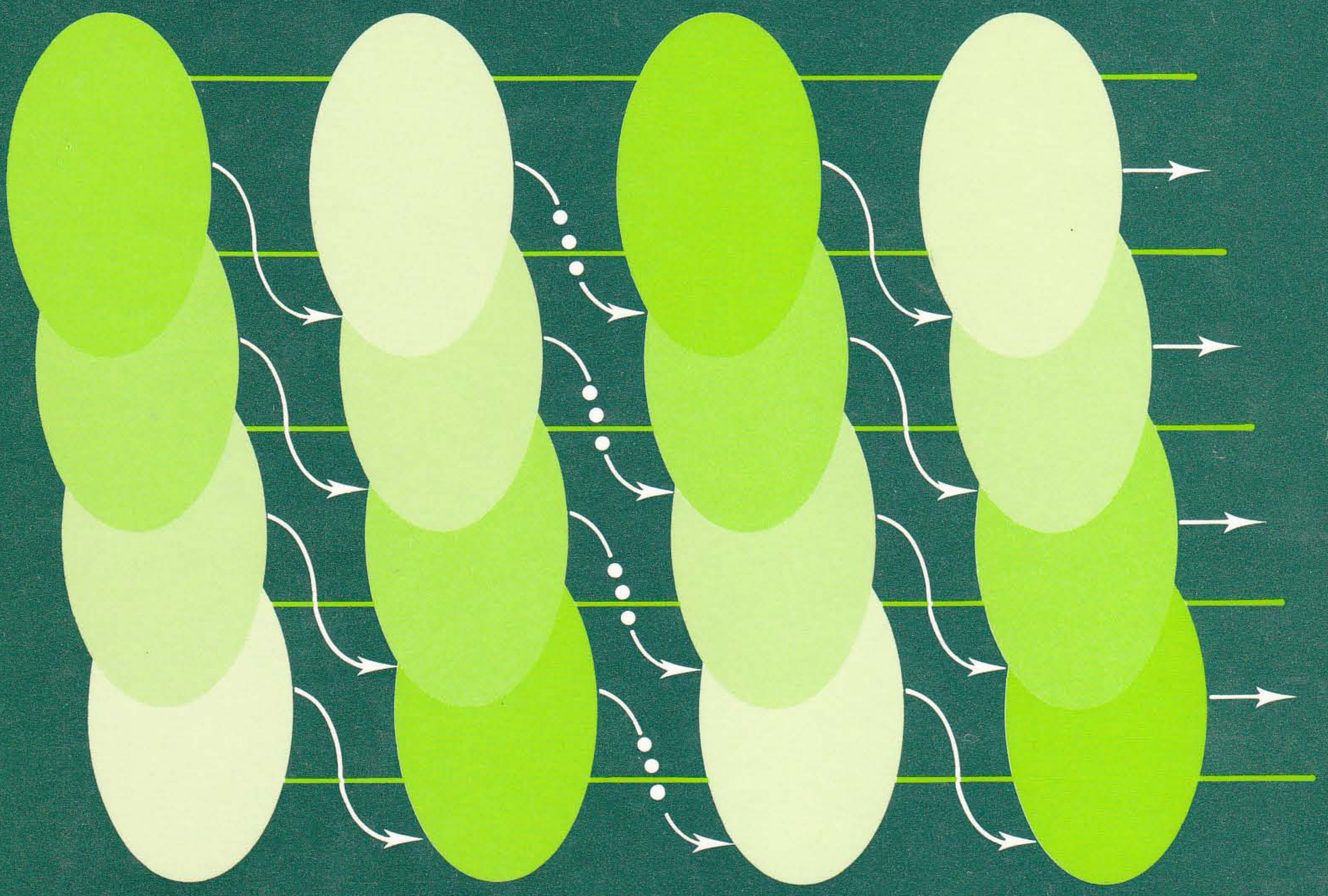
THE SPINE OF SOFTWARE

DESIGNING PROVABLY CORRECT SOFTWARE
THEORY AND PRACTICE



ROBERT L. BABER

THE SPINE OF SOFTWARE

DESIGNING PROVABLY CORRECT SOFTWARE: THEORY AND PRACTICE

or

A Mathematical Introduction to the Semantics of Computer Programs

Robert Laurence Baber

JOHN WILEY & SONS LTD
Chichester · New York · Brisbane · Toronto · Singapore

Library of Congress Cataloging-in-Publication Data:

Baber, Robert Laurence.
The spine of software: designing provably correct software, theory and practice.
Subtitle: Being a treatise on the mathematical foundations and principles of computer programming for software engineers and those who would be or become such.
Bibliography: p.
Includes index.
1. Computer software—Development. 2. Electronic digital computers—Programming. I. Title. II. Title: Mathematical introduction to the semantics of computer programs.
QA76.76.D47B33 1987 005.1 86–32483
ISBN 0 471 91474 6

British Library Cataloguing Publication Data:

Baber, Robert Laurence
The spine of software: designing provably correct software, theory and practice: or, a mathematical introduction to the semantics of computer programs: being a treatise on the mathematical foundations and principles of computer programming for software engineers and those who would be or become such.
1. Electronic digital computers—Programming
2. Electronic data processing—Mathematics
I. Title
005.13'1 QA76.6
ISBN 0 471 91474 6

Typeset by Photo-Graphics, Honiton, Devon
Printed and bound in Great Britain by Biddles Ltd, Guildford

This book is dedicated

with my deep gratitude
to those who, over the years, taught me mathematics

—that wonderful discipline which,
providing the basis and structure
for logical thought and reasoning,
is applicable to all fields of intellectual endeavour
if one is only broadminded enough
to step back and look at the forest abstractly
instead of examining the trees and leaves in isolated detail,
employing overspecialized techniques—

in the hope that they will approve of this work,
the indirect result of some of their efforts on my behalf

and

to the developers and users of tomorrow's software,
with my hopes and wishes that they will not have to resuffer
the consequences of our many past omissions and mistakes.

Acknowledgements

The material presented in this book is based extensively upon the contributions of others to the computing scientific literature. Citing such contributions at each relevant point in the text would interrupt the development of many arguments excessively and reduce the readability considerably. I decided, therefore, to include citations to the scientific literature only in rather specific cases; especially I have avoided citing one work in many places in this book. While this approach, in my opinion, best serves the interests of the reader, it clearly does not do justice to the many people who have contributed in valuable ways to the development of the mathematical and theoretical foundation of software engineering presented in this book. I would like to try to correct this injustice by thanking such contributors explicitly here. Those works of theirs which I have used are included in the bibliography.

These computing scientists are, unfortunately, too numerous to list individually here. The writings of two of them, however, have so significantly influenced my thinking leading to this book that I do want to mention them by name: Professor Edsger W. Dijkstra and Professor C. A. R. Hoare. I hope that this book increases the value of the results of their efforts by making some of them accessible to a larger fraction of the population of contemporary and future software developers.

I would like to thank Tim Denvir, Professor Andrew D. McGettrick and Roger M. Pickering, who reviewed drafts of this book and made valuable suggestions. I am also very much indebted to Drs Willem Dijkhuis, whose important contributions to this book included its title.

Finally, I would like to thank the programmers of the text processing system which I used to prepare the typescript of this book. As a result of their successful efforts, I was able to save much time and effort. As a result of the less successful aspects of their work – several significant 'bugs' – I was reminded of the great need for a book such as this one. For the sake of the users of their future programs, I hope that they will be among its readers.

ROBERT LAURENCE BABER
Landgraf Gustav Ring 5
6380 Bad Homburg v.d.H.
Federal Republic of Germany
1986 September

Guide for the reader

This book is subdivided into four main parts:

Prologue (Chapters 0 and 1)
Theory (Chapters 2, 3 and 4)
Practice (Chapters 5 and 6)
Epilogue (Chapters 7 and the appendices)

Depending upon your interests, background knowledge and motivation for reading this book, you will find one of the following approaches to this book's contents most suitable for your purposes.

If you are a serious student of software engineering, either employed as a software developer or studying at an academic institution, you will want to read and study the contents of this book in the sequence presented. Section 2.2 and Chapter 4 may be skimmed on your first reading.

If you are a software developer interested primarily in applying this material to practical programming problems, and if you are willing to accept the main theoretical results without proof, i.e. axiomatically or by intuition, then you will find the following sequence to be the quickest path to your goal:

skim the introductory material in Chapters 0 and 1,
read the definitions, theorems and lemmata (but not the proofs) in Chapter 3, concentrating especially on the first definition of a precondition and on Sections 3.0.1, 3.1.0, 3.1.1, 3.2.0, 3.3.0, 3.3.3 and 3.9,
read Chapter 5 and
study Chapter 6.

Afterward, if you are interested in examining and understanding the theoretical foundation more thoroughly, you should read Chapter 2 and the rest of Chapter 3 and skim Chapter 4. Chapter 7 will give you a preview of the future of your chosen field.

If you are not adequately familiar with the mathematical terms and notation which you encounter, you should read Appendix 0, Mathematical fundamentals, before proceeding further.

If you are a manager of software development interested primarily in assessing this material as a foundation for your subordinates' work, you will want to skim Chapter 1 and read Chapters 6 and 7.

Many readers will find Chapters 5 and 6 useful for later reference.

Contents

List of figures

Prologue

Chapter 0

Komputema Simio, the computing monkey of Moc

Monkeys ... very sensibly refrain from speech, lest they should be set to earn their livings.

– Kenneth Grahame

Give me a place to stand and I can move the earth.

– Archimedes

Accurate reckoning of entering into things, knowledge of existing things all, mysteries... secrets all.

– A'h-mose

In 2500 BC the land of the Ret Up Moc was one of the more advanced societies in the cradle of civilization. An active foreign trade had developed and several cities had been founded. A construction industry existed in which professionally trained architects and civil engineers played an important role.

Between about 2500 and 2400 BC, a major technical advance was achieved. Suddenly and unexpectedly, a group of teachers of civil engineering developed a new technique for designing buildings. Using the new method, buildings could be designed and constructed which were much larger than those previously possible. Perhaps even more importantly, the new method reduced construction costs to about a tenth of their former levels (see Baber, 1982, Ch. 0).

As a consequence, the demand for buildings of all types increased very rapidly – exploded is probably a more accurate term – and the construction industry grew correspondingly. Especially the professional architects and civil engineers had great difficulty satisfying the much increased demand for their services.

Quite extensive calculations of a somewhat repetitive nature had to be made when applying the new design approach. Akado, one of the civil

engineers who contributed significantly to the advancement of the new method, recognized that the capability to calculate faster was an essential prerequisite for any significant increase in productivity. He thought about several ways this might be achieved, including organizing the computation so that a less skilled person could perform most of the calculations, thereby freeing the designer's time for those aspects of the task requiring his special knowledge and skill. Because such specialized knowledge and skill was in short supply throughout the profession, this approach held out some promise of being useful not only to Akado himself, but also to the profession as a whole.

Among the colleagues and friends with whom Akado discussed this problem and his ideas was Kolab, a leading zoologist in Moc. Kolab's specialty was monkeys, in particular primate intelligence, communication and behavior. Kolab found the discussion with Akado interesting and made a few minor suggestions but had no really significant ideas on how to solve the problem.

A short time later, Kolab led an expedition into a jungle some days' journey from Moc. His goal was to observe primates in the wild and to collect some specimens for a zoo in Moc. Kolab's thoughts were far from Akado's calculations when suddenly he came upon a clearing in which a monkey was sitting, manipulating pebbles and making marks in the dirt with a stick. Kolab observed the monkey for quite some time, trying to figure out what it was doing. Kolab could only conclude that the monkey was counting the pebbles. This led him to wonder whether it would be possible to train this obviously intelligent monkey to perform Akado's calculations. Kolab easily captured the unusually agreeable, friendly monkey and brought him back to Moc.

In Moc, Kolab discussed his idea with Akado, who was interested in exploring its possibilities. Kolab began to train the monkey (which they named Komputema Simio) to perform, on command, tasks which were of a counting and calculating nature. While Kolab trained the monkey, Akado reorganized his calculations around particularly simple individual computational steps. After some time and effort, they succeeded in implementing the following scheme.

Akado prepared a sequence of computational instructions, wrote them on paper-like sheets and handed them to the monkey. In addition to these prepared instruction sheets, the monkey had an adequate supply of data sheets. Each data sheet was divided into three columns, the first labeled 'name', the second marked 'set' and the third called 'value'. Finally, the monkey had a supply of blank sheets (scratch paper) for making intermediate calculations as needed, a pencil-like instrument for writing on the data sheets and scratch paper and a few small pebbles which he used to keep

track of his place in the list of instructions. Sometimes the monkey would write on the dirt ground with an ordinary stick instead of using the scratch paper.

Kolab successfully trained the monkey to act in this environment as follows. The monkey would carry out the first command on the instruction sheet by performing the calculation it implied. Normally, this involved substituting certain values from the data sheet into an expression, calculating the value of the expression and writing the result in the appropriate place on the data sheet. The monkey would then proceed to the next command on the instruction sheet, carry it out in the same manner and repeat this process indefinitely. Two special commands modified the sequence of execution of the other instructions (see below).

Performing each instruction involved following certain precise rules. If the rules, when interpreted in a specific context, were inconsistent or ambiguous, the monkey would signal such an erroneous condition by ringing a bell. He would then go and play while Akado corrected the error.

The normal and most frequently occurring command was of the form

name := expression

for example,

$$m := P * a * (1 - x/L)$$

If the contents of the data sheet were as follows:

name	set	value
m	num	2500
P	num	1000
a	num	2
L	num	6
x	num	3

before the monkey started to execute the above instruction, he would substitute the corresponding values into the expression and calculate its value as follows:

$$1000 * 2 * (1 - 3/6) = 2000 * (1 - 0.5) = 2000 * 0.5 = 1000$$

This result is to become the new value of the variable m. The monkey would, therefore, verify that the calculated result is consistent with the entry opposite the name m in the column 'set' of the data sheet. This being the case in our example, Komputema would record the result calculated above as the new value of the variable m, changing the data sheet to read as follows:

name	set	value
m	num	~~2500~~ 1000
P	num	1000
a	num	2
L	num	6
x	num	3

The monkey would then proceed to the next instruction and repeat the process described above.

Whenever a calculated result was inconsistent with the result variable's set or with the operations to be performed as specified in the expression, the monkey would signal the error by ringing a bell as mentioned above.

The monkey had been trained to work with three types of variables on the data sheet: numbers (as in the above example), logical variables (which could assume only the values 'true' and 'false') and sequences of individual symbols (letters, digits and other special characters). Kolab was interested in extending the monkey's repertoire to include other types of variables, but Akado saw no real need for such improvements.

Komputema had learned to perform the following basic calculations on these three types of variables: adding (+), subtracting (−), multiplying (*), dividing (/), comparing for equality (=), comparing for order (<, >, ≤ and ≥), appending one sequence to another, i.e. concatenating (:), and the logical calculations **not**, **and** and **or**. Kolab had the impression that this collection of operations taxed the monkey's intelligence and capabilities to the limit. Fortunately, these operations were all that Akado needed or even considered desirable for his purposes.

The sequence of execution of instructions could be modified by two commands or, more properly, command structures. One was the conditional statement, which Akado wrote in the following form:

if expression
then
    ...
else
    ...
endif

When the monkey encountered this structure, he would first evaluate the expression between the **if** and the **then**. If the result was the value 'true', he would then carry out the instructions between the **then** and the **else**. If, on the other hand, the value of the expression was 'false', he would execute the commands between the **else** and the **endif**. In either event, he would continue executing the instructions following the **endif** or as determined by another, higher level command structure. If the result of evaluating the expression was neither 'true' nor 'false', Komputema would signal the error by ringing the bell as described above.

The third type of instruction which Kolab was able to teach the monkey had the form

while expression **do**
...
endwhile

Komputema would begin to execute this command by evaluating the expression. If its value was 'false', he would skip the entire construct as if it were not present and continue. If, on the other hand, the value of the expression was 'true', he would execute the instructions between the **do** and the **endwhile** and then execute the entire **while** construct again, reevaluating the **while** expression, etc. If the value of the expression was neither 'true' nor 'false', the monkey would signal the error by ringing the bell as usual.

For computational purposes, Akado found that he needed only these commands and structures. For preparing records of the calculations for inclusion in his documentation on a design, however, these seemed to be inadequate. A command something like **print report data** seemed to be required. Akado also wanted to be able to write a command akin to **get data value** in his lists of instructions at those points where the monkey should obtain a new data value directly from Akado.

Akado discussed these difficulties with Kolab and together they worked out several possible solutions. But no matter how diligently Kolab worked with the monkey, Komputema seemed to be at his limit regarding the command repertoire. Akado decided that he could continue using his previous method for inserting data values into the computation: he prepared in advance the data sheet, inserting values for those variables which represented initial data and leaving the values of the other variables blank. Thus he obviated the need for the command **get data value**.

Kolab then concentrated on teaching Komputema the last remaining command, **print report data**, but to no avail. It became completely clear the Komputema had learned about as much as he was capable of learning. Faced with the impossibility of teaching him this last command, Akado realized that he could circumvent this difficulty in the same way as he had solved the **get data value** problem: with prepared data sheets. He would define the entire report to be a family of array variables, with which the monkey already knew how to work. The indices of the array variables would indicate the page and line numbers of the data to be 'printed', which

would be represented as a sequence of characters, a type of variable with which Komputema was familiar:

name	set	value
report(1, 1)	seq	
report(1, 2)	seq	
...		
report(1, 50)	seq	

name	set	value
report(2, 1)	seq	
report(2, 2)	seq	
...		
report(2, 50)	seq	

...

When the report was complete, Akado would simply cut off the two left columns (for the names and sets) and include the remaining third column as the corresponding page in the documentation on his design.

Akado and Kolab tried this approach and found that it worked satisfactorily. The fact that Komputema now had to work with several data sheets instead of one posed no difficulty. Akado had to take care, however, that his instructions never caused the monkey to calculate a value for any of these variables a second time, for in such a case Komputema would scratch through the old value (see the example above). Akado could not, of course, submit a report with such a messy appearance to a client. Akado decided to write instructions for preparing the report in such a way that the data lines would be written sequentially, from line 1 of page 1 through to the end of the report.

Combining all of the above, Akado could write lists of instructions for performing all sorts of structural calculations and for preparing reports of the results of these computations. The first calculation made in this way for an actual building project determined the maximum shear force and the maximum moment in a beam supported at both ends and loaded at one point off center (see Fig. 0.0). The initial data sheets for this calculation were as follows:

name	set	value
s	num	
m	num	
smax	num	
mmax	num	

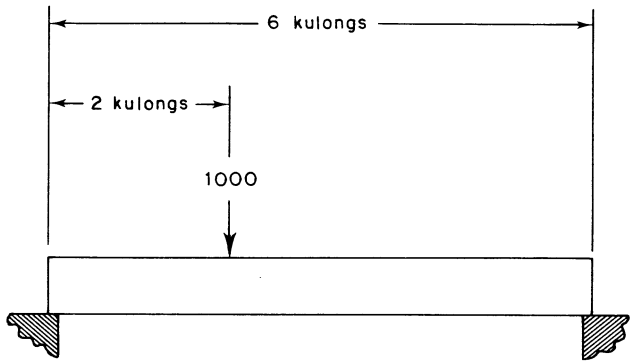


Fig. 0.0 The beam configuration first programmed by Akado

P	num	1000
a	num	2
L	num	6
x	num	
ix	num	0.1

name	set	value
report(1, 1)	seq	
report(1, 2)	seq	
report(1, 3)	seq	
report(1, 4)	seq	
report(1, 5)	seq	
report(1, 6)	seq	
report(1, 7)	seq	
report(1, 8)	seq	

The list of instructions written by Akado was as follows:

```
smax := 0
mmax := 0
x    := 0
while x ≤ L do
    if x ≤ a
    then s := P * (1 − a/L)
         m := P * x * (1 − a/L)
    else s := P * a/L
         m := P * a * (1 − x/L)
```

```
    endif
    if s > smax then smax := s else endif
    if m > mmax then mmax := m else endif
    x := x + ix
endwhile
report(1, 1) := "Stress calculation,"
report(1, 2) := "beam loaded at one point, supported at each end"
report(1, 3) := "Load =":P
report(1, 4) := "at":a:"kulongs from the left end"
report(1, 5) := "Beam length =":L:"kulongs"
report(1, 6) := "maximum shear =":smax
report(1, 7) := "maximum moment =":mmax
report(1, 8) := "End of calculation, this is."
```

Akado suspected that if he were to analyze the problem more thoroughly, he could simplify this list of instructions. But because he intended to use this list as a basis for developing a more extensive one for performing stress calculations for more complex situations, involving more and different types of loads and supports, he decided to leave his algorithm in this presumably more general form.

Akado and Kolab were proud of the results of their efforts. Akado had succeeded in organizing the calculations in a simpler and more systematic form. Consequently, he was able to prepare a set of instructions for performing the calculations so that they could be carried out by someone with no professional knowledge. Kolab had succeeded in training the monkey to execute such a set of instructions. The overall result was that these previously quite problematic calculations could now be performed by less skilled people or, even better, by especially capable, appropriately trained monkeys.

Thus, it seemed as if it might be possible to increase sufficiently the productivity of the Mocsian civil engineers so that they would finally be able to satisfy the greatly increased demand for plans for new buildings. Kolab only needed to find and train enough clever monkeys so that each engineer could have one at his disposal – and Akado, to develop some way to ensure the correctness of the lists of instructions given to them.

Exercises

1 What are the exact contents of the report page prepared by Komputema when executing the above list of instructions?
2 What are the contents of the other data sheet after Komputema has carried out the instructions in the above list in the manner described in this chapter?

Chapter 1

Introduction

We are the hollow men
We are the stuffed men
Leaning together
Headpiece filled with straw. Alas!
Our dried voices, when
We whisper together
Are quiet and meaningless
As wind in dry grass
...
Shape without form, shade without colour,
Paralysed force, gesture without motion;
...

– Thomas Stearns Eliot*

1.0 Software development as an engineering science

Characteristic of every engineering field is the existence of a body of fundamental, theoretical, scientific and mathematical material upon which most of the engineer's work is ultimately based. Such a foundation provides guidelines for the design process and, probably more importantly, enables the designer to verify systematically and precisely important characteristics of his proposed design. Such a basis for the engineer's work is a necessary condition for achieving and maintaining the reliability and correctness of a design to which the engineer, his clients and society have become accustomed.

Closely correlated with the existence of such a theoretical foundation is the fact that the engineer views the object to be designed – e.g. the structure of a building, an electrical circuit, the chassis of an automobile, the frame of an airplane, an engine, etc. – as a mathematical object. He formulates

mathematical theorems about the behavior or certain properties of the object and proves them. As a result of this approach, a building does not typically collapse during or after construction, a ship does not sink when it is launched and an airplane does not crash during its first test flight. These results are in sharp contrast to the typical experience with a new program: as a rule, it does not function properly during its first test and even after it is put into productive use, failures surprise no one. Much time and effort are expended in identifying, locating and correcting the errors in the program. While confidence in its correct functioning increases generally with productive use, one is never really sure that all errors have been found and corrected.

Examples of the theoretical foundations of engineering fields are Maxwell's equations, from which Kirchhoff's, Faraday's and Henry's laws can be derived (electrical engineering) and Newton's laws, from which various principles of statics and dynamics can be derived (civil and mechanical engineering). When designing a circuit or building and especially when verifying the design, the engineer considers his design to be a mathematical object whose component parts obey these abstract fundamental laws and equations. Considerable attention is paid to detail and the smallest basic building block. For example, the civil engineer must calculate the stresses (compression, tension, shear, torque) in each beam in a bridge resulting from the loads applied to it and the forces supporting it. After performing these calculations and verifying that the resulting stresses nowhere exceed the breaking strengths of the materials used, he is confident that the bridge will not collapse. Only when this level of confidence is achieved, based on theoretical considerations, will anyone commit resources to the construction of the bridge.

This situation did not always prevail. Especially in European art galleries with collections of paintings dealing with the industrial revolution can one find considerable evidence of bridges collapsing frequently under the load of the new locomotives. Obviously, the responsible persons were not able to calculate the stresses in the structural elements and verify that they would not exceed the breaking strength of the material used. To cite an earlier example, in 1628 the Vasa, a large, richly appointed and the most modern warship in the Swedish fleet, sank on her maiden voyage shortly after launching. Apparently, the ship's designers were not sufficiently well versed in the theory of hull stability.

Loss of human life and the extensive damage to property resulting from such technical mistakes led to various forms of pressure being brought to bear upon designers of such structures. More exacting standards for their performance were introduced. Only professionally educated designers employing scientific principles systematically in their work were able to satisfy these standards reliably and the particular field became an engineering science.

The transition of electrical technology into electrical engineering, of airplane construction into aeronautical engineering and of other technical fields into their respective engineering sciences followed courses different in detail but similar in general pattern. In each case, it was found that the less scientific, less systematic approaches ('trial and error', 'wing and a prayer') followed by the less qualified practitioners resulted in designs which were inadequate in terms of safety, reliability, efficiency and/or cost. The losses caused by and the effort required to correct the commonly faulty designs were significantly greater than the costs of training new entrants to the field sufficiently well and thoroughly that they could produce correct designs initially. That is, the serious, professional approach was less expensive in the long run.

Engineering practice and the preparatory study for it are also characterized by attention to fundamental detail. The basic, simple elements and components commonly used in the field in question are analyzed in considerable – some might even say excruciating – detail. The civil or mechanical engineer, for example, devotes much time to the study of stresses in a simple, one dimensional beam, loaded and supported in different ways. As one textbook for mechanical engineers expressed it, 'beams are undoubtedly the most important of all structural members, and the basic theory underlying their design must be thoroughly understood' (Meriam, 1951, Part 1, p. 207). Similarly, the student of electrical engineering devotes a considerable part of his time to the study of circuits containing only two elements, a resistor and a capacitor or a resistor and an inductor. After analyzing their transient as well as steady state behavior intensively, he then studies circuits containing only three components – a resistor, a capacitor and an inductor – at even greater length. These circuits are so fundamental, important and common that the electrical engineer who understands their behavior thoroughly already possesses a very substantial part of the knowledge he will need for his future work. Such an engineer usually finds the analysis of a seemingly much more complex circuit containing them as substructures to be a comparatively easy and straightforward task.

Another characteristic of engineering practice is a very conscious attempt to avoid mistakes in the first place (in contrast to correcting errors already made). Considerable attention and effort is devoted to achieving the goal of producing correct designs. In civil engineering, for example, other engineers usually review the designer's calculations, which he has organized in a standard way so that they can be easily understood by others with a similar professional training.

Especially, these two characteristics of engineering study and practice – acquiring and regularly applying a thorough understanding of basic fundamentals and a continual, conscious effort to prevent and avoid errors – result in a reliability and level of public safety well beyond that achieved

in software development practice today. These are perhaps the two prime factors distinguishing a profession from other occupations.

It seems to me that software development today is in a latter phase of its development as a non-engineering field and is beginning to enter a preliminary phase of development as an engineering science – the much acclaimed software engineering. The field of software development has probably advanced about as far as is possible without taking the approach characteristic of other, already established, engineering fields. The complexity arising in software systems seems to be greater than that with which one can successfully and reliably cope without a really thorough understanding of underlying fundamental principles. There are signs that society is not prepared to continue to accept the more or less common attitude that errors in software are in principle unavoidable. Movements in some countries to introduce legal liability in connection with software are an example of signals of such changes. At the same time, developments in computing science promise to provide a basis for achieving the reliability characteristic of the designs produced by engineers in other fields. Such developments strongly suggest, for example, that the belief of many software practitioners cited above is incorrect and that errors in software are, in fact, almost totally avoidable just as comparable errors in designing bridges, for example, can be prevented.

I should emphasize at this point that by 'software engineering' I mean an approach to the preparation for, and practice of, our vocation which engineers in other fields would recognize as exhibiting basic characteristics typical of their fields. Some of the topics subsumed today under the term 'software engineering' do not satisfy this criterion. Systematizing work in a restricted technical sense, using well-honed tools, applying management principles (e.g. relating to project management) to the organization of our work, etc., useful as these may be, do not represent the essence of engineering and are not sufficient to transform our occupation into a professional engineering field. Among other things, practitioners must acquire and apply to their work an extensive knowledge, based on a thorough understanding of fundamental, immutable principles of a mathematical and theoretical nature and of lasting validity, and they must be willing to accept responsibility for the correctness of their designs. Only then will we be able to join the ranks of the engineering professions.

Until relatively recently, the various concepts, techniques, guidelines, rules, etc. used by software developers constituted an amorphous collection of knowledge, experience, legends and superstitions with no really clear structure and, particularly, with no apparent common foundation. This state of affairs contrasts sharply with that prevailing in other professional – and especially engineering – fields. In view of this situation, the increasing use

of the term 'software engineering' in recent years led C. A. R. Hoare to ask, 'What is the great body of professional knowledge?', 'what are the theoretical mathematical ... principles which underlie the daily practice of the programmer?' (Hoare, 1984, p. 8). The physicist would have formulated the question differently: 'If software engineering really exists, then what is its equivalent of Newton's laws?' In the words of the electrical engineer, 'What are the software engineering equivalents of Maxwell's equations and Kirchhoff's laws?'

It is the thesis of this book that the fundamental principles of software engineering in this sense have been discovered and developed during the last one to two decades. They are presented in Chapter 3; the most important of them are summarized in Section 3.9. Most of the rest of the book is devoted to motivating, justifying or applying this theoretical foundation.

The software engineer who understands these principles and is able to apply them extensively and with ease will be able to avoid making the logical errors commonly committed by less qualified programmers. He will be able to verify that his proposed design (program) will function as specified – just as the civil engineer verifies his design for a bridge or a building in order to ascertain, before starting construction, that it will support itself and its planned load.

1.1 Fallacies exposed: often cited objections to an engineering approach to designing software

Whenever it is suggested that software development be practiced as an engineering science, a general reaction of approval usually follows. When specific measures, procedures, approaches and the like are then proposed for software developers to employ in software engineering practice, various objections are often raised. Usually these derive, in the final analysis, from defensive reactions of less qualified software developers who perceive – frequently only subconsciously – the personal threat represented by a transition to a true engineering science. Most such objections are logically equally applicable to well-established practices common in the traditional engineering and other professional fields. The very existence of such fields over an extended period of time, as well as the way in which they are practiced, clearly contradicts the thesis represented by such objections.

In this section, the more commonly raised objections will be presented and discussed.

Objection: I, like every other person, am imperfect. Therefore, my software must be imperfect.

One must ask the person raising this objection whether he is prepared to accept the same statement coming from a surgeon who is about to operate upon him, the structural engineers who designed the buildings in which he lives and works, the pilot who will fly him on his next business trip, the automobile mechanic who recently repaired the brakes in his car, etc. Do not the users of his software have the same right to demand a reasonably high level of quality of his services as he has with respect to the services supplied to him by others?

While most members of society recognize that the people who supply various goods and services are fallible, they are not prepared to accept a comparably high error rate in the goods and services supplied. They simply do not accept as valid the argument that the fallibility of people must imply that the results of people's work must be fraught with errors and failures.

Especially, professionals, but also non-professionals, are expected to take measures to ensure that the results of their efforts are more reliable than those efforts themselves. Both as individuals and as a group, professionals view their limitations and fallibility as a challenge to be overcome, not as a convenient excuse for indolence. The education of professionals aims to impart to the future practitioner the knowledge and ability which will enable him to minimize his errors and, importantly, to verify his results so that he can detect and correct his errors before they have led to negative consequences.

While no client expects suppliers to provide services which are always absolutely perfect, he does expect professionals, at least, to try to do so. The true professional does, in fact, consciously strive very seriously for perfection. One cannot achieve an error rate identical to zero, but one can get arbitrarily close to it.

Objection: The theoretically based approach and methods proposed by computing scientists for designing computer software amount to nothing more than that which good software developers have been doing subconsciously all along.

While there is a significant element of truth in this statement, it is doubtful whether it is completely true. In any event, it would seem to be desirable to systematize and put the approach already taken subconsciously by the best programmers on a firm, explicit foundation. This would enable other, not quite so excellent programmers to learn and apply these methods also and attain the benefits thereof. In addition, it would presumably have the advantage that the best programmers could apply these methods consciously and therefore more extensively and with greater ease and less effort than before, thus increasing also their productivity and accuracy. It would probably

also enable them to build upon this foundation more consequently and thereby to increase their capabilities to even higher levels.

Objection: The proposed new theoretically based methods are applicable only to small, toy programs. Real programs and systems are very much larger, are fundamentally different in nature and do not yield to the application of these techniques.

An approach to designing software based on a firm theoretical foundation is, in fact, relatively new and not at all widespread among practitioners. Before they can apply such a foundation to their work, they must spend a significant amount of time to learn and master this body of knowledge. As in other engineering fields, the student of this material must begin with small problems. Only after understanding the material thoroughly will he be able to apply it to larger, in practice more realistic, problems. Again as in other engineering fields, those who have completed this learning phase typically report that applying this knowledge to larger systems is less complicated than the novice would at first think.

Structuring software systems hierarchically, with each segment being limited in size, results in analyses and correctness proofs which increase in length but *not* in basic logical complexity as the size of the system increases. Not infrequently, in fact, one finds the greatest logical intricacy in the lowest level program segments, i.e. the ones which the new initiate tackles first. It is not valid to estimate the complexity of proving the entire system correct by projecting the complexity encountered initially with 'small' program segments linearly to the size of the entire system.

Again, an analogy with civil engineering is reasonably valid: The designer of a bridge with thousands of girders really must solve only two problems, calculating the stresses in one beam and resolving the forces at one joint. These he must do thousands of times, of course, but no increase in the level of complexity arises. The interaction among several beams (at each joint) must be analyzed, but each joint presents essentially the same problem. Resolving the forces at any one joint is logically no more complex than analyzing the stresses at each point in one beam. At hierarchically higher levels of the analysis, entire sections of hundreds of girders will be considered to be a single structural element with only a few points of interaction with other substructures, so that the mass of internal detail can be suppressed, simplifying the higher level analysis. At each level of analysis only a limited, easily grasped amount of detail is considered.

The student of structural engineering who starts by trying to analyze the complete bridge in its entirety as a single, indivisible unit and without first understanding *thoroughly* the statics of an individual beam and of an

individual joint will soon become very bogged down in confusing detail. He will quickly conclude that the theory is not practically applicable to large structures.

His conclusion is, of course, incorrect. The correct conclusion is that his approach is wrong. One should apply the theory to smaller, isolated subsections of the bridge, each of which is logically comparatively simple. The results of each such analysis should be combined in steps, each of which is also bounded in size and logical complexity. Theoretical considerations and practical experience clearly show that such an approach is possible and leads to practically valuable results. An essential prerequisite is that the engineer must understand the fundamentals – analyzing the individual beam and the individual joint – so thoroughly that performing such an analysis is a simple, almost subconscious reflex action for him.

All of the above comments apply correspondingly to a properly structured program system and its proof of correctness.

Objections: The proposed theoretical foundation is not really important to the practical software developer, as is evidenced by the fact that only a very tiny fraction of experienced software developers use it in their work.

Of course they do not use it in their everyday work. How can they, when they do not understand it, when they have never been shown its usefulness? Relatively few, in fact, are even aware of its existence.

This statement is not really an objection, but rather simply a comment that the material in question is not now in common use among practitioners. The reasons for the lack of such use are a separate issue not really addressed in this 'objection' at all. One possibility in principle, of course, is that it would not be useful to apply it, but this is only one possibility among many.

Another possibility, consistent with the thesis of this book, is that this would-be objection is the chicken and egg dilemma in another form: this material is not applied by practitioners because they have not learned it and they do not learn it because practitioners do not use it. This observation highlights a vicious circle which will certainly pose a major – perhaps the greatest – impediment to the transition of software development to a true engineering science: because software development has not been practiced as an engineering field, an important prerequisite for its becoming an engineering science (a certain body of professional knowledge shared by its practitioners) is not satisfied and as long as this prerequisite is not satisfied, the field cannot become an engineering science.

Objection: Compared to the program itself, a proof of correctness is excessively long and complex: For example, the subprogram for merging in

Section 5.8.0 is only some ten lines long, but its analysis and proof of correctness is some fifteen *pages* long. Similarly, much more time is required to construct the proof of correctness than to write the program.

The statements are correct but the standards of comparison contained therein are inappropriate. The goal is not a program, but a correct program, a program in whose correctness one can have confidence. Other methods for gaining that confidence are less effective and more costly.

With regard to the time required to prove correctness, a more appropriate standard of comparison would be the time required to select an appropriate set of test cases and execute the subprogram upon them. Measured against this standard, the time a professionally educated software engineer needs to construct the proof of correctness is not at all long. Another possible standard would include the time required to locate the errors, correct them and remedy their consequences. This standard certainly tips the scale in favor of the proof of correctness.

The objection to the length of the written proof calls for several comments. Firstly, a computer program is a notoriously terse description of a process and represents, therefore, a rather extreme standard of comparison. More appropriate would be the length of the complete documentation of the subprogram. Measured against this standard, the analysis and proof of correctness does not appear unduly lengthy, especially if one includes a complete listing of the test cases in the documentation.

Secondly, established engineering fields offer comparable examples. A textbook for mechanical engineering students presented an introduction to ideal (weightless) beams with concentrated loads in ten pages. This length is misleadingly short, however, because it contains a number of exercises which induce the student to develop much missing detail himself. Additional material (again, with a number of problems for the student to solve) covering a few more general situations increased the length of the introductory material on the single beam to 27 pages (Meriam, 1951, Part I, Chapter VI).

In a major textbook used by many electrical engineering students, a passage 21 pages long was devoted to an analysis of a circuit containing only two components (a resistor and an inductor). A subsequent passage ten pages long analyzed only some aspects of a circuit containing three components (a resistor, an inductor and a capacitor). This was followed by additional material approximately ten pages long on circuits with one, two or three components (Guillemin, 1953, pp. 222–62).

Compared with such material, a fifteen page analysis and proof of correctness of a subprogram containing six assignment statements, an **if** construct and a **while** loop with associated conditional expressions appears relatively compact.

Objection: The proposed theoretically based approach to software development places considerable emphasis on mathematics and logic. While this is all very nice, it is not really relevant to the practicing software developer. First and foremost, his goal is to write usable programs, not to analyze them in extreme detail and to understand every subtlety, real or imagined, in them. He needs more a handbook of software development tools and techniques with instructions on how to use them most effectively.

Such a 'Software Designer's Handbook', while useful, can never be successfully employed as a substitute for a thorough understanding of the task at hand. It is most effectively used by someone who already understands what he is doing.

A person with the attitude implied in the objection – that the designer needs only to follow a recipe, not necessarily to understand what he is doing – can never become a professional, a software engineer. He can at best become a second rate coding technician who must turn to a professional for guidance, especially when the going becomes tough. This has been found to be true for other professions, including established engineering fields, and there is no reason to believe that the situation is fundamentally different in the case of software development.

An analogy: A youngster learned a great deal from *The Radio Amateur's Handbook*, enough to become an amateur radio operator and to design and build transmitters and receivers. They worked, but not so well and reliably that they could be operated and maintained easily by others. He realized that their behavior exhibited certain idiosyncracies which he did not adequately understand and could not completely eliminate. Only after studying electrical engineering, during which time he learned the fundamental principles of electricity and physics and the mathematics necessary to work with and apply them – subjects not even mentioned in *The Radio Amateur's Handbook* – did he have the knowledge and level of competence necessary to design such equipment of commercial quality.

A set of finely honed tools can be very valuable in the hands of someone who knows how to use them and who understands what he is trying to accomplish. In the hands of someone without such understanding, they can be dangerous in the sense that he can get into more trouble faster by using them. He will often not know what problems could arise and how to avoid them. And once in real difficulty, he will typically not be able to get out of it.

Objection: The theory relates to idealized statements and commands in artificial, non-existent programming languages and is hence irrelevant to the practitioner. Only if the theory were to deal with real programming languages in their entirety and in full detail might it be of some limited use.

Most programming languages in current use were designed ('thrown together' is a more accurate description in some cases) before the theory of proving programs correct was developed. It should, therefore, be no surprise that currently used programming languages do not represent a good match with the theory of proving programs correct.

The definitions of the several statements presented in Chapter 2 of this book represent generalizations of corresponding statements in most common programming languages. These definitions provide a widely usable base for applying this theory.

Actually, most common programming languages contain unnecessary features, extensions and also restrictions. Input/output commands constitute a typical example. Sections 4.0, 4.1 and 4.2 contain a discussion of the most commonly encountered logically unnecessary constructs and how they can be reduced to the simpler fundamental statements defined in Chapter 2. Also, certain operations are implied in other structural constructs in many 'real' programming languages (the release statement is a common example).

A program designed in terms of only the fundamental statements defined in Chapter 2 is logically simpler, more readable and more easily analyzed than one expressed in a typical contemporary programming language. The software engineer who must analyze an existing program not designed in the way proposed here will often find it most convenient to translate – at least conceptually – the statements in the given program into the fundamental statements defined in Chapter 2.

Again, established engineering fields offer relevant analogies. The electrical engineer, for example, designs and analyzes his circuits in terms of ideal resistors, inductors and capacitors. He does not, at that stage, think in terms of manufacturer X's wire wound resistors with considerable distributed parasitic inductance, manufacturer Y's capacitors with leakage (parasitic resistance) or manufacturer Z's inductors with both parasitic capacitance and resistance, etc. To do so would complicate the design process unduly and would lead to an unnecessarily restrictive, specific design. Comparable examples can be found in other engineering fields.

Software design is no different in principle. The statements defined in Chapter 2 are the basic building blocks of any procedure oriented language and represent general concepts of more lasting validity than the corresponding statements in implemented programming languages of today, which usually impose rather specific restrictions of various sorts. To deal with the statements in such a language in full detail and in its entirety – as demanded in the objection – would distract the designer's attention from the real

functions of the statements in question and would lead him to concentrate instead on the idiosyncracies of the language. He would no longer see the forest for the trees and would, as a result, tend to lose sight of the logical processes being performed in his program. He would become bogged down in detail and would find it correspondingly more difficult to understand his design – and hence to prove it correct.

Another benefit of viewing programming languages in terms of their common features (e.g. as outlined in Chapter 2) is that it facilitates learning new languages quickly. The software engineer who takes this approach will also find that he will be able to use a newly learned language more effectively because he understands and concentrates his attention on the processes and other aspects of fundamental concern. The programmer who, on the other hand, concentrates his attention on the idiosyncracies and unique features of each language he learns will find that knowledge of little value when learning a new language. In fact, he will frequently find that the new, different detail he must memorize does not fit into his mental models very well, causing him to become more and more confused by the growing mass of detail he must remember.

Objection: Many software developers and students of programming are not familiar with the mathematics required to understand and apply the proposed theory and approach to designing software. They find it bewilderingly complicated and overwhelming and will, therefore, avoid it. Experience shows that by emphasizing principles of structured programming, for example, one can teach them enough so that they can become reasonably satisfactory programmers without having to learn so much mathematics and mathematical notation.

Is a programmer who cannot determine whether or not his program is correct a 'reasonably satisfactory programmer'? My answer is that he is not a reasonably satisfactory software designer. He might be a reasonably satisfactory coding technician working under the guidance of a professional software designer.

Many people can 'design' a bridge in the sense that they can sketch a configuration of girders which spans the desired distance. But only if one is able to calculate the stresses in each girder in order to verify that the material's breaking strength is not exceeded anywhere will he be able to obtain a building permit. The 'designers' of the bridges which collapsed in the last century and of the ships which sank because their hulls and weight distributions were unstable (see Section 1.0 above) were eliminated from the practice of the profession because they were incapable of ensuring an accepted level of quality of their designs.

The theory and approach to designing software to which the objection refers is the only way now known to verify analytically that a program will satisfy its specifications under all circumstances or to determine under what circumstances it will behave as specified. As is the case in other engineering fields, a certain knowledge of mathematics and the ability to use it extensively and with facility is a critical prerequisite for understanding and applying this theory. The student of software engineering who does not already have such a knowledge of mathematics or who cannot yet apply it easily must acquire such knowledge and ability.

Any theory of programming which enables one to verify analytically and exactly the behavior of a program must utilize a language suitable for expressing intricate logical arguments precisely. Ideally, such a language should also prevent one from formulating imprecise and vague statements. Mathematics was developed to fulfill these goals; extensive experience accumulated over a long period of time indicates that it satisfies these requirements well. While another language could presumably be invented for expressing a theory of programming, there seems to be little reason to do so. Furthermore, there is no reason to expect that it would be significantly simpler.

The language of mathematics has proved its usefulness in many other technically oriented areas and has assumed a permanent place of importance in every corresponding scientific and engineering field. It is already clear that the language of mathematics is similarly applicable to software development. There is considerable evidence strongly suggesting that it will assume a correspondingly important role in software engineering. If this is true, then those programmers who refuse to learn the language of mathematics will be doomed to ultimate professional extinction.

No one with the knowledge of and attitude toward mathematics expressed in the objection above could become an engineer in any other field. There is no good reason to believe that he could become a successful software engineer either. At best, he might become a coding technician, an assistant to a software engineer. But not every professional software engineer would care to hire him as such an assistant.

One can imagine that the above objection was raised many decades ago when the theory of complex variables was added to the required mathematical repertoire of the electrical engineer. This area was undoubtedly perceived by some as unnecessarily complex and irrelevant as being of no physical significance (in the real world, voltages and currents are, in the mathematical sense, real quantities, not imaginary or complex). This area of mathematics proved to be of value in understanding the physical phenomena in question. It also turned out to simplify considerably the analysis of circuits and the corresponding calculations. Introducing complex

variables necessitated significant changes in the curriculum and working electrical engineers, especially, must have found it difficult and time consuming to learn this material while simultaneously practicing their profession. But now every electrical engineer has learned this material and it is taken for granted as obligatory prerequisite knowledge.

Earlier, the introduction of algebra and differential and integral calculus into civil engineering must have met similar resistance. But this material became recognized as essential and today's structural engineer cannot imagine how one could calculate the bending of a beam under a load without applying calculus.

Objection: Software designers and programmers are already so concerned with the various restrictions imposed by the language, operating system, hardware, etc. which they are using that they cannot be burdened with still another restraint: writing code which can be proved correct.

This would-be objection is a thinly disguised pauper's oath. The software developers to whom this statement refers are, in all likelihood, so buried in detail and restrictions that they are unable to pay attention to other issues because they have not adequately mastered the language, operating system, hardware, etc. in question. In short, they are underqualified for the task they are attempting to accomplish.

The programmer who thoroughly understands the fundamental nature and structure of programming languages, computing systems, etc. will find it much easier to review and remember the idiosyncracies of any particular one than will the programmer who has no generally valid mental skeletal structure upon which to hang such details which must be mastered. The latter must memorize – in contrast to learn – these details. The resulting burden can easily become too great.

An engineer or other professional is expected to be so familiar with the fundamentals and commonly used techniques, procedures, tools, etc. of his field that they are second nature to him. He must be able to apply them almost as a reflex action. The above objection is, in effect, an admission that many software developers have not attained this level of competence in their chosen field. This is, in turn, an admission that they are not professionals.

1.2 Intended readership

This book is written for the serious student of software engineering. The term 'student' is used here in its less restrictive sense: he may already be employed as a software developer (designer or programmer) or he may be enrolled as a student in a tertiary educational institution.

The term 'engineering' is used here in its traditional sense, i.e. the application of mathematical and theoretical fundamentals and principles to the problem of designing and constructing economical and reliable systems which fulfill some real need. The term 'engineering' is not used here in the sense in which it has sometimes been used recently in the software context, specifically, in the more restricted senses of software management, software engineering management or even the mere use of some 'integrated' software development package.

This book is aimed primarily at those who design and develop software for practical use or who are preparing themselves to engage in such work. While I hope that this book will be of interest to many computing scientists and researchers also, it is not written specifically to satisfy their needs.

1.3 Prerequisites for the study of this book

It is assumed that the reader of this book is familiar with at least one programming language. He should have some experience in using it in practice, but need not have long, extensive experience in programming. In fact, the reader with extensive programming experience may be at a disadvantage, for he will probably have acquired more bad habits and false preconceived notions which he will have to unlearn. Knowledge of several programming languages will be helpful but is not necessary.

The reader should also be generally familiar with mathematics, especially with the mathematical way of thinking. He must be able to follow and understand detailed logical proofs. He must know what constitutes a mathematical proof and be able to recognize gaps and flaws in at least the simpler, more straightforward proposed proofs. Ideally, he should be able to formulate theorems on familiar topics and to construct proofs for them. If he does not already have this ability, he should concentrate on developing it while studying this book.

Relatively little specific mathematical knowledge is assumed. The reader should be familiar with basic definitions in the areas of sets, sequences, functions and Boolean algebra. He need not have advanced or particularly detailed knowledge of these areas. He should be able to manipulate expressions, especially Boolean expressions. The reader with gaps in his knowledge of these areas should study Appendix 0, Mathematical fundamentals, before proceeding with the material presented elsewhere in this book.

The reader who has successfully completed one or two years of moderately challenging courses in mathematics in secondary school will normally find

his mathematical background to be sufficient for understanding the material in this book. It is necessary, of course, that he still remember and be able to apply that knowledge with reasonable facility.

In order to make the material in this book accessible to readers with as wide a mathematical background as possible, only a relatively restricted set of mathematical symbols is used in this book. This notation is more readable than that which is more typical of, for example, the predicate calculus, especially for those with no previous exposure to that area of mathematics. After gaining facility in working with logical expressions, the reader may, however, wish to employ the more concise and mathematically traditional notation in his own work (e.g. $\wedge$ instead of **and**, $\vee$ instead of **or**, $\forall$ instead of 'for all' etc.).

1.4 Goals and contents of this book

A major goal of this book is to advance the practical application of the theory of proving computer programs correct by making that theory accessible to a larger fraction of present and future software developers. After studying the text and examples and working the exercises, the reader will, in addition to understanding the theoretical foundation, be able to apply the principles presented here to his own design and programming tasks.

The goal of this book goes deeper than merely proving programs correct, however. It also attempts to give the reader the same type of foundation for designing software that engineers in other fields already have for designing their mechanisms, systems, etc. It tries to give a more fundamental understanding of the nature of software, of the issues arising in its design, of verifying a proposed design, etc.

In effect, this book attempts to bridge the gap between theory and practice in the field of software development. It does not purport to advance significantly the theory itself. Nor does it provide a collection of omnipotent methods, tools and techniques which the reader may blindly apply to tasks he does not completely understand. In my opinion, such methods, tools, etc. are not the solution to our problems in software development. Knowledge and understanding are, and their prerequisite is a certain investment of time and effort in the study of the foundations of the field. There is no short cut.

The approach taken here is to present the practically most important parts of the theory of proving programs correct and its application to typical design problems in a language appropriate for the software engineer, in contrast to the computing scientist or mathematician. The theoretical background is presented in somewhat simpler (but still rigorous) mathematical terms than most other books on this subject employ. This development of the material leads up to presentations and discussions of several design problems, thereby highlighting and emphasizing the application to practical problems.

