, du lieber Augustin' or 'Here we go round the mulberry bush' could be performed with largely the same machine. Many automata are of this type: the pianola, the film projector, and the street organ. Again it does not become us to speak of them disrespectfully: Jacquard's loom and the modern automatic controlled milling machine come into this category, as do playback equipment for gramophone records, video discs, and tapes.

These mechanisms have been introduced to illustrate the concept of an automaton. They do not share the other aspect of the 'calculating automaton', namely that it 'calculates', so that now (with due respect) we take our leave from them.

What do we mean by 'calculate'? Let us take a very simple example: the addition of two natural numbers in the decimal system. Very simple? Maybe – though, after having learned the numbers 0 to 9, it still takes years before children get the hang of it (and some never do). Let us see what it takes.

To begin with we learn the tables of addition as shown in Figure 1.

	0	1	2	3	4	5	6	7	8	9
0	0	1	2	3	4	5	6	7	8	9
1	1	2	3	4	5	6	7	8	9	10
2	2	3	4	5	6	7	8	9	10	11
3	3	4	5	6	7	8	9	10	11	12
4	4	5	6	7	8	9	10	11	12	13
5	5	6	7	8	9	10	11	12	13	14
6	6	7	8	9	10	11	12	13	14	15
7	7	8	9	10	11	12	13	14	15	16
8	8	9	10	11	12	13	14	15	16	17
9	9	10	11	12	13	14	15	16	17	18

Figure 1

The upper row and the left-hand column are not very difficult, and gradually the child becomes familiar with the upper left corner of the table: the so-called 'calculating under ten'. After some time the bottom right corner also becomes familiar: the child now knows by heart the addition of two numbers under ten. This is very good: the child now knows the answer to 100 different additions.

However, it is also clear that we cannot go on like this. There are 10 000 different additions of two numbers less then 100, 1 000 000 different additions of two numbers less than 1000, and it would obviously be madness to try to learn such large tables by heart. Fortunately, we do

not have to, for – as any schoolchild will have noticed – the table is not without regularity. The next stage in our addition education, therefore, consists of learning to exploit this regularity.

To begin with, we do not regard larger numbers as an entity, but as a series of digits, which are dealt with one by one: we learn to construct the sequence of digits that represents the sum from the sequences of digits which represent the numbers to be added.

What are the ingredients of this construction process? First the digits of the numbers to be added are added two by two. This is represented by writing the numbers one under the other. And for this we learn that 2037 + 642 should look like

```
2037
 642
```

and *not* like

```
2037
642
```

This addition is easy, because for each pair we can do our calculations 'under ten':

```
2037
 642
---- +
2679
```

and so far it does not matter if we work from left to right or from right to left.

To be able to execute additions like

```
2037
 645
---- +
2682
```

as well, the rules are extended with 'carry 1', and to cap it all the pupil is made familiar with the cascade phenomenon which occurs when 'carry 1' must be applied in a position where the sum of the digits is 9, as in:

```
2057
 645
---- +
2702
```

This extensive examination of the decimal addition of two natural numbers is useful not because it is presupposed that the reader cannot add, but to provide an awareness of the many rules that are applied, albeit virtually unconsciously.

If formulated with sufficient precision, such a combination of rules makes up what we call an **algorithm**. (Above we have informally given an algorithm for the decimal addition of natural numbers.) An algorithm is a prescription which, provided it is faithfully executed, yields the desired result in a finite number of steps.

With reference to the addition algorithm given, we can at once make the following remark.

Remark. It is not necessarily true that an algorithm leaves nothing to the imagination of the one who carries it out: irrelevant choices may be left open. In order to add 2057 and 645, the numbers are written one under the other, but evidently,

$$\begin{array}{rl} 2057 & \\ \underline{645} & + \\ 2702 & \end{array} \qquad \text{and} \qquad \begin{array}{rl} 645 & \\ \underline{2057} & + \\ 2702 & \end{array}$$

do equally well. In the corresponding multiplication algorithm this phenomenon is more marked. Compare:

$$\begin{array}{rl} 71 & \\ \underline{28} & * \\ 568 & \\ \underline{142} & + \\ 1988 & \end{array} \qquad \text{and} \qquad \begin{array}{rl} 28 & \\ \underline{71} & * \\ 28 & \\ \underline{196} & + \\ 1988 & \end{array}$$

■

Remark. The addition algorithm can be applied in a great many different cases. The fact that, as in this example, the algorithm is applicable in an unlimited number of cases, and that, independent of the numbers to be added, there is no upper bound for the number of steps an execution of the algorithm will take, does not alter the fact that any individual execution takes only a finite number of steps. ■

Another algorithm is illustrated by the composition of differentiation rules which, for example, enables us to compute

$$\frac{\mathrm{d}}{\mathrm{d}x}(\mathrm{e}^{\sin x})$$

as:

$$(\cos x) \cdot \mathrm{e}^{\sin x}$$

The differentiation algorithm allows some freedom concerning the order in which the various rules are applied and is, in principle, also applicable in an unlimited number of cases. This example is included because, while it is customary to speak of 'the computation of a derivative', the computing is here already stripped of any specific numerical associations.

Other examples of algorithms are planimetric constructions (for the bisector of an angle, the apex of a triangle, etc.), knitting patterns, user instructions, assembly instructions, recipes, and the rules we follow to see if someone is listed in the telephone directory.

Remark. For the telephone directory of Amsterdam the rules are simpler, and searching generally does not take quite as many steps as it does for the 1982 telephone directory for Eindhoven and Region, in which the names of subscribers are listed according to their villages. The design of the latter directory may be considered to be faulty. ■

The automatic calculating machine is so called because it can 'calculate' automatically in the sense of carrying out an algorithm automatically. The computer derives its great flexibility from the fact that the selection of the algorithm to be carried out by the mechanism is up to us, and that in selecting this algorithm we have virtually unlimited freedom. (Compared with the mechanisms mentioned before, the computer represents a quantum leap.)