The primary scope of this book is limited to the case of the sequentially executed program. The most important reason for restricting attention to this case is that the development of the corresponding theory is fundamentally complete, while the corresponding theory for non-sequential processes is still under development. In any event, the correctness of a program segment when executed sequentially is a necessary condition for its correctness when executed concurrently with other processes. One can expect, therefore, that the theory presented here will be included in a more extensive theory of correctness for processes executing in parallel. Section 4.3 discusses concurrent and comparable processes and suggests some possibilities for extending the material contained in the rest of this book to such situations, but does not purport to treat that subject exhaustively.

The theory of proving computer software correct is based on the notion that programs and parts thereof can be viewed as mathematical objects about which theorems can be formulated and proved. This theory has been under development since approximately the late 1960s. The theory is well – but not widely – known. Software developers in commercial practice especially tend to have limited or no knowledge of it. This is unfortunate because one of the major sources of problems in practical software development – errors – can be largely eliminated by the application of this theory.

The major part of this book is divided into two parts: Theory (Chapters 2, 3 and 4) and Practice (Chapters 5 and 6). In the theoretical part, the state of execution of a program – its data environment – as well as the effects of executing the fundamental programming statements are defined. From these definitions various theorems and lemmata are derived which form the basis for practical applications. It is also argued that only the fundamental statements and constructs defined earlier are needed to express programs and algorithms arising in practice. The practical chapters apply the theory developed earlier to the design and analysis of programs.

Additional parts of this book present introductory material (the Prologue, Chapters 0 and 1) and draw conclusions regarding the future of the field of software development (the Epilogue, Chapter 7). The following paragraphs briefly describe the individual chapters and appendices.

Chapter 0, Komputema Simio, the computing monkey of Moc, gives a metaphorical example of the execution of a computer program. Several of the most basic ideas defined or developed later are introduced here: a variable (a line on the monkey's data sheet), a data environment (the data

sheets), a program statement (instruction) as a rule for transforming one data environment into another, the definitions of the effects of executing each type of program statement, a program or a segment thereof (a list of instructions), etc. If one were to make a copy of the data sheets after the monkey executes each instruction, the resulting temporally ordered sequence of data sheets would be the computational history (sequence of data environments) as defined in Section 2.2.

Chapter 1, Introduction, emphasizes the parallel between software development and established engineering fields, proposing that the design and development of correct software is not only theoretically but also practically possible. It argues that the various reasons commonly given for not employing a truly engineering approach are, in the final analysis and long term, invalid. It specifies the book's target audience, the prerequisite knowledge the reader is assumed to have and the goals of the book. Finally, it outlines the contents of the book.

Chapter 2, Basic semantics of computer programs and programming constructs, presents a number of definitions which provide the basis for the development of much of the material in the rest of the book. The data environment of a program and the effect upon such a data environment of executing the several types of program statements are the subjects of these definitions.

In Chapter 3, Proof rules for the individual programming constructs, a number of theoretical conclusions following from the definitions of Chapter 2 are stated and proved. These results are organized and presented in the form of proof rules applicable to the several fundamental types of program statements defined in Chapter 2. While this chapter is more theoretical than practical in nature, it builds an important basis for the application of the theory to the practical problems considered particularly in Chapters 5 and 6.

In Chapter 4, Transfundamental programming constructs, several more complex programming constructs commonly occurring in contemporary programming languages are defined in terms of the fundamental constructs introduced and treated earlier. The main point of the generally informal material in Sections 4.0 through 4.2 is that most of the more complicated types of statements in higher level programming languages can be replaced by much simpler constructs. Such substitution not only simplifies the mathematical semantics with which we must work, but – more importantly – it simplifies our programs' structures, readability and proofs of correctness. Finally, some aspects of the subject of concurrency are discussed in Section 4.3 (see the paragraph above on parallel processes).

Chapter 5, The analysis and verification of programs: methods and examples, illustrates the application of the previously presented material to

individual program statements, to isolated constructs and to program segments combining the several types of fundamental constructs. It demonstrates how to analyze an existing program or a proposed program with the goal of verifying its correctness.

Chapter 6, The construction of correct programs, the culmination of this book's presentation, begins with conclusions, drawn largely from the material in Chapter 5, which are of relevance to the design problem. These conclusions are in the form of guidelines for the designer. The rest of Chapter 6 consists of sample design problems in which the reader is led through the design process; inferences of general relevance are drawn and discussed as they arise.

Chapter 7, The practice of software engineering tomorrow, previews the software developer's future. The nature of professional software development, especially with regard to the ways in which it will differ from programming yesterday and today, is discussed. Implications for the individual software engineer, his preparation, etc. are developed. Reviewing the parallel between software development and established engineering fields drawn in Chapter 1, Chapter 7 closes with the prediction that software development will, in the foreseeable future, become an engineering science in the true sense of the term.

Appendix 0, Mathematical fundamentals, reviews all specific mathematical topics needed to understand the rest of this book. It is not a complete textbook on these subjects. It is intended to (a) refresh the reader's memory in certain key areas and (b) permit him to extend his knowledge to a limited extent into areas which are new to him. The main topics covered are sets, sequences, functions and Boolean algebra.

Appendix 1, Solutions to the exercises, contains solutions or sketches of solutions to the exercises appearing in the various sections of the book.

The Bibliography lists books, papers, etc. referenced in this book, used in its preparation or suggested as additional reading.

Theory

Chapter 2

Basic semantics of computer programs and programming constructs

Here and elsewhere we shall not obtain the best insight into things until we actually see them growing from the beginning.

– Aristotle

The very true beginning of wisdom is the desire of discipline.

– The Apocrypha, Wisdom of Solomon, 6:17

Begin at the beginning ... and go on till you come to the end: then stop.

– Lewis Carroll

Section 2.0 below introduces the notion of a data environment of a program, defining it and its component part, the program variable. The evaluation of variables and expressions (such as propositions, conditions, assertions, etc.) within the context of a particular data environment is discussed.

The rest of this chapter defines the effect of executing a program on a data environment, viewing a program (and each part thereof) as a mathematical function. Two complementary views are presented: (a) each program statement transforms one data environment into another data environment (Section 2.1) and (b) each program statement transforms a sequence of data environments into another by appending one or more newly constructed data environments to the original sequence (Section 2.2). The precise effects of the execution of each primary programming construct (declaration, release, assignment, **if** statement, **while** loop, etc.) are defined.

2.0 The data environment of a program

2.0.0 Program variables

The essence of a program variable, or simply variable, is that it associates a name, a set and a value (an element of that set) with each other. Thus,

a variable corresponds directly to the three entries on one line of the monkey's working paper (see Chapter 0). These considerations suggest the following formal definition of a variable.

Definition 2.0: A *variable* is a triple (sequence of three members) consisting of a name, a set and an element of that set. The particular element of the set is called the *value* of the variable in question.

In most programming systems, the name of a variable is a string (sequence) of characters. Often, the first character in the name must be a letter, while succeeding characters may be selected from a larger (but still precisely restricted) set of symbols. In most systems the length of a name is limited. Such restrictions vary considerably from one system to another. Because the specific syntax of a name is of no consequence for our purposes, we will not give a more detailed definition of the term 'name' in this book.

Example 1: $(x, \mathbb{Q}, 3.23)$ is a variable name x which takes on values which are rational numbers. The (current) value of x is 3.23.

Example 2: (s, strings of finite length, X7a) is a variable named s which takes on values which are strings of finite length. The current value is X7a.

The characters of which the strings are composed must be precisely defined.

An array variable can be defined in at least two ways. Perhaps the simpler is to consider it as a family of similarly named but otherwise independent variables. For example, the integer array z, with one subscript ranging in value from 0 to 3, would be the collection of variables

$$\{(z(0), \mathbb{Z}, 6), (z(1), \mathbb{Z}, 3), (z(2), \mathbb{Z}, 8), (z(3), \mathbb{Z}, -2)\}$$

This model of an array permits, in principle, the various elements of the array to be associated with different sets, i.e. to be of different 'types'. While most implementations of programming languages require all elements of an array to be of the same type, there is no fundamental, immutable reason for this restriction.

A somewhat different model of an array views it as a single variable with values in the set which is the cartesian product of the sets from which the values of the individual elements are selected. For example, using this model, the array z in the above example would be viewed as

$$(z, \mathbb{Z}^4, (6,3,8,-2))$$

The structure of this model of an array variable must be extended to include

the parameters specifying the number and the range(s) of the subscript(s), for example:

$$(z, \mathbb{Z}^4, 1, (0, 3), (6,3,8,-2))$$

Because the power 4 is determined by the number and the range(s) of the subscript(s) $(4 = 3 - 0 + 1)$, it is redundant and can be eliminated:

$$(z, \mathbb{Z}, 1, (0, 3), (6,3,8,-2))$$

This model is highly suggestive of the schemes typically employed for implementing array variables in programming systems today.

Because of its structural simplicity and greater generality, the first model of the array introduced above will normally be used in this book. In most instances, however, the two models will be equally applicable.

2.0.1 Data environments

A data environment is a collection of variables (see Section 2.0.0 above). In general, the collection need not have any particular structure, but for reasons which will be discussed later, in Section 2.0.2, it is convenient to require a data environment to be a sequence of variables. This has the effect of imposing a linearly ordered structure on the variables constituting a data environment. More formally:

Definition 2.1: A *data environment* is a sequence of variables.

Thus, a data environment corresponds directly to the contents of the monkey's working papers described in Chapter 0.

Example 3: $[(x, \mathbb{Q}, 3.23), (s,$ strings of finite length, X7a$), (y, \mathbb{Z}, -8)]$ is a data environment consisting of the three variables x, s and y with values 3.23, 'X7a' and -8 respectively.

Example 4: $[(x, \mathbb{Q}, 3.23), (x, \mathbb{Z}, 4), (y, \mathbb{Z}, -8), (x, \mathbb{Q}, 9.2), (x, \mathbb{Q}, 3.23)]$ is a data environment containing five variables, four of which have the same name, x. Two of these variables named x have the same value.

Combining and summarizing the above definitions, a data environment d is a sequence of variables, each of which is a triple consisting of a name N_i, a set S_i and a value V_i:

$$d = [(N_1, S_1, V_1), (N_2, S_2, V_2), \ldots, (N_i, S_i, V_i), \ldots]$$

where

V_i in S_i for every i.

In this book we will refer frequently to the set of all possible data environments, for which we will use the symbol $\mathbb{D}$. An element of this set, i.e. an individual data environment, will usually be denoted by d, $d0$, $d1$, etc. Note that a data environment may be empty, that is, contain no variable.

To be rigorous one must ensure that the set of all possible data environments mentioned above actually exists, i.e. that it is or can be defined precisely. In particular, one must ensure that the classical paradoxes in connection with defining a set are avoided and that the definition is logically consistent (see Appendix 0, Section A0.0.0, Basic definitions). This can be done conveniently by specifying explicitly which sets may be associated with a variable and requiring that each be well defined without reference (explicit or implicit) to the concept of a data environment. In practice, this rarely if ever poses any difficulty since typical implemented programming systems provide for only a few well-defined sets such as the following (or finite subsets thereof): $\mathbb{R}$ (the real numbers), $\mathbb{Z}$ (the integers), the set of strings (sequences) of characters selected from a finite set of symbols and certain sets defined by enumerating their elements. Throughout this book, we will assume that $\mathbb{D}$ is defined or definable for the target programming system, e.g. in the way outlined above.

2.0.2 The value of a variable in a data environment

One of the most common functions performed during the execution of a computer program determines the value of a variable, given its name, within the context of a particular data environment. Generally, that value is defined to be the value associated with the variable having the given name in the data environment in question.

The evaluation of a variable can be thought of as a function which maps a variable name and a data environment into a value:

value = valvar(name, data environment)

The value is an element of the set associated with the named variable in the data environment in question.

A given data environment can be thought of as defining a particular mapping from variable names to variable values:

value(name) = valvar(name, data environment)

The result of evaluating a variable name which is not present in the data environment in question is undefined, i.e. the domain of the function 'value' above is the set of all variable names appearing in the data environment in its definition.

Example 5: If $d = [(x, \mathbb{Q}, 3.23), (y, \mathbb{Z}, -8), (z, \text{string}, \text{X7a})]$, then

the value of x in d = valvar(x, d) = 3.23 (a rational number)
the value of y in d = valvar(y, d) = −8 (an integer)
the value of z in d = valvar(z, d) = X7a (a string)
the value of t in d, valvar(t, d), is undefined

Ambiguity can arise when the data environment in question contains more than one variable with the name for which a value is to be determined. In implemented computing systems this problem is avoided or resolved in various different ways. In some systems, each name must be unique, so the problem does not arise. In others, the scope of each variable's definition is restricted in certain ways, usually related to the program's hierarchical structure. In systems of the latter type, a variable may be defined (declared) at certain points in the program only. Furthermore, in such systems the definitional scope of the variable is often implicitly and automatically terminated (released, deallocated), leaving the programmer with no choice.

A general and convenient way to resolve this ambiguity is to require that variables with the same name be linearly ordered in a data environment; by convention, the first such variable will be used in evaluating the variable name. If all variables in the data environment are linearly ordered, i.e. if the data environment is defined to be a sequence of variables, this effect will be achieved in a particularly simple way. For this reason we defined a data environment to be a sequence of variables (see Section 2.0.1 above).

Example 6: If $d = [(y, \mathbb{Z}, -8), (x, \mathbb{Q}, 9.2), (x, \mathbb{Z}, 11)]$, then

the value of x in d = valvar(x, d) = 9.2 (a rational number)

These considerations lead us to formally define the value of a named variable in the context of a given data environment as follows.

Definition 2.3: The value of a variable named x in the context of the data environment d, written valvar(x, d), is defined to be

valvar(x, d) = V_j

where

$j = \min\{i \mid N_i = \text{“}x\text{”}\}$

The quotation marks surrounding x in the above formula signify that the name x itself is meant, not the value of the variable named x. N and V refer to the names and values respectively of the variables in the data environment d:

$$d = [(N_1, S_1, V_1), (N_2, S_2, V_2), \ldots]$$

Clearly, this definition presupposes that a variable with the name in question (x) exists in the data environment d, i.e. that there exists an i such that the data environment d (a sequence of variables) contains an ith member and N_i = "x". If this condition is not met, then the value of the variable x in d is not defined.

2.0.3 Evaluation of an expression within the context of a data environment

In mathematics in general, an expression is evaluated by substituting values for the names of the variables occurring in the expression and then performing the operations specified (see Appendix 0, Section A0.1.2, Expressions). In the case of an expression to be evaluated within the context of a data environment of a program, values are determined for the individual variables as described in Section 2.0.2 above.

The evaluation of an expression can be thought of as a function which maps the expression and a data environment into a value:

value = valexp(expression, data environment)

A given expression can be thought of as a function which maps a data environment into a value:

expression(data environment) = valexp(expression, data environment)

In general, the value of an expression is defined in this way only if the value of every variable appearing in the expression is defined in the data environment, i.e. only if every variable name appearing in the expression also occurs in the data environment in question.

Furthermore, the value of the expression is generally defined only if the values of the operands and results of all operations specified in the expression are in the defined ranges, e.g. if no 'overflow' or comparable 'run time error' arises. The specific restrictions depend upon the precise definitions of the operations involved and of the sets of which the operands and results are elements. These definitions and the restrictions and limitations resulting therefrom vary from system to system and cannot, therefore, be considered in detail here. Restrictions of this sort are, however, very important to the

programmer. He is well advised to pay close attention to them at all times, for the notation used in typical programming languages often subtly traps him into assuming (implicitly) that the implemented operations are well behaved, e.g. that the ranges of their intermediate and even final results are unbounded, that addition and multiplication as implemented are associative and obey the distributive laws, etc.

Even when the value of an expression is defined, the value determined during the execution of a program might not be the same as that expected at first by the casual programmer. Expressions are evaluated by performing the operations as they are implemented in the computing system, which is not necessarily the same as they are defined in pure mathematics. Phenomena such as rounding, approximating real numbers by elements of a discrete set, etc. must be carefully considered. Examples of apparent anomalies in evaluating expressions are given in Baber (1982, test questions 6.1 and 6.2, pp. 81, 176–7). When manipulating expressions in proofs of correctness, the software engineer must constantly remember that each operation (e.g. +, −, *, /, etc.) is to be interpreted as it is defined in the target computing system, which is not necessarily the same as it is usually defined mathematically.

An expression may evaluate to an element in the set {true, false}, in which case one speaks of a proposition, assertion, condition, relational expression, Boolean expression, etc. The values of other expressions may be real or rational numbers, integers, strings or elements of other sets.

Example 7: If $d = [(y, \mathbb{Z}, -8), (x, \mathbb{Q}, 9.2), (x, \mathbb{Z}, -11)]$, then

the value of expression $(y < x)$ in d = valexp$(y < x, d)$ = true
the value of expression $(x \leq 0)$ in d = valexp$(x \leq 0, d)$ = false
the value of expression $(2 * x + y)$ in d = valexp$(2 * x + y, d)$ = 10.4
the value of expression $(y + z)$ in d, valexp$(y + z, d)$, is undefined.

Exercise

1 Specify the domain of an expression in general but precise terms.

2.0.4 The value of an array variable in a data environment

In order to determine the value of an array variable, the subscript expression(s) are first evaluated (see Section 2.0.3 above). The resulting values are then substituted for the subscript expressions to give the actual ('reduced' or 'evaluated') variable name. Finally, the value associated with this latter variable name is determined as described in Section 2.0.2 above.

In evaluating a subscript expression, another array variable may be encountered, and, during the evaluation of its subscript expressions, still another array variable may be encountered, etc. Thus, the evaluation of a subscript expression involves, in general, a recursive process. This process will terminate if the depth of nesting in the expression is limited. If the length of the program's text is finite, such recursion arising from references to array variables within subscript expressions will necessarily be of limited depth.

The above comments on evaluating an array variable in the context of a given data environment suggest the following more formal definition.

Definition 2.4: The value of an array variable $x(s)$ in the context of the data environment d, written valvar($x(s)$, d), is defined to be

$$\text{valvar}(x(s), d) = \text{valvar}(x(\text{valexp}(s, d)), d)$$

or, interpreting the expression s as a function (see Section 2.0.3 above),

$$\text{valvar}(x(s), d) = \text{valvar}(x(\text{valexp}(s, d)), d) = \text{valvar}(x(s(d)), d)$$

If the array x has more than one subscript, i.e. if s is an n-tuple, then the value of the n-tuple of expressions is defined to be the n-tuple of the values of the individual expressions. That is, if $s = (s1, s2, \ldots)$, then

$$\begin{aligned}\text{valexp}(s, d) &= \text{valexp}((s1, s2, \ldots), d)\\ &= (\text{valexp}(s1, d), \text{valexp}(s2, d), \ldots)\end{aligned}$$

Lemma 2.0: $\text{valvar}(x(s), d) = V_j$,
where

$$j = \min\{i \mid N_i = \text{“}x(\text{valexp}(s, d))\text{”}\}$$

and N and V refer to the names and values respectively of the variables in the data environment d:

$$d = [(N_1, S_1, V_1), (N_2, S_2, V_2), \ldots]$$

Proof (sketch): This result follows directly from the definition immediately above and the definition of the value of a variable given in Section 2.0.2 ■

As discussed in Section 2.0.2 above for the case of a name of a simple (non-array) variable or a name of an array variable with constant subscripts, this definition presupposes that all variable names which appear in the various expressions encountered and which are to be evaluated in the data environment d are contained therein. Otherwise, the value of the array variable $x(s)$ in d is not defined.

2.0.5 Boolean expressions and sets of data environments

An expression E with values in {true, false} determines in a straightforward way a subset E^* of $\mathbb{D}$, i.e. a set of data environments:

$$E^* = \{d \text{ in } \mathbb{D} \mid E(d) = \text{true}\}$$

The set E^* includes every data environment d for which $E(d)$ is defined and is true. Also, $E(d)$ is defined and true for every data environment d in E^*. It is meaningful, therefore, to interpret a Boolean expression either as a function or as a set of data environments. Where the interpretation is clear from the context, no notational distinction will be made.

Conversely, a condition (Boolean function) can be defined in terms of a set:

$$\begin{aligned}C(d) &= \text{true, if } d \text{ is in } C^*\\ &= \text{false, otherwise}\end{aligned}$$

for all d in $\mathbb{D}$. Thus each of the two mathematical structures – the Boolean function and the set – captures the essence of the other, provided that one does not need to distinguish between a value of false and an undefined value of the Boolean function.

Example 8: The expression $E = (x < y)$ defines the set of data environments

$$E = \{d \text{ in } \mathbb{D} \mid \text{valvar}(x, d) < \text{valvar}(y, d)\}$$

The set E will include only data environments containing both variables x and y.

Definition 2.5: If the sets $E1^*$ and $E2^*$ are defined by the Boolean expressions (conditions) $E1$ and $E2$ respectively and $E1^*$ is a superset of $E2^*$, then it is meaningful to speak of the condition $E1$ as *less restrictive* or *weaker* than the condition $E2$. Conversely, $E2$ is said to be *more restrictive* or *stronger* than $E1$.

Loosely speaking, the 'smaller' set is the 'stronger' condition, and the 'larger' set is the 'weaker' condition.

Lemma 2.1: The following statements are then equivalent:

1 $E1^*$ is a superset of $E2^*$. ($E2^*$ is a subset of $E1^*$.)
2 $E1$ is weaker than $E2$.
3 $E2$ implies $E1$, i.e. $\text{valexp}(E2, d) \Rightarrow \text{valexp}(E1, d)$, for all d such that these values are defined.

Proof: The proof is left as an exercise for the reader. ■

Notice that the subjects of statement (1) above are the *sets* $E1^*$ and $E2^*$. The subjects of statements (2) and (3) are the *conditions* (Boolean expressions, propositions) $E1$ and $E2$.

Exercise

2 Prove lemma 2.1.

2.1 Programming constructs as functions on $\mathbb{D}$, the set of data environments

In the following sections, each of the several types of program statements and fundamental constructs (compound statements) will be defined as a function which maps a data environment into a data environment:

$$S\colon \mathbb{D} \rightarrow \mathbb{D}$$

The result of applying a statement S to the argument $d0$ (the result of 'executing' statement S upon the data environment $d0$) is a data environment $d1$ which is defined in terms of certain characteristics of $d0$ in a manner specific to each particular type of program statement or construct.

2.1.0 The assignment statement

An assignment statement consists of the name of a variable (either a simple or subscripted name), the assignment symbol (:=) and an expression involving any number of variable names. Consider, for example, the assignment statement

$$x := E(x, y, \ldots)$$

which we will call A in the paragraphs below.

The result of applying A to the argument $d0$ (the result of 'executing' statement A upon the data environment $d0$) is defined to be a data environment $d1$ which is equal to $d0$ except for the value of the first variable named x. The value of this variable in $d1$ is the value of the expression $E(x, y, \ldots)$ in $d0$. This operational definition motivates the following, mathematically more formal definition.

Definition 2.6: The effect of applying the assignment statement A above to the data environment

$$d0 = [(N0_1, S0_1, V0_1), (N0_2, S0_2, V0_2), \ldots, (N0_i, S0_i, V0_i), \ldots]$$

is

$$A(d0) = (x := E(x, y, \ldots))\,(d0) = d1$$

where

$$\begin{aligned} d1 &= [(N1_1, S1_1, V1_1), (N1_2, S1_2, V1_2), \ldots, (N1_i, S1_i, V1_i), \ldots] \\ N1_i &= N0_i, \text{ for all } i \\ S1_i &= S0_i, \text{ for all } i \\ V1_i &= \mathrm{V0}_i, \text{ for all } i \neq j \\ V1_j &= \mathrm{valexp}(E(x, y, \ldots), d0) \\ j &= \min\{k \mid N1_k = \text{“}x\text{”}\} \end{aligned}$$

provided that $V1_j$ is in $S1_j$. (Otherwise, the triple $(N1_j, S1_j, V1_j)$ would not satisfy the definition of a variable and, in turn, $d1$ would not be a data environment.)

If an array variable is referenced to the left of the := symbol, the effective variable name is determined by replacing the subscript expression(s) by its (their) values in $d0$, i.e.

$$x(s) := E(x, y, \ldots)$$

where s is a subscript expression, is to be interpreted as

$$x(\mathrm{valexp}(s, d0)) := E(x, y, \ldots)$$

$A(d0)$ is defined only when the variable to which a value is being assigned is contained in $d0$ (and therefore also in $d1$) and when the value of the expression $E(x, y, \ldots)$ in $d0$ is defined and is an element of the set associated with the variable receiving the new value. See the comments in Sections 2.0.2, 2.0.3 and 2.0.4 above regarding the conditions under which the values of named variables and expressions are defined.

It is useful to generalize the assignment statement as follows to allow several variables to be assigned various values simultaneously.

Definition 2.7: The *multiple assignment* statement has the form

$$(x1, x2, \ldots) := (E1(x1, x2, \ldots, y, \ldots), E2(x1, x2, \ldots, y, \ldots), \ldots)$$

When the multiple assignment statement above is applied to the initial data environment $d0$, each expression $E1$, $E2$, etc. is evaluated in the same, initial data environment $d0$. The value of each expression ($E1$, $E2$, etc.) becomes the value of the corresponding variable ($x1$, $x2$, etc.) in the resulting data environment $d1$ as in the case of the simple assignment statement.

If a variable name appears more than once as a member of the replacement list to the left of the := symbol and if the corresponding expressions yield different values, then the effect of executing the multiple assignment statement is ambiguous and hence undefined. Assigning values to array variables with equal subscripts can give rise to such a situation, e.g.

$$(x(i), x(j)) := \ldots$$

where i and j have the same value in the data environment in question.

Definition 2.8: The *exchange* statement

$$x :=: y$$

is a special form of the multiple assignment statement, being defined to mean

$$(x, y) := (y, x)$$

Note that the above definitions require that the expressions E, $E1$, $E2$, etc. are to be evaluated in the initial data environment $d0$. Furthermore, only the values of variables named to the left of the assignment symbol (:=) are changed. Some real systems exist which do not always satisfy these requirements. During the evaluation of one part of an expression, a new intermediate data environment may be generated which is subsequently used in the evaluation of other parts of the expression. The values of functions referenced in an expression may be calculated by procedures which alter the values of other variables. Such 'side effects', as they are often called, can and do give rise to confusion, usually make a program less readable and complicate proving its correctness. Generally, a programmer using such a system should avoid writing statements which could give rise to such side effects.

Exercise

3 Specify precisely the domain of a simple assignment statement.
4 Specify precisely the domain of a multiple assignment statement.

2.1.1 The *if* statement

Definition 2.9 The **if** statement is a compound statement of the form

if B **then** $S1$ **else** $S2$ **endif**

where B is a condition (a Boolean expression) and $S1$ and $S2$ are program statements, e.g. assignment statements, nested **if** statements, etc. The result

of executing this **if** statement, called S below, upon the data environment $d0$ is

$$S(d0) = (\textbf{if } B \textbf{ then } S1 \textbf{ else } S2 \textbf{ endif})\ (d0) = d1$$

where

$$d1 = S1(d0), \text{ if } B(d0) = \text{true}$$
$$= S2(d0), \text{ if } B(d0) = \text{false}$$

$S(d0)$ is defined whenever $B(d0)$ is defined (and true or false) and either $S1(d0)$ or $S2(d0)$ as required above is defined. Otherwise, $S(d0)$, and hence the effect of executing the **if** statement, is undefined.

Either the **then** or the **else** part may be empty, that is to say, either $S1$ or $S2$ may be a null statement, in which case $S1(d0) = d0$ or $S2(d0) = d0$ respectively.

Exercise

5 Specify precisely the domain of an **if** statement.

2.1.2 A sequence of program statements

Consider the sequence S of statements

$$S1, S2$$

Operationally, this sequence is interpreted to mean that first, $S1$ is executed upon the original data environment $d0$. Then, $S2$ is executed upon the result, i.e. upon $S1(d0)$. The motivates the following formal definition.

Definition 2.10: The effect of executing the sequence S of statements ($S1$, $S2$) upon the data environment $d0$ is

$$S(d0) = (S1, S2)(d0) = S2(S1(d0))$$

That is, the function S, describing the effect of the sequence of the two statements $S1$ and $S2$ considered as a single unit or compound statement, is the composition of the individual functions $S1$ and $S2$. Composition of two functions is associative, therefore the operation of 'executing' program statements in sequence is also associative:

$$((S1, S2), S3) = (S1, (S2, S3))$$

and one can write without introducing semantic ambiguity simply

$$S1, S2, S3$$

instead of either ((*S*1, *S*2), *S*3) or (*S*1, (*S*2, *S*3)).

Neither composition of functions nor sequencing program statements is, in the general case, commutative.

Exercise

6 Specify precisely the domain of a sequence of statements.

2.1.3 The *while* loop

The **while** loop has the following form

while *B* **do** *S* **endwhile**

where *B* is a conditional expression and *S* is any program statement (simple or compound). The effect of executing a **while** loop upon a data environment can be defined in several different but equivalent ways.

Definition 2.11: The result of applying the **while** loop above to the data environment $d0$ is defined recursively, i.e. in terms of the **while** loop itself, as follows:

(**while** *B* **do** *S* **endwhile**) ($d0$)
$= (S,$ **while** *B* **do** *S* **endwhile**$)$ ($d0$), if $B(d0) =$ true
$= d0,$ if $B(d0) =$ false

If $B(d0)$ is neither true nor false, e.g. if it is undefined, then the effect of the **while** loop is undefined.

The recursive iteration implied by this definition terminates if and only if the condition *B* is false after applying *S* to $d0$ some number of times (possibly 0 times, i.e. if *B* is false initially). More formally, this can be stated as follows: The recursive iteration implied above terminates if and only if there exists a non-negative integer n such that

$$B(S^n(d0)) = \text{false}$$

If the condition *B* is always true, the effect of the **while** loop is undefined. Operationally, one says then that the loop 'does not terminate' or is an 'infinite loop'.

Definition 2.12: Applying the definition of the sequence of program statements given in Section 2.1.2 above, definition 2.11 can be rewritten in the following slightly different form:

(**while** *B* **do** *S* **endwhile**) ($d0$)
$=$ (**while** *B* **do** *S* **endwhile**) ($S(d0)$), if $B(d0) =$ true
$= d0,$ if $B(d0) =$ false

The comments regarding values of the condition *B* and termination of the iterative steps in definition 2.11 above apply here as well.

Definition 2.13: The construct

while *B* **do** *S* **endwhile**

is equivalent to the construct

if *B* **then** (*S*, **while** *B* **do** *S* **endwhile**) **endif**

Note that here the **else** part of the **if** statement is empty, i.e. consists of the null statement (see Section 2.1.7).

The comments regarding termination of the iterative steps in definition 2.11 above apply here as well.

Definition 2.14: The data environment resulting from executing the compound statement **while** *B* **do** *S* **endwhile** upon the data environment $d0$ is

(**while** *B* **do** *S* **endwhile**) ($d0$) $= S^n(d0)$

where

$$n = \min\{i \mid i \text{ is a non-negative integer and } B(S^i(d0)) = \text{false}\}$$

provided that for all integers j in the interval $0 \leq j < n$

$$B(S^j(d0)) = \text{true}$$

If no such n exists, i.e. if there is no non-negative integer n such that $B(S^n(d0)) =$ false, or if some previous $B(S^j(d0))$ does not evaluate to true or false, then the effect of the **while** loop is undefined (cf. the condition for the termination for the recursive iteration in definition 2.11 above).

Exercise

7 Specify precisely the domain of a **while** statement.
8 Show that the four definitions above are equivalent, that is, that any one of them implies the other three.

2.1.4 The declaration statement

The simplest form of the declaration statement is

declare $(x, S, E(x, y, \ldots))$

where x is a variable name; S, a set and E, an expression with values in S. Executing this statement, called D below, upon a data environment $d0$ has the effect of prefixing a new variable named x to the sequence of variables constituting the data environment $d0$ to form the new environment $d1$.

Definition 2.15: The effect of executing the declaration statement D above upon the data environment

$$d0 = [(N0_1, S0_1, V0_1), (N0_2, S0_2, V0_2), \ldots]$$

is

$$D(d0) = (\textbf{declare}(x, S, E))(d0) = d1$$

where

$$d1 = [(x, S, \text{valexp}(E, d0)), (N0_1, S0_1, V0_1), (N0_2, S0_2, V0_2), \ldots]$$

provided that valexp$(E, d0)$ is in the set S.

Lemma 2.2: Given the data environment

$$d0 = [(N0_1, S0_1, V0_1), (N0_2, S0_2, V0_2), \ldots, (N0_i, S0_i, V0_i), \ldots]$$

and the declaration statement D above, then

$$d1 = D(d0) = [(N1_1, S1_1, V1_1), (N1_2, S1_2, V1_2), \ldots, (N1_i, S1_i, V1_i), \ldots]$$

where

$N1_1$ = "x" (i.e. the name x, not the value of x)
$S1_1 = S$
$V1_1 = \text{valexp}(E, d0)$

and for all $i > 1$,

$N1_i = N0_{i-1}$
$S1_i = S0_{i-1}$
$V1_i = V0_{i-1}$

Proof (sketch): This lemma follows directly from the definition above; it amounts to a restatement of that definition in different notation. ■

The above defines a data environment $d1$ whenever the value of the expression E in $d0$ is defined and that value is an element of the set S. Otherwise, the effect of the declaration statement is undefined.

Definition 2.16: In order to simplify our notation, we define the operation of *concatenation* of sequences, using the symbol &. By x & y we mean the single sequence formed by appending the members of y to the members of x without changing their order.

For example, if

$$x = [a, b, c]$$

and

$$y = [k, l, m]$$

then

$$x \,\&\, y = [a, b, c, k, l, m]$$

Lemma 2.3: The effect of the statement

declare (x, S, E)

referred to as D, is

$$D(d0) = (\textbf{declare}\ (x, S, E))(d0) = [(x, S, \text{valexp}(E, d0))] \,\&\, d0$$

Proof (sketch): This lemma is simply another notational form of the above definition of the effect of the declaration statement. ■

Definition 2.17: More generally, the declaration statement has the form

declare $(x1, S1, E1), (x2, S2, E2), \ldots$

The result of executing this *multiple declaration* statement D upon the data environment $d0$ is defined to be

$$D(d0) = (\textbf{declare}\ (x1, S1, E1), (x2, S2, E2), \ldots)(d0)$$
$$= [(x1, S1, \text{valexp}(E1, d0)), (x2, S2, \text{valexp}(E2, d0)), \ldots] \,\&\, d0$$

If any of the variable names above ($x1$, $x2$, etc.) refer to array variables, the effective name is determined by replacing the subscript expression(s) by its (their) values in the data environment $d0$; e.g. a reference to $x(s)$ is interpreted as a reference to $x(\text{valexp}(s, d0))$.

Notice that all expressions $E1$, $E2$, etc. are evaluated in the same, initial data environment $d0$. Therefore, the multiple declaration statement is not, in general, equivalent to a sequence of several declaration statements, each referring to only one new variable.

Exercise

9 Specify precisely the domain of a declaration statement.

2.1.5 The release statement

The release statement has the form

release x

where x is a name of a variable. Executing this statement, called R below, upon a data environment $d0$ has the effect of removing the variable named x from the sequence of variables constituting the data environment $d0$ to form the new data environment $d1$. If $d0$ contains more than one variable named x, the first such variable is removed.

Definition 2.18: The effect of executing the release statement R above upon the data environment

$$d0 = [(N0_1, S0_1, V0_1), (N0_2, S0_2, V0_2), \ldots, (N0_i, S0_i, V0_i), \ldots]$$

is

$$R(d0) = (\textbf{release } x)(d0) = d1$$

where

$$\begin{aligned} d1 &= [(N1_1, S1_1, V1_1), (N1_2, S1_2, V1_2), \ldots, (N1_i, S1_i, V1_i), \ldots] \\ N1_i &= N0_i, \text{ for all } i < j \\ &= N0_{i+1}, \text{ for all } i \geq j \\ S1_i &= S0_i, \text{ for all } i < j \\ &= S0_{i+1}, \text{ for all } i \geq j \\ V1_i &= V0_i, \text{ for all } i < j \\ &= V0_{i+1}, \text{ for all } i \geq j \\ j &= \min\{k \mid N0_k = \text{“x”}\} \end{aligned}$$

If the release statement references an array variable, the effective name is determined by replacing the subscript expression(s) by its (their) values in the data environment $d0$; e.g. a reference to $x(s)$ is interpreted as a reference to $x(\text{valexp}(s, d0))$.

This defines a data environment $d1$ whenever the data environment $d0$ contains the variable to be released. Otherwise, the effect of the release statement is undefined. Alternatively, one could define the release statement to be equivalent to the null statement when the variable to be released is not contained in the data environment, in which case $(\textbf{release } x)(d0) = d0$. See Section 2.1.7 for a discussion of the null statement.

Variables of the same name existing simultaneously in a data environment form, in effect, a 'stack', or last-in first-out store or list. The declaration and release statements are used to manipulate this stack by adding a new variable or deleting the most recently added still present variable with the stated name. Among those variables present in a data environment which bear a common name, only the first one (i.e. the one created by the most recently executed declaration which is still effective) is accessible by references in expressions, etc. A newly declared variable conceals all other variables of the same name in the data environment in question. The values of concealed variables may be rendered accessible again only by removing the concealing variables by executing one or more release statements.

Note that

$$\text{valvar}(x, (\textbf{release } x)''(d))$$

is the value of the nth concealed variable with the name x in the data environment d. We will make use of this fact when we wish to refer to concealed variables in lemmata, etc.

Exercise

10 Specify precisely the domain of a release statement.

2.1.6 The procedure call without parameters

Often, it is convenient to label a single or compound statement and to refer to (invoke, activate) that statement from one or more points in a program. A statement so labeled and referenced is variously called a procedure, program, subprogram, routine, subroutine, etc.

Definition 2.19: If a procedure named P consists of the statement S, then the result of executing the statement

call P

upon a data environment $d0$ is the same as the result of executing S upon $d0$, i.e.

(**call** P)($d0$) = $S(d0)$

We will usually specify that a procedure named P is to consist of the statement S by writing

procedure P: S **endprocedure**

In many programming languages provision is made for 'passing' values, variables, expressions, etc. ('parameters') between the calling environment and the called procedure. The semantics of calling procedures with such parameters is the subject of Section 4.0, Procedure calls with parameters.

2.1.7 The null statement

Definition 2.20: The null statement is an empty statement and maps a data environment into itself:

(null statement)($d0$) = $d0$

The null statement is an identity function. It occurs, for example, in the **if** statement when nothing is to be done in either the **then** or the **else** part.

2.1.8 Other loop constructs

Many programming languages offer other loop constructs, either instead of or in addition to the **while** loop defined in Section 2.1.3. These other loop constructs can be defined in terms of the **while** loop. Some of the more frequently encountered types of loops are defined below. Because the definitions of most of these constructs vary from one implementation to another, the following definitions should be interpreted as examples or typical definitions, not standardized ones.

The **repeat** loop

repeat S **until** C

where S is a statement and C is a conditional expression, is defined to be equivalent to the sequence of statements

S
while not C **do** S **endwhile**

The **for** loop

for i := $E1$ **to** $E2$ **step** $E3$ S **next** i

where i is a variable name, $E1$, $E2$ and $E3$ are expressions and S is a

statement, is defined in different ways in different implementations. One definition is that the above **for** statement is equivalent to the sequence

i := $E1$
while ($E3 \geq 0$ **and** $E1 \leq i$ **and** $i \leq E2$)
  or ($E3 < 0$ **and** $E2 \leq i$ **and** $i \leq E1$) **do**
S
i := i + $E3$
endwhile

Because of the variations from one implementation to another, considerable care is advised in using the **for** construct in any but the most straightforward ways. Some implementations evaluate $E1$, $E2$ and $E3$ once only, at the beginning of the loop, and save their values in internal variables during the execution of the loop. If the statement S in the above **while** loop would, for example, alter the values of variables referenced in these expressions, this model would not apply.

The following loop construct contains an internal exit and is sometimes a convenient structure for practical work:

loop
$S1$
if C **then exit**
$S2$
endloop

This loop construct is usually defined to be equivalent to

$S1$
while not C **do**
$S2$
$S1$
endwhile

Still other loop constructs can be found in various programming languages. Most are quite similar to one or more of those presented above. All can be expressed in terms of the **while** loop.

2.2 Programming constructs as functions on $\mathbb{D}^*$, the set of sequences of data environments

In Section 2.1, each program statement was considered to be a function which mapped a data environment into another data environment. Each compound statement was also considered to be a single function, the value

of which was a (single) data environment. This view of the effect of executing a program captures adequately the final result (if any) of executing that program.

In analyzing the behavior of programs, subprograms, etc. it is often useful to consider not only the final result, but also the 'computational history' or temporal development of its execution. In order to capture this history, it is useful to view the result of the execution of a program as a sequence of data environments, not just a single data environment. The number of data environments in the sequence resulting from applying the program's function to the initial data environment may be bounded or countably infinite. In the first case, the program is said to terminate. If, on the other hand, there is no end to the sequence of data environments generated (i.e. the result is an infinite sequence), one says that the program does not terminate but continues indefinitely in what is often called an 'infinite loop'.

We define the set $\mathbb{D}^*$ to be the set of all sequences of data environments (elements of $\mathbb{D}$). $\mathbb{D}^*$ contains every sequence of specific (finite) length as well as every unending sequence (every countably infinite sequence) of data environments. More formally:

Definition 2.21: The set $\mathbb{D}^*$ is defined to be

$$\mathbb{D}^* = \mathbb{D}^*_{\text{inf}} \cup_{n \text{ in } \mathbb{N}_0} \mathbb{D}^*_n$$

where $\mathbb{D}^*_{\text{inf}}$ is the set of unending ('infinite' or 'infinitely long') sequences of data environments:

$$\mathbb{D}^*_{\text{inf}} = \{[d_1, d_2, \ldots] \mid d_1 \text{ in } \mathbb{D}, d_2 \text{ in } \mathbb{D}, \ldots\}$$

and $\mathbb{D}^*_n$ is the set of sequences of exactly n data environments:

$$\mathbb{D}^*_n = \{[d_1, d_2, \ldots d_n] \mid d_1 \text{ in } \mathbb{D}, d_2 \text{ in } \mathbb{D}, \ldots, d_n \text{ in } \mathbb{D}\}$$

Above, n is a non-negative integer and $\mathbb{N}_0$ is the set of non-negative integers.

While the above definition of $\mathbb{D}^*$ includes the empty sequence, only non-empty ones will be of practical interest to us.

Functions corresponding to program statements which map a sequence of data environments to a sequence of data environments will be denoted by an asterisk (*) following the symbol referring to the statement. That is,

$$S\colon \mathbb{D} \to \mathbb{D}$$

and

$$S^*\colon \mathbb{D}^* \to \mathbb{D}^*$$

An element of $\mathbb{D}^*$, i.e. a sequence of data environments, will usually be denoted by d^*, $d0^*$, $d1^*$, etc.

We will frequently need to refer to the last data environment in a computational history (sequence of data environments). This can be facilitated by introducing a function which 'extracts' the last member from a sequence.

Definition 2.22: The function *last* maps a finite, non-empty sequence to the last member of that sequence.

For example, if

$$x = [a, b, c]$$

then

$$\text{last}(x) = c$$

In the following sections, a function S^* will be defined for each of the several types of program statements and constructs. In each case, S^* will have the effect of appending a sequence of data environments to its argument $d0^*$ to form its result $d1^*$. The sequence appended to $d0^*$ will depend only upon the last data environment in $d0^*$. This corresponds to the following operational idea. The next simple program statement is executed upon the last data environment in the already generated computational history. The resulting new data environment is appended to that computational history. Compound statements (e.g. **if** and **while** statements) are first decomposed into their individual constituent simple statements before applying them in the manner described above, so that the execution of each individual statement contributes a data environment to the computational history.

2.2.0 The assignment statement

The effect of executing an assignment statement A upon a sequence $d0^*$ of data environments is to append $A(\text{last}(d0^*))$ to $d0^*$, forming a new sequence $d1^*$ of data environments. That is, the assignment statement is applied to the last data environment in $d0^*$ as described in Section 2.1.0 above. The resulting data environment is appended to $d0^*$ to form $d1^*$. More formally:

Definition 2.23: The result of applying an assignment statement A to a computational history $d0^*$ is

$$A^*(d0^*) = d1^*$$

where

$$d1^* = d0^* \ \& \ [A(d0)]$$

and

$$d0 = \text{last}(d0^*)$$

Combining the several expressions above into one formula, we obtain the following more compact form of the definition:

$$A^*(d0^*) = d0^* \ \& \ [A(\text{last}(d0^*))]$$

Exercise

11 Specify precisely the domain of an assignment statement A^* on $\mathbb{D}^*$.

2.2.1 The *if* statement

Definition 2.24: The effect of executing the **if** statement

if B **then** $S1$ **else** $S2$ **endif**

called S in the following paragraphs, upon the sequence $d0^*$ of data environments is

$$\begin{aligned} S^*(d0^*) &= ((\textbf{if } B \textbf{ then } S1 \textbf{ else } S2 \textbf{ endif})^*)(d0^*) = d1^* \\ &= S1^*(d0^*), \text{ if } B(\text{last}(d0^*)) = \text{true} \\ &= S2^*(d0^*), \text{ if } B(\text{last}(d0^*)) = \text{false} \end{aligned}$$

$S^*(d0^*)$ is defined whenever $B(\text{last}(d0^*))$ is defined (and true or false) and either $S1^*(d0^*)$ or $S2^*(d0^*)$ as required above is defined. Otherwise, $S^*(d0^*)$, and hence the effect of executing the **if** statement, is undefined.

Either the **then** or the **else** part may be empty, that is to say, either $S1$ or $S2$ may be a null statement (see Section 2.2.7).

Compare this definition of the function of the **if** statement on $\mathbb{D}^*$ with the definition given in Section 2.1.1 of its function on $\mathbb{D}$.

Note that unlike the effect of the assignment statement described in Secton 2.2.0 above, the **if** statement does not necessarily append exactly one data environment (e.g. $[S1(\text{last}(d0^*))]$ or $[S2(\text{last}(d0^*))]$) to $d0^*$ to form $d1^*$. $S1$ or $S2$ or both may be compound statements, in which case several data environments will typically be appended to $d0^*$ to form $d1^*$.

Exercise

12 Specify precisely the domain of an **if** statement on $\mathbb{D}^*$.

2.2.2 A sequence of program statements

The operational meaning of the sequence S of statements

$S1, S2$

given in Section 2.1.2 suggests the following definition of the function S^*.

Definition 2.25: The effect of applying the sequence S of statements ($S1$, $S2$) to the computational history $d0^*$ is

$$\begin{aligned} S^*(d0^*) &= ((S1, S2)^*)(d0^*) \\ &= S2^*(S1^*(d0^*)), \text{ if } S1^*(d0^*) \text{ is of finite length} \\ &= S1^*(d0^*), \qquad \text{otherwise} \end{aligned}$$

e.g. if $S1^*(d0^*)$ is an unending sequence.

That is, the function S^* is the composition of the functions $S1^*$ and $S2^*$. The comments in Section 2.1.2 regarding associativity and commutativity apply here as well.

Exercise

13 Specify precisely the domain of a sequence of statements on $\mathbb{D}^*$.

2.2.3 The *while* loop

The definition given in Section 2.1.3 for the function of the **while** loop on $\mathbb{D}$ generalizes to a definition of its function on $\mathbb{D}^*$ with one important difference. The **while** function on $\mathbb{D}$ was not defined if the condition B always remained true. The **while** function on $\mathbb{D}^*$, on the other hand, is defined in this case (assuming that the other conditions are met, e.g. that $S(d)$ is always defined). When the condition B always remains true, the value of the **while** function on $\mathbb{D}^*$ is an infinite sequence.

The **while** function on $\mathbb{D}^*$ can be defined in several equivalent ways, corresponding to the definitions given in Section 2.1.3. Recalling the general form of the **while** loop

while B **do** S **endwhile**

we state the following four equivalent definitions.

Definition 2.26: The result of applying the **while** loop above to the sequence $d0^*$ of data environments is defined recursively as follows:

$$\begin{aligned} &((\textbf{while } B \textbf{ do } S \textbf{ endwhile})^*)(d0^*) = d1^* \\ &\quad = ((S, \textbf{while } B \textbf{ do } S \textbf{ endwhile})^*)(d0^*), \quad \text{if } B(\text{last}(d0^*)) = \text{true} \\ &\quad = ((\text{null statement})^*)\ (d0^*), \qquad\qquad\ \text{if } B(\text{last}(d0^*)) = \text{false} \end{aligned}$$

If $B(\text{last}(d0^*))$ is neither true nor false, e.g. if it is undefined, then the effect of the **while** loop is undefined.

When $B(\text{last}(d0^*))$ is false, the entire **while** construct is equivalent to the null statement. Note that this equivalence is indicated explicitly in the above definition. If the effect of the null statement were defined to be the identity function on $\mathbb{D}^*$, it would not be necessary to include the explicit reference to the null statement in the above definition (cf. Section 2.1.3). However, as discussed later, in Section 2.2.7, it is usually more appropriate to define the effect of the null statement upon a computational history differently, necessitating in turn the above form of the definition of the **while** construct as a function on $\mathbb{D}^*$.

Lemma 2.4: The result $d1^*$ will be an unending (infinite) sequence if and only if either of the following situations arises:

1 One of the applications of S leads to an infinite sequence, i.e. S is applied to a d^* such that $S^*(d^*)$ is an unending sequence. The definition of the sequence of program statements given in Section 2.2.2 leads to the termination of the iteration implied by the above recursive definition. The remainder of the sequence $((\textbf{while } B \textbf{ do } S \textbf{ endwhile})^*)(d0^*)$ is the unending sequence $S^*(d^*)$.
2 Each application of S to a d^* leads to a finite sequence $S^*(d^*)$, but condition B is always true. The loop fails to terminate, but the result $d1^*$ is precisely defined and is an unending sequence.

Proof: The proof of this lemma is left as an exercise for the reader. ■

Both of the above two situations are probably seen more clearly in the formulation of the following equivalent definitions.

Definition 2.27: Applying the definition of the sequence of program statements given in Section 2.2.2 above, definition 2.26 can be rewritten in the following form:

$$\begin{aligned}&((\textbf{while } B \textbf{ do } S \textbf{ endwhile})^*)(d0^*) = d1^*\\ &\quad = ((\textbf{while } B \textbf{ do } S \textbf{ endwhile})^*)(S^*(d0^*)),\\ &\qquad\qquad \text{if } S^*(d0^*) \text{ is a finite sequence and } B(\text{last}(d0^*)) = \text{true}\\ &\quad = S^*(d0^*), \text{ if } S^*(d0^*) \text{ is an infinite sequence and } B(\text{last}(d0^*)) = \text{true}\\ &\quad = ((\text{null statement})^*)(d0^*),\\ &\qquad\qquad \text{if } B(\text{last}(d0^*)) = \text{false}\end{aligned}$$

The comments following definition 2.26 above apply here as well.

Definition 2.28: Definition 2.13 of Section 2.1.3 applied to the compound statement itself, not to the corresponding function on $\mathbb{D}$. Therefore, the formulation of definition 2.13 remains unaltered here. The construct

while B **do** S **endwhile**

is equivalent to the construct

if B **then** (S, **while** B **do endwhile**) **endif**

Applying this definition repeatedly and simplifying in the case that condition B always remains true yields the unending sequence of program statements

$S, S, S, \ldots$

as the equivalent construct. See especially part 2 of lemma 2.4.

Definition 2.29: The computational history resulting from executing the compound statement **while** B **do** S **endwhile** upon the computational history $d0^*$ is

$$((\textbf{while } B \textbf{ do } S \textbf{ endwhile})^*)(d0^*) = ((S^n, \text{null statement})^*)(d0^*)$$

where

$$n = \min\{i \mid i \text{ is a non-negative integer } \textbf{and } [(S^*)^i(d0^*) \text{ is an infinite sequence } \textbf{or } B(\text{last}((S^*)^i(d0^*))) = \text{false}]\}$$

provided that for all integers j in the interval $0 \leq j < n$

$$B(\text{last}((S^*)^j(d0^*))) = \text{true}$$

If no such n exists but all $(S^*)^i(d0^*)$ are defined (and of finite length) and all $B(\text{last}((S^*)^i(d0^*)))$ are defined (and true), then the effect of the **while** loop is defined to be

$$((S, S, \ldots)^*)(d0^*)$$

an unending sequence of data environments. Part 2 of lemma 2.4 applies to this case.

If no such n exists because some $(S^*)^i(d0^*)$ is undefined or some $B(\text{last}((S^*)^i(d0^*)))$ is neither true nor false (e.g. undefined), then the effect of the **while** loop is undefined.

Above, $(S^*)^0(d0^*)$ is to be interpreted by convention as $d0^*$ for any S and any $d0^*$.

Exercise

14 Specify precisely the domain of a **while** statement on $\mathbb{D}^*$.

15 Show that the four definitions above are equivalent, that is, that any one of them implies the other three.
16 Prove lemma 2.4.
17 Prove or disprove the following conjecture: For any statement or construct S and for any finite, non-empty computational history $d0^*$,

$$S^*(d0^*) = d0^* \ \& \ [S(\text{last}(d0^*))]$$

2.2.4 The declaration statement

The effect of executing a declaration statement D upon a sequence $d0^*$ of data environments is comparable to that of the assignment statement. The data environment $D(\text{last}(d0^*))$ is appended to $d0^*$, forming a new sequence $d1^*$ of data environments. That is, the declaration statement is applied to the last data environment in $d0^*$ as described in Section 2.1.4. The resulting data environment is appended to $d0^*$ to form $d1^*$. More formally:

Definition 2.30: The result of applying a declaration statement D to a computational history $d0^*$ is

$$D^*(d0^*) = d1^*$$

where

$$d1^* = d0^* \ \& \ [D(d0)]$$

and

$$d0 = \text{last}(d0^*)$$

Combining the several expressions above into one formula, we obtain the following more compact form of the definition:

$$D^*(\text{d0}^*) = \text{d0}^* \ \& \ [\text{D}(\text{last}(d0^*))]$$

Exercise

18 Specify precisely the domain of a declaration statement on $\mathbb{D}^*$.

2.2.5 The release statement

The release function on $\mathbb{D}^*$ is defined in terms of the release function on $\mathbb{D}$ (see Section 2.1.5) in the same manner as was used in the cases of the assignment and declaration statements above.

Definition 2.31: The result of applying a release statement R to a computational history $d0^*$ is

$$R^*(d0^*) = d0^* \ \& \ [R(\text{last}(d0^*))]$$

Exercise

19 Specify precisely the domain of a release statement on $\mathbb{D}^*$.

2.2.6 The procedure call without parameters

Definition 2.32: If a procedure named P consists of the statement S, then the result of executing the statement

call P

upon a computational history $d0^*$ is the same as the result of executing S upon $d0^*$, i.e.

$$((\textbf{call } P)^*)(d0^*) = S^*(d0^*)$$

See Section 2.1.6 for a brief discussion of procedures, procedure calls, etc.

2.2.7 The null statement

As in the case of the null function on $\mathbb{D}$ (see Section 2.1.7), the null function on $\mathbb{D}^*$ can be defined as the identity function.

Definition 2.33: The effect of the null statement applied to a sequence $d0^*$ of data environments is $d0^*$:

$$((\text{null statement})^*)(d0^*) = d0^*$$

Definition 2.34 (alternative): For the purposes of many analyses it is more appropriate to define the null statement's function on $\mathbb{D}^*$ in such a way that it duplicates the last data environment in its argument:

$$((\text{null statement})^*)(d0^*) = \text{d0}^* \ \& \ [\text{last}(d0^*)]$$

The alternative definition above follows the pattern of all other simple statements S (i.e. the assignment, declaration and release statements):

$$S^*(d0^*) = d0^* \ \& \ [S(\text{last}(d0^*))]$$

and ensures that the execution of each statement increases the length of the computational history.

Some degenerate statements (e.g. $x := x$) are null statements according to definition 2.34, but not definition 2.33.

Where not explicitly stated otherwise, definition 2.34 applies.

2.2.8 Other loop constructs

The functions on $\mathbb{D}^*$ corresponding to the several loop constructs defined in Section 2.1.8 can be determined by applying to the equivalent structures given there the relevant parts of Sections 2.2.0 through 2.2.7.

Chapter 3

Proof rules for the individual programming constructs

Logic can be patient for it is eternal.

– Oliver Heaviside

For many parts of nature can neither be invented with sufficient subtilty, nor demonstrated with sufficient perspicuity, nor accommodated unto use with sufficient dexterity, without the aid and intervening of the mathematics; of which sort are perspective, music, astronomy, cosmography, architecture, enginery, and divers others.

– Francis Bacon

Science is built up with facts, as a house is with stones. But a collection of facts is no more a science than a heap of stones is a house.

– Jules Henri Poincaré

It is the spine that holds together the whole length of an animal and preserves its straightness.

–Aristotle

When verifying a part of a program being designed or of an already existing program, some specification exists for the subprogram in question, defining the tasks it is to perform. This specification should be a precise statement, covering all contingencies of consequence. Properly, the specification is a theorem or a collection of theorems, each of which is normally a conditional expression (proposition), whose value in the terminal data environment is to be true, or some other proposition about the terminal data environment. In some cases, the theorem may be a proposition about the computational history of the program's execution (i.e. about the sequence of data environments generated by the program) rather than about the terminal data environment alone.

Perhaps the generally most useful proof strategy is to start with the proposition of the theorem to be proved and, working backward through the part of the program of interest, find a condition whose prior truth guarantees the subsequent truth of the proposition of the theorem. Two propositions related in this way are often called a postcondition and a precondition. Showing the precondition to be true then proves the truth of the postcondition and hence the theorem. Normally, the notion of the precondition is defined in such a way that its truth does not guarantee that the statement(s) in question terminate; therefore, termination must be proved separately.

In implementing this proof strategy, it is especially helpful to view the precondition (e.g. Q) and the postcondition (e.g. P) as subsets of $\mathbb{D}$ (more specifically, the subsets containing all d such that $Q(d)$ = true and $P(d)$ = true respectively, see Section 2.0.5). Conversely, a subset can be viewed as a condition: the condition Q, for example, is true for d (i.e. $Q(d)$ = true) if and only if d is in the subset Q. (For our purposes, the value false and an undefined value of a condition will frequently be considered to be equivalent.) The statement S in question is interpreted as a function on $\mathbb{D}$ to $\mathbb{D}$ (see Section 2.1). Of interest are the sets

$$S(Q) = \{S(d) \mid d \text{ in the domain of } S \textbf{ and } d \text{ in } Q\}$$

the image of Q under S, and especially

$$S^{-1}(P) = \{d \mid d \text{ in the domain of } S \textbf{ and } S(d) \text{ in } P\}$$

the inverse image of P under S.

Definition 3.0: A subset Q of $\mathbb{D}$ is a *precondition* of a given postcondition P with respect to the statement S (under the function S) if $S(d)$ is in P for all d which are both in Q and in the domain of S (i.e. for all d in the intersection of Q and the domain of S). Symbolically,

$$\{Q\}\ S\ \{P\}$$

That is, if Q is true immediately before S is executed, P will be true immediately after S has been executed (if the effect of executing S is defined and execution terminates). In this case, one says that S is *partially correct*.

Note that the above definition allows elements d in $\mathbb{D}$ for which $S(d)$ is not defined to be included in the precondition (set) Q. In particular, the data environment resulting from executing S upon the data environment d may be undefined because either (a) execution does not terminate (i.e. $S^*([d])$ is an infinite sequence), (b) execution of the statement involves a reference to a variable not contained in the data environment in question

(i.e. $S^*([d])$ is undefined) or (c) for some other reason, an expression cannot be evaluated (also in this case, $S^*([d])$ is undefined; see Section 2.0.3). In practical work, the first of these reasons – an infinite loop – is the most important, although all must be considered.

It is sometimes useful to consider preconditions which also guarantee a defined result and termination of the execution of the statement in question. We therefore introduce the following stronger notion of a strict precondition.

Definition 3.1: A subset Q of $\mathbb{D}$ is a *strict precondition* of a given postcondition P with respect to the statement S (under the function S) if

Q is a precondition of P under S

and

Q is a subset of the domain of S

In this case, one writes

$$\{Q\}\ S\ \{P\} \text{ strictly}$$

That is, if Q is true immediately before S is executed, then the result of the execution of S is defined, execution will terminate and P will be true immediately after S has been executed. In this case, one says that S is *totally correct*.

Thus, if one proves (a) that a statement, program, etc. is partially correct and (b) that its execution yields a result (terminates), then the statement is totally correct.

Several facts follow directly from the above definitions. A strict precondition is a precondition. The intersection of any precondition Q and the domain of the statement S in question is a strict precondition. A subset of a precondition is a precondition. A subset of a strict precondition is a strict precondition.

A subset Q of $\mathbb{D}$ is a precondition of P with respect to (under) S if and only if $S(Q)$ is a subset of P. The inverse image of P under S is, of course, the weakest strict precondition of P with respect to S and every strict precondition of P under S is, therefore, a subset thereof.

A precondition Q, being a sufficient condition, has the property that

$$\text{valexp}(Q, d) \Rightarrow \text{valexp}(P, S(d)), \text{ for all } d \text{ in the domain of } S$$

The weakest precondition Qw, being a both necessary and sufficient condition, has the property that

$$\text{valexp}(Q\text{w}, d) \Leftrightarrow \text{valexp}(P, S(d)), \text{ for all } d \text{ in the domain of } S$$

The weakest precondition of a postcondition P with respect to a statement S is often written $wp(P, S)$. For the weakest strict precondition we will use the notation $wsp(P, S)$.

If all d in the domain of S are also in the set $wp(P, S)$, i.e. if the condition $wp(P, S)$ is identically true on the domain of S, then P is always true after executing S, regardless of the initial data environment.

If the set $wp(P, S)$ is empty, i.e. if the condition $wp(P, S)$ is either false or undefined for each data environment in the domain of S, then P is never true after executing S, regardless of the initial data environment; the postcondition P will never be satisfied.

The relationships among a postcondition, its preimage, a precondition and its image are shown diagrammatically in Fig. 3.0.

For any particular postcondition P and statement S it is possible and, in fact, usual that many preconditions Q exist. In order to reduce the difficulty of proving Q true, it is advantageous to find the weakest (least restrictive) of all such preconditions or a precondition sufficiently akin to the weakest precondition. In particular, this need is satisfied by any precondition which includes the weakest strict precondition, but not necessarily all data environments outside of the domain of the statement in question. Such a precondition is not, precisely speaking, necessarily the weakest precondition. Because it is weak enough, however, to include all the data environments which the statement maps into the given postcondition, it has most of the convenient practical properties of the weakest precondition. Such a precondition, which is often easily constructed, represents, loosely speaking, a fuzzy intermediate between the weakest strict precondition and the weakest

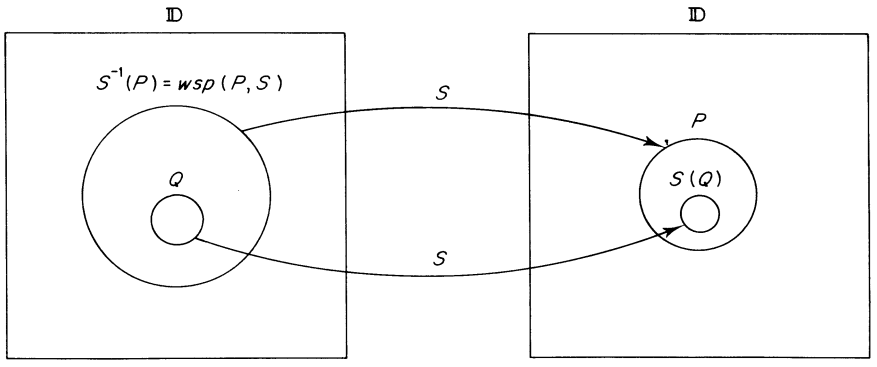


Fig. 3.0 The relationships among a postcondition, its preimage, a precondition and its image

precondition. The area of fuzziness is outside of the domain of the statement and can usually be excluded easily in another way if necessary. We therefore make the following definition.

Definition 3.2: A subset Q of $\mathbb{D}$ is a *complete precondition* of a given postcondition P with respect to the statement S if

Q is a precondition of P under S

and

$wsp(P, S)$ is a subset of Q

One writes

$\{Q\}\ S\ \{P\}$ completely

The relationships among the concepts of a precondition, a strict precondition and a complete precondition are illustrated in Fig. 3.1.

The following theorems provide the basis for 'dividing and conquering' and can often enable one to work with simpler assertions, later combining results.

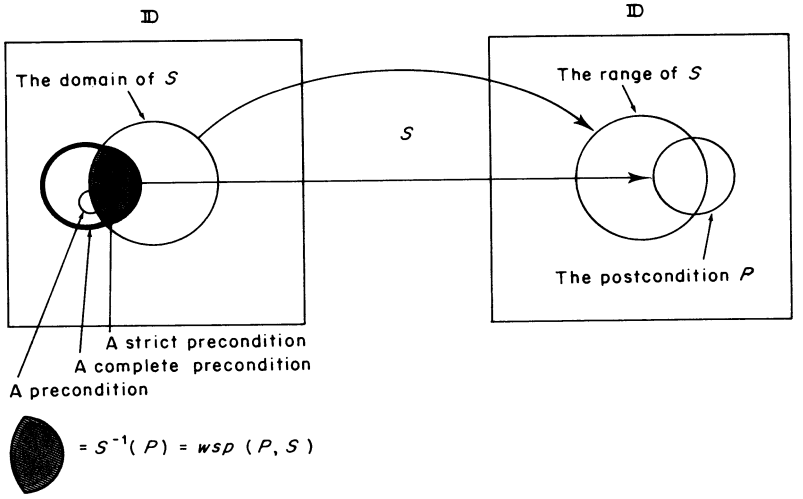


Fig. 3.1 The relationships among the domain of a statement, a precondition, a strict precondition, a complete precondition and the preimage of the postcondition

Theorem 3.0: If $Q1$ and $Q2$ are preconditions of the postconditions $P1$ and $P2$ respectively under the statement S, then

($Q1$ **and** $Q2$) is a precondition of ($P1$ **and** $P2$) under S

and

($Q1$ **or** $Q2$) is a precondition of ($P1$ **or** $P2$) under S

Theorem 3.1: If $Q1$ and $Q2$ are strict preconditions of the postconditions $P1$ and $P2$ respectively under the statement S, then

($Q1$ **and** $Q2$) is a strict precondition of ($P1$ **and** $P2$) under S

and

($Q1$ **or** $Q2$) is a strict precondition of ($P1$ **or** $P2$) under S

Theorem 3.2: If $Q1$ and $Q2$ are complete preconditions of the postconditions $P1$ and $P2$ respectively under the statement S, then

($Q1$ **and** $Q2$) is a complete precondition of ($P1$ **and** $P2$) under S

and

($Q1$ **or** $Q2$) is a complete precondition of ($P1$ **or** $P2$) under S

Exercise

1 Prove theorems 3.0, 3.1 and 3.2.
2 Show that the following three statements are equivalent for any statement S and any subsets P and Q of $\mathbb{D}$.
(a) $\{Q\}\ S\ \{P\}$
(b) $S(Q)$, the image of Q under S, is a subset of P.
(c) The intersection of Q and the domain of S is a subset of $S^{-1}(P)$, the preimage (inverse image) of P under S.
3 Show that the following three statements are equivalent.
(a) $\{Q\}\ S\ \{P\}$ strictly and completely
(b) $Q = wsp(P, S)$
(c) $Q = S^{-1}(P)$
4 Show that the weakest strict precondition is the strongest complete precondition.

In each of the following sections of this chapter, generally useful theorems, lemmata, etc. relating to a particular type of program construct will be presented.

3.0 Proof rules for the assignment statement

3.0.0 Lemma for the assignment statement

Lemma 3.0: Consider the assignment statement

$x := E$

called A below, where x is the name of a variable and E is an expression. The following equations hold for all data environments d in the domain of A:

$$\text{valvar}(x, A(d)) = \text{valexp}(E, d)$$

and

$$\text{valvar}(y, A(d)) = \text{valvar}(y, d), \text{ for all other variable names } y \text{ in } d$$

Generalizing for the case of the multiple assignment statement

$$(x1, x2, \ldots) = (E1, E2, \ldots)$$

we have

$$\text{valvar}(x1, A(d)) = \text{valexp}(E1, d)$$

$$\text{valvar}(x2, A(d)) = \text{valexp}(E2, d)$$

$$\ldots$$

and

$$\text{valvar}(y, A(d)) = \text{valvar}(y, d), \text{ for all other variable names } y \text{ in } d$$

The values of all concealed variables (see Section 2.1.5) remain unchanged by the execution of the assignment statement; i.e. for all variable names y in d and for all integers $n \geq 1$,

$$\text{valvar}(y, (\textbf{release } y)^n(A(d))) = \text{valvar}(y, (\textbf{release } y)^n(d))$$

In particular, the values of concealed variables with the same name as a variable whose value is being changed (e.g. x, $x1$, $x2$, etc. above) also remain unaffected by the execution of the assignment statement:

$$\text{valvar}(x, (\textbf{release } x)^n(A(d))) = \text{valvar}(x, (\textbf{release } x)^n(d))$$

$$\text{valvar}(x1, (\textbf{release } x1)^n(A(d))) = \text{valvar}(x1, (\textbf{release } x1)^n(d))$$

$$\text{valvar}(x2, (\textbf{release } x2)^n(A(d))) = \text{valvar}(x2, (\textbf{release } x2)^n(d))$$

$$\ldots$$

Proof: The above equations follow directly from the definition of the assignment statement (see Section 2.1.0). Note that lemma 3.0 does not, therefore, necessarily apply to those real systems which do not satisfy the requirements of the definition of the assignment statement. ■

3.0.1 Theorem for the assignment statement

Theorem 3.3: Consider a given proposition P over variable names to be evaluated in the context of a data environment and the assignment statement

$$x := E$$

as in the lemma in Section 3.0.0 above. If a proposition Q is formed by replacing every occurrence (if any) of the variable name x in P by the expression E enclosed in parentheses, then

$\{Q\}\ x := E\ \{P\}$ completely

Note: The proposition Q formed as described above is often written P_E^x.

Proof: P has, in general, the form $P(x, y1, y2, \ldots)$. Then

$$Q(x, y1, y2, \ldots) = P(E, y1, y2, \ldots)$$

For any d in the domain of the assignment statement A above,

$$\begin{aligned} \text{valexp}(Q, d) &= \text{valexp}(P(E, y1, y2, \ldots), d) \\ &= \text{P}(\text{valexp}(E, d), \text{valvar}(y1, d), \text{valvar}(y2, d), \ldots) \end{aligned}$$

Equality (=) here is understood to mean that if either side of the equation is defined, then the other side is also defined and the values are equal. If either side is undefined, the other side is also undefined.

Applying lemma 3.0, we have

$$\begin{aligned} \text{valexp}(Q, d) &= P(\text{valvar}(x, A(d)), \text{valvar}(y1, A(d)), \text{valvar}(y2, A(d)), \ldots) \\ &= \text{valexp}(P, A(d)) \end{aligned}$$

Thus, if d is in Q then $A(d)$ is in P. Q fulfills the definition of a precondition of P under A, proving the first part of the theorem

$$\{Q\}\ x := E\ \{P\}$$

Because $\text{valexp}(Q, d) = \text{valexp}(P, A(d))$ for all d in the domain of A, d is in Q if $A(d)$ is in P. Thus the inverse image of P – i.e. the weakest strict precondition – is a subset of Q. Q satisfies, then, the definition of a complete precondition. ■

The corresponding theorem for the multiple assignment statement can be proved in the same manner. Q is formed by replacing $x1$, $x2$, etc. in P by $E1$, $E2$, etc. respectively simultaneously, not one after another.

The proposition Q defined in this theorem is not necessarily a strict precondition. The truth of Q guarantees only that the variables appearing in Q are contained in the data environment upon which A is to be executed. These variables are comprised of the variables other than x appearing in P and, if x appears in P, the variables appearing in E. Neither P nor E need refer to all variables which must be present in the data environment upon which A is executed. For example, consider the assignment statement

$$x := y$$

and the proposition P

$$z = 4$$

Then

$$Q = P_y^x = (z = 4)$$

A data environment satisfying this condition need not contain either variable x or y. In particular the data environment

$$d = [(z, \mathbb{Z}, 4)]$$

is in Q but is not in the domain of A. In this case, Q is not a strict precondition; it is, however (as it must be according to the theorem), a precondition and a complete precondition.

The meaning of the statement in the theorem 'a proposition P over variable names to be evaluated in the context of a data environment' requires some elaboration. In the final analysis, it means any proposition P for which the steps in the proof are valid. In particular, the proposition P must be an expression in which the variable name to be replaced represents a value to be used in performing operations (evaluating functions) referenced (explicitly or implicitly) in the expression. Furthermore, the corresponding value is to be determined in the context of a specific data environment as described in Sections 2.0.2 and 2.0.4.

Not all meaningful logical statements satisfy this requirement. For example, the logical statement 'the variable named x is contained in the data environment d' is not a statement referring finally to a value of x but to the name itself. Substituting y for x (corresponding to the statement $x := y$) leads to a would-be precondition which is not applicable. Substituting $3 * (y + z)$ for x (corresponding to the statement $x := 3 * (y + z)$) would lead, of course, to a totally meaningless statement.

Care should also be exercised when applying this theorem to conditional expressions containing references to array variables. For example, consider the statement $x(i) := 1$ and the postcondition $x(i) = x(j)$. The sequence of symbols '$x(i)$' is not a variable name and the theorem may not be applied to such an identifier. The names of the variables comprising the array are typically '$x(1)$', '$x(2)$', etc. In order to obtain the true variable name, the subscript expression(s) must be replaced by its (their) value(s) in the data environment in question. To indicate this explicitly, we rewrite the postcondition as follows:

$$x(\text{valvar}(i, A(d))) = x(\text{valvar}(j, A(d)))$$

The values of i and j are not changed by the execution of the statement. In other words,

$$\text{valvar}(i, A(d)) = \text{valvar}(i, d)$$

and

$$\text{valvar}(j, A(d)) = \text{valvar}(j, d)$$

The postcondition can, therefore, be rewritten as

$$x(\text{valvar}(i, d)) = x(\text{valvar}(j, d))$$

Note that this is still the *post*condition, i.e. the variables $x(\cdot)$ are to be evaluated in $A(d)$, not d, even though the subscripts (i and j) are evaluated in d.

The meaning of the assignment statement can be made more explicit by writing it as follows:

$$x(\text{valvar}(i, d)) := 1$$

We may now apply the theorem to this form of the statement and the last version of the postcondition above. We must, however, distinguish between two cases: (a) $\text{valvar}(i, d) = \text{valvar}(j, d)$ and (b) $\text{valvar}(i, d) \neq \text{valvar}(j, d)$. In the first case, both $x(\text{valvar}(i, d))$ and $x(\text{valvar}(j, d))$ – being the same variable name – must be replaced by 1, resulting in the precondition $1 = 1$, which is equivalent to 'true'. In the second case, substitution in accordance with the theorem leads to the precondition $1 = x(\text{valvar}(j, d))$. Combining the results for these two cases, the precondition becomes

$$[\text{valvar}(i, d) = \text{valvar}(j, d)]$$
$$\textbf{or } [(\text{valvar}(i, d) \neq \text{valvar}(j, d)) \textbf{ and } (x(\text{valvar}(j, d)) = 1)]$$

Because a precondition is to be evaluated in d, the above expression can be simplified to

$$[i = j] \textbf{ or } [(i \neq j) \textbf{ and } (x(j) = 1)]$$

which is equivalent to

$$[i = j] \textbf{ or } [x(j) = 1]$$

This is a complete precondition of the postcondition and statement given above. Unthinkingly (and incorrectly) applying the theorem above might lead one to conclude that $(x(j) = 1)$ is the complete precondition being sought. While this expression is a precondition, it is not a complete precondition.

3.0.2 Application of the lemma and theorem for the assignment statement

By applying lemma 3.0, one can often derive a postcondition from a given precondition, working forward through the program. Generally, however, it is more useful and productive to work backward through a program, using theorem 3.3 to derive a complete precondition from a given postcondition.

3.1 Proof rules for the *if* statement

3.1.0 The progressive theorem for the *if* statement

Theorem 3.4: If

$$\{Q \textbf{ and } B\}\ S1\ \{P\}$$

and

$$\{Q \textbf{ and not } B\}\ S2\ \{P\}$$

then

$$\{Q\} \textbf{ if } B \textbf{ then } S1 \textbf{ else } S2 \textbf{ endif } \{P\}$$

Note: the above **if** statement will be called S below.

Proof: A data environment d is in the domain of the **if** statement above if and only if

1 d is in the domain of B and
2 $B(d) = \text{true} \Rightarrow d$ is in the domain of $S1$ and
3 $B(d) = \text{false} \Rightarrow d$ is in the domain of $S2$.

Consider any d in the set Q and in the domain of the **if** statement. If $B(d) = \text{true}$, then d is both in Q and in the domain of $S1$. By the first hypothesis of the theorem. $S1(d)$ is in P and, therefore, by the definition

of the **if** statement (see Section 2.1.1), $S(d)$ is in P. If, on the other hand, $B(d)$ = false, then d is both in Q and in the domain of $S2$. By the second hypothesis of the theorem and, again, the definition of the **if** statement, $S2(d)$ and $S(d)$ are in P.

That is, for all d in Q and in the domain of the **if** statement, $S(d)$ is in P. The set Q is, therefore, a precondition of P with respect to the **if** statement. ■

The progressive proof rule for an **if** statement is shown in Fig. 3.2.

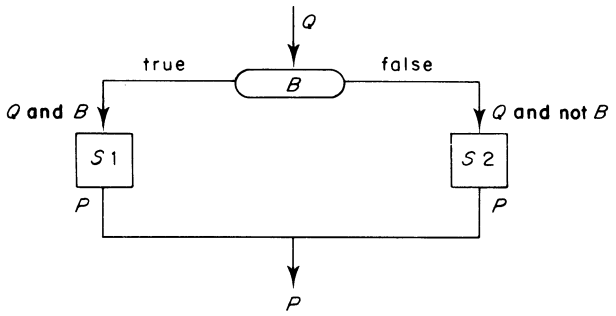


Fig. 3.2 The progressive proof rule for an **if** statement

3.1.1 The retrogressive theorem for the *if* statement

Theorem 3.5: If

$\{Q1\}\ S1\ \{P\}$ and

$\{Q2\}\ S2\ \{P\}$

then

$\{(Q1$ **and** $B)$ **or** $(Q2$ **and not** $B)\}$ **if** B **then** $S1$ **else** $S2$ **endif** $\{P\}$

Note: We define $Q = ((Q1$ **and** $B)$ **or** $(Q2$ **and not** $B))$.

Proof: From the above definition of Q, it follows that

$(Q$ **and** $B) = (Q1$ **and** $B)$

and

$(Q$ **and not** $B) = (Q2$ **and not** $B)$

By the first hypothesis of the theorem, $Q1$ is a precondition of P under $S1$. Its subset $(Q1$ **and** $B)$ is, therefore, also a precondition. But $(Q1$ **and** $B) = (Q$ **and** $B)$, so $(Q$ **and** $B)$ is a precondition of P under $S1$. Similarly, $(Q$ **and not** $B)$ is a precondition of P under $S2$. Applying the progressive theorem for the **if** statement (see Section 3.1.0), it follows that Q is a precondition of P under the **if** statement. ■

If $Q1$ and $Q2$ are complete preconditions, then Q is a complete precondition of P under the **if** statement.

The retrogressive proof rule is shown in Fig. 3.3 and the weakest strict precondition under an **if** statement is shown in Fig. 3.4.

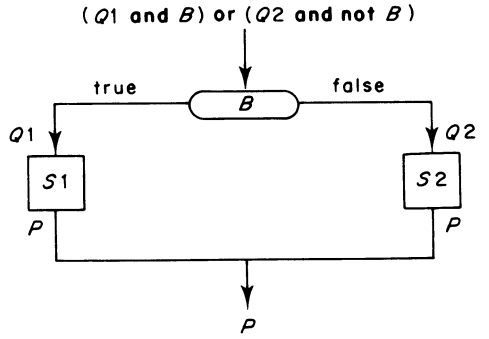


Fig. 3.3 The retrogressive proof rule for an **if** statement

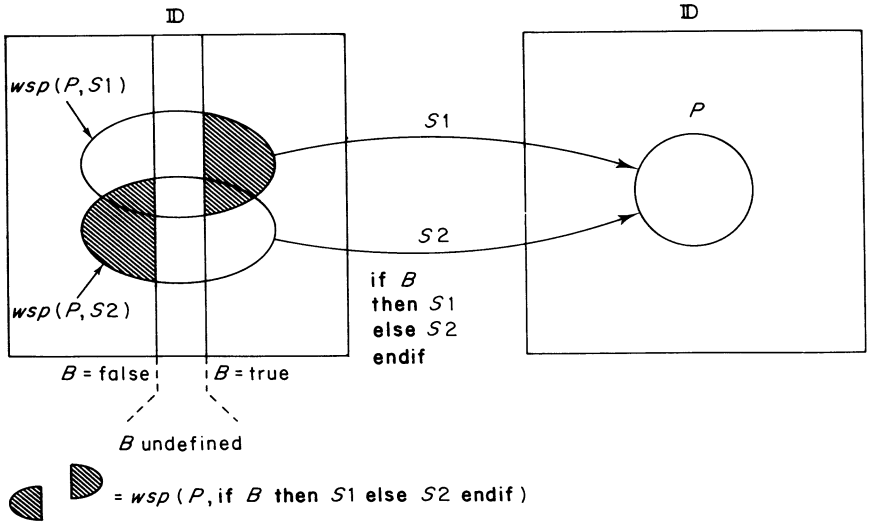


Fig. 3.4 The weakest strict precondition under an **if** statement

3.1.2 Application of the theorems for the *if* statement

Sections 3.1.0 and 3.1.1 above present two versions of fundamentally the same theorem. The form of the progressive theorem (Section 3.1.0) is adapted to the problem of deriving a postcondition from a known precondition. The form of the retrogressive theorem (Section 3.1.1) enables one to determine directly a precondition from a given postcondition. This is often the more useful procedure.

3.2 Proof rules for a sequence of program statements

3.2.0 The theorem for preconditions under a sequence of statements

Theorem 3.6: If

$\{Q\}\ S1\ \{Q1\}$

and

$\{Q1\}\ S2\ \{P\}$

then

$\{Q\}\ (S1, S2)\ \{P\}$

Proof: A data environment d is in the domain of the sequence of statements $(S1, S2)$ if and only if

1 d is in the domain of $S1$ and
2 $S1(d)$ is in the domain of $S2$.

For any d in Q and in the domain of $S1$, $S1(d)$ is in $Q1$. If this $S1(d)$ is also in the domain of $S2$, then $S2(S1(d))$ is in P; i.e. if d is in Q and in the domain of $(S1, S2)$, then $S2(S1(d))$ is in P. Thus, Q is a precondition of P under $(S1, S2)$. ■

3.2.1 The theorem for complete preconditions under a sequence of statements

Theorem 3.7: If

$\{Q\}\ S1\ \{Q1\}$ completely

and

$\{Q1\}\ S2\ \{P\}$ completely

then

$\{Q\}\ (S1, S2)\ \{P\}$ completely

Proof: By the previous theorem for preconditions (see Section 3.2.0), Q is a precondition of P under $(S1, S2)$; symbolically, $\{Q\}\ (S1, S2)\ \{P\}$. Here, we need to show only that Q is also complete, i.e. that $wsp(P, (S1, S2))$ is a subset of Q.

From the two hypotheses of this theorem it follows that

$wsp(Q1, S1)$ is a subset of Q

and

$wsp(P, S2)$ is a subset of $Q1$

For any d in $wsp(P, (S1, S2))$ (which is the inverse image of P under $(S1, S2)$), d is in the domain of $S1$, $S1(d)$ is in the domain of $S2$ and $S2(S1(d))$ is in P. Therefore, $S1(d)$ is in $wsp(P, S2)$ and hence in $Q1$. This implies, in turn, that d is in $wsp(Q1, S1)$ and hence in Q; i.e. for all d in $wsp(P, (S1, S2))$, d is in Q. Thus, $wsp(P, (S1, S2))$ is a subset of Q. ■

3.2.2 Application of the theorems for a sequence of program statements

The above theorems can be generalized and easily proved for a sequence of any number of program statements. Probably the most important implication thereof is that one can find a precondition with respect to a sequence of statements by working backward, statement by statement, from the desired postcondition. First, one finds a precondition under the last statement, which is then used as the postcondition for the next to the last statement, etc. The above theorems guarantee that a precondition with respect to the first statement found in this way is also a precondition with respect to the entire sequence of statements.

If each precondition found in the above process is a complete precondition, then the precondition under the first statement in the sequence found in the way described above is a complete precondition of the given postcondition with respect to the entire sequence of statements.

3.3 Proof rules for the *while* loop

3.3.0 The loop theorem

Theorem 3.8: If

$\{I$ **and** $B\}\ S\ \{I\}$

then

$\{I\}$ **while** B **do** S **endwhile** $\{I$ **and not** $B\}$

Note: Below we will call the statement **while** B **do** S **endwhile** simply W.

Proof: For any data environment d in the domain of W, there exists an n such that

$W(d) = S^n(d)$, (i.e. for the particular d, W is equivalent to S^n)

$B(S^n(d)) =$ false

and

$B(S^i(d)) =$ true, for all i such that $0 \leq i \leq n-1$

(see definitions 2.14 and 2.29 in Chapter 2).

By the last equation above, the condition B is satisfied before the execution of each S in the sequence of n S's. Initially, the condition I is also true by hypothesis. Thus, the precondition of the first S is satisfied and therefore the condition I will be true after the execution of the first S in the sequence. The condition (I **and** B) will be true before the execution of the second S, condition I will, therefore, be true thereafter and so on.

After the execution of the final S in the sequence, condition I will be true. By the next to last equation above, condition B will be false, i.e. condition (I **and not** B) will be true. ■

Intuitively and operationally, the flow chart in Fig. 3.5 illustrates the key idea of the proof particularly lucidly.

The condition I in this proof is called the loop invariant, because its value (true) is constant throughout the execution of the loop. It may, and often does, become false at intermediate points within S, but it is always true immediately before and after each execution of S.

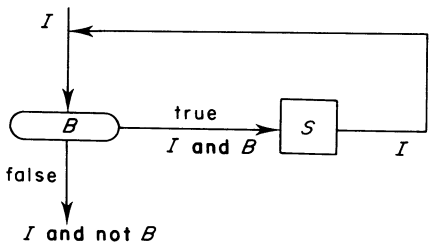


Fig. 3.5 The proof rule for a **while** loop

The loop invariant and this theorem are certainly among the most important concepts for designing and constructing computer programs and for proving them correct.

3.3.1 The theorem for the weakest strict precondition of a loop

In this section we will consider the question of the weakest strict precondition of a postcondition P with respect to the **while** statement

while B **do** S **endwhile**

which will often be abbreviated W below. Closely related is the problem of determining the weakest loop invariant.

We begin by noting that if the loop terminates, the value of the expression B in the resulting data environment $W(d)$ will be defined and false (see the loop theorem in Section 3.3.0 above). If $W(d)$ satisfies the postcondition P, then it will be in the intersection of the sets P, the domain of B and $\mathbb{D} - B$. We can restrict our consideration, therefore, to postconditions P which are subsets of the domain of B and of $\mathbb{D} - B$ without detracting from the generality of our results.

If a postcondition P' is given which does not fulfill the above requirement, then one should work with the new postcondition (Boolean expression) $P = (P'$ **and not** $B)$. This postcondition will necessarily be a subset of the domain of B and of $\mathbb{D} - B$.

It is helpful to consider the preconditions under which the loop terminates after exactly n executions of S with the resulting data environment $d1$ in P. The precondition that it terminates at all with $W(d)$ in P is then the union of the preconditions for all n. We begin the formal analysis of this section, therefore, by stating the following lemma.

Lemma 3.1: Consider a given condition B, statement S, non-negative integer n and proposition P, which is a subset of the domain of B and of $\mathbb{D} - B$. Then the **while** statement (see above), when executed upon any data environment d in $\mathbb{D}$, terminates after exactly n executions of S with $W(d)$ in P, i.e.

$A1$: $S^n(d)$ is defined and in P

and

$A2$: $B(S^n(d)) =$ false

and

$A3$: $B(S^i(d)) =$ true, for all i such that $0 \leq i \leq n-1$

if and only if

d is in $Y^n(P)$

where Y is defined as the function

$Y(M) = (B \textbf{ and } S^{-1}(M))$

for all subsets M of $\mathbb{D}$.

Proof (preliminary comments): The subset $Y^n(P)$ of $\mathbb{D}$ is the set of data environments upon which the **while** statement terminates after exactly n iterations with $W(d)$ in P. If executed upon a data environment d, the **while** statement will terminate after exactly $n+1$ iterations with $W(d)$ in P if (a) $B(d)$ is true (otherwise it would terminate immediately) and (b) when executed upon $S(d)$, the **while** statement terminates after exactly n iterations with $W(d)$ in P. The latter condition is equivalent to the condition

d in $S^{-1}(Y^n(P))$

Combining with the first condition stated above ($B(d)$ = true) leads to the definition of Y given in the statement of the lemma. The detailed proof below essentially traces this logical argument in reverse.

Fig. 3.6 illustrates the sets $Y^n(P)$ and facilitates following the proof.

Proof (detailed): We begin by noting that the propositions $A1$, $A2$ and $A3$ together repeat in more formal notation the proposition stated in words in the first clause of the second sentence of the lemma (see definition 2.14).

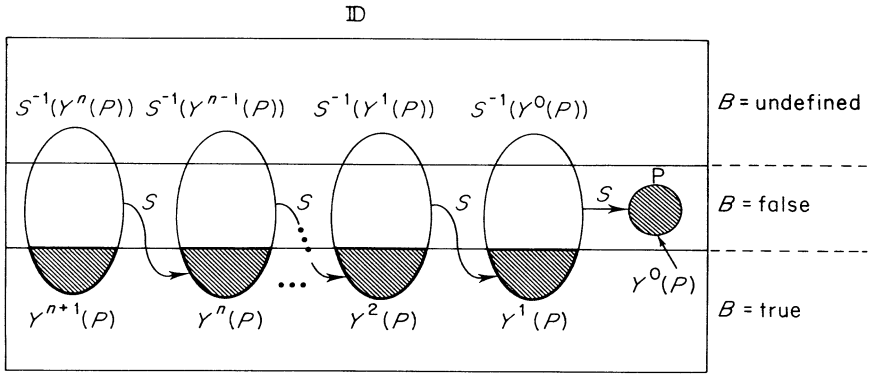


Fig. 3.6 The weakest strict precondition under a **while** loop

Also, we note that $A1 \Rightarrow A2$, because P is a subset of **not** B. Therefore, we need only prove that

($A1$ **and** $A3$) $\Leftrightarrow$ d in $Y^n(P)$

We will prove this lemma by induction on n. If $n = 0$, the proposition of the lemma reduces to a tautology, so is true.

We assume that the lemma is true for n and will prove it to hold for $n+1$ also.

$Y^{n+1}(P) = Y(Y^n(P)) = B \textbf{ and } S^{-1}(Y^n(P))$

Therefore, d is in $Y^{n+1}(P)$ if and only if d is in B and in $S^{-1}(Y^n(P))$, i.e. if and only if

$B(d)$ = true

and

$S(d)$ is defined and in $Y^n(P)$

Under the assumption that the lemma is true for n, the last proposition above is equivalent to

$B(d)$ = true

and

$S^n(S(d))$ is defined and in P

and

$B(S^i(S(d))$ = true, for all i such that $0 \leq i \leq n-1$

Simplifying, this is equivalent to

$S^{n+1}(d)$ is defined and in P

and

$B(S^i(d))$ = true, for all i such that $0 \leq i \leq n$

That is, d is in $Y^{n+1}(P)$ if and only if $A1$ and $A3$ hold for $n+1$. ■

Theorem 3.9: If the given proposition P is a subset of the domain of B and of $\mathbb{D} - B$, then the weakest strict precondition of the proposition P with respect to the statement **while** B **do** S **endwhile** is

$wsp(P, W) = P \textbf{ or } Y(P) \textbf{ or } Y^2(P) \textbf{ or } \ldots Y^i(P) \textbf{ or } \ldots$

where Y is as defined in the lemma above.

Proof: We define Q as

$$Q = P \textbf{ or } Y(P) \textbf{ or } Y^2(P) \textbf{ or } \ldots Y^i(P) \textbf{ or } \ldots$$

For any d in Q, d is in $Y^n(P)$ for some $n \geq 0$. By the lemma above and definition 2.14 $W(d)$ is defined and is in P. Therefore, Q is a strict precondition of P under W.

Consider any d in $\mathbb{D}$. If $W(d)$ is defined, that is, if the **while** statement terminates, then by definition 2.14, it terminates after some specific number n of executions of S. If $W(d)$ is in P, then, by the lemma above, d is in $Y^n(P)$ and hence in Q; i.e. if $W(d)$ is defined and in P, d is in Q. Therefore, the inverse image of P under W, which is $wsp(P, W)$, is a subset of Q.

But Q is a strict precondition. Therefore, Q must be $wsp(P, W)$, the weakest strict precondition of P under W. ■

The weakest strict precondition of P under W is also the weakest strict loop invariant.

3.3.2 The corollary for complete preconditions of a loop

The following corollary is less sharp than the theorem presented in Section 3.3.1 above, but it is often of greater practical use.

Corollary: If the given proposition P is a subset of the domain of B and of $\mathbb{D} - B$, then

$$Q = Z_0 \textbf{ or } Z_1 \textbf{ or } Z_2 \textbf{ or } \ldots Z_i \textbf{ or } \ldots$$

is a complete precondition of P with respect to the statement

while B **do** S **endwhile**

where

$$Z_0 = P$$

$$Z_i = (B \textbf{ and } C_{i-1}), \text{ for } i = 1, 2, \ldots$$

and C_{i-1} is any complete precondition of Z_{i-1} with respect to S.

Proof: As in previous sections, we will refer below to the statement **while** B **do** S **endwhile** as W.

First, we will prove that Q as defined above is a precondition of P under W. Then, we will show that the weakest strict precondition of P under W is a subset of Q. Thus, Q satisfies the requirements imposed by the definition of a complete precondition.

In order to demonstrate that Q is a precondition of P under W, we consider any data environment d in Q and in the domain of W. From the above definition of Q it follows that d must be in one of the sets Z_i ($i = 0, 1, 2, \ldots$). That is, there exists an i such that d is in Z_i.

If $i = 0$, then d is in $Z_0 = P$. It follows that $B(d) =$ false, $W(d) = d$ and, therefore, $W(d)$ is in P. Thus, Q satisfies the definition of a precondition of P under W in this case.

Otherwise, $i > 0$. The fact that d is in $Z_i = (B \textbf{ and } C_{i-1})$ implies that

$B(d) =$ true,	d is in C_{i-1} and, therefore,	$S(d)$ is in Z_{i-1}
$B(S(d)) =$ true,	$S(d)$ is in C_{i-2} and, therefore,	$S^2(d)$ is in Z_{i-2}
...		
$B(S^{i-2}(d)) =$ true,	$S^{i-2}(d)$ is in C_1 and, therefore,	$S^{i-1}(d)$ is in Z_1
$B(S^{i-1}(d)) =$ true,	$S^{i-1}(d)$ is in C_0 and, therefore,	$S^i(d)$ is in $Z_0 = P$
$B(S^i(d)) =$ false.		

Thus, $W(d) = S^i(d)$, $W(d)$ is in P and, therefore, Q is a precondition of P under W.

In order to show that $wsp(P, W)$ is a subset of Q, we next prove by induction on i that

$$Y^i(P) \text{ is a subset of } Z_i, \text{ for } i = 0, 1, 2, \ldots$$

We begin with the iterative step. If

$$Y^i(P) \text{ is a subset of } Z_i$$

then the inverse images of these sets under any function, in particular S, are in the same relation to each other:

$$S^{-1}(Y^i(P)) \text{ is a subset of } S^{-1}(Z_i)$$

But $S^{-1}(Z_i)$ is the weakest strict precondition of Z_i under S, which is a subset of any complete precondition C_i of Z_i under S (see the definition of a complete precondition). Therefore,

$$S^{-1}(Y^i(P)) \text{ is a subset of } C_i$$

and

$$(B \textbf{ and } S^{-1}(Y^i(P))) \text{ is a subset of } (B \textbf{ and } C_i)$$

Referring to the definitions of $Y(\cdot)$ (see lemma 3.1) and of Z_i, it follows that

$$Y^{i+1}(P) \text{ is a subset of } Z_{i+1}$$

completing the iterative step in the proof by induction. We obtain the starting point for the inductive proof by noting that

$$Y^0(P) = P = Z_0$$

from which it follows that

$Y^0(P)$ is a subset of Z_0

Thus,

$Y^i(P)$ is a subset of Z_i, for $i = 0, 1, 2, \ldots$

and

(P **or** $Y(P)$ **or** $Y^2(P)$ **or** $\ldots$) is a subset of (Z_0 **or** Z_1 **or** Z_2 **or** $\ldots$)

From the theorem for the weakest strict precondition of a loop (see Section 3.3.1) and the definition of Q as given in the statement of this corollary, it follows that

$wsp(P, W)$ is a subset of Q

Summarizing, Q is a precondition of P under W and includes the weakest strict precondition of P under W as a subset, thereby satisfying the definition of a complete precondition of P under W. ■

A complete precondition of P under W is also a loop invariant.

If the above corollary is modified by deleting the word 'complete' in both places, the resulting statement is also true. That is, if every C_{i-1} is a precondition of Z_{i-1} (but not necessarily a complete precondition), Q will be a precondition of P under W. Q will also be a loop invariant.

The precondition Q delivered by the above corollary guarantees further that the loop will not execute endlessly. This is shown formally in the following lemma and theorem.

Lemma 3.2: Let Z_i be defined as in the corollary above except that we require only that each C be a precondition of the corresponding Z, not necessarily a complete one. Then

$$\{Z_i\}\ S^i\ \{\textbf{not } B\}$$

for every non-negative integer i.

Proof: When $i = 0$, the thesis of this lemma reduces to

$$\{Z_0\}\ S^0\ \{\textbf{not } B\}$$

Because S^0 is a null statement and Z_0 is a subset of (**not** B), this statement is clearly true.

We begin the proof of this lemma for $i > 0$ by noting that by definition of the sets C,

$$\{C_{i-1}\}\ S\ \{Z_{i-1}\}$$

Furthermore,

$$Z_i = (B \textbf{ and } C_{i-1})$$

Thus, Z_i is a subset of C_{i-1}. Therefore

$$\{Z_i\}\ S\ \{Z_{i-1}\}$$

Similarly,

$$\{Z_{i-1}\}\ S\ \{Z_{i-2}\}$$

$$\{Z_{i-2}\}\ S\{Z_{i-3}\}$$

$$\ldots$$

$$\{Z_1\}\ S\ \{Z_0\}$$

Applying theorem 3.6 (see also Section 3.2.2), we have

$$\{Z_i\}\ S^i\ \{Z_0\}$$

Z_0 is a subset of (**not** B). It follows that

$$\{Z_i\}\ S^i\ \{\textbf{not } B\}.$$ ■

The fact that the precondition Q is a loop invariant can be seen particularly clearly from part of the above proof. Applying theorem 3.0 from the introductory part of this chapter to the sequence of statements

$$\{Z_1\}\ S\ \{Z_0\}$$

$$\{Z_2\}\ S\ \{Z_1\}$$

$$\ldots$$

$$\{Z_i\}\ S\ \{Z_{i-1}\}$$

$$\ldots$$

one obtains

$$\{Z_1 \textbf{ or } Z_2 \textbf{ or } \ldots Z_i \textbf{ or } \ldots\}\ S\ \{Z_0 \textbf{ or } Z_1 \textbf{ or } \ldots Z_i \textbf{ or } \ldots\}.$$

The above precondition is equal to (Q **and** B); the postcondition, to Q. Thus, the body of the **while** loop preserves the truth of Q and Q is a loop invariant.

We now formulate the termination theorem. Generally speaking, it states that the precondition Q of the corollary above guarantees that a loop will not execute endlessly if no subsidiary loop executes endlessly.

Theorem 3.10: Let Q be as defined in the corollary above except that we do not require the preconditions to be complete (see the lemma above). Furthermore, let the statement W

while B **do** S **endwhile**

be given, whereby S, when applied to any data environment in the intersection of Q and B, does not execute endlessly. That is, there exists no data environment $d1$ in (Q **and** B) such that $S^*([d1])$ is an infinite sequence.

If these hypotheses are fulfilled, then the result of applying W to any data environment $d0$ in Q is not an infinite loop, that is, $W^*([d0])$ is not an infinite sequence.

Proof: We will prove this theorem by contradiction. Assume that a data environment $d0$ exists in Q such that $W^*([d0])$ is defined and is an infinite sequence. Then either

1 in the course of executing the loop, the statement S is applied to a $d1$ such that $S^*([d1])$ is an infinite sequence or
2 each $S^*([d1])$ is a finite sequence (i.e. $S(d1)$ is defined) and the condition B is always true.