We can express the fact that the computer can be fed with an algorithm of our choice by saying that the computer is 'programmable'. An algorithm that could be executed by a mechanism is called a 'program', and to design programs is called 'to program'. Programming is the main subject of these lectures.

Programming merits a lecture course for a number of reasons. First of all, there is always a program needed to bridge the gap between the general-purpose computer and the specific application, and therefore the activity of programming takes a central place. Second, we know from experience that someone who has not learned to think and reason sufficiently pragmatically and soundly about the design during the programming, will irrevocably make a mess of things. To make the student completely familiar with the most effective known way of reasoning about algorithms is therefore an important objective of this course.

One warning is called for: a program is a formal text in which each letter, each digit, each punctuation mark and each operator plays its part. Programs must therefore be written with uncommon precision. Since most people grow up with the idea that they can get away with a few mistakes of spelling or grammar here and there in their writings, and behave accordingly, they are often taken aback by this requirement for precision on their first introduction to programming; so much so that they think programming is a matter of accuracy only. Once this accuracy has become second nature, they realize that the difficulty lies somewhere else completely: in the duty to prevent the subject from becoming unmanageably complicated. (Some inexperienced people regard the necessity for this accuracy as a fault in the computer, but they do not realize that the computer derives its usefulness from the very faithfulness with which it executes the algorithm assigned to it, and *no other*.)

Finally, the student should realize that what can be dealt with in the narrow scope of this introductory course cannot be representative of programming in all its possible aspects. In order to save time, and not to make things unnecessarily difficult, we shall develop our programs for a very simple machine from which elaborate trimmings (which all too often prove to be snags) are missing. The particular difficulties of the development of really large programs are outside the scope of this introductory course.

Mechanisms and their states

Let us consider a mechanism which, once started, performs something for a while, and then stops. As examples, we can think of a gramophone or a toilet cistern. The gramophone is started by putting a disc on the turntable and lowering the stylus into the groove; it stops when, at the end of the groove, the stylus comes too close to the axis of the turntable. The cistern is started by pulling the chain; when the cistern has filled up again, the feed tap is closed, and the process is stopped.

A mechanism, if started, not only does something for a while and then stops, but also, since it is a mechanism, it does it automatically, that is, without further interaction on our part. Because of this, such a mechanism is in a *different* state at any moment between start and stop: shortly after starting, it is in a state such that it will go on for quite a while, and shortly before the end it is in a state such that it nearly stops.

Someone who knows the mechanism involved and knows where to look can always see how much progress the working mechanism has made. In the case of the gramophone the state of progress is reflected by the position of the arm: one look at its position is sufficient to determine how far the playing of the record has progressed.

Remark. At different moments between start and stop, the mechanism must be in different states. When, because of a scratch in the record, the needle clicks back to the last groove and the gramophone thus returns to a state it has previously been in, this condition is not fulfilled. There is something wrong: the needle is stuck and because it cannot progress it will not finish and stop automatically. ■

In the case of the cistern, too, the state determines how far the autonomous process has progressed. The water level in the cistern, however, is only partially analogous to the position of the gramophone arm: during the whole cycle the cistern is half-full twice, once during emptying and once while filling up. The distinction between the two states is determined by the fact that the bell does or does not close off the drain pipe. We may conclude that the state of the cistern is approximately determined by two variables: the continuous variable of the 'water level' and the discrete variable 'drain', for which only the two values 'open' and 'closed' are available.

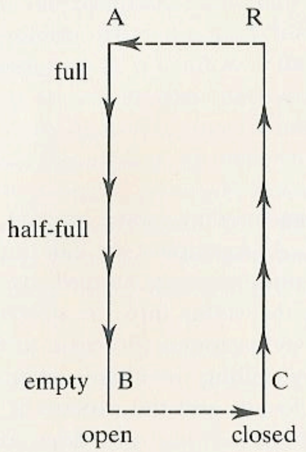

Figure 2

In Figure 2 the water level is represented vertically and the two states of the drain, open and closed, horizontally. Point R is the state of rest: cistern full and drain closed. The start, pulling the chain, opens the drain, which remains open as long as water flows through it with sufficient rapidity. When the cistern is empty, the bell drops again and the drain is closed, after which the cistern fills up. (The water supply is opened when the cistern is not full. The capacity of the supply, however, is smaller than that of the drain, so that the supply does not prevent the cistern from emptying. Verify that, when the capacity of the drain is twice that of the supply, flushing the toilet, i.e. traversing the path from A to B, takes as long as filling the cistern afterwards, i.e. traversing the path from C to R. From the fact that flushing generally takes much less time than the filling of the cistern afterwards, we may conclude that the ratio of the two capacities is usually considerably larger than 2.)

Any possible state of the cistern corresponds to a point in our two-dimensional figure, which has therefore been given the name of **state space**. (In this special case we may speak of a 'state plane', because the state space is two-dimensional. However, we shall use the more general term 'space', as in many cases our state space will have more than two dimensions.) The event which takes place in the period between start and

stop is reflected in the form of a path in the state space which must be traversed. (Note that this path does not reflect the speed with which it is traversed.)

The position of a point in the state space is here denoted by two coordinates: the water level and the fact that the drain is open or closed. Here, the water level is a continuous variable and the state of the drain has been treated as a discrete variable, which has only the values open and closed. (This means that we regard opening and closing the drain as indivisible, instantaneous events: the drain is open or closed, but not half-open. This kind of 'point event' is a useful idealization, not unlike 'point mass' in classical mechanics.) With a view to the structure of computers, discussion will be confined to state spaces in which all coordinates are discrete. If, for example, the state space is spanned by two integer coordinates, i.e. limited to integer values, we can represent the state space by the grid points in the plane shown in Figure 3, and the path by 'jumps' from one grid point to another.