(See Section 2.2.3, especially lemma 2.4.) Since Q is a loop invariant, every $d1$ to which S is applied is an element of (Q **and** B). By hypothesis, the application of S to such a $d1$ cannot result in an infinitely long sequence $S^*([d1])$. Thus, case 1 above cannot apply. From the remaining possibility it follows that for all non-negative integers n

$S^n(d0)$ is defined

and

$B(S^n(d0)) = \text{true}$

By assumption, $d0$ is in Q. Therefore, it must be in some set Z_i, that is there exists a non-negative integer i such that $d0$ is in Z_i. Since

$\{Z_i\}\ S^i\ \{\textbf{not}\ B\}$

and $S^i(d0)$ is defined, $B(S^i(d0)) = \text{false}$, in contradiction to the last sentence in the preceding paragraph. Hence, $W^*([d0])$ cannot be an infinite sequence; it must be either undefined or a finite sequence. ■

3.3.3 Application of the loop theorems

It is an old, implicitly accepted rule of thumb among programmers that every loop should be preceded by one or more statements, commonly called the 'initialization'. The loop theorem (theorem 3.8) reveals the real – and only – reason for the initialization statements preceding a loop: to establish the truth of the loop invariant, one of the postulates of the loop theorem.

To prove the correctness of a given loop, one proceeds as follows:

1 Determine the loop invariant, either by inspection or by applying the results of theorem 3.9 or the corollary in Section 3.3.2 above.
2 Prove that the initialization establishes the truth of the loop invariant.
3 Prove that the body of the loop (S in the **while** statements above) preserves the truth of the loop invariant; i.e. prove that the truth of the loop invariant and the truth of the loop condition (B above) before execution of the loop body S together imply the truth of the loop invariant after execution of S.
4 Prove that the truth of the loop invariant and the termination condition (the negation of the loop condition B) together imply the correctness of the final result.
5 Prove that the termination condition will be fulfilled in a finite number of executions of the loop body, i.e. that the loop will terminate in finite time. One must also prove that the result of each execution of the loop body is defined, i.e. that each referenced variable is present in the corresponding data environment, that the result of each evaluation of an expression is within the defined range, etc.

To design and construct a loop, the software developer proceeds as follows:

1 Define the loop invariant. This involves, explicitly or implicitly, expressing the correctness criterion in the form (I **and** E). The proposition I becomes the loop invariant. The condition E becomes the termination condition, that is (**not** E) becomes the loop condition (B in the **while** statements above). The loop invariant is a generalization of the initial and final conditions, i.e. of the precondition and the postcondition

of the loop. Expressed conversely, both the precondition and the postcondition are special cases of the loop invariant.

2 The body S of the loop is constructed so that it preserves the truth of the loop invariant, i.e. so that $\{I$ **and not** $E\}\ S\ \{I\}$.

3 The initialization A is constructed so that after it is executed, the loop invariant I is true.

4 Show that the newly constructed loop below will terminate (see step 5 of the procedure above for proving correctness).

Then the result of executing the loop

A, **while** (**not** E) **do** S **endwhile**

will be a data environment satisfying the correctness criterion.

3.4 Proof rules for the declaration statement

The declaration statement has nearly the same effect as the assignment statement (see Sections 2.1.0 and 2.1.4). The only difference is that in the case of the declaration statement, a new variable is created to which the value of the expression is assigned and, accordingly, the stack (sequence) of variables with the same name is shifted by one position. Consequently, the previous value of the variable affected is not lost, but instead becomes concealed (see Section 2.1.5). In the case of the assignment statement, on the other hand, the previous value of the variable affected is lost. The positions of concealed variables remain unchanged.

The lemma for the assignment statement applies with only one modification resulting from the creation of the new variable and the corresponding shift of concealed variables (if any) in the stack.

Lemma 3.3: Consider the declaration statement

declare ($x1$, $S1$, $E1$), ($x2$, $S2$, $E2$), ...

called D below, in which the variable names $x1$, $x2$, etc. are all different. For all data environments d in the domain of D, the following equations apply.

For accessible (not concealed) variables, we have

$$\text{valvar}(x1, D(d)) = \text{valexp}(E1, d)$$

$$\text{valvar}(x2, D(d)) = \text{valexp}(E2, d)$$

$$\ldots$$

and

$$\text{valvar}(y, D(d)) = \text{valvar}(y, d), \text{ for all other variable names } y \text{ in } d$$

For concealed (stacked) variables, the following equations apply. For all integers $n \geq 0$,

$$\text{valvar}(x1, (\textbf{release } x1)^{n+1}(D(d))) = \text{valvar}(x1, (\textbf{release } x1)^{n}(d))$$

$$\text{valvar}(x2, (\textbf{release } x2)^{n+1}(D(d))) = \text{valvar}(x2, (\textbf{release } x2)^{n}(d))$$

$$\ldots$$

and for all other variable names y in d,

$$\text{valvar}(y, (\textbf{release } y)^{n}(D(d))) = \text{valvar}(y, (\textbf{release } y)^{n}(d))$$

Proof: This lemma follows directly from the definition of the effect of executing the declaration statement given in Section 2.1.4 ■

The theorem for the assignment statement deals only with accessible (not concealed) variables and their values. Those parts of the lemmata for the declaration and assignment statements dealing with accessible variables are identical. Therefore, the theorem for the assignment statement (see theorem 3.3, Section 3.0.1) applies also to the declaration statement.

Exercise

5 The above lemma applies only to declaration statements in which all variables being newly declared have different names. Investigate how the above equations must be modified if this restriction is not met. Hint: Consider the effect of executing the declaration statement

declare (x, $S1$, $E1$), (x, $S2$, $E2$), (w, $S3$, $E3$)

upon the data environment

$$d0 = [(z, S4, v4), (x, S5, v5)]$$

6 What is the precondition of the postcondition $P1(x, z)$ under the above declaration statement? of the postcondition $P2(w, x, z)$? (In each postcondition, every variable name represents a value to be determined in the context of the resulting data environment.)

3.5 Proof rules for the release statement

The release statement eliminates the accessible variable with the stated name. Accordingly, the stack (sequence) of concealed variables with the same name is shifted by one position. The first such concealed variable (if

any) becomes accessible again. Variables with other names remain unaltered.

Reformulating this in more formal terms, we obtain the following lemma for the release statement.

Lemma 3.4: After executing the release statement

release x

called R below, the following equations apply for all data environments d in the domain of R.

For all integers $n \geq 0$,

$$\text{valvar}(x, (\textbf{release } x)^n(R(d))) = \text{valvar}(x, (\textbf{release } x)^{n+1}(d))$$

and for all other variable names y in d,

$$\text{valvar}(y, (\textbf{release } y)^n(R(d))) = \text{valvar}(y, (\textbf{release } y)^n(d))$$

Any proposition P not referring to the variable x is equally valid (or invalid, as the case may be) before and after execution of the release statement R above.

Proof: This lemma follows directly from the definition of the effect of executing the release statement given in Section 2.1.5 ■

3.6 Proof rules for the procedure call without parameters

The procedure call without parameters is semantically equivalent to the body of the procedure (see Sections 2.1.6 and 2.2.6). Therefore, the proof rules for the statement(s) constituting the body of the procedure apply.

3.7 Proof rules for the null statement

The null statement, being the identity function on $\mathbb{D}$, has only a degenerate proof rule: the precondition and postcondition are identical. If any precondition P is satisfied before execution of the null statement, P will be satisfied thereafter. Similarly, for any given postcondition P, P itself is a precondition.

3.8 Proof rules for other loop constructs

When proving programs involving loop constructs other than the **while** statement, substitute for the loop in question the equivalent **while** construct (see Section 2.1.8). Then apply the proof rules for the **while** loop (see Section 3.3).

3.9 Summary of the most important proof rules

The meaning and interpretation of each of the three types of preconditions can be summarized as follows. Given a statement S, a postcondition P,

a precondition Q of P under S, written '$\{Q\}\ S\ \{P\}$',
a strict precondition Qs of P under S, written '$\{Qs\}\ S\ \{P\}$ strictly',
a complete precondition Qc of P under S, written '$\{Qc\}\ S\ \{P\}$ completely'

and any initial data environment d, then

if d is in Q (or Qs or Qc) and if $S(d)$ is defined, then $S(d)$ is in P,
if d is in Qs, then $S(d)$ is defined and is in P and
if $S(d)$ is defined and is in P, then d is in Qc.

Of the proof rules presented in the preceding sections of this chapter, the following are of the greatest practical significance. They represent the most important 'vertebrae of the spine of software'. Every true software engineer must have a thorough knowledge of them and must be able to apply them correctly with facility and ease.

1 $\{P^x_E\}\ x := E\ \{P\}$ completely

2 $\{P^x_E\}$ **declare** $(x, S, E)\ \{P\}$ completely

3(a) $\{Q1\}\ S1\ \{P\}$ **and** $\{Q2\}\ S2\ \{P\}$
$\Rightarrow \{(Q1$ **and** $B)$ **or** $(Q2$ **and not** $B)\}$ **if** B **then** $S1$ **else** $S2$ **endif** $\{P\}$

3(b) $\{Q1\}\ S1\ \{P\}$ completely **and** $\{Q2\}\ S2\ \{P\}$ completely
$\Rightarrow \{(Q1$ **and** $B)$ **or** $(Q2$ **and not** $B)\}$
if B **then** $S1$ **else** $S2$ **endif** $\{P\}$ completely

4(a) $\{Q\}\ S1\ \{Q1\}$ **and** $\{Q1\}\ S2\ \{P\}$
$\Rightarrow \{Q\}\ (S1, S2)\ \{P\}$

4(b) $\{Q\}\ S1\ \{Q1\}$ completely **and** $\{Q1\}\ S2\ \{P\}$ completely
$\Rightarrow \{Q\}\ (S1, S2)\ \{P\}$ completely

5(a) $\{I$ **and** $B\}\ S\ \{I\}$
$\Rightarrow \{I\}$ **while** B **do** S **endwhile** $\{I$ **and not** $B\}$

5(b) $\{P$ **or** $Y(P)$ **or** $Y^2(P)$ **or** $\ldots\}$
while B **do** S **endwhile** $\{P\}$ strictly and completely
(i.e. the above precondition is the weakest strict precondition.)

5(c) $\{P$ **or** Z_1 **or** Z_2 **or** $\ldots\}$ **while** B **do** S **endwhile** $\{P\}$ completely

Chapter 4

Transfundamental programming constructs

Although to penetrate into the intimate mysteries of nature and thence to learn the true causes of phenomena is not allowed to us, nevertheless it can happen that a certain fictive hypothesis may suffice for explaining many phenomena.

– Leonhard Euler

Willst du immer weiter schweifen?
Sieh, das Gute liegt so nah.
Lerne nur das Glück ergreifen,
Denn das Glück ist immer da.

– Johann Wolfgang von Goethe

In addition to the fundamental programming constructs presented in Chapter 2, a variety of extended constructs and features exists in popular programming languages. While few, if any, of these extensions and features are really essential, many are considered to be of significant practical convenience and in any event are in common and widespread use. This chapter deals with the most important and frequently encountered groups of these extensions, typically defining them in terms of the fundamental programming constructs introduced earlier in Chapter 2.

4.0 Procedure calls with parameters

The procedure call without parameters was defined in Sections 2.1.6 and 2.2.6. Using it, values of variables are, in effect, transferred implicitly to and from the subsidiary procedure via predefined variable names contained in the commonly accessible data environment. This method of 'passing' parameters and results between the calling and called subprograms (procedures) is structurally simple and is often used in practice. The variables

accessed by both the calling and called subprogram are often called 'global' variables.

This implicit method of passing parameters and results gives rise to a minor inconvenience in practice. Often, one wishes to call a procedure many times in a program, sometimes with certain variables as parameters, sometimes with other variables as parameters. Values of the actual parameters must, therefore, sometimes be 'moved' to those variables referenced by the called procedure. For example, consider a program in which the function $2 * x + 3$ is to be evaluated for various previously calculated arguments, e.g. a, b and c, and the results are to become the values of different variables, e.g. fa, fb and fc respectively:

calling environment	*called procedure*
declare (x, ℚ, a)	**procedure** P:
call P	**declare** (y, ℚ, $2 * x + 3$)
declare (fa, ℚ, y)	**endprocedure**
release x, y	
...	
declare (x, ℚ, b)	
call P	
declare (fb, ℚ, y)	
release x, y	
...	
declare (x, ℚ, c)	
call P	
declare (fc, ℚ, y)	
release x, y	
...	

The following sequence of calls to P is easier to write and results in a more readable program:

calling environment	*called procedure*
call $P(a, fa)$	**procedure** $P(x, y)$:
...	**declare** (y, ℚ, $2 * x + 3$)
call $P(b, fb)$	**endprocedure**
...	
call $P(c, fc)$	
...	

Here the variables a, b, c, fa, fb and fc in the calling program are called 'actual parameters'. The variables x and y, which are referenced only in the called procedure, are called 'formal parameters'. Many implementations

permit any expression to be used as an actual parameter except, of course, where doing so would give rise to an implicit syntactical inconsistency (e.g. where the actual parameter receives a calculated result from the called procedure). Formal parameters are invariably variables.

Another form for invoking the above procedure is often even more convenient and descriptive of the computational effect:

```
calling environment        called procedure

  fa := P(a)               function procedure P(x):
  ...                      P := 2 * x + 3
  fb := P(b)               endprocedure
  ...
  fc := P(c)
  ...
```

Such a procedure P, which calculates a single value and which can be referenced within an expression in this way, is usually called a function procedure.

Most programming languages provide such procedure calls with parameters. The mechanisms used for passing parameters vary from one implementation to another. In fact, many programming languages provide more than one mechanism for passing parameters; the programmer must select one or another depending upon the effect he wishes to achieve. Often this design decision involves striking a compromise between logical simplicity and computational efficiency (speed and/or memory requirements). In some programming languages, this choice is expressed explicitly; in others, implicitly.

The semantic meaning of the program depends – often in subtle and intricate ways – upon the particular mechanism used for passing parameters. The effects of this dependence can and do result in confusion and can complicate the task of proving the correctness of a program. Such difficulties are particularly pronounced when the programmer is only superficially aware of the effect of the particular mechanism employed for passing parameters. While it is perfectly proper for a programmer to take advantage of such features in the system with which he is working, it is very important that he be fully aware of the precise mechanisms invoked and, in particular, of their exact effects. Availing himself of the convenience provided by these features does not relieve him of responsibility for the semantic meaning and the correctness of his programs.

Two mechanisms for passing parameters – call by value and call by name – are described in the following sections. Most mechanisms for passing parameters employed by actual systems either fit or closely resemble one

of these two models, even though different names may be used. Actual systems which provide such mechanisms for calling procedures place an important restriction on declaring and releasing variables: variables declared within the procedure must also be released therein and vice versa. This restriction will often be implicitly assumed in the sections below, in which the mechanisms of calling procedures with parameters are defined in terms of the fundamental constructs presented earlier.

4.0.0 Call by value

The basic form of the call by value views the values of the actual parameters at the point in time when the procedure is called as the only 'input' data of interest. When the procedure is called, each actual parameter is evaluated and the value assigned to the local variable representing the formal parameter. This variable is accessible only by statements within the called procedure (or procedures subsidiary to it). The value of this variable may be altered during the execution of the procedure, but such alteration does not affect the value of any variable associated with the actual parameter in the context of the calling environment.

Consider the following general example. The actual parameter a may be a constant, variable name or expression. The formal parameter f must be a variable name.

```
calling environment        called procedure

call P(a)                  procedure P(f):
                           parameter (f, Sf) value
                           S
                           endprocedure
```

In this case, the statement **call** $P(a)$ is defined to be equivalent to the following sequence:

```
declare (f, Sf, a)
S
release f
```

The variable used as the formal parameter is to be declared in this fashion even if the same variable name is used as both actual and formal parameters. In such a case, the declaration causes the variable used as the actual parameter to become concealed, thus ensuring that it cannot be altered by a reference (e.g. in an assignment statement) to the formal parameter in the procedure.

It is clear that this concept of call by value applies only to 'input' data, i.e. to data being transferred from the calling environment to the called procedure. In some systems, the call by value has been generalized somewhat to permit results to be communicated back to the calling environment. This is done by assigning the final value of the local variable representing the formal parameter to the actual parameter (which in this case must be a variable name). The assignment is performed only once, after execution of the procedure is complete. In the example below

calling environment	*called procedure*
call $P(a)$	**procedure** $P(f)$: **parameter** (f, Sf) **value result** S **endprocedure**

the statement '**call** $P(a)$' is defined to be equivalent to the following sequence

declare (f, Sf, a)
S
$a := f$
release f

if a and f are different variable names. If the actual and formal parameters have the same name, the naming conflict is resolved by renaming the formal parameter f throughout the procedure P.

4.0.1 Call by name

The call by name views the parameter being passed as a variable name or a rule (expression) for determining a value, rather than a value itself as in the case of the call by value. When the procedure is called, the formal parameter becomes associated with the variable name or expression comprising the actual parameter. The actual parameter is evaluated whenever the corresponding formal parameter is referenced in an expression being evaluated during execution. If the value of the formal parameter is changed (e.g. by the execution of an assignment statement in which its name appears in the replacement list to the left of the assignment symbol (:=)), the value of the corresponding actual parameter in the context of the calling environment is changed. In this latter case, the actual parameter must, of course, be a variable name.

The effect of calling a procedure passing parameters by name can probably be defined most simply in the following way. The procedure is considered not to be a group of statements to be executed in their original form, but

rather as a template for generating the statements to be executed. The formal parameter (a variable name) is replaced throughout the procedure by the corresponding actual parameter (a constant, variable name or expression). Where syntactically possible, the actual parameter is enclosed in parentheses in order to ensure that the resulting expressions are evaluated as intended. If such substitution would give rise to conflicts between the names of variables local to the procedure (i.e. other formal parameters or variables declared and released within the procedure) and the names of variables appearing within the actual parameter, the conflict is resolved by changing the names of the local variables involved before performing the substitution (Naur, 1962, Section 4.7.3.2). If procedure names may be passed as parameters, the preceding statements must be generalized accordingly.

In the example below

calling environment	*called procedure*
call $P(a)$	**procedure** $P(f)$: **parameter** f **name** S **endprocedure**

the statement **call** $P(a)$ is defined to be equivalent to

S'^{f}_{a}

the statement formed by replacing the formal by the actual parameter in the procedure body S as described above.

4.0.2 Recursive procedures

A recursive procedure is one which calls itself, either directly or indirectly through other procedures. The main part of a recursive procedure typically consists of three sections. Using the value of the parameter passed from the calling environment, the first section calculates the parameter to be passed to the subsidiary invocation of the procedure. The second section invokes (calls) the procedure, i.e. itself. The third section combines the value of the parameter originally passed to the currently active invocation of the procedure and the value of the result from the subsidiary call to obtain the result of the current invocation, which is passed back to the calling environment. When the procedure has more than one input or output parameter, the preceding description must be modified accordingly.

The procedure must be constructed in such a way that the parameter passed to the procedure is not altered by the action of the subsidiary invocation of the procedure. Similarly, it may be necessary to preserve the

values of variables internal to the procedure across the subsidiary call. Both requirements can be conveniently met by declaring and releasing variables accordingly, thereby 'stacking' the values to be preserved (see Sections 2.1.4 and 2.1.5).

The following example illustrates these principles. This procedure calculates the factorial of a non-negative integer. The input parameter is the value of the variable n. The value of the variable *result* after the procedure has terminated is the desired result.

```
procedure factorial:
if n = 0
then declare (result, ℤ, 1)
else declare (n, ℤ, n−1)
    call factorial
    release n
    result := n * result
endif
endprocedure
```

The declaration statement in the **else** part calculates the value for the parameter for the subsidiary call by subtracting 1 from the input parameter to the current invocation of the procedure. At the same time, the current input parameter is stacked, thereby concealing it from the subsidiary invocation of the procedure '*factorial*'. After the subsidiary invocation has terminated, the variable containing its input parameter is eliminated by the release statement. As a result of this action, the input parameter to the current invocation of the procedure becomes accessible again.

The reader should verify that both the call by value and the call by name, as defined in Sections 4.0.0 and 4.0.1 above, achieve these effects.

As mentioned in Section 4.0, some implementations require that a variable declared in one procedure also be released in that same procedure. The procedure '*factorial*' as defined above violates this restriction by declaring the variable '*result*'. If the above procedure is to be executed by such a real system, the declare statement may be replaced by the statement

```
result := 1
```

The variable *result* must then be declared before the initial call to the procedure '*factorial*' – not before the recursive call in the procedure itself.

4.0.3 Function procedures

As mentioned in Section 4.0, many real systems support function procedures. The main differences between function and ordinary procedures are (a) a function procedure is 'called' by referencing it in an expression to be evaluated instead of in a call statement and (b) the value of an (or the) output parameter is passed back to the calling environment implicitly, rather than as the value of a variable in a data environment.

In the following example, E is any expression containing references to a function procedure '*func*' and any combination of variables x, y, etc. S is a (typically compound) statement in which the result variable *func* is declared and assigned some value. The effect of executing the statement in the calling environment below

calling environment	*called procedure*
$x := E(func(a), x, y, \ldots)$	**function procedure** *func*(*f*): **parameter** *f* ... *S* **endprocedure**

is, in most real systems, typically defined to be something akin to the effect of executing the following:

```
call func(a)
x := E(func, x, y, ...)
release func
```

Subject to the restriction stated above, S may be any statement, in particular, a sequence of statements which alter the values of various variables. As a consequence of such 'side effects', the definition of the assignment statement (Section 2.1.0) and, therefore, also the lemma and theorem for the assignment statement (Sections 3.0.0 and 3.0.1 respectively) do *not* in general apply to the 'assignment' statement in the calling environment above. They do, of course, apply to the assignment statement in the equivalent sequence of statements.

If the expression contains references to two or more function procedures, an additional source of difficulty and possible confusion arises from the fact that the effect produced can depend upon the sequence in which the function procedures are called. In such systems, the sequence of execution of the several function procedures must, therefore, be defined precisely if the result is to be uniquely determined.

An alternative definition of the effect of executing such a statement containing a reference to a function procedure, which eliminates the possibility of side effects, is as follows. The value of the function is obtained by executing the function procedure upon the original data environment. All other variables appearing in the expression are evaluated in the *original* data environment – not in the data environment resulting from the execution

of the function procedure. In this way, all parts of the expression are evaluated in the original data environment. More formally, if '*func*' is the name of a function procedure, then the effect of executing the statement

$$x := E(func(a), x, y, \ldots)$$

upon the data environment $d0$ is defined to be

$$d1 = (x := E(func(a), x, y, \ldots))\ (d0)$$
$$= (x := E(\text{valvar}(func, (\textbf{call}\ func(a))(d0)), x, y, \ldots))(d0)$$

That is, the equivalent statement is obtained by substituting

$$\text{valvar}(func, (\textbf{call}\ func(a))(d0))$$

for '*func*(*a*)' in the original statement.

When this definition is generalized for more than one reference to function procedures, the result is independent of the order in which the function procedures are executed, because each function procedure is applied to the original data environment.

Except where explicitly stated otherwise, this latter definition will be used throughout this book.

4.0.4 Programming conventions and style

When calling a procedure (in the form without explicit parameters), each variable serving as a parameter may be declared in either the calling environment (i.e. outside of the current invocation of the called procedure) or within the called procedure. Similarly, each such variable may be released in either environment. Thus, the programmer must select from among four conventions for declaring and releasing variables:

	Environment in which		Comments
Convention	declared	released	
1	calling	calling	input and output
2	calling	called	input only
3	called	calling	output only
4	called	called	variables local to procedure only

It is clear that in the case of an input variable, i.e. a parameter being passed from the calling environment to the called procedure, its value must

be assigned before the call statement is executed. It follows that the variable must be declared before the call statement, i.e. in the calling environment. An input variable may be released either within the called procedure or in the calling environment after the call statement.

The value of an output (result) variable must be available in the calling environment after the call statement. It must, therefore, be released after the procedure has terminated and, therefore, in the calling environment after the call statement. It may be declared either in the calling environment prior to the call statement or in the called procedure.

When convention 1 is followed, the variable in question is declared in the calling environment before the call statement is executed and is released in the calling environment after the call statement. This permits the value of an input parameter to the procedure to be retained for subsequent use in the calling environment (e.g. for passing to another procedure called later). In the case of an output parameter, the variable in question is prematurely declared and set initially to a 'dummy' – i.e. a meaningless and superfluous – value. If the system on which the program is to be run does not permit convention 3 (the only other method for passing an output parameter) to be used, then the programmer has no alternative but to use convention 1 despite this inelegant aspect of its application, which tends to make a program less readable and more difficult to understand.

Convention 1 must be used for passing a parameter which is to be 'updated', that is, whose initial value may affect the result produced by the procedure and whose final value is such a result. Such a parameter is both an input and an output variable.

When convention 2 is employed, the variable is declared in the calling environment before the call statement and is released within the called procedure. Consequently, the value of the input variable is not available in the calling environment after the call statement. This convention is the natural and logical choice for an input parameter whose value is not required subsequently in the calling environment. It has the disadvantage that a procedure based on this convention is not suitable for later use in a new environment in which the value of the input variable is subsequently required.

With convention 3, the variable in question is declared in the called procedure and is released in the calling environment after the call statement. In the case of a pure output variable, i.e. a variable whose initial value (if any) has no effect upon the action or result of the called procedure, this convention is the natural and obvious choice.

Applying convention 4, a variable is both declared and released in the called procedure. Being unavailable in the calling environment, it may serve neither as an input nor as an output parameter. It can be only an internal

variable, local to the procedure. Convention 4 should be used resolutely and consistently to isolate a procedure's internal variables from the outer environment. Such a programming style improves the readability and understandability of programs because the reader can clearly see that certain types of interactions cannot occur. For the same reason, it tends to reduce the effort required to design and verify the correctness of the interfaces between procedures and their calling environments.

The following example shows all four conventions applied to a single procedure. The variable $v1$ is declared and released according to convention 1; $v2$, convention 2; $v3$, convention 3 and $v4$, convention 4. Variable $v1$ can be used as input and/or output. Variable $v2$ is an input variable only; $v3$, output only. Variable $v4$ is a local variable, internal to the procedure P.

calling environment	*called procedure*
declare ($v1$, $S1$, ...)	**procedure** P:
declare ($v2$, $S2$, ...)	**declare** ($v4$, $S4$, ...)
call P	...
...	**declare** ($v3$, $S3$,...)
release $v3$, $v1$	**release** $v4$, $v2$
	endprocedure

As mentioned in Section 4.0, many actual programming languages and systems require that a variable be both declared and released in the same environment. This restriction effectively prohibits the explicit use of conventions 2 and 3. Therefore, in such systems convention 1 is used for passing all parameters and convention 4 is used for the procedure's local (internal) variables. Convention 3 is used implicitly and only for passing the formal result of a function procedure to the calling environment, but this is not normally visible to or under the control of the programmer. Convention 2 is not common in practice.

Dogmatic principles aside, it does not matter greatly which conventions a programmer uses, but he should have a sound, logical reason for his choice in every single instance. Logical simplicity, clarity, consistency and, of course, correctness are the most important principles upon which programming style should be based, both in general and with respect to passing parameters to and from called procedures.

4.1 Input/output

In most actual computing systems the programmer must use quite different constructs to refer to program variables stored in the computer's central

memory and those stored in so-called peripheral or external storage devices. The reasons for this requirement are historical. While those reasons probably cannot be rationally justified today, this distinction has become so deeply ingrained in programming languages and tradition that it is now accepted unquestioningly as a basic tenet of computing.

At least one implementation of the Basic programming language permitted the programmer to specify that an array should be stored in a peripheral storage file, but otherwise the array variables in question were referenced in exactly the same manner as any other 'internal' variable. Although this convention simplifies the programming task and the program, sometimes to a very considerable extent, this approach has not become generally accepted. Despite its advantages, there are currently no signs that it will gain in popularity in the foreseeable future.

Especially when proving programs correct, it is convenient to consider variables stored in peripheral files in the same manner as internal variables. One can then apply the proof rules developed in Chapter 3 for the fundamental constructs; one need not derive a large collection of special proof rules applicable only to the input/output constructs. Such a simplified and simplifying approach to the problem of proving the correctness of programs containing input/output statements will be taken in this book.

Although the basic concept applies to all 'external' variables, it is useful to discuss certain types of them separately. Auxiliary functions specific to each type of external variable are common and are often built into the corresponding input/output statement.

4.1.0 Backing storage

Typical backing storage devices are magnetic disks, diskettes and tapes. These devices are used because they are less expensive, usually much less expensive, than central memory devices. The disadvantage of backing storage is that it is slower than central memory. Furthermore, the time to locate and transfer data to and from backing storage may vary depending upon which data was accessed in the previously executed operation.

Data stored on backing storage devices are typically processed by programs in essentially the same manner as data stored in central memory. Values of variables stored in backing storage are used in computations and the results of computations become the new values of such variables. Probably the only logical distinction between data stored in backing storage and data stored in the computer's central memory is that the former are often retained between executions of a program or programs, whereas the latter are usually discarded when the program which created them terminates.

Data stored in backing storage are usually organized in files. A number of different types of files are found in actual computing systems. The most commonly encountered are the sequential file, the direct access file (sometimes called 'random access file') and the indexed file. Most other types, including many structures implemented in data base systems, are variants (often only minor variants) of one of these three types for which special processing functions (procedures) are provided for use by the programmer.

When the programmer wants to use the value of a variable stored in a file in a computation, he must typically 'read' the desired value and transfer it to an internal variable. Depending upon the file type, he must in one way or another indicate to the system's reading procedure which data element in the file is desired. He must also indicate to which internal variable the value in question is to be transferred. Such operations are equivalent to simple references to the corresponding array variables in an assignment statement as discussed in earlier sections of this book.

When the programmer wants to transfer the result of a computation to a variable stored in a file, he must write statements similar to those outlined in the above paragraph, the main difference being that the computed value is 'written' to the variable stored in the file. The 'write' operation corresponds to an ordinary assignment statement, where the variable to which a value is being assigned is the array variable corresponding to the data element in question in the file.

In addition, the programmer using such a file oriented system must normally be concerned with other technical details such as 'opening' and 'closing' the file, indicating via which 'channel number' the file is to be accessed and data transferred, etc. When the programmer views files as collections of ordinary arrays, such concerns do not arise.

Any of the three common types of files mentioned above and discussed in more detail in the following sections can be viewed as an appropriate collection of arrays as defined in Section 2.0.0. Doing so usually simplifies greatly the proof of correctness of program segments involving variables stored in backing storage. This approach is, therefore, generally to be recommended.

4.1.0.0 Sequential files

A sequential file is typically a sequence of strings, i.e. a sequential file is an array. Each array variable assumes a value which is a string, i.e. a sequence of characters, often thought of as a 'line'. In some systems all strings in a sequential file must be of the same length. In other systems, the lengths of such strings may vary but a maximum length is imposed. In

still other systems, no restriction on the lengths applies, other than, of course, that set by the physical size of the system's memory.

Example 1: Consider a sequential file named '*sfile*' which is read and processed by a program segment with the following structure. It is assumed that the variable '*channel*' has been declared to be an integer variable and x, a string variable.

```
open channel, "sfile"
while not eof(channel) do
    read channel, x
    (process x)
endwhile
close channel
```

The file can be thought of as an array named '*sfile*' together with a variable '*sfilelength*' which indicates the length of the file. More specifically, the sequential file '*sfile*' is considered to consist of the non-negative integer variable

sfilelength

and the string variables

$sfile(i),\ i = 1, 2, \ldots, sfilelength$

The above program segment is then equivalent to the following:

```
i := 0
while i < sfilelength do
    i := i + 1
    x := sfile(i)
    (process x)
endwhile
```

In this program segment the variable x is logically superfluous and can be eliminated to yield

```
i := 0
while i < sfilelength do
    i := i + 1
    (process sfile(i))
endwhile
```

This program captures the logic of the computational process inherent in the original program above. Only technicalities specific to accessing data values stored in a peripheral file have been suppressed – technicalities which

should be considered in a completely different stage of the design process and which should not be intermixed with considerations related to the logical structure of the algorithm. Rewriting the program segment in this way separates technical, system oriented considerations from the logical aspects of the application and the requirements it poses. Separating these quite different aspects of the design problem simplifies the designer's task – both in the creative phase of synthesizing the system's structure and in the more systematic, mechanistic phase of analyzing and verifying the proposed design.

Example 2: Consider the following program segment, which creates a sequential file as described in example 1 above.

```
open channel, "sfile"
while (processing incomplete) do
    x := ...
    write channel, x
endwhile
close channel
```

The above program segment is equivalent to:

```
i := 0
while (processing incomplete) do
    x := ...
    i := i + 1
    declare (sfile(i), set of strings, x)
endwhile
sfilelength := i
```

As in example 1 above, the variable x is superfluous here and can be eliminated to yield

```
i := 0
while (processing incomplete) do
    i := i + 1
    declare (sfile(i), set of strings, ...)
endwhile
sfilelength := i
```

If the array variables *sfile*(i) are already declared, the declare statements above should be replaced by the corresponding assignment statement

```
sfile(i) := ...
```

In this case, the file is being rewritten, not created anew.

4.1.0.1 Direct access files

A direct access file is usually defined as a sequence of records. Each record is identified by a number. Normally, the integers beginning with 0 or 1 are used as record numbers. Each record contains the values of several variables. For example, a record might consist of a name field (a string of 40 characters), a five digit integer (indicating the named person's employee number) and a four digit integer (the year of his birth). Such a file may be defined mathematically as a collection of three arrays: name, employee number and year of birth. The number of the record in the file is the value of the subscript of each array variable; i.e. any direct access file can be viewed as a collection of arrays with subscripts in a common set.

Example 3: To read record number *recn* from an already opened direct access file, statements such as the following must typically be executed.

```
seek channel, recn
read channel, x, y, z
(process x, y and z)
```

where x, y and z are internal variables corresponding to the data fields in each record in the file.

As outlined before, the file is, in effect, a collection of arrays. If the file name is '*dfile*', the arrays might be named '*dfile.x*', '*dfile.y*' and '*dfile.z*'. Then the above program segment is equivalent to

```
x := dfile.x(recn)
y := dfile.y(recn)
z := dfile.z(recn)
(process x, y and z)
```

In a program segment, these assignment statements will usually be superficial. As in the examples for the sequential file, the variables, x, y and z can be eliminated by replacing them by *dfile.x*(*recn*), *dfile.y*(*recn*) and *dfile.z*(*recn*) respectively in the succeeding statements in the program segment in question to yield simply

```
(process dfile.x(recn), dfile.y(recn) and dfile.z(recn))
```

Example 4: To write new values into the direct access file described in example 3, statements such as the following must be executed.

```
x := ...
y := ...
z := ...
seek channel, recn
write channel, x, y, z
```

If a new record is being created, the above statements are equivalent to the following:

declare (*dfile.x*(*recn*), . . .)
declare (*dfile.y*(*recn*), . . .)
declare (*dfile.z*(*recn*), . . .)

If an existing record is being rewritten with new values, the first program segment above is equivalent to the following assignment statements:

dfile.x(*recn*) := . . .
dfile.y(*recn*) := . . .
dfile.z(*recn*) := . . .

4.1.0.2 Indexed files

An indexed file, as typically implemented, consists of a direct access file and one or more indices. An index is a table of record numbers (subscript values) which serves logically to order ('sort' in commercial data processing terminology) the main file. The index is, in effect, also an array, each variable of which assumes a value which is a subscript of the arrays constituting the main file. Mathematically speaking, the index is a permutation of those arrays. In some real systems the index also contains the values of the variables of one of the arrays, but such information is redundant and can be omitted. If it is present, searches can often be performed faster.

An indexed file, then, is a collection of arrays related structurally to one another in a specific way.

Example 5: Consider an indexed file in which each record contains a name, employee number and address. The name and the employee number are specified as key fields, that is, the programmer may access a record by specifying either the name or the employee number. The system will maintain automatically an index for each of these fields.

We will view this indexed file as a collection of the following variable and arrays. The value of the variable *ifilelength* is a non-negative integer which indicates how many records are present in the file. The array variables comprising the file are as follows:

name(*i*),
empnumber(*i*),
address(*i*),
index.name(*i*) and
index.empnumber(*i*),

where $i = 1, 2, \ldots,$ *ifilelength*. The sequences

[*index.name*(1), *index.name*(2), . . . , *index.name*(*ifilelength*)]

and

[*index.empnumber*(1), *index.empnumber*(2), . . . , *index.empnumber*(*ifilelength*)]

are permutations of the integers 1 through *ifilelength* inclusive. These two indexes order, or sort, the file into two sequences, one ascending with *name*, the other ascending with *empnumber*. More precisely, for every pair of integers m and n such that

$$1 \leq m < n \leq \textit{ifilelength}$$

it is true that

name(*index.name*(*m*)) ≤ *name*(*index.name*(*n*)) **and**
empnumber(*index.empnumber*(*m*)) ≤ *empnumber*(*index.empnumber*(*n*))

The program segment

read *channel*, **key**(*name*) = *kname*, *rname*, *rempnumber*, *raddr*
if (record found) **then** (process *rname*, *rempnumber*, *raddr*) **endif**

is equivalent to

call *locatename*
if *recn* > 0
then *rname* := *name*(*recn*)
rempnumber := *empnumber*(*recn*)
raddr := *address*(*recn*)
(process *rname*, *rempnumber*, *raddr*)
endif

which can be simplified to

call *locatename*
if *recn* > 0
then (process *name*(*recn*), *empnumber*(*recn*), *address*(*recn*))
endif

The procedure *locatename* is a system routine which assigns a value to the variable *recn* as follows. The value of the variable *kname* (an input variable to the procedure) is sought among the values of *name*(·). If this value exists in the file, i.e. if there exists an i with *name*(i) = *kname*, then the value of *recn* is the record number (subscript value) of the first such entry in the file. Otherwise, i.e. if the name being sought is not present in the file, the

resulting value of *recn* is 0. Expressed in more mathematical language, the postcondition of the procedure *locatename* is

[*recn* = *index.name*(min{*i* | *name*(*index.name*(*i*)) = *kname*})]

or [(*recn* = 0) **and** (there exists no *i* such that *name*(*i*) = *kname*)]

Similarly, a procedure *locateempnumber* would also exist which would perform the analogous function for the data field employee number. Such an indexed file system would also include procedures for writing a new record to the file, rewriting over an existing record, deleting a record, etc. In particular, such procedures would maintain the indexes so that the corresponding conditions specified above are maintained. Such conditions are invariants and represent the specifications which the file management system must meet.

4.1.1 Display output

Display output can be thought of as a sequence of strings, i.e. an array. Structurally, it is much like a sequential file (see Section 4.1.0.0).

If the display in question has a particular structure, this can be reflected in the structure of the equivalent array, if desired. For example, a printed report might consist of a sequence of pages, each page consisting of a sequence of lines (strings). The length of each line and the length of each page are limited but, in general, no limit is imposed on the number of pages in a report. The report, then, would be equivalent to a two dimensional array, one dimension (subscript) indicating the page in the report and the other dimension, the line on a page.

A comparable structure could be used for output to a video display unit. Alternatively, the display output could be viewed more simply as a sequence of lines, corresponding to a one dimensional array.

Example 6: A report is printed by executing a statement of the following form

print *channel*, *x*, *y*, *z*

appropriately.

If one views the printed report as the family of array variables

reportline(*pagenumber*, *linenumber*)

where

$1 \leq pagenumber$

and

$1 \leq linenumber \leq pagelength$

the **print** statement above is equivalent to the program segment

```
call incrementlinenumber
declare (reportline(pagenumber, linenumber),
    strings of maximum length linelength,
    x & y & z)
```

where the procedure *incrementlinenumber* is as follows.

```
procedure incrementlinenumber:
linenumber := linenumber + 1
if linenumber > pagelength
then pagenumber := pagenumber + 1
    linenumber := 1
endif
endprocedure
```

In the above program segment, '&' is the concatenation operator.

At first glance, this may appear more complex than the single **print** statement above. However, the explicit presence of variables for the page and line numbers has an advantage: it enables one to state, and to prove more easily, conditions on the positions of variables on the printed page. This, in turn, facilitates proving formally that the desired format of the report will actually be maintained under all conditions. It also eliminates the need to establish and apply proof rules for another construct, in this case, the **print** statement. Furthermore, if the program provides for additional format features such as a header (title block) and/or a footer, variables for the page and line numbers must normally appear explicitly anyway.

4.1.2 Input from the external environment

Input from the external environment, e.g. keyboard input, can be viewed in a similar way as a sequential file. In most applications, it is probably most convenient to view keyboard input as a sequence of strings, each string being terminated by the character corresponding to the carriage return key (on many keyboards this key is marked 'enter', 'transmit' or 'end of line', etc.) The statement

input *x*

is then equivalent to

inputline := *inputline* + 1
x := *keyboardinput*(*inputline*)

where the array *keyboardinput* is deemed to exist and contain the data entered (and, possibly, still to be entered in the 'future') via the keyboard. As in previous examples, the variable *x* in the above program segment will often be superfluous and can be eliminated by substituting subsequent references to it by *keyboardinput*(*inputline*).

In some applications, it may be desirable to view keyboard input as a sequence of individual characters. In input procedures in the system software will this approach be especially appropriate. One important routine in typical computer systems, in fact, converts between these two views; its input is the direct keyboard in the form of a string of individual characters and its output is a sequence of strings, each of which is terminated by a particular end of line character.

4.1.3 Interactive communication

Interactive communication, such as a man–machine dialog conducted via a keyboard and a video display, a two-way communication line, two-way pipelines between computational tasks, etc., can generally be modelled as a combination of two single direction input/output channels or devices as described above. While the interaction between the two one-way channels may be meaningful to the human sitting at a terminal consisting of a keyboard and a video display unit, to the machine, the program or the computational task, such a logical interaction is irrelevant and not meaningful. As far as the mechanistic execution of the program segment is concerned, the two-way interactive communication channel might just as well be independent data streams.

System software, for example, rarely, if ever, processes the data flowing in the two directions jointly. When, as in the case of some communication devices, hardware considerations dictate that the data flow in the two directions be coordinated in some way, the system software designer's task often involves logically separating these two communication channels.

4.2 Data structures

A number of programming languages provide for naming a structured group of variables. The programmer can subsequently refer in expressions to the entire group as a single entity. Such features can sometimes reduce considerably the clerical effort involved in writing programs and can contribute to their readability. Handy as they are, these facilities amount,

in the final analysis, to a type of shorthand; they add no fundamental, new capability to a language.

When such compound variables are encountered in a program to be proved correct, they can be replaced by references to the individual component variables of which they are composed. Typically, an assignment statement must be replaced by a multiple assignment statement or broken down into a series of assignment statements.

Example 7: Consider the variable *gr*, defined to consist of a structured group of variables as follows:

gr
  gra
    *gra*1
    *gra*2
  grb
    *grb*1
    *grb*2
      *grb*2a
      *grb*2b
  grc

That is, the compound variable *gr* is the triple of variables *gra*, *grb* and *grc*. The compound variable *gra* is, in turn, the pair of variables *gra*1 and *gra*2, which are not subdivided further. The compound variable *grb* is also a pair of variables (*grb*1 and *grb*2), one of which is not subdivided further and one of which is a compound variable, consisting of *grb*2a and *grb*2b. The variable *grc* is an elemental (not compound) variable.

Tracing the hierarchical structure of the variable *gr* downward to its end points, i.e. to the variables which are not themselves compound variables, we see that *gr* consists ultimately of the elemental variables *gra*1, *gra*2, *grb*1, *grb*2a, *grb*2b and *grc*. A reference to the variable *gr* is, in reality, a reference to the *n*-tuple of variables (*gra*1, *gra*2, *grb*1, *grb*2a, *grb*2b, *grc*) and may be replaced accordingly. The assignment statement

gr := ...

is, therefore, equivalent to the multiple assignment statement

(*gra*1, *gra*2, *grb*1, *grb*2a, *grb*2b, *grc*) := ...

4.3 Non-sequentially executed program statements

When designing programs to be executed on certain actual computing systems, it is often useful to identify parts of programs which can be

executed 'concurrently' or 'in parallel'. As a result, the efficiency (usually speed) of execution can often be improved. In order to model program structures which cater for the concurrent execution of parts of programs, the concepts introduced earlier must be extended. When doing so, it is important to remember that it is not so much the temporal relationship between the execution of the program segments involved which is of interest, but rather the functional relationship between them. That is, the question of interest is not so much 'when is which program segment executed?' but instead 'which data environments are derived from which other data environments by the application of which program segments?'

Much theoretical work has been done – and is currently still in progress – in the area of non-sequential processes. Much of this work deals with synchronizing concurrently executing processes at selected points. Typically, such processes operate on the single data environment, that is, they modify the values of variables in one data environment. They share data not only in the form of a common initial data environment, but also continually while they are executing. Frequently, the goal of such synchronization is to ensure that the results obtained are identical to those which would be obtained by executing the program segments in some (often an arbitrary) sequence. Viewed in this way, such an executional process is not really characterized by true concurrence, but rather is fundamentally a sequential process characterized by some degree of arbitrariness in the order in which the various parts of the program may be executed. Restricting this arbitrariness in such a way that the desired results are obtained is often the subject of concern.

In Sections 4.3.0 through 4.3.2 below, we distinguish between the above mentioned two aspects of this subject: (a) situations in which the order of execution of program segments is of no consequence and (b) concurrently executing programs and the collections of data environments generated thereby.

In the following sections, some extensions of concepts presented earlier are outlined briefly. A more thorough treatment of the subject of non-sequential processes is beyond the scope of this book. For more information, the reader is referred to the professional literature (for example, Dijkstra, 1968, appendix on semaphores; Herzog, 1984; Dal Cin, 1984; Filman, 1984; Gries, 1978, Part III; Hoare, 1985; Milner, 1986; the references contained therein, more recently published papers and manuals on programming languages supporting concurrent execution).

4.3.0 Sequential interchangeability

Often statements appearing in sequence in a program may be interchanged without altering the results they produce when executed. That is, it often occurs that statements $S1$ and $S2$, for example, have the property that

$$(S1, S2)(d) = (S2, S1)(d), \text{ for all } d \text{ in } \mathbb{D}$$

Similarly, a collection of more than two statements can exhibit the property that all permutations, when executed, give rise to the same data environment. This motivates the following definition.

Definition 4.0: A collection of statements

$$S_1, S_2, \ldots, S_n$$

is *sequentially interchangeable* if for every permutation p of the integers 1 through n inclusive and for every d in $\mathbb{D}$

$$(S_{p(1)}, S_{p(2)}, \ldots S_{p(n)})(d) = (S_1, S_2, \ldots, S_n)(d)$$

Equality here is meant in the usual sense that if either term is defined, they are both defined and equal; if either term is undefined, both are undefined.

It should be noted that the sequences of data environments generated by executing the various permutations of sequentially interchangeable statements are not, in general, equal. That is, it is *not* generally true that for every permutation p and every d^* in $\mathbb{D}^*$.

$$(S_{p(1)}, S_{p(2)}, \ldots S_{p(n)})^*(d^*) = (S_1, S_2, \ldots, S_n)^*(d^*)$$

even when the statements $S_1, S_2, \ldots, S_n$ are sequentially interchangeable.

The following theorem often reduces the effort required to show that a given collection of statements is sequentially interchangeable.

Theorem 4.0: If every pair of statements selected from the collection of statements

$$S_1, S_2, \ldots, S_n$$

is sequentially interchangeable, then the entire collection is also sequentially interchangeable.

Proof: (sketch) Any permutation of the collection of statements can be formed from any other permutation by a sequence of steps, each of which interchanges two adjacent statements. Each step results in a new sequence of statements which yields the same result as the previous sequence when executed. ■

The converse of this theorem is not always true; i.e. the condition in this theorem (the statements are pairwise interchangeable) is a sufficient, but

not a necessary condition. Degenerate situations exist in which a certain pair of statements is not sequentially interchangeable but the collection is. For example, the pair of statements

$x := x * y$
$y := 0$

is clearly not sequentially interchangeable, but the following collection is:

$x := x * y$
$y := 0$
$x := 0$

Two assignment statements in which different variables receive new values are sequentially interchangeable when the variable referenced on the left side of one statement is not referenced (explicitly or implicitly) in the expression on the right side of the other statement. For example, the two statements

$x := E1$
$y := E2$

are sequentially interchangeable if x does not appear in the expression $E2$ and y does not appear in $E1$. Otherwise, e.g. if x appears in $E2$ or if y appears in $E1$, the statements are sequentially interchangeable only in special cases which are not of general interest.

The variables on the left sides of the statements may, of course, be array variables. In this case, an additional, related restriction applies: x may not appear in the subscript expression for y and vice versa. That is, the result of executing each statement should be independent of the value of the other variable being modified.

Generalizing to situations involving more than two assignment statements, a collection of assignment statements is sequentially interchangeable if the variables whose values are being modified are all different and if the value of a variable being modified in any one statement has no influence on the value of any expression appearing in any other statement.

It should be noted that these conditions for sequential interchangeability are sufficient, but not necessary conditions. Typically, however, if they are not met, the statements in question are sequentially interchangeable only in special, often degenerate situations, if at all.

The effects of declaration statements are similar to those of assignment statements. Therefore, a collection of declaration statements or of assignment and declaration statements intermixed is sequentially interchangeable under corresponding conditions.

In practice, situations arise in which a certain condition is fulfilled prior

to the execution of a given sequence of statements. Furthermore, that condition is a precondition of the specified postcondition with respect to every permutation of the given statements. It is clear that, in such a situation, the statements may be executed in any sequence, even if they are not sequentially interchangeable. In such a case, the different results obtained will all satisfy the postcondition, i.e. will be correct. The statements are, in a certain restricted sense, interchangeable with respect to the specific precondition and postcondition. This motivates the following two definitions.

Definition 4.1: A collection of statements

$S_1, S_2, \ldots, S_n$

is *sequentially interchangeable with respect to the condition* Q (a subset of $\mathbb{D}$), if for every permutation p of the integers 1 through n inclusive and for every d in Q

$$(S_{p(1)}, S_{p(2)}, \ldots, S_{p(n)})(d) = (S_1, S_2, \ldots, S_n)(d)$$

Equality here is meant in the same sense as in definition 4.0 above.

Definition 4.2: A collection of statements

$S_1, S_2, \ldots, S_n$

is *sequentially interchangeable with respect to the precondition* Q *and the postcondition* P (both subsets of $\mathbb{D}$), if for each d in Q either of the following two conditions is met:

1 For every permutation p of the integers 1 through n inclusive

$(S_{p(1)}, S_{p(2)}, \ldots, S_{p(n)})(d)$ is defined and in P.

2 For every permutation p of the integers 1 through n inclusive

$(S_{p(1)}, S_{p(2)}, \ldots, S_{p(n)})(d)$ is undefined.