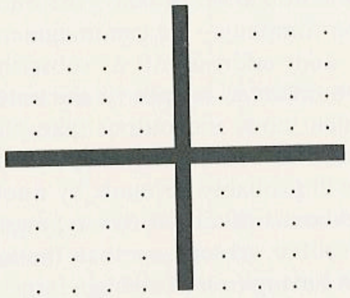

Figure 3

Remark. There are machines in which values are represented by continuously variable physical quantities, such as voltage, current intensity, or the rotation angle of an axis. These are the so-called **analogue** machines. They are completely outside the scope of this book. Analogue machines have always had the drawback that it is technically impossible in this way to represent values with very high precision. They have largely lost their earlier advantage of speed, as digital computers have become faster, so that what used to be done by analogue equipment is now carried out more and more by digital equipment – for example, the digital recording of music. ■

Computation as a change of state

A computer reacts to signals it receives from the external world by giving signals in response to the signals received. Taking in information from the external world is called the **input**, and yielding information is called **output**. As part of the automation of 008 – the number for 'Information' of the Dutch telephone company – we can imagine a machine which takes as input the name and address of a subscriber, and returns the corresponding phone number as output to the external world.

The way in which input and output take place often differs from one computer to another. In automating telephone information, the input of name and address will probably be made by a telephone operator who 'types' this in via a keyboard (like that of a typewriter), while the output takes place by means of a screen, so that the operator can give the desired answer without having to leaf through large telephone directories. It is also conceivable that the computer could produce the answer in an audible form. The Girobank gives us an example where input and output take place through quite different channels: input takes place by means of punched cards (or other machine-readable forms), and output takes place by means of addressed and printed 'statements of account'.

Because of the large variety of input and output media we shall largely abstract from input and output in this course, and focus our attention on what goes on from the moment the input is completed and the computations can commence, to the moment the computations are completed, and the answer is ready for output.

Remark. For sake of simplicity we pass over the fact that the computation process can start in part before the input is completed, and that sometimes output of part of the answer is possible before the process of computation is completed. ■

The reasons for confining ourselves to the period from the end of the input to the beginning of the output are many. Firstly, different

computers resemble each other in what goes on internally far more than in the ways in which they communicate with the external world; what goes on internally is therefore a subject of a far more universal nature. Secondly, it is during this period that the real process of computation, on which we should focus our attention, takes place. (A third reason, which can be mentioned now but not yet explained, is that it is a simplification which enables us to treat partial computations on the same level as the total computation.)

In what follows we shall consider computational processes which start from an **initial state** of the mechanism and lead to a **final state** of the mechanism. If it concerns the whole computation, we shall tacitly assume from now on that the initial state is directly determined by the input and that the final state is directly determined by what must be the output.

A little more precisely: the initial and final states are described by the same set of coordinates for each computation, the input determines the value of one or more coordinates for the initial state, and in the final state the value of one or more coordinates represents the desired answer.

Remark. The input does not have to specify the values of all the coordinates. ■

In deciding to regard computations as changes of state, we can give the **functional specification** of a program by stating the relation between the initial and the final state. A fixed plan for this functional specification will be used, and its description illustrated by a series of small examples.

Such a functional specification consists of four ingredients:

(i) the **declaration** of **local variables**;
(ii) the precondition, traditionally put in braces;
(iii) the name of the program, traditionally separated from what went before by a semicolon;
(iv) the postcondition, also traditionally put in braces.

All this is preceded by the opening bracket |[(pronounced 'begin') and followed by the corresponding closing bracket]| (pronounced 'end').

A simple example of a functional specification is the following (in which, for the sake of discussion, the lines are numbered):

```
(i)    |[ x : int
(ii)      {x = X}
(iii)     ; skip
```

```
(iv)     {x = X}
   ]|
```

Remark. If the opening and closing brackets are not on the same line, they are vertically aligned for the sake of clarity. The functional specification given above is so small that there would have been no objection at all to the layout:

```
|[ x: int {x = X}; skip {x = X} ]|
```

■

We should read this functional specification as follows. Line (i) tells us that it concerns a state space with only one coordinate, which is denoted by the name `x`, and whose value range is limited to integers. This is the purpose of the **type indication** `: int`, which concludes the declaration. (`int` is short for the Latin word *integer*.) The rest tells us that the original validity of precondition (ii) is sufficient for the execution of the program `skip` in line (iii), to ensure that afterwards postcondition (iv) will hold.

When, as is the case here, the conditions contain a quantity like `X`, which actually comes out of the blue, it means that the functional specification holds for any *possible* value of `X`. (Since `x` is integer and `x = X`, we do not have to worry about $X = 3\frac{1}{2}$.) In other words, the above functional specification of `skip` tells us that `skip` must leave the value of `x`, whatever it may be, unchanged.

Another simple example of a functional specification of a program, which we shall also conveniently call `skip`, is:

```
(i)    |[ x, y, z: int
(ii)      {x = X  ∧  y = Y  ∧  z = Z}
(iii)     ; skip
(iv)      {x = X  ∧  y = Y  ∧  z = Z}
       ]|
```

This specifies that in a three-dimensional state space with coordinates `x`, `y` and `z` execution of `skip` should leave the values of each of the coordinates, whatever they may be, unchanged.

Remark. The order in which, separated by commas, the local names `x`, `y` and `z` are listed in the declaration is irrelevant. Equivalent forms for line (i) are therefore:

```
|[ x, z, y: int
|[ z, y, x: int  etc.
```


We have introduced the three local variables here together in one declaration. We could also have introduced each with its own declaration; in that case, however, such independent declarations must be separated by semicolons. For example:

```
|[ x: int; y: int; z: int
```

Mixed forms are also permitted, as in:

```
|[ x, y: int; z: int
```

■

EXERCISE

Check that, with the above-mentioned liberties, line (i) can be written in 24 different, but equivalent, ways.

If in a complicated program variables obviously belong together in groups, the clarity of the text may be enhanced by a corresponding declaration in groups. ■

The declaration(s) is (are) always preceded by the opening bracket `|[`. The corresponding closing bracket `]|` indicates the point of the text up to which the meaning, which is assigned to the names introduced by the declaration, is applicable. In this way the pair of brackets `|[` and `]|` defines the scope of the names in the text.