Note that this definition is more restrictive than merely requiring that Q be a precondition of P under the sequence

$(S_{p(1)}, S_{p(2)}, \ldots, S_{p(n)})$

for every permutation p. For example, a data environment d may exist such that

$(S_{p(1)}, S_{p(2)}, \ldots, S_{p(n)})(d)$

is defined and in P for some permutations p and is undefined for all others.

Such a d may be in a precondition of P under the above sequence of statements for every permutation p, but it is excluded as an element of a precondition Q by definition 4.2.

It is true, however, that a Q which satisfies definition 4.2 above is a precondition of P under the sequence

$$(S_{p(1)}, S_{p(2)}, \ldots, S_{p(n)})$$

for every permutation p. This fact follows directly from definition 4.2 and the definition of a precondition (see Chapter 3).

In the following section, situations involving statements which are sequentially interchangeable in the senses of definitions 4.1 and 4.2 are examined from a somewhat different standpoint.

4.3.1 Pseudoconcurrent execution of program statements

As mentioned in Sections 4.3 and 4.3.0, it is, in some situations, permissible for a collection of statements to be executed in any sequence, even if the results obtained may vary depending upon the actual order of execution. Such a collection of statements represents an obvious candidate for 'parallel' or 'concurrent' execution. This motivates the following definition, whereby we will use the term 'pseudoconcurrent' in order to distinguish between such a situation and the quite different structure for non-sequential execution outlined in Section 4.3.2 below.

Definition 4.3: The result of executing the statements

$$S_1, S_2, \ldots, S_n$$

pseudoconcurrently upon a data environment d is defined to be an arbitrarily selected member of the set

$$|[S_1, S_2, \ldots, S_n]|\ (d) = \{(S_{p(1)}, S_{p(2)}, \ldots, S_{p(n)})(d) \mid p \text{ is a permutation of the integers 1 through } n \text{ inclusive}\}$$

provided that

$$(S_{p(1)}, S_{p(2)}, \ldots, S_{p(n)})(d)$$

is defined for every permutation p.

By 'an arbitrarily selected member' above we mean that the selection is not under the control or influence of the programmer. The particular member (resulting data environment) is selected by the 'system' in an undefined way, e.g. randomly, and can vary from one pseudoconcurrent execution of these statements to another in any manner. All that can be

said about the data environment $d1$ resulting from executing the above statements pseudoconcurrently upon the data environment $d0$ is that $d1$ is a member of the above set, i.e. that there exists a permutation p such that

$$d1 = (S_{p(1)}, S_{p(2)}, \ldots, S_{p(n)})(d0)$$

The operational notion underlying this definition is that a permutation of the given statements is selected at random; that sequence of statements is then executed. Murphy's law implies that if the effect of executing any one of the possible permutations is undefined, that permutation will be selected. This suggests that if the effect of executing any permutation is undefined, then the effect of pseudoconcurrently executing the collection of statements should also be undefined, hence the requirement that the effect of every permutation of the statements be defined.

This requirement implies that the domain of the pseudoconcurrent construct

$$|[S_1, S_2, \ldots, S_n]|$$

is the intersection of the domains of the sequences

$$(S_{p(1)}, S_{p(2)}, \ldots, S_{p(n)})$$

over all permutations p.

Note also that the function corresponding to the pseudoconcurrent construct

$$|[S_1, S_2, \ldots S_n]|$$

does not, as the other programming constructs introduced before, map a data environment into a data environment, but rather it maps a data environment into a set of data environments, i.e. into a subset of $\mathbb{D}$.

It is clear that the data environment resulting from executing a collection of statements pseudoconcurrently is uniquely determined if the collection is sequentially interchangeable, because then the set of possible results contains only one member. If the statements in the pseudoconcurrent construct are not sequentially interchangeable, the set of possible results will, at least for some initial data environment $d0$, contain more than one member. In this case, the result of executing the collection of statements pseudoconcurrently will not be uniquely determined.

A classical example of the 'concurrent' execution of programs which requires synchronization so that execution is pseudoconcurrent in the above sense is the airline seat reservation problem (Baber, 1982, p. 169 ff.) The functions 'reserve' (also called 'lock') and 'release' (or 'unlock') serve to coordinate different, concurrently executing tasks (program executions).

Without such synchronization, reservations can be effectively lost or overbooking can occur, even if the program is written to prevent it. When synchronized properly, some sections of the 'concurrent' tasks may execute temporally independently, but certain sections are restricted to strictly sequential execution. That is, the critical section of one task is executed completely, then the critical section of another task is executed completely; the execution of these sections of the programs may not be overlapped or interleaved. The final result of the 'concurrent' execution of the several tasks is the same as that which would have been produced by executing the tasks strictly sequentially, that is, one after the other in some order.

A collection of statements may be executed in any order, that is pseudoconcurrently, if the required postcondition is fulfilled regardless of the order of execution. Whether the result of such a pseudoconcurrent execution of the statements is uniquely determined or not is of no consequence in such a situation. The fact that such situations arise in programming practice leads to the need to determine a precondition with respect to a collection of pseudoconcurrently executed statements. The following theorems provide a way to derive such a precondition.

Theorem 4.1: If, for each permutation p of the integers 1 through n inclusive, Q_p is a precondition of the postcondition P under the sequence of statements

$$(S_{p(1)}, S_{p(2)}, \ldots S_{p(n)})$$

then

$$Q = \mathbf{and}_p\ Q_p$$

is a precondition of P under the pseudoconcurrent construct

$$|[S_1, S_2, \ldots, S_n]|$$

Symbolically,

$$\{Q_p\}\ (S_{p(1)}, S_{p(2)}, \ldots, S_{p(n)})\ \{P\} \text{ for every permutation } p$$
$$\Rightarrow \{\mathbf{and}_p\ Q_p\}\ |[S_1, S_2, \ldots, S_n]|\ \{P\}$$

Proof: Let $d0$ be an element of the domain of the pseudoconcurrent construct

$$|[S_1, S_2, \ldots, S_n]|$$

and of Q. For any element $d1$ of the set

$$|[S_1, S_2, \ldots, S_n]|\ (d0)$$

there exists a permutation p such that

$$d1 = (S_{p(1)}, S_{p(2)}, \ldots, S_{p(n)})(d0)$$

Q is a subset of Q_p. It follows that $d0$ is an element of Q_p, a precondition of P under the above sequence of statements. Therefore, $d1$ is in P. Thus, Q fulfills the definition of a precondition of P under the given pseudoconcurrent construct. ■

The above proof assumes a natural generalization of the definition of a precondition for the case of a pseudoconcurrent programming construct. Specifically, we require that all possible results of pseudoconcurrently executing the statements in question satisfy the specified postcondition. If some, but not all, possible results of executing a pseudoconcurrent programming construct upon a data environment $d0$ are in the specified postcondition, then no set which includes $d0$ can be a precondition. In other words, we require that the set of possible results be a subset of the specified postcondition.

Theorem 4.2: If all the Q_p in theorem 4.1 are complete preconditions, then also Q is a complete precondition.

Proof: The fact that Q is a precondition follows directly from theorem 4.1. We will show here that the precondition Q is complete.

Consider any d in the domain of the pseudoconcurrent construct such that

$$|[S_1, S_2, \ldots, S_n]|\ (d)$$

is a subset of P. This implies that

$$(S_{p(1)}, S_{p(2)}, \ldots, S_{p(n)})(d) \text{ is defined and in } P$$

for all permutations p. This, in turn, implies that d is in the complete precondition Q_p for every p. Therefore, d is also in Q, the intersection of these sets. Therefore, Q must be a complete precondition. ■

There is clearly a connection between

> the sequential interchangeability of a collection of statements with respect to a precondition and a postcondition (see definition 4.2) and a precondition under a pseudoconcurrent construct.

This connection is clarified in the following paragraphs, in which the logical relationships between the following three statements are examined.

1 The collection of statements

$$S_1, S_2, \ldots, S_n$$

is sequentially interchangeable with respect to the precondition Q and the postcondition P.

2 Q is a precondition of the postcondition P under the sequence

$$(S_{p(1)}, S_{p(2)}, \ldots, S_{p(n)})$$

for every permutation p of the integers 1 through n inclusive.

3 Q is a precondition of the postcondition P under the pseudoconcurrent construct

$$|[S_1, S_2, \ldots, S_n]|$$

Theorem 4.3: The first statement above implies the second.

Proof: (sketch) This fact, which was stated following definition 4.2 above, follows directly from that definition and the definition of a precondition (see Chapter 3). ■

Theorem 4.4: The second statement above implies the third.

Proof: This theorem is a special case of theorem 4.1, in which $Q_p = Q$ for every permutation p. ■

The converse of this theorem is not, in general, true. There exist data environments which, being outside the domain of the pseudoconcurrent construct, may be included in its precondition Q but which cannot be included in a precondition Q satisfying the second statement. An example is any data environment d for which

$$(S_{p(1)}, S_{p(2)}, \ldots, S_{p(n)})\,(d)$$

is undefined for some permutation p and is outside the postcondition P for another.

However, if such data environments are excluded from the precondition Q, the converse of theorem 4.4 is true, as the following theorem states.

Theorem 4.5: If Q is a subset of the domain of the pseudoconcurrent construct

$$|[S_1, S_2, \ldots, S_n]|$$

then the third statement above implies the second. Furthermore, Q is a strict precondition of each permuted sequence of the given statements.

Proof: Consider any data environment $d0$ in Q. Because Q is a subset of the domain of the pseudoconcurrent construct,

$$d1 = (S_{p(1)}, S_{p(2)}, \ldots, S_{p(n)})\,(d0)$$

is defined for every permutation p. Because Q is a precondition of P under the pseudoconcurrent construct, $d1$ is in P. Thus, Q satisfies the definition of a strict precondition of P under every permuted sequence of the given statements. ■

Theorem 4.6: If Q is a subset of the domain of the sequence

$$(S_{p(1)}, S_{p(2)}, \ldots, S_{p(n)})$$

for every permutation p, then the second statement above implies the first.

Proof: Consider any data environment d in Q. Then, for every permutation p, the data enviornment

$$(S_{p(1)}, S_{p(2)}, \ldots, S_{p(n)})(d)$$

is defined and in P. Thus, definition 4.2 for the sequential interchangeability with respect to a precondition and a postcondition is fulfilled. ■

Summarizing theorems 4.3 through 4.6, it is always true that

statement 1 $\Rightarrow$ statement 2 $\Rightarrow$ statement 3

If Q is a subset of the domain of the pseudoconcurrent construct, then the three statements are equivalent:

statement 1 $\Leftrightarrow$ statement 2 $\Leftrightarrow$ statement 3

4.3.2 The computational history of contemporaneously executed statements

In Section 2.2, the computational history of the execution of a program was viewed as a sequence of data environments:

$$[d0, d1, d2, d3, \ldots]$$

or, in diagrammatical form:

$$d0 \rightarrow d1 \rightarrow d2 \rightarrow d3 \rightarrow \ldots$$

The statements (functions) transforming each data environment into its successor can be added to the diagram:

$$d0 \xrightarrow{S1} d1 \xrightarrow{S2} d2 \xrightarrow{S3} d3 \xrightarrow{S4} \ldots$$

Such a structure, which reflects the notion of the temporal development of the execution of a program, has the important property that each data environment has exactly one predecessor and one successor, except the first (if any) which has no predecessor and the last (if any) which has no successor. Such a sequence reflects the fundamental notion that program statements (mathematical functions) are applied to their data environments (arguments) one after another, in a specified order.

Generalizing to allow contemporaneous execution of program statements (non-sequential application of functions) requires that the above restriction – one predecessor, one successor – be relaxed. At least branching (more than one successor) must be allowed. Allowing joining (more than one predecessor of a single data environment) is a logical next step and is necessary if the results of the contemporaneously executed program segments are to be combined and used in a subsequent execution of some program segment. If the concept of the computational history of the execution of a progam is generalized in this way, structures such as that shown in Fig. 4.0 may arise:

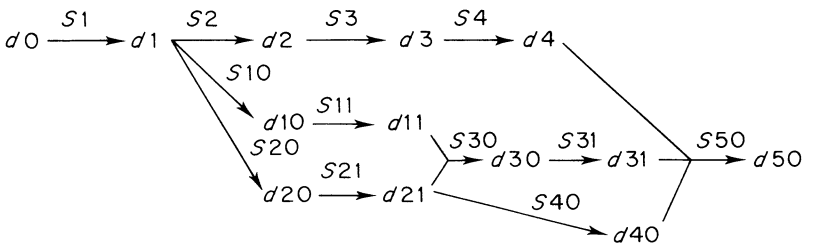


Fig. 4.0 A non-sequential computational history

In functional terms, this computational history may be described as follows:

$$d50 = S50(d4, d31, d40),$$

where

$$d4 = S4(d3)$$
$$d3 = S3(d2)$$

$$d2 = S2(d1)$$
$$d1 = S1(d0)$$
$$d31 = S31(d30)$$
$$d30 = S30(d11, d21)$$
$$d11 = S11(d10)$$
$$d10 = S10(d1)$$
$$d21 = S21(d20)$$
$$d20 = S20(d1)$$

and

$$d40 = S40(d21)$$

Combining into one expression yields

$$d50 = S50(S4(S3(S2(S1(d0)))),\ S31(S30(S11(S10(S1(d0))),S21(S20(S1(d0)))\ )),\ S40(S21(S20(S1(d0))))\ )$$

The concept of branching is natural and its definition obvious enough; several different statements are applied to (executed upon) the same data environment to form as many different data environments. In more mathematical terms, several different functions are applied to the same argument, yielding as many values (results).

Not so obvious is the action of joining as shown above. A program construct is needed which has not just one, but in general many data environments as arguments. No such construct has yet emerged as a general consensus in programming practice, although in some specific situations, the one or other particular mechanism may appear to be a natural choice. Several possibilities exist for such a general program construct, such as concatenation of the arguments (data environments) in a predetermined order, transferring the values of variables in one or more arguments (data environments) to variables of the same name in another argument, which becomes the resulting data environment, etc.

A contemporaneous computational history such as the one illustrated above may contain sequential substructures. In the above diagram, the sequences (*S*2, *S*3, *S*4), (*S*10, *S*11) and (*S*20, *S*21) are examples of such sequential substructures of the computational history. In such cases, the computational history can be reduced by combining such sequences of statements into a single compound statement and considering only the initial and final data environments of the sequence. The resulting reduced contemporaneous computational history illustrates only the branching and joining structure.

More formally, if in a contemporaneous computational history a data environment *d* exists such that

1 d has exactly one predecessor,
2 d has exactly one successor and
3 d's successor has only one predecessor (i.e. d is its successor's only predecessor),

then the data environment d can be eliminated by combining the statement giving rise to it and the statement which is applied to (executed upon) it into a single compound statement. If a contemporaneous computational history contains no such data environment d, it is said to be reduced.

If the example above is reduced in this way, the result is as shown in Fig. 4.1.

In functional terms, this computational history may be described as follows:

$$d50 = S50(S4'(S1(d0)),\ S31(S30(S11'(S1(d0)), S21'(S1(d0)))),\ S40(S21'(S1(d0))))$$

where

$$S4' = (S2, S3, S4)$$

$$S11' = (S10, S11)$$

and

$$S21' = (S20, S21)$$

The various proof rules, concepts of preconditions and postconditions, etc. covered elsewhere in this book can be applied straightforwardly to the

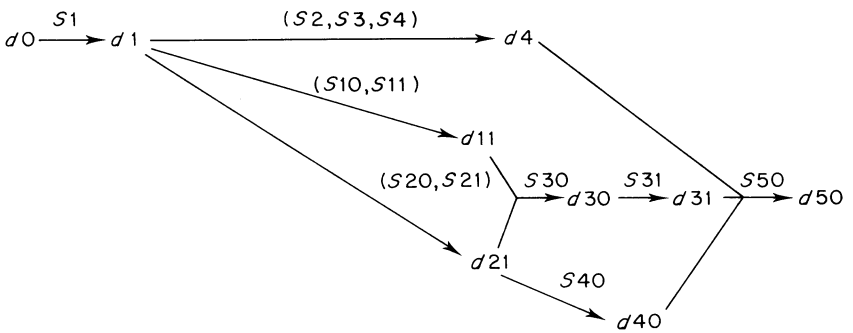


Fig. 4.1 A reduced non-sequential computational history

individual program segments and sequences in contemporaneous program structures of the types discussed in this section.

4.3.3 The interleaved execution of sequential programs upon a common sequence of data environments

A typical mechanism for executing several sequential processes 'in parallel' in effect interleaves (merges) the several sequences of statements to be executed into a single sequence. The resulting single sequence is then executed as defined in Chapter 2. The several distinct program segments operate upon a shared data environment or, more precisely, a common sequence of data environments. Thus, a single sequence of data environments constitutes the computational history of such a mechanism for executing program segments in parallel.

The several program segments are combined so that the original relative order of statements in each is preserved. Otherwise, no restrictions on the interleaving mechanism are imposed. It is, in general, not reproducible and may be viewed as a random process.

Usually, the several individual sequential processes being executed in parallel in this manner interact through shared variables. Other variables are referenced only by one process. In some schemes, the shared variables constitute a common pool referenced by any number of such computational processes. In others, variables are shared by only two processes.

The interaction among processes being executed upon a shared data environment in an interleaved way can give rise to major problems. Even if each individual program segment, when executed independently, yields correct results, it may no longer do so when executed interleaved with other programs. Some of the difficulties which can arise and some approaches to avoiding or solving them are outlined below.

In order to discuss these phenomena mathematically precisely, one must define the mechanism for the interleaved execution of separate programs accordingly. In practice, such mechanisms differ in important details such as the smallest interleavable piece of code. For example, in some systems, the smallest executional unit might be an assignment, declaration or release statement. In others, such a statement might be subdivided into still smaller pieces (e.g. machine language instructions) before interleaving these pieces with comparable units of other processes for execution. Still others might execute an entire **if** construct, for example, without interruption. Such differences can be of consequence to the software designer.

For the sake of simplicity, we will assume in the following examples that the indivisible unit of execution is the elementary statement (assignment, declaration or release statement). Furthermore, we will consider specific situations involving only sequences of assignment statements.

Example 8: Consider the following two program segments, each consisting of a sequence of three statements.

Program segment 1: ($S11$, $S12$, $S13$)

Program segment 2: ($S21$, $S22$, $S23$)

If these program segments are executed in sequence (i.e. not in parallel), the first segment followed by the second, the statements would be executed in the order

($S11$, $S12$, $S13$, $S21$, $S22$, $S23$)

If interleaved, the following executional sequences would be among the 20 ((3+3)!/(3!3!)) possible ones:

($S11$, $S12$, $S21$, $S13$, $S22$, $S23$)
($S11$, $S21$, $S12$, $S13$, $S22$, $S23$)
($S11$, $S21$, $S12$, $S22$, $S13$, $S23$)
($S11$, $S21$, $S22$, $S12$, $S13$, $S23$)
($S11$, $S12$, $S21$, $S22$, $S13$, $S23$)

Note that any such sequence can be derived from any other by a sequence of steps, each of which interchanges adjacent statements from different program segments. The non-interleaved sequence ($S11$, $S12$, $S13$, $S21$, $S22$, $S23$) provides a convenient starting point.

From the observation in the last paragraph above, the following generally valid conclusion can be deduced. If each pair of statements from different program segments is sequentially interchangeable (see Section 4.3.0, especially definition 4.0), then every interleaved executional sequence has the same effect and the program segments may be executed in an interleaved manner. This requirement is a rather restrictive one, however, and is generally satisfied only by program segments which do not interact at all (i.e. which refer to no common variables) or in rather special, often uninteresting, cases.

This difficulty can be circumvented effectively by (a) considering pairs of statements which are sequentially interchangeable with respect to a suitable precondition and postcondition, i.e. in the weaker sense of definition 4.2, and (b) permitting the programmer to restrain the interleaving mechanism. Some systems allow him, for example, to specify that certain groups of statements are not to be subdivided by the interleaving mechanism. A variety of constructs exist in programming languages for this purpose, for example, data locking, semaphores, synchronization or rendezvous points, waiting mechanisms (implicit or explicit, applied to the exchange of messages

between communicating sequential processes or in more general situations), etc.

Such mechanisms, while in common use, are often unnecessarily restrictive. It is sufficient to prevent the interleaving mechanism from subdividing a group of statements by interleaving a statement which is not sequentially interchangeable with the appropriate component of the group. Interleaving other statements with components of the said group can be permitted. The following example illustrates this situation.

Example 9: Consider the following instance of the structure considered in example 8. Each program segment represents a simplified form of part of a program for recording airline seat reservations. Initially, the values of the variables *dem*1 and *dem*2 indicate the number of seats requested by the customers being processed by terminals 1 and 2 respectively.

program segment 1	program segment 2
$S11$: $temp1 := avail$	$S21$: $temp2 := avail$
$S12$: $avail := temp1 - dem1$	$S22$: $avail := temp2 - dem2$
$S13$: $res1 := res1 + dem1$	$S23$: $res2 := res2 + dem2$

The value of the shared variable *avail* gives the number of seats available on the flight in question. The variables *res*1 and *res*2 accumulate the number of seats reserved via terminals 1 and 2 respectively.

If the postcondition P is

$$avail + res1 + res2 = cap$$

where *cap* is the capacity of the flight in question, it can be easily shown that this same condition is a complete precondition under either program segment 1 or 2. This condition can be viewed as a program invariant or, probably more appropriately, a data invariant.

Consider the interleaved sequence ($S11$, $S21$, $S22$, $S12$, $S13$, $S23$). We derive the complete preconditions at the various points in the interleaved execution as follows:

program segment 1	program segment 2
$\{avail + res1 + res2 + dem2 = cap\}$	
$S11$: $temp1 := avail$	
	$\{temp1 + res1 + res2 + dem2 = cap\}$
	$S21$: $temp2 := avail$
	$\{temp1 + res1 + res2 + dem2 = cap\}$
	$S22$: $avail := temp2 - dem2$

$\{temp1 + res1 + res2 + dem2 = cap\}$
$S12$: $avail := temp1 - dem1$
$\{avail + res1 + res2 + dem1 + dem2 = cap\}$
$S13$: $res1 := res1 + dem1$
$\{avail + res1 + res2 + dem2 = cap\}$
$S23$: $res2 := res2 + dem2$
$\{avail + res1 + res2 = cap\}$

Similarly, the complete precondition under the sequence ($S11$, $S21$, $S12$, $S22$, $S13$, $S23$) is $\{avail + res1 + res2 + dem1 = cap\}$. Summarizing, we have

$\{avail + res1 + res2 = cap\}$ ($S11$, $S12$, $S13$, $S21$, $S22$, $S23$) $\{P\}$ completely

$\{avail + res1 + res2 + dem2 = cap\}$ ($S11$, $S21$, $S22$, $S12$, $S13$, $S23$) $\{P\}$ completely

and

$\{avail + res1 + res2 + dem1 = cap\}$ ($S11$, $S21$, $S12$, $S22$, $S13$, $S23$) $\{P\}$ completely

If the interleaved execution is to yield a data environment in P regardless of which of the above three sequences is actually executed, the initial data environment must be in the intersection of the three preconditions above. Their intersection is a subset of $\{dem1 = dem2 = 0\}$. Thus the interleaved execution of program segments 1 and 2 is correct only for this uninteresting special case.

One source of the problem is clearly the fact that the execution of $S12$ negates the effect of $S22$. This is, in turn, due to the fact that $S12$ and $S22$ are not sequentially interchangeable. A similar difficulty arises because $S12$ and $S21$ are not sequentially interchangeable.

If one attempts to interchange adjacent statements from different program segments in the sequence

($S11$, $S12$, $S13$, $S21$, $S22$, $S23$)

in order to obtain

($S11$, $S21$, $S22$, $S12$, $S13$, $S23$),

one must at some point interchange $S12$ and $S21$, e.g. transform ($S11$, $S12$, $S21$, $S13$, $S22$, $S23$) into ($S11$, $S21$, $S12$, $S13$, $S22$, $S23$)

Because the functions ($S12$, $S21$) and ($S21$, $S12$) are not equal, the functions

corresponding to the above two longer sequences will not, in general, be equal.

It is true, however, that the sequence ($S11$, $S12$), viewed as a single construct, is sequentially interchangeable with the sequence ($S21$, $S22$) with respect to any precondition and any postcondition involving neither $temp1$ nor $temp2$. Thus, if the sequence ($S11$, $S12$) is not interrupted to execute either $S21$ or $S22$, and, correspondingly, if the sequence ($S21$, $S22$) is not interrupted to execute either $S11$ or $S12$, no difficulty will arise. Subject to this restriction, which is typically implemented by 'locking' the variable *avail* (temporarily reserving it for one process), the interleaved execution of program segments 1 and 2 will always yield correct results.

Note that the sequence ($S11$, $S12$) may be interrupted by $S23$ without giving rise to an incorrect result. Correspondingly, the sequence ($S21$, $S22$) may be interrupted by $S13$, e.g. in the sequence ($S11$, $S12$, $S21$, $S13$, $S22$, $S23$).

Inserting the locking and unlocking commands to achieve the effect described above, the program segments 1 and 2 become:

program segment 1	program segment 2
lock *avail*	**lock** *avail*
$S11$: $temp1 := avail$	$S21$: $temp2 := avail$
$S12$: $avail := temp1 - dem1$	$S22$: $avail := temp2 - dem2$
unlock *avail*	**unlock** *avail*
$S13$: $res1 := res1 + dem1$	$S23$: $res2 := res2 + dem2$

Locking a single shared variable does not, however, guarantee correct results if another variable is shared with another process – not even if only one process modifies the shared variable. The following example illustrates such a situation.

Example 10: Consider a situation in which a third program is added to the system in example 9. The third program contains a segment which 'inputs' or 'reads' the values of *avail*, $res1$ and $res2$ and displays this information.

program segment 3

$S31$: $report(page, linea) := avail$
$S32$: $report(page, line1) := res1$
$S33$: $report(page, line2) := res2$

If no restraint is imposed on the relationship among the values displayed on the three lines of the report, this program segment will function correctly

when its execution is interleaved with the execution of program segments 1 and 2 in example 9.

If, however, it is required that the sum of the numbers displayed on the three lines be *cap* (the capacity of the flight), i.e. if

report(*page*, *linea*) + *report*(*page*, *line*1) + *report*(*page*, *line*2) = *cap*

is a postcondition under program segment 3, the interleaved execution of program segments 1, 2 and 3 will not, in general, yield correct results. As in example 9, the problem arises because certain pairs of statements are not sequentially interchangeable – neither in the strict sense of definition 4.0 nor in the weaker sense of definition 4.2 with respect to the appropriate conditions.

In particular, the following pairs of statements are not sequentially interchangeable: (*S*31, *S*12), (*S*31, *S*22), (*S*32, *S*13) and (*S*33, *S*23). The following groups including these statements are, however, sequentially interchangeable with respect to the appropriate preconditions and postconditions:

(*S*12, *S*13) and (*S*31, *S*32)
(*S*22, *S*23) and (*S*31, *S*32, *S*33)

One can show in several different ways that each above pair of groups of statements is sequentially interchangeable. Most simply, it follows from the sequential interchangeability of the entire program segments (*S*11, S12, S13) and (*S*31, *S*32, *S*33) in the case of the first pair above and, correspondingly, from the sequential interchangeability of the program segments (*S*21, *S*22, *S*23) and (*S*31, *S*32, *S*33) in the case of the second pair.

Adding appropriate locking statements to the final versions of program segments 1 and 2 in example 9 and to program segment 3, we obtain the following versions of the three program segments:

program segment 1

lock *avail*
*S*11: *temp*1 := *avail*
*S*12: *avail* := *temp*1 − *dem*1
lock *res*1
unlock *avail*
*S*13: *res*1 := *res*1 + *dem*1
unlock *res*1

program segment 2

lock *avail*
*S*21: *temp*2 := *avail*
*S*22: *avail* := *temp*2 − *dem*2
lock *res*2
unlock *avail*
*S*23: *res*2 := *res*2 + *dem*2
unlock *res*2

program segment 3

lock *avail*
*S*31: *report*(*page*, *linea*) := *avail*
lock *res*1
lock *res*2
unlock *avail*
*S*32: *report*(*page*, *line*1) := *res*1
unlock *res*1
*S*33: *report*(*page*, *line*2) := *res*2
unlock *res*2

This locking scheme is minimally restrictive in the sense that it permits execution of every sequence of the statements which always yields a correct result while it prevents the execution of every sequence which could yield an incorrect result. Other locking schemes exist which guarantee correct results but which are unnecessarily restrictive, i.e. they prevent the execution of certain sequences which would always yield correct results.

Note that the addition of the new program segment 3 to the system necessitated modifications to the already existing programs. Such interactions among programs to be executed interleaved are not unusual in program development.

The problem in example 10 arose not from the precise form of the third program segment, but from its postcondition, i.e. from its criterion of correctness. Thus, in order to design a system of programs which are to give correct results when executed interleaved, the designer of any one program must consider every other program with which variables are shared as well as its precondition, its various intermediate conditions and its postcondition. In general, this constitutes a very serious practical constraint on the design process and considerably complicates adding a new program to the system at a later time.

If such a system of programs is structured so that no variable is referenced by more than two programs, the complexity of interaction can be reduced considerably. This consideration argues for a system structure in which any particular shared variable or group of such variables is maintained and referenced by one data management program only. Any other program wishing to access the shared data does so by exchanging appropriate messages with the corresponding data management program. The messages are exchanged via variables, each of which is referenced by only two programs, the data management one and the one requesting (indirect) access to the shared data. Via such message variables, each individual program sends its requests to the data management program, which returns acknowledgements and the values of shared data elements as requested. Maintaining the truth of conditions imposed on the variables in the common data pool is the responsibility of the data management program. A data base system is one example of such a program.

In this section, we have considered certain specific situations involving only sequences of assignment statements. The conclusions drawn are more generally valid, however. The theory of interleaved execution outlined here can be extended to include all of the fundamental constructs defined in Chapter 2.

Practice

Chapter 5

The analysis and verification of programs: methods and examples

It would be a serious error to think that one can find certainty only in geometrical demonstrations or in the testimony of the senses.
– Augustin Louis Cauchy

It should always be required that a mathematical subject not be considered exhausted until it has become intuitively evident.
– Felix Klein

Nature is not embarrassed by difficulties of analysis.
– Augustin Fresnel

Mathematical Analysis is as extensive as nature herself.
– Joseph Fourier

Characteristic of every engineering discipline are two quite distinct aspects of the design activity: creativity and systematic verification. When deciding upon the general structure and form of the object to be designed, the engineer exercises creative skills in a non-mechanistic process. General principles based on theoretical considerations are of some value, but guidelines based on practice and experience are heavily employed. The process is neither precisely determined nor systematic nor reproducible. After making preliminary design decisions, the engineer then analyzes his proposed design systematically, precisely and in detail to verify that the object will fulfill the specifications, e.g. that stresses will not exceed the breaking strengths of the materials to be used, etc. This phase of the design effort is predominantly mechanistic and systematic in nature; it is reproducible by other engineers who have received a similar education and who share a common body of knowledge.

In practice, the creative, non-systematic work is usually performed first. When studying the engineering field in question, however, it is advantageous to examine the systematic, more mechanical aspects of the design process

first. Only afterward, the less systematic, more creative processes are considered.

There are several reasons for this approach. The requirements of the verification process tend to restrain in certain fundamental ways the freedom of choice in the creative phase. The prerequisites for the systematic analysis give rise to important and useful principles for conducting and recording the results of the creative phase. The underlying theory and the way in which it is applied in the analytical process gives insight into the fundamental nature of the design problem; this insight is of value when making the preliminary, more subjective design decisions.

We begin our examination of the engineering design of computer programs, therefore, with the analysis of given programs or program segments. We assume that the segments in question represent tentative, proposed designs. Our goal is to verify their validity, correctness and appropriateness for the specified task and to identify any possible shortcomings or restrictions which must be imposed upon their application.

For each of the constructs used in the program segment, we will apply the corresponding results of the theory presented in Chapters 2, 3 and 4. As the word 'analysis' implies, we will take the program apart and verify each part individually. Our analysis will exhibit a structure (usually hierarchical) corresponding to that of the program itself.

5.0 The assignment statement

5.0.0 Assignment to an ordinary variable

The correctness of an ordinary assignment statement is verified by straightforwardly applying either lemma 3.0 for the assignment statement (see Section 3.0.0) or theorem 3.3 (see Section 3.0.1). In either case, one must also demonstrate that the initial data environment is in the domain of the given assignment statement (see Section 2.1.0). This is done by showing that the variables referenced in the statement are contained in the initial data environment, that the value of the expression on the right side of the assignment symbol (:=) is defined and that it is an element of the set associated with the variable to which the value is being assigned.

If the postcondition is given, it is usually preferable to derive a complete precondition by applying the theorem. The given precondition (if stated) must imply the derived complete precondition, i.e. the given precondition must be a subset of the derived precondition.

Example 1: Consider the assignment statement

$$z := z + x$$

and the postcondition

$$z \leq max$$

Symbolically, this verification problem can be written as follows:

$$\{?\}\ z := z + x\ \{z \leq max\}$$

Applying theorem 3.3 for the assignment statement (see Section 3.0.1), we obtain

$$z \leq max - x$$

as a complete precondition. If this condition is met, if variables named x and z are present in the initial data environment and if the values of x and z are such that the value of $(z + x)$ is defined (is in the range of the operation '+' as implemented) and if that value is a member of the set associated with the variable z, then this part of the program in question is correct.

Alternatively, one can often apply lemma 3.0 to a given precondition to derive a postcondition. The derived postcondition must imply (be a subset of) the stated postcondition (if one is given).

Example 2: Consider the statement

$$x := y - 3 * x$$

and the precondition

$$x \geq 0$$

Call the initial data environment $d0$ and the final data environment $d1$. Then applying lemma 3.0, we have

$$x(d1) = y(d0) - 3 * x(d0)$$

and

$$y(d1) = y(d0)$$

Rewriting the precondition gives

$$x(d0) \geq 0$$

which is equivalent to

$$3 * x(d0) \geq 0$$

Combining the above, we derive

$$3 * x(d0) = y(d0) - x(d1) = y(d1) - x(d1) \geq 0$$

or, more simply,

$$y - x \geq 0$$

as the postcondition.

One must also show that $d0$ is in the domain of the statement, as described in example 1 above.

5.0.1 Assignment to an array variable

In principle, an assignment statement in which an array variable receives a new value is verified in the same way as described in Section 5.0.0. However, as stated earlier in Section 3.0.1, extreme care should be exercised when the subscript expression involves one or more variables. This point deserves particular attention, as it gives rise to probably the most subtle trap in proving programs correct. When in doubt or if confusion arises, it is advisable to rewrite the given condition in a form which explicitly indicates in which data environment each variable is to be evaluated. The resulting expression should then be manipulated in a way corresponding to the proof of theorem 3.3 in Section 3.0.1. Whenever a variable belonging to the array receiving a new value is referenced in a condition, one must distinguish between two cases depending upon whether or not the referenced variable is the one receiving a new value.

At this point the reader should review the example given in Section 3.0.1.

Example 3: The following statement appears in a program which merges two arrays of sorted values.

$$c(ic) := a(ia)$$

The required postcondition is that the resulting array is in sequence, i.e.

$$c(1) \leq c(2) \leq \ldots c(ic)$$

with the convention that this condition is considered to be true if $ic \leq 1$. This postcondition can be written in various equivalent ways, e.g.

$$[ic \leq 1]$$

$$\textbf{or } [(ic > 1) \textbf{ and}_{i=1}^{ic-1} c(i) \leq c(i+1)]$$

We are to derive a complete precondition. In order to simplify the analysis by case, distinguishing between subscript values equal to and not equal to the value of ic in the initial data environment, we rewrite the above postcondition so that these two subscript values appear in separate terms:

$$[ic \leq 1]$$

$$\textbf{or } [(ic > 1) \textbf{ and } (\textbf{and}_{i=1}^{ic-2} c(i) \leq c(i+1)) \textbf{ and } c(ic-1) \leq c(ic)]$$

In this form, the last reference to c is the only reference to an element of the array c with a subscript value equal to ic. Thus, we can apply theorem 3.3 for the assignment statement (see Section 3.0.1) and substitute $a(ia)$ for $c(ic)$ in the above postcondition to obtain the precondition:

$$[ic \leq 1]$$

$$\textbf{or } [(ic > 1) \textbf{ and } (\textbf{and}_{i=1}^{ic-2} c(i) \leq c(i+1)) \textbf{ and } c(ic-1) \leq a(ia)]$$

It is not always possible to separate references to array variables so directly and straightforwardly as illustrated above. In such situations, an exhaustive analysis by case must be performed.

In the following example, we will employ an abbreviated notation in order to improve the readability of the various expressions. A single prime symbol ($'$) will indicate evaluation in the initial data environment and a double prime symbol ($''$), evaluation in the resulting data enviornment. That is, if an assignment statement A is applied to an initial data environment d, then the meanings of y', y'', E' and E'', where y is a variable name and E is an expression, are as follows:

$$y' = \text{valvar}(y, d)$$

$$y'' = \text{valvar}(y, A(d))$$

$$E' = \text{valexp}(E, d)$$

$$E'' = \text{valexp}(E, A(d))$$

If x is an array name and i is the name of an ordinary variable, then, for example, $x''(i')$ indicates that the subscript i is to be evaluated in the initial data environment and that the array variable $x(i')$ is to be evaluated in the resulting data environment:

$$x''(i') = \text{valvar}(x(\text{valvar}(i, d)), A(d))$$

Example 4: Consider the postcondition

$$x(x(2)) = y$$

and the assignment statement

$$x(i) := E$$

where x is an array name, y is the name of an ordinary variable and E is an expression involving any variables. Rewriting the postcondition to indicate explicitly the data environment in which each variable is to be evaluated, we obtain

$x''(x''(2)) = y''$

Applying lemma 3.0 for the assignment statement (see Section 3.0.0) yields

$x''(i') = E'$

$x''(j) = x'(j)$ for all subscript values $j \neq i'$

and

$y'' = y'$

The postcondition is, therefore, equivalent to

$x''(x''(2)) = y'$

We wish to proceed by substituting expressions involving singly primed variables for doubly primed ones, thereby deriving a precondition. In taking the next step, we must distinguish between the two mutually exclusive and exhaustive cases:

Case 1: $i' \neq 2$

and

Case 2: $i' = 2$

In case 1, we can rewrite our intermediate postcondition above into the following form:

$x''(x'(2)) = y'$ (case 1 only)

To proceed further, we must again distinguish between two mutually exclusive and exhaustive cases:

Case 1(a): $i' \neq x'(2)$

and

Case 1(b): $i' = x'(2)$

In case 1(a), our condition becomes

$x'(x'(2)) = y'$ (case 1(a) only)

In case 1(b), we obtain

$x''(i') = y'$

and finally,

$E' = y'$ (case 1(b))

In case 2, we rewrite the equivalent postcondition

$x''(x''(2)) = y'$

(see above) to

$x''(x''(i')) = y'$

and finally,

$x''(E') = y'$ (case 2 only)

Again, we must distinguish between two mutually exclusive and exhaustive cases:

Case 2(a): $E' \neq i'$

and

Case 2(b): $E' = i'$

In case 2(a), we can write our condition as

$x'(E') = y'$ (case 2(a) only)

In case 2(b), we have

$x''(i') = y'$

and finally,

$E' = y'$ (case 2(b))

Combining the above four cases, we have for our precondition

$x'(x'(2)) = y'$, if $(i' \neq 2)$ **and** $(i' \neq x'(2))$, i.e. in case 1(a)

E' = y', if $(i' \neq 2)$ **and** $(i' = x'(2))$, i.e. in case 1(b)

$x'(E') = y'$ if $(i' = 2)$ **and** $(E' \neq i')$, i.e. in case 2(a)

and

$E' = y'$, if $(i' = 2)$ **and** $(E' = i')$, i.e. in case 2(b)

This can be written in the following equivalent form:

$(i' \neq 2)$ **and** $(i' \neq x'(2))$ **and** $(x'(x'(2)) = y')$

or $(i' \neq 2)$ **and** $(i' = x'(2))$ **and** $(E' = y')$

or $(i' = 2)$ **and** $(E' \neq i')$ **and** $(x'(E') = y')$

or $(i' = 2)$ **and** $(E' = i')$ **and** $(E' = y')$

Note that, by definition, the logical **and** function takes precedence over the logical **or** function, i.e. (a **or** b **and** c) means (a **or** (b **and** c)).

The above expression involves only variables to be evaluated in the initial data environment. It is, therefore, the precondition which we sought. Being logically equivalent to the given postcondition, it is a complete precondition. Because it is understood that a precondition is to be evaluated in the initial data environment, we can eliminate the prime symbols to obtain our precondition in conventional notation:

$$(i \neq 2) \textbf{ and } (i \neq x(2)) \textbf{ and } (x(x(2)) = y)$$
$$\textbf{or } (i \neq 2) \textbf{ and } (i = x(2)) \textbf{ and } (E = y)$$
$$\textbf{or } (i = 2) \textbf{ and } (E \neq i) \textbf{ and } (x(E) = y)$$
$$\textbf{or } (i = 2) \textbf{ and } (E = i) \textbf{ and } (E = y)$$

This expression can be rewritten in several different equivalent ways, which may be convenient for certain purposes, e.g.

$$(i \neq 2) \textbf{ and } (\mathrm{i} \neq \mathrm{x}(2)) \textbf{ and } (x(x(2)) = \mathrm{y})$$
$$\textbf{or } (i \neq 2) \textbf{ and } (i = x(2)) \textbf{ and } (E = y)$$
$$\textbf{or } (i = 2) \textbf{ and } (E \neq 2) \textbf{ and } (x(E) = y)$$
$$\textbf{or } (i = 2) \textbf{ and } (E = 2) \textbf{ and } (y = 2)$$

In the above derivation, we have implicitly assumed that subscript expressions are evaluated just as any other expressions. In many actual systems, subscript values must be integers. After evaluating a subscript expression in the normal way, such systems usually round a non-integral result to an integer. Such rounding, if performed automatically by the system of interest, should be explicitly considered in our analysis. If the automatic rounding is expressed by the function 'round' and the above analysis is adjusted according, our precondition becomes:

$$(\mathrm{round}(i) \neq 2) \textbf{ and } (\mathrm{round}(i) \neq \mathrm{round}(x(2))) \textbf{ and } (x(x(2)) = y)$$
$$\textbf{or } (\mathrm{round}(i) \neq 2) \textbf{ and } (\mathrm{round}(i) = \mathrm{round}(x(2))) \textbf{ and } (E = y)$$
$$\textbf{or } (\mathrm{round}(i) = 2) \textbf{ and } (\mathrm{round}(E) \neq \mathrm{round}(i)) \textbf{ and } (x(E) = y)$$
$$\textbf{or } (\mathrm{round}(i) = 2) \textbf{ and } (\mathrm{round}(E) = \mathrm{round}(i)) \textbf{ and } (E = y)$$

Note that the condition $(E = y)$ in cases 1(b) and 2(b) remains unchanged, i.e. the subscript rounding function is not applied to E in this expression. In the specification of the four cases, on the other hand, the rounding function is applied to all expressions, whereby we assume that round(2) = 2. The rounding function is, of course, understood to be implicitly applied to the subscript expressions in the references to the array variables $x(x(2))$ and $x(E)$ above.

Summarizing the conclusions to be drawn from the above examples, two different, alternative strategies are useful for deriving a precondition from a given postcondition for an assignment statement in which an array variable receives a new value:

1 Separate terms in the postcondition involving variables belonging to the array in question so that references to the array variable receiving a new value are isolated from references to array variables not receiving a new value. Then apply theorem 3.3 for the assignment statement (see Section 3.0.1).
2 Rewrite the given condition so that the data environment in which each variable is to be evaluated is explicitly indicated. Using lemma 3.0 for the assignment statement (see Section 3.0.0), substitute equal expressions for the various variables so that all variables are evaluated in the same data environment. If the goal is to derive a precondition, all variables should be evaluated in the initial data environment; if a postcondition is sought, all variables should be evaluated in the resulting data environment. Using the '/' notation has the advantage that the expressions being manipulated have a particularly simple form.

When the first strategy above can be applied, if often leads more quickly to the desired solution. The second strategy, being applicable in all situations, represents a general method for solving the problem. Furthermore, it can be used not only for deriving a precondition from a given postcondition, but also vice versa.

5.1 The *if* statement

The correctness of an **if** construct is verified by applying the progressive theorem for the **if** statement (theorem 3.4, Section 3.1.0) or the retrogressive theorem for the **if** statement (theorem 3.5, Section 3.1.1). If a postcondition is to be derived from a precondition, the progressive theorem is applicable. In the more usual situation, in which a precondition is to be derived from a given postcondition, the retrogressive theorem is used.

Example 5: We wish to determine a precondition of the postcondition

$$x \geq 0$$

under the statement

$$\textbf{if } x < 0 \textbf{ then } x := -x \textbf{ endif}$$

In this case, the statement $S1$ of the retrogressive theorem (theorem 3.5) is the statement '$x := -x$'; the statement $S2$ is the null statement.

A precondition under $S1$ is

$$-x \geq 0$$

which is equivalent to

$$x \leq 0$$

A precondition under $S2$ is the postcondition itself.

Applying the retrogressive theorem for the **if** statement, we obtain

$$((x \leq 0) \textbf{ and } (x < 0)) \textbf{ or } ((x \geq 0) \textbf{ and not } (x < 0))$$

as the precondition. Simplifying yields

$$(x < 0) \textbf{ or } (x \geq 0)$$

which is always true. Thus, the precondition is simply the logical constant 'true' and the postcondition is always satisfied, provided, of course, that the result of executing the above **if** statement on the given initial data environment is defined at all.

The domain of an **if** statement, in general, is the intersection of the domain of its condition and, if this condition is true, the domain of the statement in the **then** part or, if the **if** condition is false, the domain of the statement in the **else** part. That is,

$$\text{domain}(\textbf{if } B \textbf{ then } S1 \textbf{ else } S2 \textbf{ endif})$$

$$= \text{domain}(B) \textbf{ and } ((B \textbf{ and } \text{domain}(S1)) \textbf{ or } ((\textbf{not } B) \textbf{ and } \text{domain}(S2)))$$

Because the set B and the set (**not** B) are both subsets of the domain of B, the above condition (set) is equal to the following:

$$(B \textbf{ and } \text{domain}(S1)) \textbf{ or } ((\textbf{not } B) \textbf{ and } \text{domain } (S2))$$

The **if** statement in this example is defined for all data environments which contain a numerical variable named x, provided that whenever the value of x is negative and in the implemented range, then the (positive) value of $-x$ is also in the implemented range. This requirement is not fulfilled by all real systems.

Example 6: Consider the following **if** statement

$$\textbf{if } x < 0 \textbf{ then } x := -x \textbf{ else } x := 2 * x - 80 \textbf{ endif}$$

and the postcondition

$$x > 0$$

The precondition under the **then** part

$$x := -x$$

is

$$-x > 0$$

or, equivalently,

$$x < 0$$

The precondition under the **else** part

$$x := 2 * x - 80$$

is

$$2 * x - 80 > 0$$

or, in an equivalent, simpler form

$$x > 40$$

Applying the retrogressive theorem for the **if** statement (theorem 3.5), we obtain

$$((x < 0) \textbf{ and } (x < 0)) \textbf{ or } ((x > 40) \textbf{ and } (x \geq 0))$$

or, simplifying,

$$(x < 0) \textbf{ or } (x > 40)$$

as the precondition of the **if** statement given above.

The preconditions of the **then** and **else** parts of the **if** statement as derived above are complete preconditions. Therefore,

$$\{(x < 0) \textbf{ or } (x > 40)\}$$

$$\textbf{if } x < 0 \textbf{ then } x := -x \textbf{ else } x := 2 * x - 80 \textbf{ endif } \{x > 0\} \text{ completely}$$

(see Section 3.1.1).

5.2 A sequence of program statements

The correctness of a sequence of statements is verified by applying either theorem 3.6 or theorem 3.7 (Sections 3.2.0, 3.2.1), depending upon whether an ordinary or a complete precondition is involved.

The following example is taken from the body of a **while** loop which calculates the sum of the values stored in an array.

Example 7: Determine a complete precondition of the given postcondition

$$(s = \sum_{j=1}^{i} x(j)) \textbf{ and } (i \leq n)$$

under the sequence of statements

$i := i + 1$

$s := s + x(i)$

A complete precondition under the last statement can be found by applying theorem 3.3 for the assignment statement (see Section 3.0.1):

$$(s + x(i) = \sum_{j=1}^{i} x(j)) \textbf{ and } (i \leq n)$$

or, simplifying,

$$(s = \sum_{j=1}^{i-1} x(j)) \textbf{ and } (i \leq n)$$

This complete precondition under the last statement becomes the postcondition with respect to the first statement. Applying the theorem for the assignment statement again, we obtain for a complete precondition under the first statement

$$(s = \sum_{j=1}^{i} x(j)) \textbf{ and } (i \leq n - 1)$$

This is the desired complete precondition under the given sequence of statements.

The next example, which involves array variables, is taken from a program which permutes (rearranges) the values in an array so that they are partially sorted.

Example 8: Given the postcondition

$(gl < gr)$ **and** for every integer i such that $gl < i \leq gr$, $x(i) = x(gr)$

or, equivalently,

$$(gl < gr) \textbf{ and}_{i=gl+1}^{gr} x(i) = x(gr)$$

and the sequence of statements

$x(gl) :=: x(gr)$

$gr := gr - 1$

$gl := gl - 1$

find a complete precondition. The values of the variables gl and gr may be assumed to be integers, e.g. because gl and gr were declared integer variables.

Applying theorem 3.3 for the assignment statement (Section 3.0.1) to the last two statements above, we obtain as the precondition of the second (and the postcondition for the first) statement

$$(gl - 1 < gr - 1) \textbf{ and}_{i=gl}^{gr-1} x(i) = x(gr - 1)$$

Before applying theorem 3.3 to the first statement in the given sequence, we rewrite the above condition so that subscript values equal to those appearing in the exchange statement are isolated (see Section 5.0.1, especially example 3). The first step results in the expression

$$(gl < gr) \textbf{ and } [\textbf{and}_{i=gl+1}^{gr-1} x(i) = x(gr - 1)] \textbf{ and } x(gl) = x(gr - 1)$$

Clearly, none of the references to $x(i)$ above can refer to either $x(gl)$ or $x(gr)$. The two references to $x(gr - 1)$ cannot refer to $x(gr)$, but they may refer to $x(gl)$ if $gl = gr - 1$. The reference to $x(gl)$ must, of course, refer to $x(gl)$ but cannot refer to $x(gr)$. Thus, we must distinguish between the two situations $(gl = gr - 1)$ and $(gl \neq gr - 1)$ only. Rewriting the above postcondition for the first statement accordingly and simplifying leads to

$(gl = gr - 1)$

or

$$(gl < gr - 1) \textbf{ and } [\textbf{and}_{i=gl+1}^{gr-1} x(i) = x(gr - 1)] \textbf{ and } x(gl) = x(gr - 1)$$

as the postcondition of interest for the first statement in the sequence.