Two specifications are given above, one for `skip` in a one-dimensional state space, and one for `skip` in a three-dimensional space. In this way we could define `skip` every time we introduce state spaces. Because this would become far too tedious, we consider `skip` as defined in *any* state space: from now on `skip` is the universal name of the action which does not effect any changes of state, irrespective of the state space used to define the state.

Remark. On the face of it `skip` does not seem to be terribly useful. Later we shall see that `skip` is just as useful as the digit 0 in the positional number system. ■

Further examples of functional specifications are:

```
|[ x, y: int
   {x = X  ∧  y = Y}
 ; swap0
   {x = Y  ∧  y = X}
]|
```

and

```
|[ x, y: int
  {x = Y  ∧  y = X}
 ; swap1
  {x = X  ∧  y = Y}
]|
```

EXERCISE

Show that the functional specifications of swap0 and swap1 are equivalent.

Programs swap0 and swap1 have the property that, on knowing the final state, the initial state can be deduced. If, for example, after execution of swap1 the state is given by x = 2 ∧ y = 3, we can conclude that in the initial state x = 3 ∧ y = 2. We can express this by saying that the execution of swap1 does not destroy any information, in contrast to the following programs, copy and euclid, for example.

```
|[ x, y: int
  {x = X}
 ; copy
  {x = X  ∧  y = X}
]|
```

If the outcome of copy is x = 5 ∧ y = 5 we may conclude that originally x = 5. With regard to the original value of y, which is *not* mentioned in the precondition, we cannot draw any conclusions; the value of y may therefore have been anything at the beginning.

```
|[ x, y: int
  {x = X  ∧  y = Y  ∧  x > 0  ∧  y > 0}
 ; euclid
  {x = y  ∧  x = gcd(X, Y) }
]|
```

In euclid, gcd stands for 'greatest common divisor of'. If the outcome of euclid is x = 5 ∧ y = 5, the initial state cannot have been anything whatsoever. For example, x = 40 ∧ y = 70 is out of the question, but there are many possibilities. We could have specified the same program euclid as follows:

```
|[ x, y: int
  {X = gcd(x, y)  ∧  x > 0  ∧  y > 0}
 ; euclid
  {x = X  ∧  y = X}
]|
```


Examination of the final state does determine the value of X, for example 5, but for a given value of X the precondition, regarded as an equation in the two unknowns x and y, has many solutions. So euclid also destroys information.

Remark. It is important to realize that an incorrect functional specification can make impossible demands, as in the following incorrect specification for root:

```
|[ x: int
  {x = X}
 ; root
  {x = √X}
]|
```

If originally x = 43, then for X = 43 the precondition is neatly fulfilled, but with this value of X the postcondition cannot be satisfied with integer x. (And the same holds when initially x is negative.) In short, the precondition above is too weak and must be strengthened with the additional condition that x is a square. If, as in this and most cases, the final state is uniquely determined by the initial state, the naming of the values in the final state is, as a rule, the easiest way to overcome this:

```
|[ x: int
  {x = X²  ∧  X ≧ 0}
 ; root
  {x = X}
]|
```

■

Programs and their construction

In the preceding section we saw that it is the objective of a computation to effect a change of state as laid down in a functional specification. Because, as a rule, such a functional specification contains a number of unspecified parameters – denoted in our examples by X, Y or Z – a functional specification usually describes a large class of changes of state. The program must indicate how these changes of state are to be effected.

For a limited class of changes of state this can be effected in a single step: the initial state is then directly rendered into the final state. Such computations take very little time; they correspond to the building blocks with which we can build complicated programs, which can give rise to longer computations. During such a longer computation the total change of state is effected by the succession of a (possibly large) number of 'small' changes, that is, changes that can be effected directly in one step. The corresponding blocks from which the program is constructed are called **assignment statements**, and the following section discusses how assignment statements are written in a program, and which changes of state correspond to each assignment statement.

Remark. Two-fold use is made of our formalism for functional specification. For `euclid`, the functional specification should be seen as the formulation of a programming exercise. For `skip` the functional specification was used for the definition of a building block available to the programmer. This latter use will be followed in the introduction of the assignment statement.

The assignment statement

Consider the functional specifications of the following programs which, for lack of imagination, we shall call S0, S1 and S2:

```
|[ x, y: int {X = 0  ∧  Y = y}; S0 {X = x  ∧  Y = y} ]|
|[ x, y: int {X = 88  ∧  Y = y}: S1 {X = x  ∧  Y = y} ]|
|[ x, y: int {X = x + 3  ∧  Y = y}; S2 {X = x  ∧  Y = y} ]|
|[ x, y: int {X = 7 * y  ∧  Y = y}; S3 {X = x  ∧  Y = y} ]|
|[ x, y: int {X = 10 *(x − y)  ∧  Y = y}; S4 {X = x  ∧  Y = y} ]|
```

Remark. In itself it would have been simpler to specify, for S0:

```
|[ x, y: int {y = y}; S0 {x = 0  ∧  y = Y} ]|
```

This specification also expresses neatly that the value of x is zero after execution of S0 (regardless of its initial value: after all, x does not occur in the precondition), and that the value of y has remained unchanged. (Check that the functional specification of S1 could be simplified in a similar way.) In this list such a simplification has deliberately not been carried out, so that the functional specifications are as similar as possible. ■

The functional specifications given above all have the same pattern, namely:

```
|[ x, y: int {X = E  ∧  Y = y}; Si {X = x  ∧  Y = y} ]|
```

with E taking the values 0, 88, x + 3, 7 * y and 10 *(x − y) respectively, and with the name Si.