The exchange statement

$x(gl) :=: x(gr)$

was defined in Section 2.1.0 to be equivalent to the multiple assignment statement

$(x(gl), x(gr)) := (x(gr), x(gl))$

Applying theorem 3.3 for the assignment statement in its generalized form to this multiple assignment statement and to the last expression above for the postcondition of the exchange statement, we obtain

$(gl = gr - 1)$

or

$$(gl < gr - 1) \textbf{ and } [\textbf{and}_{i=gl+1}^{gr-1} x(i) = x(gr - 1)] \textbf{ and } x(gr) = x(gr - 1)$$

which reduces to

$$(gl < gr) \textbf{ and}_{i=gl+1}^{gr} x(i) = x(gr)$$

a complete precondition under the given sequence of statements. Note that the precondition is identical in form to the postcondition. It is part of the invariant of the loop from which this sequence of statements was taken.

5.3 The *while* loop

It is generally easy to prove a **while** loop correct – provided the designer has specified the loop invariant as well as the postcondition. Since deciding upon the loop invariant constitutes the major design decision in the construction of a loop, as we will see in Chapter 6, a loop invariant will be available for every properly designed and documented loop. The postcondition must, in any event, be specified, for without it, the very notion of 'correctness' has no meaning and hence it cannot be verified.

To verify the correctness of the loop, one applies the loop theorem (theorem 3.8, Section 3.3.0). This requires showing that

> the initialization establishes the truth of the loop invariant,
> the body of the loop preserves it and
> the loop invariant and the negation of the loop condition together imply the postcondition.

Finally, in common with proving all other constructs correct, one must show that the initial data environment is in the domain of the **while** loop. In particular, this involves demonstrating that the loop terminates (see Section 3.3.3).

Of the several steps in verifying the correctness of a loop, proving that the body of the loop preserves the truth of the loop invariant usually requires the greatest effort. Frequently, moderately complicated logical expressions arise. Appropriate, convenient notation can often help considerably to reduce the clerical effort.

Example 9: Consider the program segment

$i := 0$

$s := 0$

while $i < n$ **do**

$i := i + 1$
$s := s + x(i)$
endwhile

where n is an integer variable. The required postcondition is

$$s = \sum_{j=1}^{n} x(j)$$

and the loop invariant specified by the loop's designer is

$$(s = \sum_{j=1}^{i} x(j)) \textbf{ and } (i \text{ is an integer}) \textbf{ and } (i \leq n)$$

We begin by noting that the initialization establishes the truth of the loop invariant provided that in the initial data environment

$$n \geq 0$$

This is, therefore, a precondition which must be satisfied by the prior segment of the program.

We now prove that the body of the loop preserves the truth of the loop invariant. Referring to the loop theorem (theorem 3.8) this necessitates showing that

$$\{I \textbf{ and } B\}\ S\ \{I\}$$

where I is the above loop invariant, B is the **while** condition $(i < n)$ and S is the sequence of two statements comprising the body of the loop.

We note that the condition (I **and** B) is equal to

$$(s = \sum_{j=1}^{i} x(j)) \textbf{ and } (i \text{ is an integer}) \textbf{ and } (i < n)$$

Because n is an integer, this condition can be rewritten as follows:

$$(s = \sum_{j=1}^{i} x(j)) \textbf{ and } (i \text{ is an integer}) \textbf{ and } (i \leq n - 1)$$

By applying theorem 3.3 for the assignment statement (Section 3.0.1) and simplifying, it is easily shown that

$$\{i \text{ is an integer}\}\ S\ \{i \text{ is an integer}\}$$

In Section 5.2, example 7, it was shown that

$$\{(s = \sum_{j=1}^{i} x(j)) \textbf{ and } (i \leq n - 1)\}\ S\ \{(s = \sum_{j=1}^{i} x(j)) \textbf{ and } (i \leq n)\}$$

Applying theorem 3.0, we can combine the last two statements above to obtain

$$\{(\mathrm{s} = \sum_{j=1}^{i} \mathrm{x(j)}) \textbf{ and } (i \text{ is an integer}) \textbf{ and } (i \leq n - 1)\}$$

$$S\{(s = \sum_{j=1}^{i} x(j)) \textbf{ and } (i \text{ is an integer}) \textbf{ and } (i \leq n)\}$$

thereby verifying that $\{I$ **and** $B\}\ S\ \{I\}$, i.e. that the body of the loop preserves the loop invariant.

The loop theorem states that when (and if) the loop terminates, the condition (I **and not** B) will be satisfied, i.e. that

$$(s = \sum_{j=1}^{i} x(j)) \textbf{ and } (i \text{ is an integer}) \textbf{ and } (i \leq n) \textbf{ and } (\textbf{not } (i < n))$$

will be true. This condition simplifies to

$$(s = \sum_{j=1}^{i} x(j)) \textbf{ and } (i \text{ is an integer}) \textbf{ and } (i = n)$$

which implies

$$s = \sum_{j=1}^{n} x(j)$$

the specified postcondition. We conclude, therefore, that the loop is partially correct provided that $n \geq 0$.

Termination can be shown easily. Initially, the value of i is 0. It is increased by one each time the body of the loop is executed. Therefore, after the body of the loop has been executed n times, $i = n$, and the loop will terminate.

One must also show that the effect of executing each statement is defined. This will be the case if the initial data environment contains all referenced variables (appropriately declared), if the array x exists with subscripts ranging from 1 to n inclusive (at least) and if each intermediate sum of the $x(i)$ and the final value are within the range of the numerical set specified in the declaration of s.

The above method can be used only when the loop invariant is known. In general, it will yield an ordinary, but not a complete precondition. If the loop invariant is not known or if a complete precondition is required, we must employ a different approach based on the corollary for complete preconditions of a loop (see Section 3.3.2).

Example 10: Consider the program segment and postcondition as in example 9. Here, we assume that a complete precondition is desired and that a corresponding loop invariant is not available. We will, therefore, apply the corollary of Section 3.3.2 and derive both a complete precondition and a loop invariant.

We begin by restating the postcondition:

$$s = \sum_{j=1}^{n} x(j)$$

This postcondition is not a subset of (**not** B), so we must construct the effective postcondition (see Section 3.3.1) by forming the intersection of the above postcondition and (**not** B). The resulting condition will be the P and Z_0 of the corollary:

$$Z_0 = [(s = \sum_{j=1}^{n} x(j)) \textbf{ and } (i \geq n)]$$

Applying the definitions of C and Z stated in the corollary in Section 3.3.2, we obtain

$$C_0 = [(s + x(i+1) = \sum_{j=1}^{n} x(j)) \textbf{ and } (i + 1 \geq n)]$$

$$Z_1 = [(s = \sum_{j=1}^{i} x(j)) \textbf{ and } (i = n - 1)]$$

$$Z_2 = [(s = \sum_{j=1}^{i} x(j)) \textbf{ and } (i = n - 2)]$$

$$\ldots$$

$$Z_k = [(s = \sum_{j=1}^{i} x(j)) \textbf{ and } (i = n - k)]$$

$$\ldots$$

The complete precondition Q of the loop and the loop invariant is, therefore,

$$Q = [(s = \sum_{j=1}^{n} x(j)) \textbf{ and } (i \geq n)] \textbf{ or } [(s = \sum_{j=1}^{i} x(j)) \textbf{ and } (i < n)]$$

$$= [(s = \sum_{j=1}^{n} x(j)) \textbf{ and } (i > n)] \textbf{ or } [(s = \sum_{j=1}^{i} x(j)) \textbf{ and } (i \leq n)]$$

Applying the theorem for the assignment statement to each of the two initializing statements, we obtain

$$[(0 = \sum_{j=1}^{n} x(j)) \textbf{ and } (n < 0)] \textbf{ or } [n \geq 0]$$

as a complete precondition under the entire program segment. Compare this with the precondition derived in example 9.

The method used in example 10 can, in principle, be applied to every loop. In practice, however, the complexity of the expressions which can arise sometimes precludes its application.

5.4 The declaration statement

As pointed out in Section 3.4, the declaration statement has a very similar effect as the assignment statement. As far as non-concealed variables are concerned, the effect is, in fact, identical. Thus, the same analytical techniques as those used for the assignment statement are often applied to the declaration statement.

The declaration statement is frequently used to preserve the values of variables which, while not required in the immediate program segment, may be required later in hierarchically superior program segments. Before returning control to such a superior segment, a release statement is executed, making the concealed value accessible again.

In order to prove the correctness of the superior segment, the special effects of declare statements must, in general, be considered explicitly. This can often be done conveniently by noting that if, for example, the declare statement D

declare $(x, \mathbb{Z}, E)$

is applied to a data environment $d0$, the resulting data environment $d1$ is given by the equation

$$d1 = D(d0) = [(x, \mathbb{Z}, \text{valexp}(E, d0))] \,\&\, d0$$

This notational form will often facilitate proving that part or all of the initial data environment is left undisturbed by the action of a particular program segment.

Most frequently a progressive approach will be found appropriate when analyzing a section of a program consisting of one or more declare statements. That is, one will typically derive a postcondition from a given precondition. The postcondition will usually involve statements about the structure of the resulting data environment, as illustrated by the above example; the postcondition will not be simply a conditional expression over variable names. Section 5.5 and the example of a recursive procedure in Section 5.8.1 below treat this subject further.

5.5 The release statement

The release statement is used to undo the effect of a previously executed declare statement. The goal is typically to restore the data environment to a previous state or to an approximation thereto. For example, if the data environment $d1$ has the structure

$$d1 = [(x, \mathbb{Z}, \text{valexp}(E, d0))] \,\&\, d0$$

and the release statement R

release x

is applied to $d1$, then the resulting data environment $d2$ is given by the equation

$$d2 = R(d1) = d0$$

Equations and conditions in this and similar form will be used in the proof of correctness of the recursive procedure in Section 5.8.1 below.

5.6 The procedure call

The procedure call without parameters was defined in Sections 2.1.6 and 2.2.6 to be semantically equivalent to the body of the procedure. Therefore, the analytical techniques appropriate for the constructs appearing in the body of the procedure should be applied.

Procedure calls with parameters may be replaced by the equivalent procedure calls without parameters (see Section 4.0). The equivalent program can then be verified by the methods described above.

Normally, a procedure is designed to perform one particular function, whereby a specific precondition and a specific postcondition are presumed. A theorem about the precondition and the postcondition can be formulated and proved. Such a theorem constitutes a proof rule for a call to the procedure in question and can be used in the proofs of the correctness of the program segments which call the procedure. In fact, one can argue that such an approach to proving the correctness of a program is the most important – or even the only real – reason for subdividing the program into individual procedures. This approach will be used in examples in Section 5.8 and in Chapter 6.

5.7 Other loop constructs

To prove a loop other than a **while** loop correct, replace it by the equivalent **while** loop and prove the latter correct as described in Section 5.3 above. Section 2.1.8 gives definitions for several other loop constructs in terms of the **while** loop.

5.8 Examples of the analysis of entire program segments

The following examples illustrate how the proof techniques described in earlier sections may be combined to prove the correctness of a program segment consisting of a combination of assignment statements, **if** statements,

loops, sequences thereof, etc. They illustrate the level of complexity which can arise in such proofs as well as ways of limiting and dealing with it. While some of the conditional expressions in the following examples may appear complicated to the novice, one should keep in mind that (a) skill in interpreting and manipulating such expressions can be acquired quickly by appropriate study and exercise, (b) it is quite possible to cope with even the very complicated expressions by 'dividing and conquering' them in suitable ways and (c) other engineers must deal with problems at least as complex (e.g. the civil engineer when calculating the statics of a proposed design for a bridge or a skyscraper, the aeronautical engineer when analyzing the characteristics of an airplane at both subsonic and supersonic speeds, the electrical engineer when calculating the response function of an intricate circuit performing both band-pass and impedance matching functions, etc.).

5.8.0 Merging two sorted arrays

The following program merges the values in two sorted arrays a and b. The values are copied into the resulting array c so that c is in sequence when the program terminates.

$(ia, ib, ic) := (1, 1, 1)$

while $(ia \leq na)$ **or** $(ib \leq nb)$ **do**

  if $(ib > nb)$ **or** $((ia \leq na)$ **and** $(a(ia) \leq b(ib)))$
  then $c(ic) := a(ia)$
    $ia := ia + 1$
  else $c(ic) := b(ib)$
    $ib := ib + 1$
  endif
  $ic := ic + 1$
endwhile

Formally, we state the precondition as follows. Given are two families of array variables, $a(ia)$ and $b(ib)$, where $ia = 1, 2, \ldots, na$ and $ib = 1, 2, \ldots, nb$. The values of the variables na and nb are non-negative integers. The two arrays are sorted in ascending sequence, i.e.

for all integers i such that $1 \leq i < na$, $a(i) \leq a(i + 1)$

and for all integers i such that $1 \leq i < nb$, $b(i) \leq b(i + 1)$

The postcondition can be formally stated as follows:

the sequence $[c(1), c(2), \ldots, c(na + nb)]$ is a permutation (rearrangement of the terms) of the sequence $[a(1), a(2), \ldots a(na), b(1), b(2), \ldots, b(nb)]$

and for all integers i such that $1 \leq i < na + nb$, $c(i) \leq c(i + 1)$

whereby we also require that the merging program not modify either array a or b.

The general idea behind the program's variables and statements is that $a(ia)$ is the next element of array a which should be copied to the array c; in other words, ia 'points' to the next element of the array a to be copied. Similarly, ib points to the next element of array b to be copied to array c. The variable ic points to the element of array c which is to receive the next value from either array a or array b. The lesser of the next elements from a or b is copied to c as long as an uncopied element remains in the array in question. Whenever an element of a or b is copied to c, the pointers are updated accordingly.

The designer specified that the loop invariant consist of the following terms combined with the logical **and** operator:

$I1$: $1 \leq ia \leq na + 1$

$I2$: $1 \leq ib \leq nb + 1$

$I3$: $(ic - 1) = (ia - 1) + (ib - 1)$

$I4$: the sequence $[c(1), c(2), \ldots, c(ic - 1)]$ is a permutation of the sequence $[a(1), a(2), \ldots, a(ia - 1), b(1), b(2) \ldots, b(ib - 1)]$

$I5$: $(1 < ic)$ **and** $(ia \leq na) \Rightarrow c(ic - 1) \leq a(ia)$

$I6$: $(1 < ic)$ **and** $(ib \leq nb) \Rightarrow c(ic - 1) \leq b(ib)$

$I7$: for all integers i such that $1 \leq i < ic - 1$, $c(i) \leq c(i + 1)$

The individual terms or groups thereof are intended to express in mathematically precise language the following notions. $I1$ states that the value of the pointer ia is either in the defined range of subscripts for array a or, if all elements of array a have already been copied to array c, the value of ia is 1 greater than the greatest allowed subscript value. $I2$ is the corresponding statement for the pointer ib. $I3$ defines ic in terms of ia and ib. (Since ic is a function of ia and ib, it is redundant and could be eliminated from the program.) $I4$ states that the values in array c must have come from arrays a and b, i.e. that they were not generated in some other way. $I5$ and $I6$ state that the next values to be copied from arrays a and b (if any) are greater than or equal to the last value copied to array c (if any). $I7$ states that the values already in array c are in ascending sequence.

Note that each of the above terms in the loop invariant depends as follows upon program variables whose values are changed by the merging program:

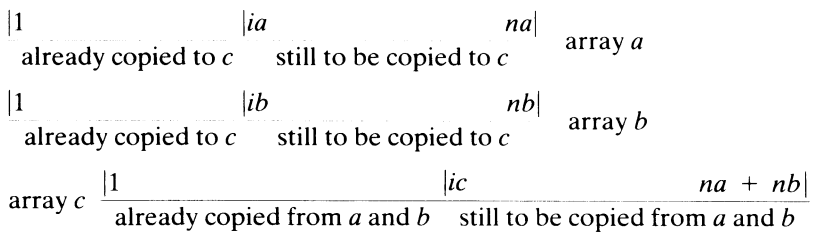
$I1(ia)$
$I2(ib)$
$I3(ia, ib, ic)$
$I4(ia, ib, ic, c)$
$I5(ia, ic, c)$
$I6(ib, ic, c)$
$I7(ic, c)$

Thus, the loop invariant is

$I1(ia)$ **and** $I2(ib)$ **and** $I3(ia, ib, ic)$ **and** $I4(ia, ib, ic, c)$

and $I5(ia, ic, c)$ **and** $I6(ib, ic, c)$ **and** $I7(ic, c)$

The loop invariant can be represented diagrammatically as follows:



To simplify the expressions which arise and especially to prevent losing sight of the structure of our correctness proof, we label the several conditions and statements which appear in the program as follows:

$B1$: $(ia \leq na)$ **or** $(ib \leq nb)$

$B2$: $(ib > nb)$ **or** $((ia \leq na)$ **and** $(a(ia) \leq b(ib)))$

$S1$: $(c(ic) := a(ia), ia := ia + 1)$

$S2$: $(c(ic) := b(ib), ib := ib + 1)$

$S3$: **if** $B2$ **then** $S1$ **else** $S2$ **endif**

$S4$: $ic := ic + 1$

$S5$: $(S3, S4)$

Using this notation, the merge program then becomes:

$(ia, ib, ic) := (1, 1, 1)$

while $B1$ **do**

$S5$

endwhile

To demonstrate that the loop is correct, we must show that (see Sections 3.3.0 and 5.3)

1 the initialization establishes the truth of the loop invariant,
2 the loop invariant and the negation of the loop condition $B1$ together imply the postcondition,
3 the body $S5$ of the loop preserves the truth of the loop invariant and
4 the initial data environment is in the domain of the loop, in particular, the loop terminates in finite time.

By substituting 1 for ia, ib and ic in each of the terms $I1$, $I2$, ..., $I7$ of the loop invariant and noting that the precondition requires that na, $nb \geq 0$, one can verify directly that the initialization establishes the truth of the loop invariant.

The second step above can also be verified easily. The negation of the loop condition $B1$ is

not $((ia \leq na)$ **or** $(ib \leq nb))$

which is equivalent to

$(ia > na)$ **and** $(ib > nb)$

This condition and the several terms of the loop invariant imply that

$ia = na + 1$

$ib = nb + 1$

and

$ic = na + nb + 1$

Substituting these values into the terms $I4$ and $I7$ of the loop invariant leads directly to the postcondition.

Proving that the body $S5$ of the loop preserves the truth of the loop invariant involves a large number of individual steps, each of which is, however, relatively simple. We will use several different theorems stated and proved in earlier sections. They all serve to 'divide and conquer'. Our strategy is to consider the loop invariant as the given postcondition of the body of the loop and, working backward by applying the retrogressive proof rules, derive preconditions at the various intermediate points in the loop, ending with the precondition at the beginning of the body of the loop. Finally, we will show that the loop invariant together with the **while** condition implies that precondition. The general structure of this process is illustrated in Fig. 5.0.

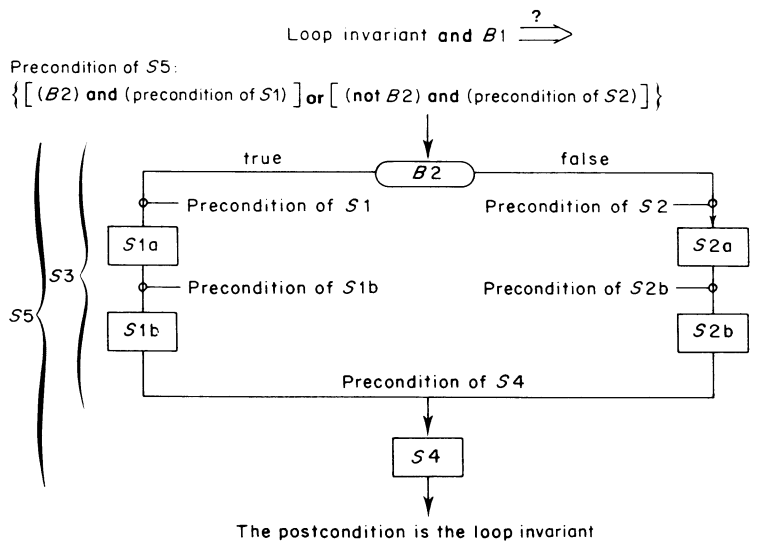


Fig. 5.0 The structure of the proof of correctness of the subprogram for merging two sorted arrays

Lemma 5.0: $\{I1(ia)$ **and** $I2(ib)\}$ $S4$ $\{I1(ia)$ **and** $I2(ib)\}$

Proof: The condition $[I1(ia)$ **and** $I2(ib)]$ does not refer to the variable ic, whose value is modified by statement $S4$. Therefore, by the theorem for the assignment statement, $[I1(ia)$ **and** $I2(ib)]$ is a precondition of itself under $S4$. ■

Lemma 5.1: $\{I1(ia)$ **and** $I2(ib)$ **and** $B1$ **and** $B2\}$ $S1$ $\{I1(ia)$ **and** $I2(ib)\}$

Proof: Applying the theorem for the assignment statement to the sequence $S1$ of statements with $[I1(ia)$ **and** $I2(ib)]$ as the postcondition, we obtain as a precondition

$$I1(ia + 1) \textbf{ and } I2(ib)$$

which is, by definition of $I1$ and $I2$,

$$(0 \leq ia \leq na) \textbf{ and } (1 \leq ib \leq nb + 1)$$

Expanding ($B1$ **and** $B2$) and simplifying yields

$$(ia \leq na) \textbf{ and } (ib > nb)$$

or $(ia \leq na)$ **and** $(a(ia) \leq b(ib))$

Thus,

$$(B1 \textbf{ and } B2) \Rightarrow (ia \leq na)$$

and therefore

$$[I1(ia) \textbf{ and } I2(ib) \textbf{ and } B1 \textbf{ and } B2]$$

$$\Rightarrow [(1 \leq ia \leq na) \textbf{ and } (1 \leq ib \leq nb + 1)]$$

Because this latter condition is stronger than (is a subset of) the precondition derived in the first step of this proof above, it follows that also $[I1(ia)$ **and** $I2(ib)$ **and** $B1$ **and** $B2]$ is a precondition of $[I1(ia)$ **and** $I2(ib)]$ under $S1$. ■

Lemma 5.2: $\{I1(ia)$ **and** $I2(ib)$ **and** $B1$ **and not** $B2\}$ $S2$ $\{I1(ia)$ **and** $I2(ib)\}$

Proof: The proof of this lemma is analogous to that of the preceding lemma 5.1. Applying the theorem for the assignment statement to the sequence $S2$ of statements with $[I1(ia)$ **and** $I2(ib)]$ as the postcondition, we obtain

$$(1 \leq ia \leq na + 1) \textbf{ and } (0 \leq ib \leq nb)$$

as a precondition of $[I1(ia)$ **and** $I2(ib)]$ under $S2$.

Expanding ($B1$ **and not** $B2$) and simplifying yields

$$(ib \leq nb) \textbf{ and } (ia > na)$$

or $(ib \leq nb)$ **and** $(a(ia) > b(ib))$

Thus,

$$(B1 \textbf{ and not } B2) \Rightarrow (ib \leq nb)$$

and therefore

$$[I1(ia) \textbf{ and } I2(ib) \textbf{ and } B1 \textbf{ and not } B2]$$

$$\Rightarrow [(1 \leq ia \leq na + 1) \textbf{ and } (1 \leq ib \leq nb)]$$

Because this latter condition is stronger than (is a subset of) the precondition derived in the first step of this proof above, it follows that also $[I1(ia)$ **and** $I2(ib)$ **and** $B1$ **and not** $B2]$ is a precondition of $[I1(ia)$ **and** $I2(ib)]$ under S2. ■

Lemma 5.3: $\{I1(ia)$ **and** $I2(ib)$ **and** $B1\}$ $S3$ $\{I1(ia)$ **and** $I2(ib)\}$

Proof: By lemma 5.1,

$\{I1(ia)$ **and** $I2(ib)$ **and** $B1$ **and** $B2\}$ $S1$ $\{I1(ia)$ **and** $I2(ib)\}$

and by lemma 5.2,

$\{I1(ia)$ **and** $I2(ib)$ **and** $B1$ **and not** $B2\}$ $S2$ $\{I1(ia)$ **and** $I2(ib)\}$

It follows by theorem 3.4 for the **if** statement that

$\{I1(ia)$ **and** $I2(ib)$ **and** $B1\}$

if $B2$ **then** $S1$ **else** $S2$ **endif** $\{I1(ia)$ **and** $I2(ib)\}$

But by definition, $S3$ is the above **if** statement. ■

Lemma 5.4: The condition $[I1(ia)$ **and** $I2(ib)]$ is a loop invariant, i.e.

$\{I1(ia)$ **and** $I2(ib)$ **and** $B1\}$ $S5$ $\{I1(ia)$ **and** $I2(ib)\}$

Proof: By lemma 5.3,

$\{I1(ia)$ **and** $I2(ib)$ **and** $B1\}$ $S3$ $\{I1(ia)$ **and** $I2(ib)\}$

and by lemma 5.0,

$\{I1(ia)$ **and** $I2(ib)\}$ $S4$ $\{I1(ia)$ **and** $I2(ib)\}$

It follows by theorem 3.6 for a sequence of statements that

$\{I1(ia)$ **and** $I2(ib)$ **and** $B1\}$ $(S3, S4)$ $\{I1(ia)$ **and** $I2(ib)\}$

But by definition, $S5$ is the sequence $(S3, S4)$. ■

The reader should pay particular attention to the structure of the proof above. Starting with the postcondition of the body $S5$ of the loop, we worked backward through the sequence of statements $(S3, S4)$ comprising $S5$. We derived a precondition under the statement $S4$, which became the postcondition for the statement $S3$. Because $S1$ and $S2$ are the components of the **if** statement $S3$, we derived preconditions under $S1$ and $S2$ and combined them to form a precondition of $S3$ and of the entire body $S5$ of the loop.

We will use the same strategy for verifying the other terms of the loop invariant. The structure of the entire proof systematically repeats the pattern in the first five lemmata above. The following table shows for each term or group of terms of the loop invariant and for each program construct which lemma deals with the corresponding precondition and postcondition:

	$S4$	$S1$	$S2$	$S3$	$S5$
($I1$ **and** $I2$)	5.0	5.1	5.2	5.3	5.4
$I3$	5.5	5.6	5.7	5.8	5.9
$I4$	5.10	5.11	5.12	5.13	5.14
($I5$ **and** $I6$)	5.15	5.16	5.17	5.18	5.19
($I7$ **and** $I5$ **and** $I6$)	5.20	5.21	5.22	5.23	5.24

Lemma 5.5: $\{I3(ia, ib, ic + 1)\}$ $S4$ $\{I3(ia, ib, ic)\}$

Proof: This lemma follows directly from the application of the theorem for the assignment statement. ■

Lemma 5.6: $\{I3(ia, ib, ic)$ **and** $B1$ **and** $B2\}$ $S1$ $\{I3(ia, ib, ic + 1)\}$

Proof: Applying the theorem for the assignment statement to $S1$, we obtain

$\{I3(ia + 1, ib, ic + 1)\}$ $S1$ $\{I3(ia, ib, ic + 1)\}$

From the definition of $I3$, it follows that

$$I3(ia + 1, ib, ic + 1) = [ic = ia + (ib - 1)]$$

But

$$[ic = ia + (ib - 1)] = [(ic - 1) = (ia - 1) + (ib - 1)] = I3(ia, ib, ic)$$

i.e.

$$I3(ia + 1, ib, ic + 1) = I3(ia, ib, ic)$$

Thus,

$\{I3(ia, ib, ic)\}$ $S1$ $\{I3(ia, ib, ic + 1)\}$

The condition $(I3(ia, ib, ic)$ **and** $B1$ **and** $B2)$ is a subset of $I3(ia, ib, ic)$ and is, therefore, also a precondition under $S1$. ■

Lemma 5.7: $\{I3(ia, ib, ic)$ **and** $B1$ **and not** $B2\}$ $S2$ $\{I3(ia, ib, ic + 1)\}$

Proof (sketch): The proof of this lemma is analogous to that of lemma 5.6. ■

Lemma 5.8: $\{I3(ia, ib, ic)$ **and** $B1\}$ $S3$ $\{I3(ia, ib, ic + 1)\}$

Proof (sketch): Analogously to the proof of lemma 5.3, this lemma follows from the above two lemmata and theorem 3.4 for the **if** statement. ■

Lemma 5.9: The condition $I3(ia, ib, ic)$ is a loop invariant, i.e.

$\{I3(ia, ib, ic)$ **and** $B1\}\ S5\ \{I3(ia, ib, ic)\}$

Proof (sketch): Analogously to the proof of lemma 5.4, this lemma follows from the lemmata 5.5 and 5.8 and theorem 3.6 for a sequence of statements. ■

Lemma 5.10: $\{I4(ia, ib, ic + 1, c)\}\ S4\ \{I4(ia, ib, ic, c)\}$

Proof: This lemma follows directly from the application of the theorem for the assignment statement. ■

Lemma 5.11: $I4(ia, ib, ic, c)$ **and** $B1$ **and** $B2\}\ S1\ \{I4(ia, ib, ic + 1, c)\}$

Proof: Applying the theorem for the assignment statement, we obtain

$\{I4(ia + 1, ib, ic + 1, c\}\ ia := ia + 1\ \{I4(ia, ib, ic + 1, c)\}$

By the definition of $I4$,

$I4(ia + 1, ib, ic + 1, c)$

$=$ (the sequence $[c(1), c(2), \ldots, c(ic)]$ is a permutation of the sequence $[a(1), a(2), \ldots, a(ia), b(1), b(2), \ldots, b(ib - 1)]$)

Applying the theorem for the assignment statement again, we obtain

{the sequence $[c(1), c(2), \ldots, c(ic - 1), a(ia)]$ is a permutation of the sequence $[a(1), a(2), \ldots, a(ia), b(1), b(2), \ldots, b(ib - 1)]$}

$c(ic) := a(ia)\ \{I4(ia + 1, ib, ic + 1, c)\}$

The above precondition is equivalent to

(the sequence $[c(1), c(2), \ldots, c(ic - 1)]$ is a permutation of the sequence $[a(1), a(2), \ldots, a(ia - 1), b(1), b(2), \ldots, b(ib - 1)]$)

which is $I4(ia, ib, ic, c)$. Referring to the definition of $S1$ and theorem 3.6 for a sequence of statements and combining terms, we have

$\{I4(ia, ib, ic, c)\}\ S1\ \{I4(ia, ib, ic + 1, c)\}$

The condition ($I4(ia, ib, ic, c)$ **and** $B1$ **and** $B2$) is a subset of $I4(ia, ib, ic, c)$ and is, therefore, also a precondition under $S1$. ■

Lemma 5.12: A precondition of $I4$ under $S2$ is as follows:

$\{I4(ia, ib, ic, c)$ **and** $B1$ **and not** $B2\}\ S2\ \{I4(ia, ib, ic + 1, c)\}$

Proof (sketch): The proof of this lemma is analogous to that of lemma 5.11. ■

Lemma 5.13: $\{I4(ia, ib, ic, c)$ **and** $B1\}\ S3\ \{I4(ia, ib, ic + 1, c)\}$

Proof (sketch): Analogously to the proof of lemma 5.3, this lemma follows from the above two lemmata and theorem 3.4 for the **if** statement. ■

Lemma 5.14: The condition $I4(ia, ib, ic, c)$ is a loop invariant, i.e.

$\{I4(ia, ib, ic, c)$ **and** $B1\}\ S5\ \{I4(ia, ib, ic, c)\}$

Proof (sketch): Analogously to the proof of lemma 5.4, this lemma follows from the lemmata 5.10 and 5.13 and theorem 3.6 for a sequence of statements. ■

Lemma 5.15: A precondition of ($I5$ **and** $I6$) under $S4$ is as follows:

$\{I5(ia, ic + 1, c)$ **and** $I6(ib, ic + 1, c)\}\ S4\ \{I5(ia, ic, c)$ **and** $I6(ib, ic, c)\}$

Proof: This lemma follows directly from the application of the theorem for the assignment statement. ■

Lemma 5.16: If the array a is in ascending sequence (see the precondition of the entire program segment given in the beginning of this Section 5.8.0), then a precondition of ($I5$ **and** $I6$) under $S1$ is as follows:

$\{I5(ia, ic, c)$ **and** $I6(ib, ic, c)$ **and** $B1$ **and** $B2\}$

$S1\ \{I5(ia, ic + 1, c)$ **and** $I6(ib, ic + 1, c)\}$

Proof: Applying the theorem for the assignment statement, we obtain

$\{I5(ia + 1, ic + 1, c)$ **and** $I6(ib, ic + 1, c)\}$

$ia := ia + 1\ \{I5(ia, ic + 1, c)$ **and** $I6(ib, ic + 1, c)\}$

By the definition of $I5$,

$I5(ia + 1, ic + 1, c)$

$= [(1 < ic + 1)$ **and** $(ia + 1 \leq na) \Rightarrow c(ic) \leq a(ia + 1)]$

Applying the theorem for the assignment statement again, we obtain

$\{(1 < ic + 1)$ **and** $(ia + 1 \leq na) \Rightarrow a(ia) \leq a(ia + 1)\}$

$c(ic) := a(ia)\ \{I5(ia + 1, ic + 1, c)\}$

Because the array a is in ascending sequence, the precondition above is always true, i.e.

$\{\text{true}\}\ c(ic) := a(ia)\ I5\{ia + 1, ic + 1, c)\}$

and, therefore, trivially

$\{I5(ia, ic, c)$ **and** $B1$ **and** $B2\}\ c(ic) := a(ia)\ \{I5(ia + 1, ic + 1, c)\}$

Similarly, by the definition of $I6$,

$I6(ib, ic + 1, c)$

$= [(1 < ic + 1)$ **and** $(ib \leq nb) \Rightarrow c(ic) \leq b(ib)]$

Applying the theorem for the assignment statement again, we obtain

$\{(1 < ic + 1)$ **and** $(ib \leq nb) \Rightarrow a(ia) \leq b(ib)\}$

$c(ic) := a(ia)\ \{I6(ib, ic + 1, c)\}$

By applying the definition of the implication function ($\Rightarrow$), the precondition above can be simplified and we have

$\{(ic \leq 0)$ **or** $(ib > nb)$ **or** $(a(ia) \leq b(ib))\}$
$c(ic) := a(ia)\ \{I6(ib, ic + 1, c)\}$

Because a subset of a precondition is a precondition, we can simplify the above to

$\{(ib > nb)$ **or** $(a(ia) \leq b(ib))\}\ c(ic) := a(ia)\ \{I6(ib, ic + 1, c)\}$

Examining the expansion and simplification of ($B1$ **and** $B2$) given in the proof of lemma 5.1, we see that

$(B1$ **and** $B2) \Rightarrow [(ib > nb)$ **or** $(a(ia) \leq b(ib))]$

so that

$\{B1$ **and** $B2\}\ c(ic) := a(ia)\ \{I6(ib, ic + 1, c)\}$

and

$\{I6(ib, ic, c)$ **and** $B1$ **and** $B2\}\ c(ic) := a(ia)\ \{I6(ib, ic + 1, c)\}$

By applying theorem 3.0, we can combine this with

$\{I5(ia, ic, c)$ **and** $B1$ **and** $B2\}\ c(ic) := a(ia)\ \{I5(ia + 1, ic + 1, c)\}$

which was shown above to be true, to obtain

$\{I5(ia, ic, c)$ **and** $I6(ib, ic, c)$ **and** $B1$ **and** $B2\}$

$c(ic) := a(ia)\ \{I5(ia + 1, ic + 1, c)$ **and** $I6(ib, ic + 1, c)\}$

By applying theorem 3.6 for a sequence of statements, we can combine the above with

$\{I5(ia + 1, ic + 1, c)$ **and** $I6(ib, ic + 1, c)\}$

$ia := ia + 1\ \{I5(ia, ic + 1, c)$ **and** $I6(ib, ic + 1, c)\}$

(see the first step in the proof of this lemma) to obtain

$\{I5(ia, ic, c)$ **and** $I6(ib, ic, c)$ **and** $B1$ **and** $B2\}$

$S1\ \{I5(ia, ic + 1, c)$ **and** $I6(ib, ic + 1, c)\}$ ■

Lemma 5.17: If the array b is in ascending sequence (see the precondition of the entire program segment given in the beginning of this Section 5.8.0), then a precondition under $S2$ is as follows:

$\{I5(ia, ic, c)$ **and** $I6(ib, ic, c)$ **and** $B1$ **and not** $B2\}$

$S2\ \{I5(ia, ic + 1, c)$ **and** $I6(ib, ic + 1, c)\}$

Proof (sketch): This lemma is proved in the same manner as lemma 5.16 above. Additionally, the obvious fact that $[b(ib) < a(ia)] \Rightarrow [b(ib) \leq a(ia)]$ is used. ■

Lemma 5.18: If the arrays a and b are in ascending sequence (see the precondition of the entire program segment given in the beginning of this Section 5.8.0), then a precondition of ($I5$ **and** $I6$) under $S3$ is as follows:

$\{I5(ia, ic, c)$ **and** $I6(ib, ic, c)$ **and** $B1\}$

$S3\ \{I5(ia, ic + 1, c)$ **and** $I6(ib, ic + 1, c)\}$

Proof (sketch): Analogously to the proof of lemma 5.3, this lemma follows from the above two lemmata and theorem 3.4 for the **if** statement. ■

Lemma 5.19: If the arrays a and b are in ascending sequence (see the precondition of the entire program segment given in the beginning of this Section 5.8.0), then the condition $[I5(ia, ic, c)$ **and** $I6(ib, ic, c)]$ is a loop invariant, i.e.

$\{I5(ia, ic, c)$ **and** $I6(ib, ic, c)$ **and** $B1\}$
$S5\ \{I5(ia, ic, c)$ **and** $I6(ib, ic, c)\}$

Proof (sketch): Analogously to the proof of lemma 5.4, this lemma follows from the lemmata 5.15 and 5.18 and theorem 3.6 for a sequence of statements. ■

Lemma 5.20: $\{I7(ic + 1, c)\}\ S4\ \{I7(ic, c)\}$

Proof: This lemma follows directly from the application of the theorem for the assignment statement. ■

Lemma 5.21: A precondition of $I7$ under $S1$ is as follows:

$\{I7(ic, c)$ **and** $I5(ia, ic, c)$ **and** $B1$ **and** $B2\}\ S1\ \{I7(ic + 1, c)\}$

Proof: Applying the theorem for the assignment statement, we obtain

$\{I7(ic + 1, c)\}\ ia := ia + 1\ \{I7(ic + 1, c)\}$

By the definition of $I7$,

$I7(ic + 1, c)$

$=$ [for all integers i such that $1 \leq i < ic, c(i) \leq c(i + 1)$]

Rewriting to isolate the reference to $c(ic)$, we have

$I7(ic + 1, c)$

$=$ $[((1 < ic) \Rightarrow (c(ic - 1) \leq c(ic)))$
and (for all integers i such that $1 \leq i < ic - 1, c(i) \leq c(i + 1))$]

Applying the theorem for the assignment statement again, we obtain

$\{((1 < ic) \Rightarrow (c(ic - 1) \leq a(ia)))$

and (for all integers i such that $1 \leq i < ic - 1, c(i) \leq c(i + 1))\}$

$c(ic) := a(ia)\ \{I7(ic + 1, c)\}$

Because a subset of a precondition is a precondition, we may modify the above to become

$\{((1 < ic) \Rightarrow (c(ic - 1) \leq a(ia)))$

and (for all integers i such that $1 \leq i < ic - 1, c(i) \leq c(i + 1))$

and $B1$ **and** $B2\}$

$c(ic) := a(ia)\ \{I7(ic + 1, c)\}$

In the proof of lemma 5.1 it was shown that

$(B1$ **and** $B2) \Rightarrow (ia \leq na)$

Thus, we can rewrite the precondition under the statement $(c(ic) := a(ia))$ above to obtain

$\{((1 < ic)$ **and** $(ia \leq na) \Rightarrow (c(ic - 1) \leq a(ia)))$

and (for all integers i such that $1 \leq i < ic - 1, c(i) \leq c(i + 1))$

and $B1$ **and** $B2\}$

$c(ic) := a(ia)\ \{I7(ic + 1, c)\}$

which is

$\{I7(ic, c)$ **and** $I5(ia, ic, c)$ **and** $B1$ **and** $B2\}\ c(ic) := a(ia)\ \{I7(ic + 1, c)\}$

Referring to the definition of $S1$ and theorem 3.6 for a sequence of statements, we can combine the above with

$\{I7(ic + 1, c)\}\ ia := ia + 1\ \{I7(ic + 1, c)\}$

(see the first step in the proof of this lemma) to obtain

$\{I7(ic, c)$ **and** $I5(ia, ic, c)$ **and** $B1$ **and** $B2\}\ S1\ \{I7(ic + 1, c)\}$ ■

Lemma 5.22: A precondition under $S2$ is as follows:

$\{I7(ic, c)$ **and** $I6(ib, ic, c)$ **and** $B1$ **and not** $B2\}\ S2\ \{I7(ic + 1, c)\}$

Proof (sketch): This lemma is proved in the same manner as lemma 5.21. ■

Lemma 5.23: A precondition under $S3$ is as follows:

$\{I7(ic, c)$ **and** $I5(ia, ic, c)$ **and** $I6(ib, ic, c)$ **and** $B1\}\ S3\ \{I7(ic + 1, c)\}$

Proof (sketch): First, strengthen the preconditions proved in lemmas 5.21 and 5.22 by **and**ing them with the terms $I6$ and $I5$ respectively. Analogously to the proof of lemma 5.3, this lemma then follows from theorem 3.4 for the **if** statement. ■

Lemma 5.24: The condition $[I7(ic, c)$ **and** $I5(ia, ic, c)$ **and** $I6(ib, ic, c)]$ is a loop invariant, i.e.

$\{I7(ic, c)$ **and** $I5(ia, ic, c)$ **and** $I6(ib, ic, c)$ **and** $B1\}$

$S5\ \{I7(ic, c)$ **and** $I5(ia, ic, c)$ **and** $I6(ib, ic, c)]$

Proof: Analogously to the proof of lemma 5.4, it follows from the lemmata 5.20 and 5.23 and the theorem 3.6 for a sequence of statements that

$\{I7(ic, c)$ **and** $I5(ia, ic, c)$ **and** $I6(ib, ic, c)$ **and** $B1\}\ S5\ \{I7(ic, c)\}$

Applying theorem 3.0 for the intersection of preconditions and postconditions to the above and the result of lemma 5.19, we obtain the thesis of this lemma. ■

Theorem 5.0: The condition

$I1(ia)$ **and** $I2(ib)$ **and** $I3(ia, ib, ic)$ **and** $I4(ia, ib, ic, c)$

and $I5(ia, ic, c)$ **and** $I6(ib, ic, c)$ **and** $I7(ic, c)$

is a loop invariant, that is,

$\{I1(ia)$ **and** $I2(ib)$ **and** $I3(ia, ib, ic)$ **and** $I4(ia, ib, ic, c)$

and $I5(ia, ic, c)$ **and** $I6(ib, ic, c)$ **and** $I7(ic, c)$ **and** $B1\}$

$S5$

$\{I1(ia)$ **and** $I2(ib)$ **and** $I3(ia, ib, ic)$ **and** $I4(ia, ib, ic, c)$

and $I5(ia, ic, c)$ **and** $I6(ib, ic, c)$ **and** $I7(ic, c)\}$

Proof (sketch): Apply theorem 3.0 for the intersection of preconditions and postconditions to the results of lemmata 5.4, 5.9, 5.14 and 5.24. ■

This completes the proof that the program segment is partially correct, i.e. that if it yields a result, that result fulfills the stipulated postcondition. To prove this, we have shown that the initialization establishes the truth of the loop invariant, that the loop invariant and the termination condition (the negation of the loop condition) together imply the postcondition and that the body of the loop preserves the truth of the loop invariant.

The last step mentioned – showing that the body of the loop preserves the truth of the loop invariant – was the most complex step. In practice, it is almost always the longest part of the proof of correctness of a loop. Therefore, the reader should at this point review the structure of this proof. Refer especially to the diagram immediately preceding lemma 5.0.

Fig. 5.1 reviews and summarizes the structure of the above proof. In particular, it highlights the functional forms of the pre- and postconditions encountered at key points in the body of the loop. The condition $P[ia, ib, ic, (c(1),\ldots\ c(ic-1))]$ represents the loop invariant being considered. In lemmata 5.0–5.4, P represents ($I1$ **and** $I2$); in lemmata 5.5–5.9, P represents $I3$, etc. (see the table following lemma 5.4). Notice how each precondition is derived from the corresponding postcondition by the application of the appropriate retrogressive proof rule for each assignment and **if** statement.

Finally, we must show that the initial data environment is in the domain of the program segment. An important part of such a demonstration is a proof that the loop terminates. This can be shown in several essentially equivalent ways. Perhaps the simplest is to note that the terms $I1$, $I2$ and $I3$ of the loop invariant imply that

$$1 \leq ic \leq na + nb + 1$$

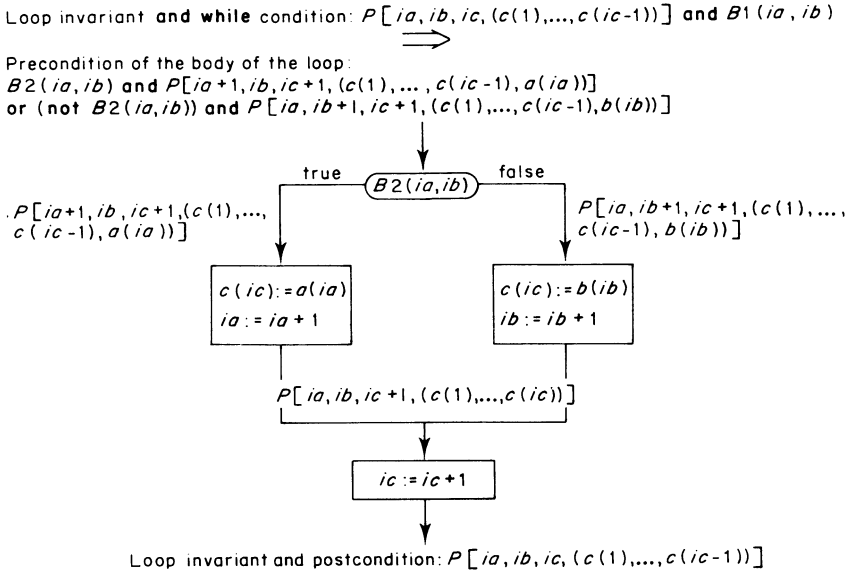


Fig. 5.1 Functional forms in the proof of correctness of the subprogram for merging two sorted arrays

Each execution of the body of the loop increases the value of ic by 1. The initial value of ic is 1. Thus the loop must terminate after its body has been executed at most (na + nb) times. In fact, it will be executed exactly (na + nb) times.

This argument can be expressed more formally by applying the definitions of Section 2.2 and considering the length of the computational history d^* (a sequence of data environments) at each point in the executional process. Each time the body of the loop is executed, ic is incremented by one and three individual statements are executed which contribute to the computational history. It follows, therefore, that the condition

$$\text{length}(d^*) = \text{length}(d0^*) + 1 + 3 * (ic - 1)$$

is a loop invariant, that is, is satisfied before and after each execution of the body of the loop. The function length(·) by definition maps its argument, a sequence of data environments, into an integer, the number of data environments in that sequence. The sequence d^* of data environments is the computational history developed up to the point in question in the execution of our subprogram. When evaluating the above condition, the

value of ic in the data environment last(d^*) is to be used. The sequence $d0^*$ of data environments is the previously developed computational history to which our subprogram is applied.

Combining this equation with the upper bound on ic above and taking into consideration the data environment appended to the computational history upon termination of the loop (see Definitions 2.26 and 2.34), we obtain

$$\text{length}(d^*) - \text{length}(d0^*) \leq 2 + 3 * (na + nb)$$

Noting that the length of our subprogram's contribution to the computational history is length(d^*) − length($d0^*$), we see that this length – and hence the execution time of the subprogram – is bounded (finite). Furthermore, it increases at most linearly with ($na + nb$). One says that the time complexity of the subprogram is 'of the order of ($na + nb$)', written O($na + nb$) (Baber, 1982, pp. 81–2, 177–80).

In addition, one must, strictly speaking, show that each execution of each statement yields a defined result. If ia, ib and ic are declared to be numerical variables with a sufficiently large range, the corresponding assignment statements will always yield a defined result. Similarly, the arrays a, b and c must be suitably declared, with sufficiently large ranges for their subscripts.

The condition $B2$ in the **if** statement can, depending upon conventions of the implementation of the actual system used, be quite problematic. Consider the situations in which $ia = na + 1$ or $ib = nb + 1$. In these cases, the **if** condition refers to the array variables $a(na + 1)$ and $b(nb + 1)$ respectively, variables which may not exist and whose values are, in any event, irrelevant.

Some real systems will, when evaluating ($ib > nb$) or ($ia \leq na$), recognize that the value of the subexpression ($a(ia) \leq b(ib)$) has no effect on the value of the **if** condition in the special situations identified above. These systems will bypass the process of evaluating the potentially problematic subexpression and no difficulty will arise. The given program segment will yield a defined result when executed on such a system. These systems, in effect, evaluate a logical expression such as (true **or** undefined) as true and (false **and** undefined) as false.

Other real systems exist, however, which evaluate each term of the **if** condition, including the potentially problematic one, before combining the intermediate results to determine the value of the entire expression. These systems will typically abort execution of the program when an array variable is referenced and the value of the subscript expression is 'out of range'. These systems, in effect, evaluate a logical expression such as (true **or** undefined) as undefined and (false **and** undefined) as undefined. Consequently, the result of executing this program on such a system is not defined.

The **if** statement must be rewritten so that the subexpression ($a(ia) \leq b(ib)$) is not evaluated in the problematic special cases. This can be done easily, but results in an **if** construction with a somewhat cumbersome appearance.

Note that the complexity (amount of detail) in this proof arose not just from the length or intricacy of the program segment being analyzed. The table after lemma 5.4 suggests that the amount of detail in such a proof is proportional to the product of the complexity of the specification (in particular, of the number of terms in the loop invariant) and the structural complexity of the program segment (in particular, the number of fundamental constructs constituting it).

Interaction between terms of the loop invariant increases further the logical complexity of a proof, in particular of the individual lemmata. This type of interaction arose in the above example in the case of the term $I7$. The invariance of both terms $I5$ and $I6$ was a hypothesis essential to the proof of the invariance of $I7$.

This is a general result: the complexity of a proof of correctness depends not only, not even primarily, upon the length or complexity of the *algorithm* or program performing the task in question. It strongly depends also upon the complexity of the *specification* of the task the algorithm performs and the interrelationships among the parts of that specification as well as between them and the components of the algorithm.

After the programmer has specified the pre- and postconditions of this subprogram and has proved it correct, others may use this subprogram without having to reprove its correctness each time they use it. They need only ensure that its precondition is satisfied before calling it and that the subsequent parts of the calling program assume only that the called subprogram's postcondition is fulfilled. The proof of correctness of the calling program will refer to the already proved theorem of correctness of the called subprogram – represented by its pre- and postconditions.

5.8.1 A recursive procedure

Consider a procedure named '*factorialprocedure*' defined as follows:

```
procedure factorialprocedure:
if n = 0
then declare (factorialvalue, ℤ, 1)
else declare (n, ℤ, n − 1)
     call factorialprocedure
     release n
     factorialvalue := n * factorialvalue
endif
endprocedure
```

The function of this procedure is formulated in the form of a mathematical proposition or theorem.

Theorem 5.1: If a data environment $d0$ contains a numerical variable n whose value $n(d0)$ (= valvar(n, $d0$)) is a non-negative integer, then the result of applying the above procedure '*factorialprocedure*' to $d0$ is the data environment $d1$, where

$$d1 = (\textbf{call}\ \textit{factorialprocedure})(d0) = [(\textit{factorialvalue}, \mathbb{Z}, n(d0)!)]\ \&\ d0$$

Proof: We prove this theorem by induction on $n(d0)$. If $n(d0) = 0$, then the conclusion of the theorem follows directly from the definitions of the **if** and declare statements.

We now assume that the theorem is true for $n(d0) = k$ and show that it is true for $n(d0) = k+1$. When $n(d0) = k+1 \neq 0$, the statement '**call** *factorialprocedure*' is equivalent to the **else** part of the **if** statement above. The result of executing the declare statement is the data environment

$$da = [(n, \mathbb{Z}, k)]\ \&\ d0$$

By the inductive assumption, the proposition of this theorem applies when $n(d0) = k$. The result of executing the call statement upon da is, therefore, the data environment

$$\begin{aligned} db &= [(\textit{factorialvalue}, \mathbb{Z}, n(da)!)]\ \&\ da \\ &= [(\textit{factorialvalue}, \mathbb{Z}, k!), (n, \mathbb{Z}, k)]\ \&\ d0 \end{aligned}$$

The result of executing the release statement is the data environment

$$dc = [(\textit{factorialvalue}, \mathbb{Z}, k!)]\ \&\ d0$$

After executing the assignment statement, the final data environment is

$$\begin{aligned} (\textbf{call}\ \textit{factorialprocedure})(d0) &= [(\textit{factorialvalue}, \mathbb{Z}, n(d0)*k!)]\ \&\ d0 \\ &= [(\textit{factorialvalue}, \mathbb{Z}, n(d0)!]\ \&\ d0 \end{aligned}$$

Thus, the conclusion of the theorem holds provided that the value of the variable n in the initial data environment $d0$ is 0 or 1 or 2, etc., i.e. provided that it is a non-negative integer. ■

This example illustrates how declare and release statements can be used to pass a parameter to a procedure without causing the loss of the values of parameters from prior calls. It also demonstrates a common and generally useful way of handling declare and release statements in a proof of correctness.

Chapter 6

The construction of correct programs

Weil ein Vers dir gelingt in einer gebildeten Sprache,
Die für dich dichtet und denkt, glaubst du schon Dichter zu sein.
– Friedrich Schiller

Science surpasses the old miracles of mythology.
– Ralph Waldo Emerson

God made the integers; all else is the work of man.
– Leopold Kronecker

In Chapter 5, we analyzed short program segments and proved that they fulfilled certain specifications and exhibited certain characteristics. While such analytical steps are an important part of the engineering design process, they are not really representative of it for the following reasons. Firstly, the examples in Chapter 5 are relatively short; they were isolated and extracted from a larger real environment in order to introduce and illustrate the applicable analytical procedures. Secondly, and more importantly, in the proper practice of software engineering, one does not typically develop a proof of correctness for an already existing, finished program. Instead, the software engineer develops the program and its proof of correctness together, adding detail to each concurrently and step by step.

The examples in Sections 6.1 through 6.10 below are more typical of actual design tasks arising in the development of computer programs. While they are not large, they do illustrate the construction of both subsidiary procedures and high level control programs. These examples are not artificial – all are typical of practically useful programs. Most, in fact, are extracts from application software written for and used in productive commercial systems.

The examples in this chapter cover a variety of design problems and application areas. Sections 6.1 through 6.5 illustrate the design of low

level subprograms and their interfaces with the procedures calling them. These subprograms perform specific tasks of limited scope and of a somewhat technical nature. In Sections 6.6 through 6.10, on the other hand, we design an entire small program and higher level control procedures and structures for larger systems. Here, the emphasis is on the coordination of many lower level subprograms which perform quite different individual tasks.

The example in Section 6.1 deals with designing the subprogram already analyzed in Section 5.8.0 and thus serves as a bridge between the topics of Chapter 5 (analysis) and Chapter 6 (design). The subjects of Sections 6.2 and 6.3 are similar types of subprograms involving searching and rearranging an array. In Section 6.4 a recursive procedure is designed which uses (calls) an already defined subprogram. The interface between the two subprograms and how it is considered and handled in the proof of correctness of the calling subprogram are illustrated. Section 6.5 deals with searching and updating a linked linear list and involves the interaction of three closely related procedures and their common data structure.

In Section 6.6, the application of ideas presented in earlier chapters to the problem of coordinating several different subprograms within the scope of part of one application program is illustrated. In Sections 6.7 and 6.8 we derive a simple, flexible and general solution to the problem of coordinating several subprograms in order to print a report correctly – an old, often encountered task for which many an erroneous program has been written and is still in use. In Section 6.9 a small program is designed in its entirety. In Section 6.10, the last in the chapter, we design the main control program for a moderately large system consisting of a number of different types of subprograms and data files.

Before discussing the examples in detail, it is advantageous to summarize a number of useful guidelines for the software designer. They derive from the theoretical results presented in the previous chapters as well as from design experience. These guidelines are presented in Section 6.0 below.

6.0 Guidelines for the designer

Both theory and practical experience suggest that it is desirable to observe the following guidelines when designing software, programs and segments thereof. These guidelines should not be viewed as absolute, inviolable laws of software development, but rather as strong suggestions. The designer should deviate from them only with compelling reason and after careful consideration.

0 Identify restrictions imposed by the target programming language.
1 Subdivide the system's functions into small, hierarchically organized units.

2 Define the postcondition and the precondition of each program segment before starting to design its code.
3 Specify the loop invariant before starting to write the body of any loop.
4 Write general pseudocode, then develop it into precise, unambiguous pseudocode and finally translate the latter into the target programming language.
5 Develop precise pseudocode and its correctness proof hand in hand.
6 When developing a detailed proof of correctness, work from the postcondition backwards to the precondition.
7 Design and prove the correctness of each subprogram as an isolated entity.
8 Use a small number of simple but general purpose constructs.

These guidelines are discussed individually and in more detail below.

6.0.0 Identify restrictions imposed by the target programming language

The target programming language selected will usually impose certain restrictions on the structure of the program or software system to be designed. By taking such restrictions into account early in the planning phase, one can avoid considerable effort and difficulty later, especially in the coding phase.

Typically, such restrictions pertain to declaring and releasing variables, to the multiple use of variable names, to calling procedures, to passing parameters to and from procedures and to file operations (I/O).

6.0.1 Subdivide the system's functions into small, hierarchically organized units

In any area of endeavour, not just software development, complexity is most effectively mastered by pursuing the simple and simplifying strategy of 'divide and conquer'. The task or system should be subdivided in a manner which facilitates understanding the entire system, permits different people to develop the different parts more or less independently, minimizes the effects of design changes to one part on other parts and eases the task of proving the system's programs correct.

These goals can usually be best achieved by subdividing the system hierarchically into modules, subprograms, procedures, etc. The function of each unit is subdivided into a small number of subfunctions which interact in simple ways and which exchange relatively little data with one another. The subfunctions within one subdivision should be logically closely related while the subfunctions of different subdivisions should be logically relatively independent of one another. The interfaces should involve as few variables

as possible. The specification of each subdivision should consist of a logically simple precondition, a logically simple postcondition and a rule for mapping the initial data environment into the final data environment. Often this rule is implied by the precondition and the postcondition, in which case the rule need not be stated separately and explicitly.

6.0.2 Define the postcondition and the precondition of each program segment before starting to design its code

The postcondition defines the set of final data environments, i.e. the range of the program segment in question. The postcondition also defines, at least partially, the task which the program segment is to accomplish (its output). The precondition defines the set of valid initial data environments, i.e. the domain of the program segment in question (its input). The designer cannot meaningfully begin to design the program segment, write its code, etc., until both of these points have been clarified.

Most frequently, the designer will begin by stating the postconditions and preconditions of interacting program segments in rather general terms. He will then refine them, adding detail and making them more precise.

The final version of the postcondition and the precondition must be unambiguous and mathematically precise. As long as this requirement is satisfied, they may be written in any suitable form and in any suitable language, although they will normally be expressed in mathematical terms or in a very similar form. Only after reaching this point will the designer start to code the program segment in question.

Often, one program or subprogram will reference data (variables) which other program segments also reference. Typically, such data can be guaranteed to be correct and consistent only if certain conditions are met. Because these conditions relate to the data and are not specific to any particular programs or subprograms, it is meaningful to think of them as 'data invariants'. The data invariants must be explicitly considered when designing any program or part thereof which can modify the subject data. The data invariants will become pre- and postconditions of most subprograms which reference the subject data.

Frequently the designer of a program will find it useful to specify that a certain condition be met at many points in the program. Such a condition, which may be thought of as a 'program invariant', will be both a precondition and a postcondition of many parts of the program in question – in particular, of procedures.

The various postconditions and preconditions constitute an important part of the documentation on each program segment. Together, they represent a theorem which the program segment satisfies. This theorem will be used

(referenced) in the proof of correctness of any other program using (e.g. calling) the subject program segment or procedure.

Postconditions and preconditions are to the software engineer much as voltages and currents are to the electrical engineer, as forces and stresses are to the structural engineer. These represent the key entities with which the engineer is concerned when designing his program, electrical system, structure of a building or bridge, etc.

6.0.3 Specify the loop invariant before starting to write the body of any loop

This guideline is essentially the same as 'that in Section 6.0.2 above, reformulated for the loop. It is so important, however, that repeating and emphasizing it is warranted.

The loop invariant expresses the most important design decision regarding a loop. It is the key to understanding the loop, its function and its operation. It is the key to proving the correctness of the loop. It is, therefore, an indispensable part of the documentation of a program segment containing a loop.

The loop invariant is a generalization of the precondition and the postcondition of the loop. In other words, the loop's precondition is a special case of the loop invariant. The loop's postcondition is also a special case of the loop invariant.

The software engineer uses the loop invariant as a guide to writing the code for the loop. He cannot, therefore, begin to write that code until he has specified the loop invariant unambiguously, precisely and completely.

6.0.4 Write general pseudocode, then develop it into precise, unambiguous pseudocode and finally translate the latter into the target programming language

The process of designing and coding a program segment consists of three steps:

1 Write an outline of the program segment in a general form of 'pseudocode'. At this stage, the program is specified only coarsely. This version of the program may be very condensed and much detail may be missing. It is in general incomplete, imprecise and ambiguous. This version of the program is only of temporary value to the designer.
2 Refine the first, general version of the program segment into detailed, complete, precise and unambiguous pseudocode. The programming constructs presented in Sections 2.1.0 through 2.1.6 inclusive (assign-

ment, **if**, sequence, **while**, declaration and release constructs and the procedure call without parameters) constitute a suitable pseudocode language for this purpose. Avoid constructions and combinations of statements which cannot be translated easily and directly into the target programming language.

3 Translate the second version of the program segment into the target programming language.

In step 1, a combination of imprecise natural language and more systematic terminology may be used for writing the general pseudocode as the designer deems most appropriate for his purpose. References to already defined preconditions, postconditions and invariants often facilitate the subsequent refinement of the pseudocode (e.g. 'assuming that condition Q is true, establish the truth of P', 'decrease the size of the unprocessed data region while maintaining the truth of condition P', etc.).

Because the result of step 1 above is incomplete and ambiguous, steps 1 and 2 should be performed by the same person, who should be a qualified software engineer. Step 3 is a relatively straightforward, more mechanistic process and can, therefore, be performed by a less qualified coding technician. Step 3 is analogous to the work of a technician in an engineering design team or even of a draftsman in an architectural or structural engineering project.

The detailed proof of correctness should be prepared for the second version of the program (i.e. for the result of step 2 above). By doing so, one avoids complications resulting from technicalities and idiosyncracies of the target language which are logically irrelevant to the algorithm. Input/output is a common example, see Section 4.1.

The entire programming process involves two very different levels of abstraction: one relating to the algorithmic solution to the application problem at hand and the second relating to the specific technicalities of the target programming language, its restrictions, idiosyncracies, etc. The main goal of the approach outlined above is to separate these two areas of concern. Steps 1 and 2 are concerned with the logic of the algorithm being programmed but not with the specific technicalities of the target language. Step 3 begins with a complete solution at the algorithmic level and translates that into an operational program. The person performing step 3 is concerned almost exclusively with the technicalities of the target language and not with logical aspects of the algorithm.

The person performing step 3 need only ensure that the results of steps 2 and 3 are logically equivalent. He need not concern himself explicitly with proving the correctness of the final program in terms of the original specifications.

6.0.5 Develop precise pseudocode and its correctness proof hand in hand

When the general form of the program is developed (see step 1 in Section 6.0.4 above), the designer should have a rough sketch of the proof of correctness in mind. The general pseudocode and the sketch of the correctness proof should be refined, detailed and developed into precise pseudocode and a logically complete proof together, step by step.

The proof should be structured and subdivided in the same way that the program segment itself is structured and subdivided.

A logically complex or extensive proof should be broken down into a number of simple, structurally identical or similar steps (see e.g. Section 5.8.0). Usually, a proof involving even a large number of simple lemmata exhibiting a repetitive pattern is to be preferred over a proof involving many structurally unique steps with logically complex interactions.

6.0.6 When developing a detailed proof of correctness, work from the postcondition backwards to the precondition

For each fundamental construct, a retrogressive proof rule exists which can be applied conveniently in practice to yield meaningful results. These proof rules may be applied in a largely straightforward manner to a given postcondition to derive a precondition, even a complete precondition when required. See Chapter 3, especially Section 3.9, Summary of the most important proof rules.

An important exception to this guideline arises in the case of a release statement and a postcondition referring (explicitly or implicitly) to the variable name appearing in the release statement. Such a postcondition typically states essentially that a previous data environment (e.g. one existing before the corresponding declaration statement was executed) has been restored. In such cases, it is usually more convenient to start from the initial data environment and, working forward through the code, prove that the postcondition follows. Section 5.8.1 contains a good example of this approach..

When developing the proof of correctness of a program segment, the role of each statement in the proof and each statement's contribution to establishing the truth of the postcondition should be evident. If it is not, the designer should ask himself (a) if the statement is necessary at all, (b) if the postcondition is complete and (c) if the precondition is more restrictive than intended. If the statement is necessary or really desired, then it may be contributing to a hitherto unexpressed part of the postcondition (which should be explicitly added to the postcondition). Alternatively, the statement may deal with a situation which is excluded by the explicitly stated

precondition but which the designer intended to allow. In this case, the precondition should be explicitly relaxed (weakened) to include such situations which the program segment handles correctly. If the statement is not needed, it should, of course, be eliminated.

6.0.7 Design and prove the correctness of each subprogram as an isolated entity

Typically, the statement of a theorem is significantly simpler logically than its complete proof. One should take advantage of this fact by proving the correctness of a program segment once and using (referencing) the correctness theorem in the proofs of all other programs which use the program segment in question (e.g. by calling it as a procedure).

The postcondition and the precondition constitute a theorem which the program segment satisfies, i.e. its correctness theorem. The postcondition is the thesis of the theorem and the precondition is its hypothesis.

In effect, the correctness theorem for a program segment, once proved, becomes a proof rule for that program segment or a call to it.

6.0.8 Use a small number of simple but general purpose constructs

The difficulty of writing a program segment, the difficulty of understanding it and the intricacy and logical complexity of its correctness proof are related and typically roughly equal. All can be reduced by restricting oneself to a small number of simple but generally applicable and universally combinable constructs.

When complexity arises, the designer should limit it by isolating its source, by restricting it to closed, narrowly bounded regions and/or by subdividing the function giving rise to it.

In the design phase, avoid logically unnecessary constructs. Generalize or combine constructs which, although different, perform essentially the same logical function (e.g. the assignment statement and I/O statements, see Section 4.1).

6.1 Merging two sorted arrays (example)

We begin our examination of examples of program construction by considering the design phase of the problem analyzed in detail in Section 5.8.0, merging two sorted arrays.

Initially, two arrays a and b are given, with subscripts ranging from 1 to na and nb respectively. The values of na and nb are non-negative integers. Each array is already sorted, i.e.

for all integers i such that $1 \leq i < na$, $a(i) \leq a(i+1)$
and for all integers i such that $1 \leq i < nb$, $b(i) \leq b(i+1)$

This constitutes the precondition. It may be expressed in various equivalent ways, for example

$\mathbf{and}_{i=1}{}^{na-1}\ a(i) \leq a(i+1)$
$\mathbf{and}_{i=1}{}^{nb-1}\ b(i) \leq b(i+1)$

The goal of the program segment to be written is to merge the values from the two arrays a and b, forming an array c whose values are copied from a and b in such a way that c is in sequence. More precisely, the postcondition is:

the sequence $[c(1), c(2), \ldots, c(na+nb)]$ is a permutation (rearrangement of the terms) of the sequence $[a(1), a(2), \ldots, a(na), b(1), \ldots, b(nb)]$

and for all integers i such that $1 \leq i < na + nb$, $c(i) \leq c(i+1)$

whereby the program segment to be written is not permitted to modify any value in either array a or b.

The precondition can be represented diagrammatically as follows:

$\underline{|1 \qquad\qquad na|}$ array a
to be copied to c

$\underline{|1 \qquad\qquad nb|}$ array b
to be copied to c

array c $\underline{|1 \qquad\qquad na+nb|}$
to be copied from a and b

The corresponding diagram for the postcondition is:

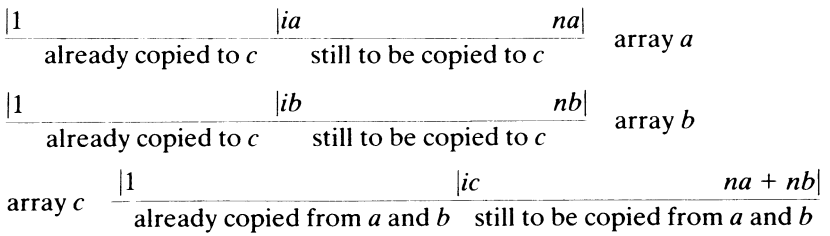
$\underline{|1 \qquad\qquad na|}$ array a
already copied to c

$\underline{|1 \qquad\qquad nb|}$ array b
already copied to c

array c $\underline{|1 \qquad\qquad na+nb|}$
already copied from a and b

We require that the values in the array c be in sequence.

It seems natural to copy the values from arrays a and b one by one to array c. This suggests a loop as the basic structure in our program segment. Generalizing the precondition and the postcondition above leads to the following diagrammatical representation of the loop invariant:



At such an intermediate stage of the merge, $(ia - 1)$ values have been copied from array a and $(ib - 1)$ values have been copied from array b. The sum,

$$(ia - 1) + (ib - 1)$$

must be equal to the number of values copied to array c, which is $(ic - 1)$. This requirement is term $I3$ of the complete loop invariant (see Section 5.8.0).

At an intermediate stage of the merge, the values already copied to array c must be in sequence. In order to ensure that the next element copied from a or b to c will not destroy the sequence of the array c, we must also require that

$$c(ic - 1) \leq a(ia)$$

and that

$$c(ic - 1) \leq b(ib)$$

Adding the appropriate conditions to cover the cases in which ia, ib or ic is beyond the range of the subscripts of the corresponding array (representing the cases that all values have been copied from a or b or that no value has yet been copied to c) leads to the terms $I5$ and $I6$ in the formal statement of the complete loop invariant (see Section 5.8.0).

The other terms of the loop invariant as given in Section 5.8.0 follow directly from the diagrammatical definitions of ia and ib above (e.g. $I1$ and $I2$) or from the postcondition applied to that portion of array c to which values have already been copied (e.g. $I4$ and $I7$).

We write the first version of our program segment in rather general terms as follows:

Establish the truth of the loop invariant; i.e. initialize ia, ib and ic so that the loop invariant reflects the precondition.
while uncopied values remain in a or b **do**
    Reduce the number of uncopied values in the arrays a and b while maintaining the truth of the loop invariant.
endwhile

Refining this very coarse pseudocode one step, we write:

Establish the truth of the loop invariant; i.e. initialize ia, ib and ic so that the loop invariant reflects the precondition.
while uncopied values remain in a or b **do**
    if value from a should be copied next
    then copy value from a
    else copy value from b
    endif
endwhile

The initialization follows directly from the diagrams above:

$$(ia, ib, ic) := (1, 1, 1)$$

The **while** condition also follows from the diagrams above:

$$(ia \leq na) \textbf{ or } (ib \leq nb)$$

If we had originally stated the **while** condition as 'values remain to be copied to c', we would have written the **while** condition precisely as

$$ic \leq na + nb$$

These two forms of the **while** condition are, under the conditions guaranteed by the loop invariant, logically equivalent, that is, one is true if and only if the other is true. Either is correct; they simply represent different views of one and the same terminal condition.

The initial part of the segment 'copy value from a' is obvious: $c(ic) := a(ia)$. After executing this statement, the values of ia and ic no longer satisfy the definitions implied in the diagram representing the loop invariant. This can be set right again by increasing ia and ic by one. The program segment 'copy value from a' then becomes

$$c(ic) := a(ia)$$
$$ia := ia + 1$$
$$ic := ic + 1$$

By symmetry, we write the corresponding program segment for 'copy value from b':

$$c(ic) := b(ib)$$
$$ib := ib + 1$$
$$ic := ic + 1$$

Using less intuition and more formalism, one can obtain this result in another way. The objective of the program is to increase ia, ib and ic so that $ia = na + 1$, $ib = nb + 1$ and $ic = na + nb + 1$, in which case the loop invariant implies the postcondition. We begin by writing

$ic := ic + 1$

This invalidates the loop invariant, in particular term *I*3 (see Section 5.8.0). The validity of the loop invariant can be reestablished in one of only three ways, e.g. by executing either

$ic := ic - 1$
$ia := ia + 1$

or

$ib := ib + 1$

We reject the first possibility because it exactly reverses the effect of our original statement, $ic := ic + 1$. Thus we are left with two possibilities for the latter part of our program segment:

$ia := ia + 1$	$ib := ib + 1$
$ic := ic + 1$	$ic := ic + 1$

Consider term *I*4 of the loop invariant (see Section 5.8.0):

> the sequence $[c(1), c(2), \ldots, c(ic - 1)$ is a permutation of the sequence $[a(1), a(2), \ldots, a(ia - 1), b(1), b(2), \ldots, b(ib - 1)]$

Deriving the precondition of *I*4 with respect to the pair of statements on the left above, we obtain

> the sequence $[c(1), c(2), \ldots, c(ic)]$ is a permutation of the sequence $[a(1), a(2), \ldots, a(ia), b(1), b(2), \ldots, b(ib - 1)]$

Our immediate goal is to reduce this to term *I*4 of the loop invariant. If $c(ic)$ were replaced by $a(ia)$ or vice versa, we could eliminate these terms to obtain an equivalent proposition, which would be identical in form with *I*4 above. This suggests prefixing either

$c(ic) := a(ia)$

or

$a(ia) := c(ic)$

to the pair of statements above. The statement of the postcondition forbids us to alter the value of any element of the array a, leaving us with only the first possibility above.

The same approach applied to the other pair of statements leads to the third, initial statement $c(ic) := b(ib)$.

The two possible segments are, therefore:

$c(ic) := a(ia)$	$c(ic) := b(ib)$
$ia := ia + 1$	$ib := ib + 1$
$ic := ic + 1$	$ic := ic + 1$

Combining the above, our complete program segment becomes

$(ia, ib, ic) := (1, 1, 1)$
while $(ia \leq na)$ **or** $(ib \leq nb)$ **do**
  if value from a should be copied next
  then $c(ic) := a(ia)$
    $ia := ia + 1$
    $ic := ic + 1$
  else $c(ic) := b(ib)$
    $ib := ib + 1$
    $ic := ic + 1$
  endif
endwhile

Only the **if** condition remains to be made more explicit. Intuitively, it is clear that the lesser of the two values $a(ia)$ or $b(ib)$ should be copied to the array c, provided, of course, that ia or ib respectively is in the appropriate range. Formally, this can be seen reasonably easily. If the greater of the two values were copied first, then this last value copied to c would be greater than the other value remaining in either $a(ia)$ or $b(ib)$. As a consequence, *I*5 or *I*6 would be violated. The lesser value would be copied to the array c later, i.e. an element of the array c with a greater subscript. Such action would clearly destroy the desired order of the values of the elements of the array c.

If the two values $a(ia)$ and $b(ib)$ are equal, it makes no difference which is copied to the array c.

Listing all possibilities in detail, we have:

1	$(ia > na)$ **and** $(ib > nb)$	all values from a and b already copied to c
2	$(ia \leq na)$ **and** $(ib > nb)$	only a contains values to be copied
3	$(ia > na)$ **and** $(ib \leq nb)$	only b contains values to be copied
4	$(ia \leq na)$ **and** $(ib \leq nb)$	both a and b contain values to be copied

Condition 1 above is the negation of the **while** condition, so cannot apply when the **if** condition is evaluated. We can, therefore, exclude it from further consideration.

In case 2, values to be copied to c remain only in a. This situation can be distinguished from the others (3 and 4) by the condition $(ib > nb)$.

In case 3, no values remain in a to be copied to c.

In case 4, the lesser of the two values $a(ia)$ and $b(ib)$ should be copied to array c. If they are equal, either may be copied next.

Thus, the value from a should be copied next if either

$(ib > nb)$ (no values remain in array b)

or

$(ia \leq na)$ **and** $(ib \leq nb)$ **and** $a(ia) \leq b(ib)$

Simplifying the above by applying one of the rules of Boolean algebra (in particular, $(X$ **or** $((\textbf{not } X)$ **and** $Y)) = (X$ **or** $Y))$, we obtain

$(ib > nb)$ **or** $((ia \leq na)$ **and** $(a(ia) \leq b(ib)))$

for the **if** condition and our final program segment becomes

```
(ia, ib, ic) := (1, 1, 1)
while (ia ≤ na) or (ib ≤ nb) do
    if (ib > nb) or ((ia ≤ na) and (a(ia) ≤ b(ib)))
    then c(ic) := a(ia)
         ia := ia + 1
         ic := ic + 1
    else c(ic) := b(ib)
         ib := ib + 1
         ic := ic + 1
    endif
endwhile
```

The two statements $ic := ic + 1$ can be brought outside the **if** statement to obtain the version of this program which was analyzed in Section 5.8.0.

At this point the designer would perform the analysis contained in Section 5.8.0 in order to verify rigorously and formally the correctness of the final version of the program designed above.

6.2 Searching a sorted array (example)

Let $k(i)$, for $i = first, first + 1, \ldots, last$, be a given array whose values are in ascending (more accurately, non-descending) order, i.e.

$k(first) \leq k(first + 1) \ldots \leq k(last)$

The values of *first* and *last* are integers with $first - 1 \leq last$. (If $first - 1 = last$, the given array is empty.)

Our task is to design a program segment which locates all values in k which are equal to the value of the given variable *skey*.

More precisely, the postcondition is

for all j such that $first \leq j \leq last$, $k(j) = skey \Leftrightarrow il \leq j \leq ih$
and $first \leq il$

and $il - 1 \leq ih$
and $ih \leq last$

where il and ih are calculated by the program segment to be designed. If $first = il$, $il - 1 = ih$ or $ih = last$, the corresponding range of subscript values is empty.

The first part of the postcondition above can be expressed in other equivalent forms, for example

$$\textbf{and}_{j=first}^{il-1}\, k(j) \neq skey$$

$$\textbf{and}_{j=il}^{ih}\, k(j) = skey$$

$$\textbf{and}_{j=ih+1}^{last}\, k(j) \neq skey$$

or, in view of the fact that the values of $k(.)$ are in sequence,

$$\textbf{and}_{j=first}^{il-1}\, k(j) < skey$$

$$\textbf{and}_{j=il}^{ih}\, k(j) = skey$$

$$\textbf{and}_{j=ih+1}^{last}\, k(j) > skey$$

The following diagram illustrates the postcondition:

first		*il*		*ih*		*last*
$k(i)<skey$		$k(i)=skey$		$k(i)>skey$		

Similarly, the precondition can be represented diagrammatically:

first		*last*
$k(i)?$		

It seems natural to locate the array values which are equal to *skey* by repeatedly comparing various individual array values with it and recording the results in a suitable way. This suggests a loop as the fundamental structure in our program segment. Generalizing the precondition and the postcondition above, we obtain the following diagram to represent the loop invariant.

We must distinguish between case 1

first	*a*	*il*	*ih*	*b*	*last*
$k(i)<skey$	$?(\leq)$	$k(i) = skey$	$?(\geq)$	$k(i)>skey$	

and case 2, when the central interval of values equal to *skey* is empty: known to be

first	*ih* *a*		*b* *il*	*last*
$k(i)<skey$	$?$		$k(i)>skey$	

In either case, b marks the lower boundary of the region in which $k(i) > skey$ (the 'greater' region). Similarly, in either case, a marks the upper

boundary of the 'less' region. The variable *il* marks the lower boundary of the equal region while *ih* marks the upper boundary of the equal region.

The unknown region consists of those subscript values *i* for which

$(a \leq i < il)$ **or** $(ih < i \leq b)$

In case 2, these two subregions are the same. This is one of the reasons for selecting the above form for case 2 among the various possibilities.

The program to be written will increase, never decrease, the values of *a* and *ih*. Correspondingly, it will decrease, never increase, the values of *b* and *il*. Thus, the unknown regions always decrease in size.

The fact that this form for case 2 never requires *ih* to be decreased or *il* to be increased is another important reason for selecting it.

Our first version of the searching loop is

Establish the truth of the loop invariant; i.e. initialize *a*, *b*, *il* and *ih* so that the loop invariant reflects the precondition.
while the unknown region is not empty **do**
    Reduce the size of the unknown region while maintaining the truth of the loop invariant.
endwhile

which we refine to

Establish the truth of the loop invariant; i.e. initialize *a*, *b*, *il* and *ih* so that the loop invariant reflects the precondition.
while the unknown region is not empty **do**
    Select some subscript value in the unknown region.
    Compare the value of the selected array element with *skey*.
    Modify *a*, *b*, *il* and/or *ih* to reflect the information gained from the above comparison, i.e. in order to reestablish the truth of the loop invariant.
endwhile

Initially, the region known to contain values equal to *skey* is empty, so case 2 of the loop invariant applies. The initialization is then:

$a := first$
$b := last$
$il := b + 1$
$ih := a - 1$

or, if these variables have not been declared or are to be declared anew within our program segment, we must write:

declare $(a, \mathbb{Z}, first)$
declare $(b, \mathbb{Z}, last)$

declare $(il, \mathbb{Z}, b + 1)$
declare $(ih, \mathbb{Z}, a - 1)$

In the above, $\mathbb{Z}$ represents the set of integers.

The selection of one of the above two forms of the initialization is a design decision relating not just to this program segment, but also to its interaction with its caller. It should be clarified with those responsible for the specification of the module we are designing, as the specification stated above is unambiguous in this respect.

The unknown region is not empty if and only if

$(a < il)$ **or** $(ih < b)$

(see above). This is, then, our **while** condition.

The comparison step is simple; we merely compare $k(j)$ with *skey* in one or more **if** statements, where *j* is any subscript value selected from the unknown region, i.e. $a \leq j < il$ or $ih < j \leq b$. Three situations can arise,

$k(j) < skey$
$k(j) = skey$

or

$k(j) > skey$

In each situation, we must consider both the case 1 and case 2 forms of the loop invariant.

If $k(j) < skey$ and case 1 of the loop invariant applies, the value of *j* must lie in the '? (≤)' region (see diagram above). We must increase the value of *a* to mark the newly determined upper boundary of the '<' region:

$a := j + 1$

If $k(j) < skey$ and case 2 of the loop invariant applies, we must increase the values of both *a* and *ih*:

$a := j + 1$
$ih := j$

It would be convenient if both of these program segments were identical. A first step in this direction can be made if we rewrite the code for case 1 to be:

$a := j + 1$
$ih := ih$

We stated above that the value of *ih* is increased, never decreased. This suggests rewriting the code for case 2 to be:

$a := j + 1$
$ih := \max(ih, j)$

(Note that in case 2, $j > ih$, so that $\max(ih, j) = j$.)

In case 1, $j < ih$, $\max(ih, j) = ih$, so that we may rewrite the code for case 1 to be:

$a := j + 1$
$ih := \max(ih, j)$

Our program segments for cases 1 and 2 are now identical for the situation in which $k(j) < skey$.

If $k(j) > skey$, symmetry leads us to write

$b := j - 1$
$il := \min(il, j)$

for both cases 1 and 2. The reader should verify this code by repeating the above argument, applying it to this situation.

If $k(j) = skey$, only the values of il and/or ih need be adjusted. The facts that the value of il may only be decreased and the value of ih may only be increased suggest the code

$il := \min(il, j)$
$ih := \max(ih, j)$

for any and all cases. The reader should verify that this is correct for each of the three situations:

case 1: '? ($\leq$)' region ($a \leq j < il$)
case 1: '? ($\geq$)' region ($ih < j \leq b$)

and

case 2: ($ih + 1 = a \leq j \leq b = il - 1$)

Combining the above, we refine our previous version of the program to be designed to become:

declare (a, $\mathbb{Z}$, *first*)
declare (b, $\mathbb{Z}$, *last*)
declare (il, $\mathbb{Z}$, $b + 1$)
declare (ih, $\mathbb{Z}$, $a - 1$)
while ($a < il$) **or** ($ih < b$) **do**
    Select a value for j such that ($a \leq j < il$) **or** ($ih < j \leq b$).
    if $k(j) < skey$
    then $a := j + 1$
        $ih := \max(ih, j)$

    else
    if $k(j) = skey$
    then $il := \min(il, j)$
        $ih := \max(ih, j)$
    else
    if $k(j) > skey$
    then $b := j - 1$
        $il := \min(il, j)$
    endif
    endif
    endif
endwhile

We must refine the statement selecting a value of j, in particular, we must decide whether it should be an assignment or declaration statement. The variable j is used only within the body of the **while** loop, so it is logical to declare it at the beginning and release it at the end of the body of the loop.

The result ('output') variables of our program are il and ih; they are mentioned in the postcondition. Therefore, we may not release them at the end of our program.

The variables a and b are not mentioned in the postcondition, but are used only internally. We may release them at the end of our program. If we declare them in the initialization, as we have done above, we should in fact, release them.

The third **if** condition will, of course, always be true at that point. Therefore, it and the following '**then**' can be omitted if desired.

Our program then becomes:

declare (a, $\mathbb{Z}$, *first*)
declare (b, $\mathbb{Z}$, *last*)
declare (il, $\mathbb{Z}$, $b + 1$)
declare (ih, $\mathbb{Z}$, $a - 1$)
while ($a < il$) **or** ($ih < b$) **do**
    declare (j, $\mathbb{Z}$, any value such that ($a \leq j < il$) **or** ($ih < j \leq b$))
    if $k(j) < skey$
    then $a := j + 1$
        $ih := \max(ih, j)$
    else
    if $k(j) = skey$
    then $il := \min(il, j)$
        $ih := \max(ih, j)$

```
        else {k(j) > skey}
            b := j − 1
            il := min(il, j)
        endif
        endif
        release j
    endwhile
    release b
    release a
```

The relation in braces ({}) above is a comment.

Exercise

1 State the loop invariant formally.
2 Prove formally that the body of the loop preserves the loop invariant.
3 Prove the correctness of the **while** loop with its initialization.
4 State completely and prove the theorem which describes the effect of executing the final version of the subprogram above.

6.3 Partitioning an array (example)

An array $x(i)$, $i = 1, 2, \ldots, n$, is given. Our task is to design a subprogram which will rearrange (mathematically, permute) the values of the array variables $x(i)$ so that (a) those values which are less than some initially selected value are in the lower part of the array, (b) those values which are equal to the selected value are in the middle of the array and (c) those values which are greater than the selected value are in the upper part of the array. The middle part of the array may not be empty upon termination.

We begin by generalizing the specification of the given array so that both the initial and final subscripts are given by variables, il and ig. The postcondition can then be represented by the following diagram:

$\lvert il$	$eq\rvert$	$gr\rvert$	$ig\rvert$
$x(i) < key$	$x(i) = key$	$x(i) > key$	

More formally, we require that our subprogram declare and calculate values for the variables eq and gr such that

1 $il - 1 \leq eq < gr \leq ig$
2 **and** for all i such that $il \leq i \leq ig$
  (a) $il \leq i \leq eq \Rightarrow x(i) < x(gr)$
  (b) **and** $eq < i \leq gr \Rightarrow x(i) = x(gr)$
  (c) **and** $gr < i \leq ig \Rightarrow x(i) > x(gr)$
3 **and** the sequence of values $[x(il), x(il + 1), \ldots, x(ig)]$ after termination of the subprogram is a permutation of the values of the same array elements before execution of the subprogram.

This postcondition can be expressed in other forms, e.g.

$il - 1 \leq eq < gr \leq ig$
and$_{i=il}{}^{eq}\ x(i) < x(gr)$
and$_{i=eq+1}{}^{gr}\ x(i) = x(gr)$
and$_{i=gr+1}{}^{ig}\ x(i) > x(gr)$
and $[x''(il), x''(il + 1), \ldots, x''(ig)]$ is a permutation of $[x'(il), x'(il + 1), \ldots, x'(ig)]$

where $x'(il)$ stands for the value of $x(il)$ before execution of the subprogram and $x''(il)$, for the value thereafter, etc.

The precondition can be represented by the diagram

$\lvert il$	$ig\rvert$
$x(i)?$	

The postcondition requires that the central region not be empty. This condition can, of course, be satisfied only if the given array is not empty. More formally, the postcondition implies, i.e. can be satisfied only if $il \leq ig$. This condition is, therefore, a necessary part of the precondition.

The goal of our subprogram is to permute the values of the given array so that it is coarsely sorted. Probably the simplest way to permute the values is by repeatedly exchanging pairs of values, successively moving individual values into the regions in which they must be in the final state. Such a strategy suggests a loop as the fundamental structure of the subprogram.

There are several ways to generalize the precondition and the postcondition to obtain the loop invariant. The major design decisions concern the number of unknown regions to form and their location(s) relative to the three known regions ('less', 'equal' and 'greater'). Efficiency considerations suggest that once a value is placed in a known region, it should never be moved again if one can avoid doing so. This, in turn, suggests that the 'less' region should be at the lower end of the subscript range and the 'greater' region, at its upper end.

Intuitively, it seems simpler to have only one unknown region rather than two. Then, the only decision left is whether to place the unknown region between the 'less' and 'equal' regions or between the 'equal' and 'greater' regions. These two possibilities are symmetric with respect to each other

and are structurally equivalent. The choice between them is, therefore, arbitrary. These considerations lead to the following diagrammatic form for our loop invariant:

il	k	eq	gr	ig
$x(i) < key$	$x(i)?$	$x(i) = key$	$x(i) > key$	

Initially, the equal interval contains one arbitrarily selected element and the unknown interval contains all others. In the final state, the unknown interval is empty.

The formal form of the loop invariant is:

$il \leq k$
and $k - 1 \leq eq < gr \leq ig$
and$_{i=il}{}^{k-1}$ $x(i) < x(gr)$
and$_{i=eq+1}{}^{gr}$ $x(i) = x(gr)$
and$_{i=gr+1}{}^{ig}$ $x(i) > x(gr)$
and $[x(il), x(il + 1), \ldots, x(ig)]$ is a permutation of $[x'(il), x'(il + 1), \ldots, x'(ig)]$

The goal of the loop is to reduce the size of the unknown region until it is empty, in which case, the loop invariant reduces to the postcondition. The body of the loop should, therefore, assign some array element in the unknown region to the appropriate known region. These considerations lead to the following first version of our subprogram:

Establish the truth of the loop invariant; i.e. initialize k, eq and gr so that the loop invariant reflects the initial condition (the precondition).
while unknown region not empty **do**
1 Select some subscript value in the unknown region.
2 Compare the value of the selected array element with the value of some element in the 'equal' region.
3 Exchange the value of the selected and some other array elements as appropriate to move the former into the region in which it belongs.
4 Update the regional pointers (k, eq and gr) appropriately in order to reestablish the truth of the loop invariant.
endwhile

The only real design decision to be made in writing the initialization regards the selection of the value defining the 'equal' region. The specification and the postcondition say nothing about how this is to be done. If we arbitrarily choose it to be the value $x(ig)$, the initialization is particularly simple:

declare (k, $\mathbb{Z}$, il) {'less' region empty}
declare (gr, $\mathbb{Z}$, ig) {'greater' region empty}

declare (eq, $\mathbb{Z}$, $gr - 1$) {'equal' region contains one element, $x(ig)$}

Passages enclosed in braces ({}) above are comments. They are not part of the program statements.

We might make explicit the fact that the choice of the value defining the 'equal' region is arbitrary by adding the statement

$x(i) :=: x(gr)$, $il \leq i \leq ig$, i otherwise an arbitrary integer

to the initialization.

The **while** condition 'unknown region not empty' translates directly into

$k \leq eq$

Next, we must decide how to select a subscript value in the unknown region, i.e. how to select an array element to be moved into the region in which it belongs. We have three fundamentally different choices: k, eq and some intermediate value. In each case, the value of the selected array element will have to be moved into one of the three regions 'less', 'equal' or 'greater'.

If we select k as the subscript value and $x(k)$ belongs in the 'less' region, no exchanging is necessary; we need only to increase the value of k. If $x(k)$ belongs in the 'equal' region, the values of $x(k)$ and $x(eq)$ must be exchanged. If $x(k)$ belongs in the 'greater' region, $x(k)$ must be moved to position gr, $x(gr)$ must be moved to position eq and $x(eq)$ must be moved to position k. Two exchanges will be necessary to achieve this.

If we select a value i between k and eq as the subscript value ($k < i < eq$), the number of exchanges required is always at least as great as in the case above. The reader should identify the exchanges required in each possible situation.

If we select eq as the subscript value, the situation is somewhat simpler. In two cases ('less' or 'greater'), one exchange is required. In one case ('equal'), no exchange is required.

We will, therefore, choose $x(eq)$ as the array element to move into the region in which it belongs. First, we must determine into which region it must be assigned. This we do by comparing $x(eq)$ with some array element in the equal region. The element $x(gr)$ is always in this region; it is, in fact, the only one which is always in this region. We will, therefore, choose it as the basis for comparison.

We must distinguish between three possible, mutually exclusive cases: $x(eq) < x(gr)$, $x(eq) = x(gr)$, and $x(eq) > x(gr)$.

The simplest situation arises when $x(eq) = x(gr)$. We need only decrease eq to insert $x(eq)$ into the equal region:

$eq := eq - 1$

If $x(eq) < x(gr)$, then exchanging the values of $x(eq)$ and $x(k)$ will place the tested value into position for insertion into the 'less' region:

$x(eq)$:=: $x(k)$
$k := k + 1$

If $x(eq) > x(gr)$, then the original value of $x(eq)$ must be brought into position gr, so that it can be put into the 'greater' region by decreasing gr. Before doing so, however, the entire 'equal' region must effectively be shifted downward by one position. This effect can be achieved most simply by exchanging $x(eq)$ and $x(gr)$ and adjusting the regional pointers appropriately:

$x(eq)$:=: $x(gr)$
$gr := gr - 1$
$eq := eq - 1$

Combining the above, we transform our first version of the subprogram into the following:

```
declare (k, ℤ, il) {'less' region empty}
declare (gr, ℤ, ig) {'greater' region empty}
declare (eq, ℤ, gr − 1) {'equal' region contains one element, x(ig)}
x(i) :=: x(gr), il ≤ i ≤ ig, i otherwise an arbitrary integer
while k ≤ eq do
    if x(eq) < x(gr)
    then x(eq) :=: x(k)
         k := k + 1
    else
    if x(eq) = x(gr)
    then eq := eq − 1
    else
    if x(eq) > x(gr)
    then x(eq) :=: x(gr)
         gr := gr − 1
         eq := eq − 1
    endif
    endif
    endif
endwhile
```

The third **if** condition will, of course, always be true at that point. Therefore, it and the following '**then**' can be omitted if desired.

The variable k is an internal variable; its final value is of no interest to the calling program. We should, therefore, release it at the end of our subprogram. The variables eq and gr appear in the postcondition; their values are results of this subprogram, so they may not be released here.

These considerations lead to the final form of our subprogram for partitioning an array into three subsections:

```
procedure partition:
declare (k, ℤ, il) {'less' region empty}
declare (gr, ℤ, ig) {'greater' region empty}
declare (eq, ℤ, gr − 1) {'equal' region contains one element, x(ig)}
x(i) :=: x(gr), il ≤ i ≤ ig, i otherwise an arbitrary integer
while k ≤ eq do
    if x(eq) < x(gr)
    then x(eq) :=: x(k)
         k := k + 1
    else
    if x(eq) = x(gr)
    then eq := eq − 1
    else {x(eq) > x(gr)}
         x(eq) :=: x(gr)
         gr := gr − 1
         eq := eq − 1
    endif
    endif
endwhile
release k
endprocedure
```

In the original statement of the problem, the range of subscripts was specified to be 1 to n, not il to ig. We can partition the array as originally given by calling the above form of the subprogram *partition* appropriately:

```
declare (il, ℤ, 1)
declare (ig, ℤ, n)
call partition
release ig
release il
```

The subject of this design exercise is a slight modification of 'The Problem of the Dutch National Flag' (Dijkstra, 1976, Ch. 14), in which a collection of red, white and blue pebbles is to be separated by color.

Exercise

5 Draw a diagram representing the conditions satisfied at each intermediate point in the above subprogram '*partition*'. Your diagrams should be of the

type used above to illustrate the postcondition, the precondition and the loop invariant.

6 Identify the exchanges which would be necessary if an intermediate subscript value i ($k < i < eq$) were selected instead of eq as the subscript of the next array element to be placed in its proper region.

7 State completely and prove the theorem which describes the effect of executing the final version of the subprogram '*partition*' above.

6.4 Quicksort, a recursive sorting algorithm (example)

The subprogram '*partition*' developed in Section 6.3 above subdivides one unsorted array into two smaller unsorted subarrays and a third subarray which, because it contains only elements equal to each other, is effectively sorted. If each of the two unsorted subarrays were to be sorted in place, then the entire original array would be sorted.

This suggests a recursive procedure for sorting an array: By calling the subprogram '*partition*', subdivide the array to be sorted. Then the sorting procedure calls itself recursively to sort each of the two unsorted subarrays:

procedure *sort*:
call *partition* for the entire array
call *sort* for one of the two unsorted subarrays
call *sort* for the other unsorted subarray
endprocedure

We specify this design task, therefore, as follows. Design a recursive program which will sort the values in a given array, using the subprogram '*partition*' (see Section 6.3 above).

The correctness of such a recursive procedure is most naturally and conveniently proved by induction. Because the subprogram '*partition*' returns a non-empty 'equal' region (which does not need to be sorted), the total size of the two unsorted subarrays is truly less than the size of the original array. Each subarray must, therefore, be truly smaller than the original array. This suggests a proof by induction on the number of elements in the array to be sorted.

If the procedure '*sort*' sorts any input array containing fewer than n elements, then the above version will clearly also sort an input array with n elements. Thus, the requirements of the inductive step in the proof are satisfied.

In an inductive proof, one must also prove the thesis of the theorem for some particular value of the variable of induction, here, the number of elements in the array to be sorted. A particulary simple starting point is provided by noting that an array containing 0 or 1 element is already sorted. We refine somewhat our initial coarse design above and write:

procedure *sort*:
if the array to be sorted contains 2 or more elements
then call *partition* for the entire array
  call *sort* for one of the two unsorted subarrays
  call *sort* for the other unsorted subarray
else The array contains 0 or 1 element and is, therefore, already sorted, so do nothing.
endif
endprocedure

Before making our code more precise, we must define the interface between it and its caller precisely. The notation and conventions used in Section 6.3 are convenient: Design a procedure which will sort a given array $x(i)$, $i = il, il + 1, \ldots, ig$, where il and ig are two given integers with $il - 1 \leq ig$. (If $il - 1 = ig$, then the array is empty; if $il = ig$, the array contains one element.)

This enables us to refine our pseudocode above one step further to obtain:

procedure *sort*:
if $il < ig$
then call *partition* (entire array, subscript values il to ig inclusive)
  call *sort* ('less' subarray, subscript values il to eq inclusive)
  call *sort* ('greater' subarray, subscript values $gr + 1$ to ig inclusive)
endif
endprocedure

The parameters of our procedure '*sort*' are il and ig. We must set them up appropriately in our pseudocode for each recursive call. The parameters for *partition* and *sort* are the same, so no adjustment is needed for the call to *partition*. Our final pseudocode then becomes:

procedure *sort*:
if $il < ig$
then call *partition*
    declare (ig, $\mathbb{Z}$, eq) {original ig becomes concealed}
      call *sort* {il = original il; ig = eq from *partition*)
    release ig {restore original ig}
    declare (il, $\mathbb{Z}$, $gr + 1$) {original il becomes concealed}
      call *sort* {il = $gr + 1$ from *partition*; ig = original ig}
    release il {restore original il}
  release eq {release result from *partition*}
  release gr {release result from *partition*}
endif
endprocedure

The recursive sorting algorithm 'Quicksort' is due to C. A. R. Hoare. Its original form uses a different partitioning algorithm.

Exercise

8 State and prove the correctness theorem for the above procedure '*sort*'. Hint: Use the correctness theorem for the subprogram '*partition*' (see Exercise 7).

6.5 Searching and updating a linked list (example)

Our task is to design subprograms which will

1 locate an element equal to a given search key *skey*,
2 delete an element whose value is equal to *skey* and
3 insert a new element

in a given ordered linked linear list.

We begin by defining two types of linked linear lists and specifying conditions which they must satisfy. The condition for an ordered linked linear list – which may perhaps be most meaningfully thought of as a data invariant – will be a precondition and a postcondition of each of our three subprograms.