Our program notation offers the possibility of writing such programs as follows, by means of **assignment statements**:

```
for S0:  x:= 0
for S1:  x:= 88
for S2:  x:= x + 3
for S3:  x:= 7 * y
for S4:  x:= 10 *(x − y)
```

(Pronounce the assignment operator := as 'becomes', that is, 'x becomes zero', 'x becomes eighty-eight', 'x becomes x plus three', 'x becomes seven times y' and 'x becomes ten times opening bracket x minus y closing bracket.)

The postulate of the assignment implies that for each permissible expression E the program x:= E satisfies the functional specification

```
|[ x, y: int {X = E  ∧  Y = y}; x:= E {X = x  ∧  Y = y} ]|
```

Any declared variable may be chosen for x; the postulate of the assignment also implies that for any permissible expression E the program y := E satisfies the functional specification

```
|[ x, y: int {X = x  ∧  Y = E}; y:= E {X = x  ∧  Y = y} ]|
```

Even more variables could have been declared; the postulate of the assignment then also implies that for any permissible expression E the program x:= E satisfies the functional specification

```
|[ x, y, z: int
   {X = E  ∧  Y = y  ∧  Z = z}
 ; x:= E
   {X = x  ∧  Y = y  ∧  Z = z}
]|
```

In the above 'permissible expressions' have been mentioned several times without being defined further. However, a definition of which expressions can be regarded as permissible will be postponed. In the meantime, it is sufficient to know that the given examples of expressions are permissible in places where the declaration x, y, z: int is in force. First an example will show how the postulate of the assignment is used to make more particular assertions about a specific assignment statement, such as x:= x + 3; for example, under which precondition x ≧ y will hold after execution. For the sake of clarity, consider this problem in the state space described by x, y, z: int: z then denotes the other variables.

By substituting the permissible expression x + 3 for E in the last formulation of the postulate of the assignment, we get

```
|[ x, y, z: int
   {X = x + 3  ∧  Y = y  ∧  Z = z}
```



```
 ; x:= x + 3
   {X = x  ∧  Y = y  ∧  Z = z}
]|
```

an assertion which holds for all X, Y and Z, in particular for all X, Y and Z which satisfy X ≧ Y. Restricting the application to these cases, we derive

```
|[ x, y, z: int
   {X = x + 3  ∧  Y = y  ∧  Z =z  ∧  X ≧ Y}
 ; x:= x + 3
   {X = x  ∧  Y = y  ∧  Z = z  ∧  X ≧ Y}
]|
```

The validity of the postcondition implies x ≧ y, so that the assertion

```
|[ x, y, z: int
   {X = x + 3  ∧  Y = y  ∧  Z = z  ∧  X ≧ Y}
 ; x:= x + 3
   {x ≧ y}
]|
```

holds for all X, Y and Z. By rewriting the precondition we may draw the same conclusion for

```
|[ x, y, z: int
   {X = x + 3  ∧  Y = y  ∧  Z = z  ∧  x + 3 ≧ y}
 ; x:= x + 3
   {x ≧ y}
]|
```

Since:

(i) this latter assertion holds for all X, Y and Z;

(ii) X, Y and Z occur only in the initial part of the precondition

   X = x + 3 ∧ Y = y ∧ Z = z

   and this initial part, regarded as an equation in the unknowns X, Y and Z, can be solved for all values of x, y and z;

we can eliminate X, Y and Z by dropping this initial part. We then get:

```
|[ x, y, z: int
   {x + 3 ≧ y}
 ; x:= x + 3
   {x ≧ y}
]|
```

In words: the initial validity of $x + 3 \geqq y$ justifies the conclusion that, after execution of the assignment statement x:= x + 3, the relation $x \geqq y$ holds. (This conclusion is obviously weaker than what we knew already: it no longer expresses that the execution of x:= x + 3 leaves the values of y and z intact.)

Here, x + 3 was a very particular choice for the right-hand side of the assignment statement. Analogously, we could have derived

```
|[ x, y, z: int {E ≧ y}; x:= E {x ≧ y} ]|
```

from the postulate of the assignment for any permissible expression E.

The choice of the postcondition $x \geqq y$ was also arbitrary. If we had, for example, chosen $z \cdot (x + 1) \leqq (y + 3) \cdot x$, then we would have derived:

```
|[ x, y, z: int
    {z · (E +1) ≦ (y + 3) · E}
 ; x:= E
    {z · (x + 1) ≦ (y + 3) · x}
]|
```

The general pattern for finding the corresponding precondition for the assignment statement x:= E and given postcondition R is obviously that we substitute the expression E for x in R, if necessary in brackets. It is common practice to denote this substitution result by R^x_E. With this convention we can sum up our rule as:

$$|[\ x, y, z: \text{int}\ \{R^x_E\};\ x:= E\ \{R\}\ ]|$$

Remark. Application of the rule to the postcondition $\neg R$ yields the assertion:

$$|[\ x, y, z: \text{int}\ \{\neg R^x_E\};\ x:= E\ \{\neg R\}\ ]|$$

From this we see that the initial validity of R^x_E is not only sufficient, but also necessary for x:= E to effect a state in which R holds. ■

Permissible expressions

A program is a set of instructions that can be executed by a computer. This means that there must be no misunderstanding about what the instructions imply. After reading this far, few people will question that the meaning of the assignment statement

```
x:= 2 * x
```

is that the value of x is doubled. But opinions about the aim of

```
x:= x / 2 * 6
```

are divided (if you ask a large enough number of people). In the Netherlands, where traditionally multiplication comes before division, the view that

```
x:= x / (2 * 6)
```

is meant will prevail. In countries with other traditions, however, it will be assumed that

```
x:= (x / 2) * 6
```

is meant. It is clear that these kinds of ambiguities must be ruled out by exact definitions. This inevitably requires us to define just as exactly *which* expressions have an unambiguous meaning. This section is concerned with this definition. In passing, the most common formalism used for giving such definitions will be introduced, i.e. BNF (Backus–Naur Form, named after John Backus and Peter Naur). BNF became widely known by the way in which it was used in the famous *Algol 60 Report* of January 1960.

Just about the simplest permissible expression is the natural number. BNF will now be used to define what natural numbers look like on paper. Since the notation of natural numbers consists of digits, we shall first define what forms there are for digits. In BNF this is given by the **syntax rule**:

$$\langle \textit{digit} \rangle ::= 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$$

To the left of the sign ::= (pronounced 'is defined as') is the name of the syntactic unit to be defined, placed in angle brackets; to the right of the ::= sign are the forms of the syntactic unit, separated by a vertical mark, | (pronounced 'or'). The rule states which ten characters are digits and, moreover, that if later on we come across ⟨*digit*⟩ in a syntactic formula, this can denote any one of these ten characters.

Remark. The order in which alternative forms are listed in a syntax rule is irrelevant. We could have defined the syntactic unit *digit* just as well by:

$$\langle \textit{digit} \rangle ::= 9 \mid 8 \mid 7 \mid 6 \mid 5 \mid 4 \mid 3 \mid 2 \mid 1 \mid 0 \qquad \blacksquare$$

Now we have the tool to define, if we should want to, which character sequences belong to the syntactic unit *number under thousand*:

$$\begin{array}{l} \langle \textit{number under thousand} \rangle \\ \quad ::= \langle \textit{digit} \rangle \\ \quad\quad \mid \langle \textit{digit} \rangle \langle \textit{digit} \rangle \\ \quad\quad \mid \langle \textit{digit} \rangle \langle \textit{digit} \rangle \langle \textit{digit} \rangle \end{array}$$

The above definition tells us that a *number under thousand* has three alternative forms: a single digit, two digits in a row, or a sequence of three digits. It would be a dismal writing lesson to write out in this way the forms of *number under billion*. It would be hopeless to write in this way the forms of a natural number. In BNF the syntactic unit of a natural number is given by:

$$\begin{array}{l} \langle \textit{natural number} \rangle \\ \quad ::= \langle \textit{digit} \rangle \\ \quad\quad \mid \langle \textit{digit} \rangle \langle \textit{natural number} \rangle \end{array}$$

This is a **recursive definition**: the syntactic unit defined occurs in its own definition (namely in the second alternative)! The first confrontation with a recursive definition usually sends a shiver down one's spine: one cannot help thinking of the snake that bites its own tail and begins to eat itself, continuing until nothing is left. Having overcome this shiver, however, people learn to appreciate recursive definitions; without them we would

not be able to define a syntactic unit with an unlimited number of forms.

On thinking it over, this definition of (the syntax of) *natural number* is not as uncanny as it may seem at first sight, thanks to the presence of the first alternative. In the definition of *digit*, the first alternative presents us with ten forms of a natural number (and by this the first alternative is, as it were, exhausted). With these ten forms as possible substitutes for *natural number* in the second alternative, 100 new forms are yielded, with these 100 as possible substitutes in the second alternative we get 1000 new forms, etc.

Remark. The syntax rule

$$\begin{array}{l} \langle \textit{natural number} \rangle \\ \quad ::= \langle \textit{digit} \rangle \\ \quad\quad \mid \langle \textit{natural number} \rangle \langle \textit{digit} \rangle \end{array}$$

is equivalent to the one given before. Both define the set of finite, non-empty sequences of digits. ■

In an analogous way to our definition of *digit* we define:

$$\begin{array}{ll} \langle \textit{letter} \rangle ::= & \mathrm{a \mid b \mid c \mid d \mid e \mid f \mid g \mid h \mid i \mid j \mid k \mid l \mid m} \\ & \mathrm{n \mid o \mid p \mid q \mid r \mid s \mid t \mid u \mid v \mid w \mid x \mid y \mid z} \\ & \mathrm{A \mid B \mid C \mid D \mid E \mid F \mid G \mid H \mid I \mid J \mid K \mid L \mid M} \\ & \mathrm{N \mid O \mid P \mid Q \mid R \mid S \mid T \mid U \mid V \mid W \mid X \mid Y \mid Z} \end{array}$$

Remark. With this it has been defined that an alphabet of 52 different characters will be used. Note that:

- the digit 0 and the letters o and O are different characters;
- the digit 1 and the letters l and I are different characters;
- the digit 9 and the letter g are different characters.

Furthermore, since the order of the alternatives in a syntax rule has no meaning, the 52 letters are not combined as pairs, and therefore the letter sequence *dog* has nothing to do with the letter sequences *Dog* or *DOG*. ■

These examples have introduced local variables, for instance by the declaration

```
x, y, z: int
```

introducing their 'names' (x, y and z respectively) in passing. Just as the forms of a natural number have been defined, those of a name will now be defined.

Remark. In the English literature the standard term for *name* is 'identifier'. We shall stick to the syntax of Algol 60 identifiers for names. ■

⟨*name*⟩ ::= ⟨*letter*⟩
| ⟨*name*⟩ ⟨*letter*⟩
| ⟨*name*⟩ ⟨*digit*⟩

EXERCISE

Check that there are 199 888 different names of three characters.

Now we are ready for the definition of the syntax of expressions which are permissible. At this stage we shall confine ourselves to the syntactic unit called **integer expression**; the syntax describes how integer expressions are constructed from:

- names and natural numbers;
- additive operators;
- multiplicative operators;
- pairs of brackets.

Remarks. At this stage the text is deliberately confined to a modest syntax for integer expressions. Later this will be expanded a little. ■

⟨*integer expression*⟩
::= ⟨*intterm*⟩
| ⟨*addop*⟩ ⟨*intterm*⟩
| ⟨*integer expression*⟩ ⟨*addop*⟩ ⟨*intterm*⟩

This syntax rule tells us that an integer expression is a sequence in which specimens of the syntax unit *addop* and specimens of the syntax unit *intterm* alternate, and which ends with a (specimen of the syntax unit) *intterm*. All we have to do now is to state what an *addop* and an *intterm* may lok like. For the former this is more simple than it is for the latter.

⟨*addop*⟩ ::= + | −

⟨*intterm*⟩
::= ⟨*intfactor*⟩
| ⟨*intterm*⟩ ⟨*multop*⟩ ⟨*intfactor*⟩

This completes the definition of *addop*: a plus sign or a minus sign. The following syntax rule tells us that an *intterm* consists of one or more specimens of the syntactic unit *intfactor*, separated by a specimen of the syntactic unit *multop*. And all we have to do now is to state what a *multop* and an *intfactor* may look like; here again this is simpler for the former than it is for the latter:

⟨*multop*⟩ ::= * | / | **div** | **mod**

⟨*intfactor*⟩
::= ⟨*natural number*⟩
| ⟨*name*⟩
| (⟨*integer expression*⟩)

This completes the syntactic definition of the integer expression. Analysis of an example will show that

```
- ab - x mod 3 + 8 *(y + 1)
```

is an integer expression.

This holds because:

(0) - ab - x mod 3 is an *integer* expression
(1) + is an *addop* (self-evident)
(2) 8 *(y + 1) is an *intterm*

Assertion (0) holds because:

(0.0) − ab is an *integer* expression
(0.1) − is an *addop* (self-evident)
(0.2) x mod 3 is an *intterm*

Assertion (0.0) holds because:

(0.0.0) − is an addop (self-evident)
(0.0.1) ab is an *intterm*

Assertion (0.0.1) holds because:

(0.0.1.0) `ab` is an *intfactor*

Assertion (0.0.1.0) holds because:

(0.0.1.0.0) `ab` is a *name*

Assertion (0.0.1.0.0) holds because:

(0.0.1.0.0.0) `a` is a *name*
(0.0.1.0.0.1) `b` is a *letter* (self-evident)

Assertion (0.0.1.0.0.0) holds because:

(0.0.1.0.0.0.0) `a` is a *letter* (self-evident)

Thus assertion (0.0) is proved; (0.1) is self-evident, and (0.2) holds because:

(0.2.0) `x` is an *interm*
(0.2.1) <u>mod</u> is a *multop* (self-evident)
(0.2.2) `3` is an *infactor*

Assertion (0.2.0) holds because:

(0.2.0.0) `x` is an *infactor*

Assertion (0.2.0.0) holds because:

(0.2.0.0.0) `x` is a *name*

Assertion (0.2.0.0.0) holds because:

(0.2.0.0.0.0) `x` is a *letter* (self-evident)

Thus assertion (0.2.0) is proved; (0.2.1) is self-evident, and (0.2.2) holds because:

(0.2.2.0) `3` is a *natural number*

Assertion (0.2.2.0) holds because:

(0.2.2.0.0) `3` is a *digit* (self-evident)

With this (0.2.2) is proved, and thus (0.2), and thus (0.2), and thus (0); (1) is self-evident and the proof of (2) is left to the reader.

We do not suggest that he or she produce such a long-winded proof as this for (0), but that the reader should be aware of each appeal to formal syntax. The analysis is given in such small steps in order that: (1) the reader can imagine that this can be done by a mechanism; and (2) the reader may appreciate the fact that a formal definition of the syntax is essential for this. (The development and first implementation of FORTRAN in the mid-1950s took 100 times more man-years than the first implementation of Algol 60 in 1960. The two projects differed enough to warrant some caution in interpreting this factor of 100; however, they were similar enough to tend to explain this factor, among other things, by the circumstance that there was no formal definition of FORTRAN.)

Furthermore, this syntax of integer expressions makes it explicity clear that additive operators are 'left-associative'; that is, an integer expression like `– a – b + c`, for example, is an integer expression only in the analysis

```
( (– a) – b) + c
```

and that interpretations such as

`(– a) – (b + c)` and `–(a – (b + c)`

are not permitted. So the syntax for integer expressions does more than merely define what sequences of symbols are integer expressions: it also indicates how the expression must be interpreted; that is, which constants and which intermediary results must be the operands of which operators when the expression is being computed.

EXERCISE

Determine why `2k + 1` is not a syntactically correct integer expression.

Remark. The multiplicative operators are also defined left-associatively and no priority has been assigned to multiplication over division: `m / 2 * h` is thus short for `(m / 2)* h` and *not* for `m /(2 * h)`. If the latter is intended, the brackets cannot be left out. The advice not to be too stingy with pairs of brackets has a much wider application, however. There is often no consensus about what can be left out without a change of meaning. ■

It only remains for us now to define the operators. If the additive operators occur as binary operators, the addition is denoted by + and the subtraction by −, and if they occur as a unary operator – that is, as the first symbol of an integer expression – the + has no effect and the − denotes a change of sign. The reader presumably knows what is meant by addition, subtraction and change of sign.

The remaining multiplicative operators, /, div and mod are **partial operators**, that is x / y, x div y and x mod y are not defined for all pairs of integer values (x, y).

x / y denotes the quotient of x and y, which naturally is defined only if $y \neq 0$; furthermore, in using x / y, we agree to confine ourselves to those situations in which x is an integer number of times y.

For x div y and x mod y there is only the restriction $y \neq 0$: x div y = q and x mod y = r, where integer q and r satisfy:

$$x = q \cdot y + r \wedge 0 \leq r < \mathrm{abs}(y)$$

Note that:

$$x \ \underline{\mathrm{mod}}\ y = x \ \underline{\mathrm{mod}}(-y)$$
$$(-x)\underline{\mathrm{div}}\ y \neq -x \ \underline{\mathrm{div}}\ y, \text{ unless } x \ \underline{\mathrm{mod}}\ y = 0$$
$$(x + y)\underline{\mathrm{mod}}\ y = x \ \underline{\mathrm{mod}}\ y$$
$$(x + y)\underline{\mathrm{div}}\ y = 1 + x \ \underline{\mathrm{div}}\ y$$

This completes, for now, the description of what we had indicated as permissible expressions.

Concatenation of statements

Until now we have only come across the assignment statement in the form x := E, and so far, the programs we can write are rather limited for two reasons. Firstly, the change of state they can effect is restricted to the change of the value of one variable of the state space; secondly, its new value is restricted to what we can express by means of a permissible expression. We shall now see how the first restriction is overcome.

In all places where we can write a statement, we can also write a *statement list*

⟨*statement list*⟩
  ::= ⟨*statement*⟩
   | ⟨*statement list*⟩ ⟨*statement*⟩

that is, a list of one or more statements, in the latter case connected by the semicolon. The semicolon, which is written between two consecutive statements, connects those two statements in the sense that execution of the left-hand statement must be followed by that of the right-hand statement, and that the postcondition of the left-hand statement is identified as the precondition of the right-hand statement.

At the end of our discussion about the assignment statement (page 22) the general structure of assertions about a single assignment statement was given as:

```
|[ x, y, z: int {Q}; x:= E0 {R} ]|
```

This means that if, before the execution of x := E0, Q is satisfied, then after execution of x := E0 the state satisfies R; this assertion is true if Q equals R^{x}_{E0}, i.e. the condition R in which the expression E0 is substituted for x – in brackets if necessary, but we shall no longer repeat this every time.

With $P = Q^{Y}_{E1}$

```
|[ x, y, z: int {P}; y := E1; x:= E0 {R} ]|
```

is also a correct assertion, this time about the assignment statement y := E1. The postulate about concatenation implies that these two assertions may be combined to

```
|[ x, y, z: int {P}; y:= E1 {Q}; x:= E0 {R} ]|
```

or, if we eliminate Q by suppressing it, to the assertion about y := E1; x := E0:

```
|[ x, y, z: int {P}; y := E1; x:= E0 {R} ]|
```

Now we can also check what assertions we can make if we switch the two statements, that is, assertions of the form:

```
|[ x, y, z: int {P'}; x := E0; y := E1 {R} ]|
```

Working from right to left, we first form $Q' = R^{Y}_{E1}$ and then $P' = Q'^{X}_{E0}$. In general, $P \neq P'$ – namely if x occurs in E1, or y in E0. In other words, concatenation of statements is generally *not* commutative.

As an example we shall derive P, so that the assertion

```
|[ x, y, z: int {P}
 ; x:= x +y; y:= x - y; x:= x - y
   {x = X  ∧  y = Y  ∧  z = Z}
]|
```

holds.

To begin with we introduce the suppressed intermediary conditions – two by now:

```
|[ x, y, z: int {P}
 ; x:= x + y {Q1}
 ; y:= x - y {Q0}
 ; x:= x - y
   {x = X  ∧  y = Y  ∧  z = Z}
]|
```

Working back to front, by substituting x − y for x we get:

```
Q0:  x - y = X  ∧  y = Y  ∧  z = Z
```

Substituting y by x − y in this, we get

```
Q1:  x - (x - y) = X  ∧  x - y = Y  ∧  z = Z
```


and after simplification

```
Q1:  y = X  ∧  x - y = Y  ∧  z = Z
```

Substituting x by x + y in this, we get

```
P:  y = X  ∧  (x + y) - y = Y  ∧  z = Z
```

and after simplification

```
P:  y = X  ∧  x = Y  ∧  z = Z
```

If we compare P with the postcondition, we see that the 'continued concatenation'

```
x:= x + y; y:= x - y; x:= x - y
```

swaps the values of x and y and leaves all other variables undisturbed.

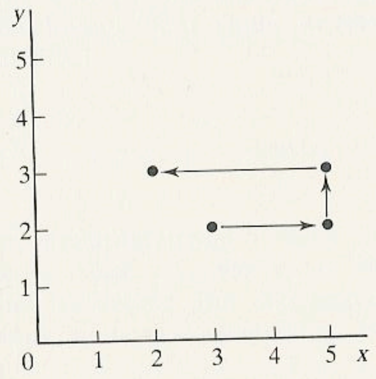

Figure 4

In Figure 4, showing a projection of the state space on the plane *x*, *y*, the path for the special case X = 2 ∧ Y = 3 is indicated. The figure is given only as an illustration of the fact that the statement concatenation enables us to construct programs that effect the desired change of state not at one go, but in a number of consecutive steps. In this metaphor the computation becomes a path through the state space which leads from the starting point to the end point. We shall see later that in actual practice these paths tend to be traversed in a great many steps.

Figure 4 is *only* an illustration of the metaphor. In programming practice such figures are never drawn. They would have the practical drawback of becoming terribly complicated and the fundamental drawback of referring only to a very specific case (in this case X = 2 ∧ Y = 3).

Remark. The continued concatenation above was produced only for reasons of illustration. Later we shall come across more realistic solutions to exchange the values of two variables. ■

The alternative statement

In the preceding section we saw that for a given program the path through the state space depends on the initial state, but as long as we have only assignment statements and their concatenation at our disposal, the same series of statements will always be executed, independently of the initial state. We shall now show that for a given program the initial state can also determine *which* statements will be executed. A study of the functional specification of the program `largest` will reveal the need for this greater flexibility:

```
|[ x, y, z: int {x = X  ∧  y = Y}
 ; largest {x = X  ∧  y = Y  ∧  z = max(x, y)}
]|
```

The problem would be trivial if `max(x, y)` were among the permissible expressions, since `z := max(x, y)` would then satisfy the functional specification of `largest`. But since `max(x, y)` is not among the permissible expressions, we must do something else.

Remark. There are programming languages that allow the programmer to add `max(x, y)` to the set of permissible expressions. However, the addition requires a solution to the problem of how to write a program that satisfies the functional specification of `largest` in terms of the expressions already permissible. ■

To begin with, we observe that with the postcondition for `largest` we can derive, by means of the postulate of the assignment statement for `z := x`:

```
|[ x, y, z: int {x = X  ∧  y = Y  ∧  x = max(x, y)}
 ; z:= x {x = X  ∧  y = Y  ∧  z = max(x, y)}
]|
```