6.5.0 Linked linear lists

Consider a pair of arrays $e(i)$ and $p(i)$, where the values of i are in some set S of subscripts, e.g. $i = 1, 2, \ldots$. The value of $p(i)$ is the subscript of the successor of $e(i)$, i.e. $p(i)$ is the 'pointer' to the successor of $e(i)$. The sequence of values

$$e(f), e(p(f)), e(p(p(f))), \ldots$$

is called a linked linear list if the sequence does not contain a loop, i.e. if no two terms appearing in the sequence have equal subscript values. The value of the variable f is the subscript of (the pointer to) the first element in the list. If the value of $p(i)$ is a special value not in S, then $e(i)$ has no successor and is the last element in the list. We will call that special value '*endvalue*' below. If $f = endvalue$, then the list is empty.

For convenience, we will employ the following notation:

$p^2(f)$ means $p(p(f))$,
$p^3(f)$ means $p(p(p(f)))$, etc. and
$p^0(f)$ means f.

Note that

$p^0(f)$ points to (is the subscript of) the first term in the list,
$p^1(f)$ points to the second term in the list, etc. and
$p^{i-1}(f)$ points to the ith term in the list.

If the number of elements $e(i)$ is limited, e.g. if the set S of subscript values is a finite set, then the requirement that no loop be present implies that the list has an end (a last element). Because memory is limited in any real computing system, this situation always applies in such systems.

If the sequence of terms in the list is in a specific order (e.g. non-descending), the list is called 'ordered'.

If a linked linear list contains exactly n elements, then $p^n(f) = endvalue$. Every previous pointer in the sequence must be a valid subscript, i.e. must be in S. Finally, some non-negative integer n must exist which fulfills these conditions. This condition, which must be satisfied by any finite linked linear list, can be written more formally as follows:

There exists a non-negative integer n (the number of elements in the list) such that
1 $p^n(f) = endvalue$ **and**
2 for every integer j in the interval $0 \leq j < n$, $p^j(f)$ is in S

If the list is also ordered in ascending sequence with equal elements permitted, then we must require that every pair of consecutive elements be in order, i.e.

$$e(k) \leq e(p(k))$$

for every k which is the subscript of an element in the list other than the last. This requirement can be rewritten in the following form and added (logically '**and**ed') to the criterion for the unordered linked linear list above:

3 for every integer j in the interval $0 \leq j < n - 1$
$e(p^j(f)) \leq e(p^{j+1}(f))$

The above condition can be written in different, equivalent forms, e.g.

$$\mathbf{or}_{n=0}^{\inf} [p^n(f) = endvalue \; \mathbf{and}_{j=0}^{n-1} p^j(f) \text{ in } S \; \mathbf{and}_{j=0}^{n-2} e(p^j(f)) \leq e(p^{j+1}(f))]$$

6.5.1 Searching an ordered linked linear list

We begin with the subprogram for locating the element in the list with a given value, whereby the possibility must be considered that the value being

sought is not present in the list. Therefore, after our subprogram has been executed, its result variable(s) should identify two successive elements in the list which bracket the value being sought, i.e.

$$e(i) < skey \leq e(p(i))$$

or, alternatively,

$$e(i) \leq skey < e(p(i))$$

depending upon whether we want to find the first or last element in the list which is equal to the given search key in the event that there are more than one. In this regard, the specification is ambiguous. We will choose the first alternative above to be our design goal.

The above tentative version of the postcondition is inadequate. There may be no $e(i)$ which is less than *skey*. Similarly, there may be no $e(p(i))$ which is greater than or equal to *skey*. Such elements of the list may not exist either because the list is empty, because the element being sought is less than or equal to the first element or because the element being sought is greater than the last element. Thus, our postcondition must encompass the following four mutually exclusive and exhaustive possibilities:

1 The list is empty (so the value being sought cannot appear in it).
2 The value being sought is less than or equal to the first element of the list (which is not empty).
3 The value being sought is greater than the last element of the list (which is not empty).
4 Two successive elements of the list exist which bound the value being sought, i.e.

$$e(i) < skey \leq e(p(i))$$

If we adopt the convention that the result variable *pred* will point to the predecessor of the element equal to *skey*, if any, we can refine the above tentative postcondition to become:

1 $f = endvalue$ **or**
2 $f \neq endvalue$ **and** $skey \leq e(f)$ **or**
3 $f \neq endvalue$ **and** $pred \neq endvalue$ **and** $p(pred) = endvalue$ **and** $e(pred) < skey$ **or**
4 $f \neq endvalue$ **and** $pred \neq endvalue$ **and** $p(pred) \neq endvalue$ **and** $e(pred) < skey \leq e(p(pred))$

The main factor distinguishing between subconditions 1 and 2 above on the one hand and subconditions 3 and 4 on the other hand is the existence of

an element in the list preceding *skey* in order, i.e. an element such that $e(pred) < skey$. Under conditions 1 and 2, no such predecessor exists. If our subprogram would set *pred* equal to *endvalue* in such cases, the calling program's test to distinguish between subconditions 1 and 2 versus 3 and 4 would be particularly simple.

The main factor distinguishing between subcondition 3 on the one hand and subcondition 4 on the other is the existence of an element in the list succeeding *skey* in order, i.e. an element such that $skey \leq e(p(pred))$. The calling program can distinguish between these two situations most easily by testing whether $p(pred) = endvalue$ or not. In a generalized sense, the calling program can distinguish between subconditions 1 and 2 in the same way, where $p(pred)$ corresponds to f.

These considerations suggest revising the postcondition above so that our subprogram returns two results: *pred*, a pointer to the predecessor of *skey*, and *succ*, a pointer to the successor of *skey*. The variable *succ* would represent – i.e. be equal to – f in subconditions 1 and 2 and $p(pred)$ in subconditions 3 and 4. The final version of the postcondition is then:

1 $pred = endvalue$ **and** $succ = endvalue$ **and** $f = endvalue$ **or**
2 $pred = endvalue$ **and** $succ \neq endvalue$ **and** $f = succ$ **and** $skey \leq e(succ)$ **or**
3 $pred \neq endvalue$ **and** $succ = endvalue$ **and** $p(pred) = endvalue$ **and** $e(pred) < skey$ **or**
4 $pred \neq endvalue$ **and** $succ \neq endvalue$ **and** $p(pred) = succ$ **and** $e(pred) < skey \leq e(succ)$

Now the calling program can easily distinguish among these four cases by comparing *pred* and *succ* with *endvalue*.

A general strategy for our subprogram's operation follows naturally from the structure of a linked list: successively test pairs of adjacent elements in the list to determine whether the postcondition is satisfied. If the pair does not satisfy it, step to the next pair. This implies a loop as the fundamental structure of our subprogram and suggests the following general form for it.

> Initialize the loop variable(s) to point to the first pair of elements in the list which could bracket the position being sought.
> **while** postcondition not satisfied **do**
> Step the loop variable(s) to point to the next pair of elements in the list.
> **endwhile**

Next, we must decide upon a loop invariant. In order to do this, it is useful first to ponder the initial situation and second to generalize the initial

situation and the postcondition. Leaving aside for the moment the possibility of an empty list (subcondition 1 above); the first possible 'pair' of elements which could satisfy the postcondition corresponds to subcondition 2. This suggests as the initialization

pred := *endvalue*
succ := *f*

(or the equivalent declaration statements), which is also appropriate for subcondition 1. After this initialization, either subcondition 1 or 2, depending upon whether *f* is equal to endvalue or not, would be true if subcondition 2 did not contain the term '*skey* ≤ *e*(*succ*)'. This suggests generalizing (relaxing) the postcondition to obtain a loop invariant by deleting this term. It appears not only in subcondition 2, but also in subcondition 4 and should be deleted from both. This leads to the following tentative loop invariant:

1 *pred* = *endvalue* **and** *succ* = *endvalue* **and** *f* = *endvalue* **or**
2 *pred* = *endvalue* **and** *succ* ≠ *endvalue* **and** *f* = *succ* **or**
3 *pred* ≠ *endvalue* **and** *succ* = *endvalue* **and** *p*(*pred*) = *endvalue* **and** *e*(*pred*) < *skey* **or**
4 *pred* ≠ *endvalue* **and** *succ* ≠ *endvalue* **and** *p*(*pred*) = *succ* **and** *e*(*pred*) < *skey*

Noting that the conditions

(*succ* = *endvalue* **and** *f* = *endvalue*)

and

(*succ* = *endvalue* **and** *f* = *succ*)

(see subcondition 1) are logically equivalent, as are

(*succ* = *endvalue* **and** *p*(*pred*) = *endvalue*)

and

(*succ* = *endvalue* **and** *p*(*pred*) = *succ*)

(see subcondition 3), we can simplify the above tentative loop invariant to obtain the loop invariant *I*:

pred = *endvalue* **and** *f* = *succ* **or**
pred ≠ *endvalue* **and** *p*(*pred*) = *succ* **and** *e*(*pred*) < *skey*

We determined this loop invariant by removing the condition

skey ≤ *e*(*succ*)

from the postcondition, subconditions 2 and 4, i.e. when *succ* ≠ *endvalue*. When *succ* = *endvalue*, the loop invariant *I* is equal to the postcondition. This suggests the condition

succ = *endvalue* **or** [*succ* ≠ *endvalue* **and** *skey* ≤ *e*(*succ*)]

which is equivalent to

succ = *endvalue* **or** *skey* ≤ *e*(*succ*)

as the termination condition *E* (the 'end' condition, cf. the latter half of Section 3.3.3).

At this point, the reader should verify that (*I* **and** *E*) equals (i.e. is logically equivalent to) the postcondition. Having done so, we can complete the design of our subprogram as described in the latter half of Section 3.3.3.

The **while** condition is (**not** *E*), the negation of the termination condition, i.e.

succ ≠ *endvalue* **and** *e*(*succ*) < *skey*

Combining all of the above, we write the final version of our subprogram, which we will call '*locate*' in Sections 6.5.2 and 6.5.3 below:

procedure *locate*:
declare (*pred*, subscripts, *endvalue*)
declare (*succ*, subscripts, *f*)
while *succ* ≠ *endvalue* **and** *e*(*succ*) < *skey* **do**
  pred := *succ*
  succ := *p*(*pred*)
endwhile
endprocedure

Exercise

9 Show that (*I* **and** *E*) is equal to the postcondition.
10 One programmer's initial design specified the postcondition as

$1 \leq m \leq n + 1$
and $_{i=1}{}^{m-1}\ e(p^{i-1}(f)) < skey$
and$_{i=m}{}^{n}\ skey \leq e(p^{i-1}(f))$

and the program as

m := 1
while ? **do**
  m := *m* + 1
endwhile

(a) What is the meaning of the variable m?
(b) Show that this postcondition corresponds to the one given earlier in this section.
(c) Complete the design of this program by determining the loop invariant and the **while** condition.
(d) Transform the resulting version of the program into the one given earlier in this section.

6.5.2 Deleting an element from an ordered linked linear list

The subprogram '*locate*' (see Section 6.5.1 above) can be used to find the element in the list which is to be deleted. Then, pointers must be modified appropriately.

If neither *pred* nor *succ* is equal to *endvalue* and $p(pred) = succ$, then the element to which *succ* points can be deleted from the list by executing the statement

$p(pred) := p(succ)$

(see Fig. 6.0).

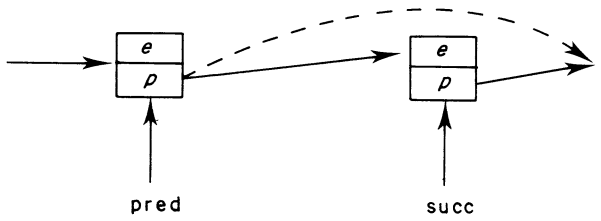


Fig. 6.0 Deleting an element other than the first from a linked list

If $pred = endvalue$, $succ \neq endvalue$ and $succ = f$, then the element to which *succ* points is the first element in the list. It can be deleted by executing the statement

$f := p(succ)$

(see Fig. 6.1).

If $succ = endvalue$, then *succ* does not point to any element which can be deleted.

Combining the above and utilizing the subprogram '*locate*' of Section 6.5.1 above, we write the following subprogram for deleting an element which has the same value as *skey*, if one is present in the list:

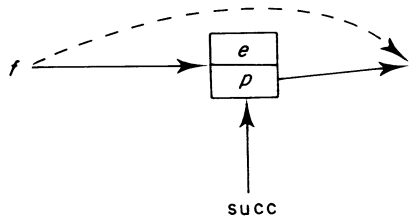


Fig. 6.1 Deleting the first element of a linked list

```
procedure delete:
call locate
if succ ≠ endvalue
then if e(succ) = skey
     then if pred ≠ endvalue
          then p(pred) := p(succ)
          else f := p(succ)
          endif
     endif
endif
release pred
release succ
endprocedure
```

Refer especially to the postcondition of the subprogram '*locate*' (see Section 6.5.1) and to the condition which an ordered linked linear list must fulfill (see Section 6.5.0).

Exercise

11 Why is the condition '**if** $e(succ) = skey$' required?

6.5.3 Inserting an element into an ordered linked linear list

The subprogram '*locate*' (see Section 6.5.1 above) can be used to locate the place in the list where the new element should be inserted. Then, pointers must be modified appropriately.

If neither *pred* nor *succ* is equal to *endvalue* and $p(pred) = succ$, then a new element to which the variable *pnew* points may be inserted into the list between the elements to which *pred* and *succ* point by executing the statements

$p(pnew) := succ$
$p(pred) := pnew$

(see Fig. 6.2).

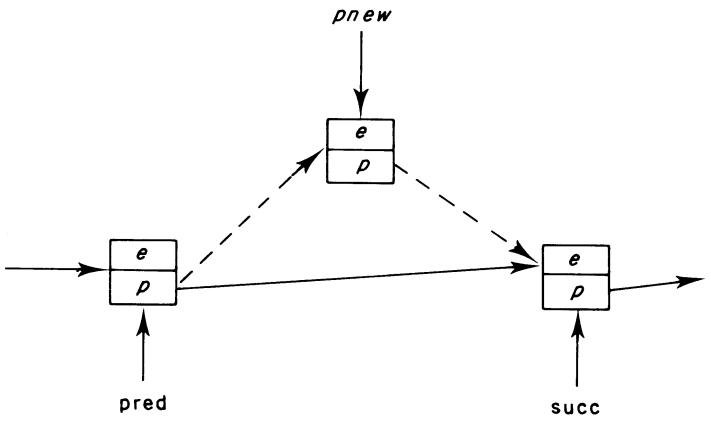


Fig. 6.2 Inserting a new element into a linked list other than at the beginning

If $pred \neq endvalue$ and $p(pred) = succ = endvalue$, then executing the above statements would insert the new element at the end of an existing, non-empty list.

If $pred = endvalue$, $succ \neq endvalue$ and $succ = f$, then the following, corresponding sequence of statements would insert the new element at the beginning of a previously non-empty list:

$p(pnew) := succ$
$f := pnew$

(see Fig. 6.3).

If $pred = endvalue$, $succ = endvalue$ and $succ = f$, then the above statements would insert the new element into a previously empty list.

The subprogram for inserting the new element $e(pnew)$ into an ordered linked linear list is thus:

procedure *insert*:
declare (*skey*, list elements, *e*(*pnew*))
call *locate*
$p(pnew) := succ$
if $pred \neq endvalue$

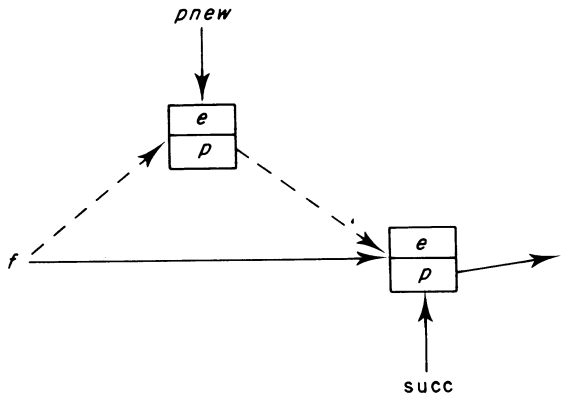


Fig. 6.3 Inserting a new element at the beginning of a linked list

then $p(pred) := pnew$
else $f := pnew$
endif
release *pred*
release *succ*
release *skey*
endprocedure

Refer especially to the postcondition of the subprogram '*locate*' (see Section 6.5.1) and to the condition which an ordered linked linear list must fulfill (see Section 6.5.0).

Exercise:

12 Prove that if the condition for an ordered linked linear list (see the end of Section 6.5.0) is satisfied before the above subprogram is executed, then it will be satisfied afterward as well.

6.6 File positioning (example)

The contents of a file in a medical laboratory's computing system indicate which analyses should be performed on each blood sample. The file consists of a sequence of lines, each of which contains a number. Each blood sample and each possible analysis is identified by a positive integer (>0). The number 0 indicates the end of a sequence of analysis numbers for one

sample or the end of a sequence of sample groups (and hence the end of the file).

For each blood sample the sequential file contains a line with the sample number, then one or more lines, each with an analysis number, and finally a line containing the number 0 (indicating the end of the sequence of analysis numbers for the sample). The analyses indicated in the group are to be performed on the sample indicated in the first line. The end of the file is indicated by a line containing 0 instead of a sample number:

Sample number
Analysis number
...
Analysis number
0 (End of the sample group)
...
Sample number
Analysis number
...
Analysis number
0 (End of the sample group)
0 (End of the file)

In this application system a central computer is connected directly with an automatic analyzer, which contains its own small computer. From time to time the analyzer sends to the central computer a request for instructions. The request indicates the position of the sample in the sample tray, which is the same as the position of the sample's data group in the file. The central computer responds by sending to the analyzer the list of numbers identifying the analyses to be performed.

The sample's position in the file is not necessarily the same as the sample number, which is of no consequence in this part of the program.

Normally the analyzer requests instructions for the samples in the order in which they appear in the file. Exceptions can, however, occur, e.g. in the event of an operational disturbance, an error on the part of the operator of the analyzer, repeating an analysis, etc.

Our task is to design subroutines for the central computer which read the data from this file and transmit the requested instructions to the analyzer. Probably the most important design decision in this connection concerns the interrelationship between the several subroutines.

In order to simplify logically the interaction among the several subroutines, we define conditions which are to be fulfilled whenever any of these subprograms is called or returns control to its caller:

1 The file is positioned immediately after a sample number.
2 The value of the variable *snr* is that sample number.
3 The variable *pos* indicates the sample's position in the file.

The 'sample number' after which the file is positioned may be the zero marking the end of the file.

The three conditions above are pre- and postconditions for each procedure to be designed except one (for opening the file), for which they are only postconditions. Such an interface condition is comparable to a loop invariant. It can be thought of as a 'program invariant'.

The subprograms for processing this file follow straightforwardly from this specification and the description of their functions given above:

procedure open file:
open *file*
read *snr*
pos := 1
endprocedure

procedure position file to the sample group indicated by *posreq*:
precondition: *posreq* is an integer, *posreq* ≥ 1.
if *pos* > *posreq*
then rewind *file*
  read *snr*
  pos := 1
endif
Prop:: *pos* ≤ *posreq* (invariant of the following **while** loop)
while *pos* < *posreq* **and** *snr* > 0 **do**
  repeat
    read *analysisnumber*
  until *analysisnumber* ≤ 0
  Prop: The file is positioned before a sample number.
  read *snr*
  pos := *pos* + 1
endwhile
endprocedure
Theorem: At the end of this subprogram either
1. *snr* > 0 and *pos* = *posreq* (The desired sample has been found.)
or
2. *snr* ≤ 0 (The desired sample is not in the file.).

procedure answer request for the sample position *posreq*:
Precondition: *posreq* is an integer, *posreq* ≥ 1.
call position file (see above)
Send message to analyzer: "Analysis numbers follow"
if *snr* > 0
then loop
read *analysisnumber*
if *analysisnumber* ≤ 0 **then exit endif**
Send *analysisnumber* to analyzer.
endloop
Prop.: The file is positioned before a sample number.
read *snr*
pos := *pos* + 1
endif
Prop.: All analysis numbers (if any) for the sample in the position *posreq* of the file have been transmitted to the analyzer.
Send message to the analyzer: "End of the analysis numbers"
endprocedure

This application represents a relatively simple example of data processing. Nevertheless errors are often made when designing and constructing such programs. Those errors cause unnecessary costs for 'testing' (finding and correcting the errors), relieving the effects of the errors, etc.

Exercise

13 Prove each proposition and theorem stated in the above subprograms.
14 Why did the designer select the particular file position (see condition 1 above) as the standard one? What would be the effect upon the subprograms of a different choice for the standard file position?

6.7 Control logic for printing a report (example)

It is interesting to ponder how many times this 'wheel has been reinvented' – and erroneously – in the last several decades of computing. In the program libraries of computer installations throughout the world programs for printing reports abound, many of which still contain errors which occasionally manifest themselves. The last page of a report may contain a header but no data; the footer may or may not be printed. Or the header is printed on the first page of an otherwise empty report, leaving the paper positioned

in the middle of the page instead of at the top. As a result, the next report is not positioned properly on the page; etc. *ad taedium.*

In this section, we will design a set of short subprograms which correctly control printing data, headers and footers. We will begin by stating more precisely what we mean by 'correct'. From our criteria for correctness – in effect, the postcondition – we will deduce invariant conditions which must be satisfied at particular points in the execution of the program. These conditions represent pre- and postconditions for the several subprograms, which we will more or less straightforwardly derive from those pre- and postconditions.

We will design the printing subprograms to be used (called) by a program which generates a sequence of data groups. Each data group contains a number of lines of data which are to be printed on one page, i.e. which may not be split and printed on two or more pages. The using program will call an initializing subprogram in the printing subsystem before the first data group is to be printed. It will then call the main printing subprogram repeatedly, once for each group of lines to be printed. After calling this subprogram for the last data group to be printed, the main program will call a subprogram for terminating (closing) the printing operation.

We will define a report to be correct when each page is printed correctly, when its pages are numbered in sequence beginning with a certain initial page number (usually 1) and when no data group is split onto two or more pages. A page is printed correctly when it contains

the header in a predefined format and with the correct page number,
at least one line of data,
at most a certain maximum number of lines of data and
a footer in a predefined format and with the correct page number

and when the header, data and the footer are printed in the predefined positions (lines) on the page. Note that the above definition of correctness implies that if a report contains no data, neither a header nor a footer may be printed.

Subject to the above restrictions, as much data as possible should be printed on each page. This goal can be achieved, of course, by skipping to a new page only when absolutely necessary to do so in order to avoid violating the conditions stated above, i.e. when the next data group to be printed will not fit in the space remaining in the data area of the current page.

The format of a page can be described in the following, more detailed manner:

Report page	Line number	Description
Header	1	first line on page
	llh	last line in header
Data	*llh* + 1	first line of data
	...	
	lld	last line of data
Footer	*lld* + 1	first line in footer
	llp	last line on page (and in footer)

The values of the above parameters delimiting the three regions must satisfy the following conditions:

$$0 \leq llh < lld \leq llp$$

The header, including blank lines for spacing, is to be printed on lines 1 through *llh* inclusive. If $llh = 0$, then the header is empty. Data is to be printed on lines $llh + 1$ through *lld* inclusive. This region may not be empty. The footer, including blank lines for spacing, is to be printed on lines $lld + 1$ through *llp* inclusive. If $llp = lld$, then the footer is empty.

Pages are to be numbered beginning with the value of the parameter *firstpage*.

We will assume that the target computer system has a printer which is capable of printing line by line in a forward direction only. Thus, if the last line was printed on line L of page P, the next line must be printed on line $L + 1$ of page P, unless L is the last line on a physical page, in which case the next line must be printed on line 1 of page $P + 1$. We assume that the printer is not capable of other movements, such as skipping a number of lines in a single operation, advancing the paper to a particular line, feeding the paper backwards, etc.

Our printing subsystem will consist of three higher level subprograms, for initialization, printing a group of data lines and termination. These will call additional subprograms for controlling page overflow, printing a header, printing a footer, etc.

We further define two control variables:

pageno	the current page number
lineno	the number of lines already printed on the current page

We define an invariant which is to be true whenever statements outside the printing subsystem are being executed after initialization and before termination of printing. In other words, the invariant is a postcondition of

the initialization subprogram, both a pre- and a postcondition of the subprogram for printing a group of data lines and a precondition of the termination subprogram. The first term in the invariant reflects the above definition of the control variables *pageno* and *lineno*:

> *I*1: The report has been completely printed to and including line number *lineno* on page number *pageno*. Subsequent lines have not been printed.

This term in the invariant suggests that the initialization subprogram should consist of the following statements:

> **procedure** initialize printing:
> *pageno* := *firstpage* − 1
> *lineno* := *llp*
> Ensure that the printing mechanism is positioned so that the next line transmitted to the printer will be printed on the first line of a new page (e.g. by asking the operator to adjust the paper feeding mechanism, if necessary).
> **endprocedure**

One of the conditions of correctness specified above requires that each page contain at least one line of data. This requirement can be incorporated into our invariant by **and**ing the following to term *I*1 above:

> *I*2: If any lines have been printed in the current report, then *llh* < *lineno*.

This term precludes the possibility that a header has been printed with no data following it on the page. This term can be rewritten in several different, equivalent forms, e.g.

> *I*2: lines have been printed in the current report ⇒ *llh* < *lineno*,
> *I*2: no lines have been printed in the current report **or** *llh* < *lineno*

or

> *I*2: (*pageno* = *firstpage* − 1 **and** *llp* = *lineno*) **or** *llh* < *lineno*.

The requirement that data not be printed beyond line *lld* can be incorporated into the invariant in a similar manner:

> (*pageno* = *firstpage* − 1 **and** *llp* = *lineno*) **or** *lineno* ≤ *lld*

Combining the above, we obtain the complete invariants:

> *I*1: The report has been completely printed to and including line number *lineno* on page number *pageno*. Subsequent lines have not been printed.

and
I2: [(*pageno* = *firstpage* − 1 **and** *llp* = *lineno*) **or** *llh* < *lineno* ≤ *lld*]

The subprogram for printing a group of data lines must, of course, contain the following statements or the equivalent thereof:

i := *first* − 1
while *i* < *last* **do**
    i := *i* + 1
    lineno := *lineno* + 1
    print *dataline*(*i*) on page *pageno*, line *lineno*
endwhile

The data lines constituting the group to be printed must be contained in the array variables *dataline*(*first*), *dataline*(*first* + 1), ..., *dataline*(*last*). The calling program must assign appropriate values to these array variables and to the variables *first* and *last* before calling this printing subprogram. Note that the number of lines to be printed is given by the value of the expression

last − *first* + 1

We require that the calling program ensure that the values of *first* and *last* are such that the value of this expression is non-negative. We explicitly permit it to be zero, in which case the data group to be printed is empty.

The body of the above loop will be executed exactly *last* − *first* + 1 times. The precondition of term *I2* of the invariant under this loop is, therefore,

(*pageno* = *firstpage* − 1 **and** *llp* = *lineno* + *last* − *first* + 1)
or *llh* < *lineno* + *last* − *first* + 1 ≤ *lld*

If *last* − *first* + 1, the number of lines in the data group to be printed, is zero, this precondition is the same as term *I2* of the invariant (corresponding to the fact that the body of the loop will not be executed). The subprogram as written above is complete for this case.

Consider the case that *last* − *first* + 1 > 0. The first of the two terms connected by the **or** will be satisfied only if *pageno* = *firstpage* − 1 and *lineno* < *llp*. This would correspond to an internal point within the preceding report, a situation which we will not permit to arise in our program. We require, therefore, that this term be false. Then the precondition above reduces to

llh < *lineno* + *last* − *first* + 1 ≤ *lld*

The invariant (as a precondition) implies the first part of this condition. The second part of the above condition will not necessarily be satisfied. If it is not satisfied, i.e. if

lineno + *last* − *first* + 1 > *lld*

then the next data group to be printed will not fit on the current page. The current page must be terminated and the next one started. This suggests prefixing our tentative subprogram above by

if *lineno* + *last* − *first* + 1 > *lld* **then call** new page **endif**

The subprogram 'new page' should print the footer and header, increment *pageno* accordingly and set *lineno* to *llh*. In this case, the precondition derived above implies

llh + *last* − *first* + 1 ≤ *lld*

or, equivalently,

last − *first* + 1 ≤ *lld* − *llh*

This condition expresses the fact that it is possible to print a data group on one page only if the page's data region is large enough. While clearly essential, such a requirement is easily overlooked if one does not systematically analyze a proposed design. The calling program which generates the data groups to be printed must ensure that this condition is satisfied. This condition will, therefore, be part of the precondition of our subprogram for printing a data group.

A short analysis of our tentative subprogram above extended to check for page overflow leads to the conclusion that it is still incomplete. As shown above, a precondition of term *I2* of the invariant with respect to the loop is

(*pageno* = *firstpage* − 1 **and** *llp* = *lineno* + *last* − *first* + 1)
or *llh* < *lineno* + *last* − *first* + 1 ≤ *lld*

This condition is the postcondition of the subprogram 'new page'. The only variables (as opposed to constant parameters) in this postcondition are *pageno* and *lineno*; the statements in the subprogram 'new page' affecting these variables will be

pageno := *pageno* + 1
lineno := *llh*

so that the precondition with respect to the subprogram 'new page' is

(*pageno* = *firstpage* − 2 **and** *llp* = *llh* + *last* − *first* + 1)
or *llh* < *llh* + *last* − *first* + 1 ≤ *lld*

The value of *pageno* will never be *firstpage* − 2, so this precondition reduces to

0 < *last* − *first* + 1 ≤ *lld* − *llh*

The truth of the first part of this condition cannot be guaranteed in our subprogram in its present tentative form. In particular, *lineno* = *llp* initially. If *lld* < *llp*, then the subprogram 'new page' will be called the first time the **if** statement is executed. If *last* − *first* + 1 = 0, the precondition of the subprogram 'new page' is not met and its postcondition will not be met. The body of the subprogram we are designing for printing a data group can be embedded in an appropriate **if** statement to prevent calling the subprogram 'new page' in this case. Our complete subprogram then becomes:

procedure print a data group:
if *last* − *first* + 1 > 0
then if *lineno* + *last* − *first* + 1 > *lld* **then call** new page **endif**
    i := *first* − 1
    while *i* < *last* **do**
        i := *i* + 1
        lineno := *lineno* + 1
        print *dataline*(*i*) on page *pageno*, line *lineno*
    endwhile
endif
endprocedure

Next, we must design the subprogram 'new page'. Basically, it consists of a call to a subprogram to print the footer followed by a call to a subprogram to print the header. However, the footer should be printed only when data has already been printed in the report, i.e. not initially (when *pageno* < *firstpage*). The subprogram is, then:

procedure new page:
if *pageno* ≥ *firstpage* **then call** print footer **endif**
call print header
endprocedure

The subprogram 'print footer' has a relatively simple structure. It must skip lines in the data region and print the lines in the footer:

procedure print footer:
while *lineno* < *lld* **do**
    lineno := *lineno* + 1
    print a blank line on page *pageno*, line *lineno*
endwhile
Print the individual lines in the footer (*llp* − *lld* lines).
lineno := *llp*
endprocedure

The subprogram 'print header' also follows straightforwardly from its assigned task:

procedure print header:
pageno := *pageno* + 1
Print the individual lines in the header (*llh* lines).
lineno := *llh*
endprocedure

After the main program has called the subprogram 'print a data group' for the last data group, it must call the subprogram 'terminate printing' to ensure that the report is properly completed. This subprogram has the function of printing the footer on the last page, but only, of course, if there is a last page, i.e. only if one or more lines have been printed in the current report:

procedure terminate printing:
if *pageno* ≥ *firstpage* **then call** print footer **endif**
endprocedure

Exercise

15 Show that *lineno* ≤ *lld* must be true immediately before calling the subprogram 'print footer'. Show further that this condition is not always met if *pageno* < *firstpage*, thus necessitating the **if** condition in the subprogram 'new page'.
16 What is the precondition of the subprogram 'print footer'? of the subprogram 'print header'? What are their postconditions?
17 Show formally the need for the **if** condition in the subprogram 'terminate printing'. Deduce the precise form of this condition. Which other condition would also be correct? Why would a designer choose the one condition instead of the other?
18 What is the loop invariant of the **while** loop in the subprogram 'print a data group'?
19 Prove rigorously the correctness of this system of subprograms for controlling the printing of a report. Pay particular attention to each requirement in the specifications (postconditions). Show that each print statement causes the data in question to be printed on the correct line. Hint: Replace each print statement by an equivalent assignment statement of the form

report(*pageno*, *lineno*) := ...

20 Which statement in the subprograms designed in this section is redundant? What indications are there of its redundancy?
21 What is the postcondition of the subprogram 'terminate printing'? What can be said about the position of the paper in the printer after this subprogram has executed?

6.8 Printing several data elements on one line (example)

In this example, the main program generates a sequence of data elements or groups. Several such data elements or groups are printed on each line. Our task is to construct the program and appropriate subprograms so that printing is controlled correctly. By 'correct' we mean that – in addition to the criteria specified in the preceding example (see section 6.7) – each printed line contains at least 1 and at most N (a constant) data groups.

We assume that the programming language being used contains commands for (a) printing data on the current line (without terminating the line) and for (b) terminating the current line (e.g. Basic). Subprograms from the example in Section 6.7 above will be used here.

The primary design decision involves specifying the interface between the main program generating the data groups and our subprograms for controlling their printing. A convenient form for the specification of this interrelationship is a condition restricting the number of data groups on the current line and a definition of one or more variables which describe completely the state of the line currently being built. We choose these such that outside of the subprograms to be designed for printing the data, the following conditions are met:

1 The variable *online* indicates how many data groups have been printed on the line currently being prepared but not yet terminated.
2 The printed line currently being prepared contains fewer than the maximum allowed number N of data groups, i.e.
$0 \leq online < N$

The variable *lineno* (see Section 6.7 above) will indicate here the number of complete (i.e. already terminated) lines which have been printed on the current page.

The relevant parts of the main program then become:

call initialize printing
online := 0
while ... **do**
    Generate a data group.
    call output a data group
endwhile
if *online* > 0 **then call** close current line **endif**
call terminate printing

The subprograms for building and printing the current line follow from the conditions set down above:

procedure output a data group:
if *online* = 0 **and** *lineno* ≥ *lld* **then call** new page **endif**
Print the data group without terminating the line.
online := *online* + 1
Prop.: $0 < online \leq N$
if *online* ≥ *N* **then call** close current line **endif**
endprocedure

procedure close current line:
Terminate the current printed line.
lineno := *lineno* + 1
online := 0
endprocedure

Exercise

22 Prove rigorously that the two interface conditions defined above are both pre- and postconditions of the subprogram 'output a data group'.
23 Prove that the proposition in the subprogram 'output a data group' is always true at that point in the execution of the program.
24 Prove that a blank (an empty) line will never be printed by the above program, i.e. that the statement 'terminate the current printed line' will never be executed with no data groups on the line.
25 Prove that a line will never be printed with more than N data groups.
26 Under which circumstances, if any, will a line be printed with fewer than N data groups?
27 What is the postcondition of the subprogram 'close current line'? What, therefore, must its precondition be? Is its precondition always satisfied? If so, prove; if not, give a counterexample.
28 Is the condition '*online* = 0' in the first **if** statement in the subprogram 'output a data group' necessary? If it is, why? If not, prove that it is not required and explain why the designer included it.

6.9 The game of thirteen matchsticks (example)

A correct program is to be written which guides two persons playing the game of 'Thirteen Matchsticks'. Initially, 13 matches are placed on a table. Two players alternately remove one, two or three matches at a time. The player who removes the last match loses.

A general structure for a program to guide two players through any game in which they alternate turns is:

Initialization
while Game not yet ended **do**
    Process one player's turn.
    Switch to the other player's turn.
endwhile
Display result (winner, loser).

We define the two control variables

N: the number of matchsticks still in play (on the table) and
S: the identification of the player whose turn it is (1 or 2)

From the definitions of the game and the variables the loop invariant follows directly:

($N \geq 0$) **and** (N is an integer) **and** ($S = 1$ **or** $S = 2$)
and (It is player S's turn.)

The initialization must establish the truth of the loop invariant. By the definition of the game, 13 matches are initially in play. It was not specified which player starts. In the initialization one can, therefore, write either $S := 1$, $S := 2$, $S :=$ a random selection of 1 or 2 or $S :=$ the players' selection of 1 or 2. For the sake of simplicity we will write $S := 1$. Thus the initialization is:

$N := 13$
$S := 1$

The **while** condition, "Game not yet ended", is, expressed differently, 'there are still matches in play' or, more simply, $N > 0$.

The change of players' turns can be written in various ways:

if $S = 1$ **then** $S := 2$ **else** $S := 1$ **endif** or
if $S = 2$ **then** $S := 1$ **else** $S := 2$ **endif** or
$S := 3 - S$

Notice that:

$\{S{=}1$ **or** $S{=}2\}$ **if** $S = 1$ **then** $S := 2$ **else** $S := 1$ **endif** $\{S{=}1$ **or** $S{=}2\}$
$\{S{=}1$ **or** $S{=}2\}$ **if** $S = 2$ **then** $S := 1$ **else** $S := 2$ **endif** $\{S{=}1$ **or** $S{=}2\}$
$\{S{=}1$ **or** $S{=}2\}$ $S := 3 - S$ $\{S{=}1$ **or** $S{=}2\}$

i.e. executing any of these statements preserves the loop invariant.

When the loop ends, it would be player S's turn to remove a match if any were left. The other player must, therefore, have taken the last match, thereby losing, i.e. upon termination of the loop, player S is the winner.

The above considerations lead to the following more detailed version of the program:

$N := 13$
$S := 1$
while $N > 0$ **do**
    Process player S's turn.
    $S := 3 - S$
endwhile
Display a message indicating that player S won.

The processing of one player's turn must still be described in more detail. It is reasonable to structure it as follows:

1 Display the state of the game (the number of matches in play and whose turn it is).
2 Ask player S how many matches he wishes to remove. His decision becomes the value of the variable W.
3 $N := N - W$.

It is necessary that the execution of these three steps preserves the loop invariant. Step 1 does not change the value of any variable and can therefore be neglected.

After executing step 3 it should be true that

($N \geq 0$) **and** (N is an integer)

If we replace $N - W$ for N in this proposition, we obtain the following condition, which must be satisfied before step 3 is executed:

($N - W \geq 0$) **and** ($N - W$ is an integer)

Before the execution of step 3, N is an integer. Therefore, W must be an integer. W must, therefore, fulfill the following condition:

($W \leq N$) **and** (W is an integer)

The rules of the game require that each player remove 1, 2 or 3 matches in turn. The complete condition which W must satisfy is, therefore,

($W = 1$ **or** $W = 2$ **or** $W = 3$) **and** ($W \leq N$)
and (W is an integer)

Because

($W = 1$ **or** $W = 2$ **or** $W = 3$) $\Rightarrow$ (W is an integer)

the condition

($W = 1$ **or** $W = 2$ **or** $W = 3$) **and** ($W \leq N$)

suffices as a precondition with respect to the statement $N := N - W$.

The program segment which asks the player for his decision must ensure that this condition is satisfied. This suggests the following pseudocode:

loop
    Ask player S how many matches he wishes to remove (W).
if ($W = 1$ **or** $W = 2$ **or** $W = 3$) **and** ($W \leq N$) **then exit endif**
    Inform player S of his error and the allowed choices.
endloop

Does this loop terminate? In general one cannot answer this question positively, because in principle the player can repeatedly enter an invalid choice. More important in this practical situation is the question, *can* this loop terminate, i.e. is there always some value of W which satisfies the termination condition? It is obvious that $W = 1$ satisfies the termination condition if the integer $N > 0$. This is guaranteed by the **while** condition (see above).

Our program has now taken on the following form, from which it need only be translated into the desired programming language:

$N := 13$
$S := 1$
while $N > 0$ **do**
    Display the state of the game (N matches remain and it is player S's turn).
    loop
        Ask player S how many matches he wishes to remove (W).
    if ($W = 1$ **or** $W = 2$ **or** $W = 3$) **and** ($W \leq N$) **then exit endif**
        Inform player S of his error and the allowed choices.
    endloop
$N := N - W$
$S := 3 - S$
endwhile
Display a message indicating that player S won.

Does the **while** loop terminate? Since $W \geq 1$, N will be reduced by at least 1 during each execution of the body of the **while** loop. After at most 13 (the initial value of N) passes through the loop, $N = 0$ and the termination condition of the **while** loop will be satisfied. The **while** loop will, therefore, always terminate.

Provided that no player prevents the progress of the game by repeatedly entering an invalid choice, the program will always terminate. If this possibility is to be excluded, corresponding statements must be added to the program. Such a program segment might generate a valid choice (e.g. $W = 1$) after a player has entered invalid decisions a certain number of times consecutively.

An additional 'cosmetic improvement' to this program would be to suppress the request for the player's choice when $N = 1$. In this case the player whose turn it is has only one valid choice: he must remove the one remaining matchstick. The program could set W equal to 1 and inform the player of his only allowed choice.

The reader should pay particular attention to the way in which the proof of correctness (which was only sketched here) and the program were developed hand in hand. From specific requirements of the proof, certain program statements and conditions were derived. In this manner one can construct a program with a relatively simple proof and with a logically clean structure. Perhaps more importantly, all eventualities are covered. Often, appropriate extensions and improvements are identified while designing the program.

Exercise

29 What is the loop invariant of the inner loop (**loop/endloop**)? Hint: Transform this loop into a **while** loop.
30 Extend the program as suggested above so that (a) a player is prevented from entering invalid choices endlessly and (b) the player's decision is preempted when $N = 1$.

6.10 Control program for a management game (example)

In this section we will consider the problem of designing a control program for a management game and, of course, proving it correct. Our design for the control program must be unambiguous and sufficiently detailed that it can be directly translated into a particular programming language.

6.10.0 Statement of the problem: design requirements

The management game for which the control program is to be constructed simulates the development of several competing companies over time. At the beginning of each round of play (which represents one period of time) each company's management team makes a series of decisions (e.g. production volume, advertising budget, prices, etc.) which are entered into the computer system. After the decisions for all companies for the coming time period have been entered, the system calculates and prints the results (sales, costs, profit, etc.) for each company. On the basis of this information, the management teams make their decisions for the succeeding time period and the cycle is repeated.

The management game is intended to be used in a seminar environment. Each round of play may last several hours. Lectures, discussions, exercises

on related topics, etc. may be interspersed between rounds of play. A single game may extend over several days and may even be continued weeks or months later.

Between the rounds of play the trainer also enters decisions (e.g. gross national product, total market demand, etc.). This enables him to exercise a certain amount of influence on the management game and the companies' performance.

The number of companies remains constant throughout the entire game.

The control program is to be designed and constructed in such a way that

1 the several management teams and the trainer can enter their decisions in any sequence,
2 the decisions of the individual management teams and of the trainer can be revised as desired as long as the calculations for the affected time period have not been initiated,
3 the management game can be reset to a previous time period (e.g. in order to investigate the effects of alternative decisions) and
4 the management game can be interrupted and restarted any time later at the same point in the game.

In addition, the control program and its several subprograms are to be written so that they offer the most appropriate default values available whenever decisions or other input from the operator is requested.

Results for one time period can be calculated only if the decisions for the time period in question have been made and entered by all management teams and by the trainer. The results of the previous time period are also required by the algorithm for the computation. (Status information carried over from one time period to the next includes the opening balance sheets, factory capacities, numbers of employees, etc.)

Seen from the standpoint of the trainer, who will operate the computer, the management game should run in a cycle consisting of the following steps:

1 Information on the state of the game is displayed.
2 The functions permitted at the current point in the game are displayed. The system suggests one function to the operator.
3 The system asks the trainer to select a function. Only a valid choice is accepted by the machine.
4 The selected function is executed.

The functions required are:

enter the decisions of one management team for one time period,
enter the trainer's decisions for one time period,
calculate and print the results for the coming time period,
reset the game to a previous time period,
interrupt the management game.

6.10.1 Files

The various pre- and postconditions and the loop invariant will refer mainly to the values of variables. It is, therefore, appropriate to give some thought to the program's variables (data) and the structure thereof before specifying the conditions and loop invariants in detail.

The description of the management game (see Section 6.10.0) above states that each of the following groups of data (variables) is generated in a single logical step:

the decisions of the trainer for one time period,
the decisions of one management team for one time period and
the results of the game calculations for one time period (but for all companies).

At any point in the game, it must be possible to interrupt the program, switch off the computer, switch it back on later and restart the game program at the point at which the game was interrupted. The first function selected may be to reset the game to an earlier time period. This can be done only if the above data groups are still available for the corresponding time periods. Because the game may be reset to any previous time period, all data groups for all previous time periods must still be available.

Thus, all of these data groups for all time periods must be stored on non-volatile memory devices such as disk. Using currently common terminology in connection with such 'peripheral' equipment, we will refer to each logical data group defined above as a 'file'. For example, one file will contain all decisions for one management team for one time period; another file, the decisions for the same management team for another time period, etc.

Also the status of the game must, of course, be maintained by the system. The values of the status ('control') variables must also be stored in non-volatile memory, at least between an interruption of the game and the subsequent restart. We will, therefore, specify that the values of such control variables be maintained in one control file.

Our system's collection of non-volatile data consists, then, of the following files:

one trainer decision file for each time period,
one management decision file for each company and for each time period,
one result file for each time period and
one control file.

It should be reiterated that the term 'file' here refers to a single logical group of data which is, from the viewpoint of the control program at least, an indivisible unit. The values of variables stored in a file are maintained even though the computer is switched off, i.e. these variables are not in effect released upon interrupting the game, switching off the computer, etc. These are, for our purposes, the relevant characteristics of a file, not the details of its storage format, access paths, etc.

6.10.2 Functional subprograms

The subdivision of the entire management game program into the subfunctions (subprograms) below follows naturally from the description of the desired operational cycle and the functions required (cf. guideline in Section 6.0.1). Each subprogram below performs a logically complete and relatively isolated (closed) function. The control logic required within each subprogram is independent of the overall control logic chosen for the operation of the system as a whole.

These subprograms will be called by the control program which we are designing. Note that certain prerequisites are assumed; each subprogram can be assumed to function properly only when those preconditions are satisfied. When they are met, the subprogram in question is to deliver results in accordance with the stated postconditions. These pre- and postconditions, which constitute the specifications of the respective subprogram, represent design decisions which attempt (a) to simplify each subprogram and their interaction and (b) to separate distinct operations (e.g. file I/O, dialog with the operator, etc.).

The subprograms which we will use are as follows:

1 Enter the decisions of one management team. This subprogram conducts a dialog in which the operator enters the decisions of one management team for one time period. Before this subprogram is called, a complete set of decisions made by this team must already be stored in main memory. These decisions will be used as default values. They may be either decisions for the previous time period or previously entered decisions for the current time period which are to be revised.

2 Enter the trainer's decisions. This subprogram conducts a dialog in which the operator enters the trainer's decisions for one time period. Before this subprogram is called, a complete set of trainer's decisions must already be stored in main memory. These decisions will be used as default values. They may be either decisions for the previous time period or previously entered decisions for the coming time period which are to be revised.
3 Read the decisions of one management team from a file into main memory. This subprogram reads the decision file for the management team *team* and for the time period *rdtime*. The subprogram assumes that the file exists and contains valid data.
4 Read the trainer's decisions from a file into main memory. This subprogram reads the trainer decision file for the time period *rdtime*. The subprogram assumes that the file exists and contains valid data.
5 Write decisions of one management team from main memory into a file. This subprogram writes the decisions of the management team *team* for the time period *wrtime* into the appropriate file. If this file does not already exist, this subprogram will create it (i.e. in effect declare the corresponding variables).
6 Write the trainer's decisions from main memory into a file. This subprogram writes the trainer's decisions for the time period *wrtime* into the appropriate file. If this file does not already exist, this subprogram will create it (i.e. in effect declare the corresponding variables).
7 Calculate and print one time period. This subprogram and its subsidiary routines (a) perform the calculations for the time period *calctime*, (b) store the results into the appropriate file and (c) print the reports for the trainer and the management teams. This subprogram assumes that all required files exist and contain valid data. This subprogram requires the decisions of the trainer and of all management teams for the time period to be processed (*calctime*) as well as the results from the preceding time period (*calctime* − 1). This subprogram will create a file for the newly calculated results (i.e. in effect declare the corresponding variables) if necessary.

Note that the subprograms which write files (as specified in items 5, 6 and 7 above) are more conveniently programmed for operation on a system whose file management system contains open for write commands with automatic file creation (cf. guideline in Section 6.0.0 regarding identifying restrictions imposed by the target programming language). If the target system does not offer such a form of open statement, the designer should, at this stage of the design process, specify how this function should be

implemented. If additional control variables are required (e.g. indicating which files physically exist, regardless of the validity of their contents), they should be explicitly included in the pre- and postconditions above and in the loop invariant for the control program below.

6.10.3 Control variables and the loop invariant

Generally speaking, the files described in Section 6.10.1 above reflect the state of the game. More precisely, a complete specification of the state of the game would indicate exactly which files exist and contain valid data. Because files of any one type must exist for all prior time periods (see the argument in Section 6.10.1 above), it suffices to maintain information indicating the last time period for which

a trainer decision file exists,
a management decision file for company 1 exists,
a management decision file for company 2 exists,
a management decision file for company ... exists and
a result file exists.

Therefore, we define the following control variables:

ncomp Number of companies (management teams)

ltpmd(c) Last time period for which valid decisions of the management team of company c ($c = 1, 2, \ldots,$ *ncomp*) are present and stored in the appropriate file

ltptd Last time period for which valid trainer's decisions are present and stored in the appropriate file

ltpres Last time period for which valid computed results are present and stored in the appropriate file. 'Valid' here is construed to mean that the values of the variables in the result file for a time period were calculated from the values of the variables in all decision files for the same time period and in the result file for the previous time period.

Strictly speaking, *ncomp* is not a control variable but rather a fixed parameter. Because its value indicates the number of variables *ltpmd*(c) which are present, it is convenient to store it in the control file.

The values of these control variables are maintained both in a file (the 'control file') and in main memory.

The time period *ltpres* is the last one which has been completely processed. The time period of current interest is, therefore, *ltpres* + 1. We will

presumably refer frequently to this time period in the control program. The program will be more readable if we introduce a variable with this value. This redundant variable will be maintained only in main memory, i.e. it is, in effect, released when the game is interrupted and is declared anew when the game is restarted:

curtime The current time period, i.e. the time period for which results are to be calculated and printed next and for which decisions must be entered (if not already present).

The loop invariant is:

1 *ltpres* $\leq$ *ltpmd*(c), for all c, **and**
2 *ltpres* $\leq$ *ltptd* **and**
3 *ltpres* ≥ 0 **and**
4 the values of the control variables (see above) in main memory and in the control file are equal **and**
5 for every company c and for every time period from 0 to *ltpmd*(c) inclusive a management decision file with valid contents exists **and**
6 for every time period from 0 to *ltptd* inclusive a trainer decision file with valid contents exists **and**
7 for every time period from 0 to *ltpres* inclusive a result file with valid contents exists (see the definition of the variable *ltpres* above) **and**
8 *curtime* = *ltpres* + 1.

Terms 1 and 2 above reflect the requirement that all decision files must be present before the result file for any particular time period can be calculated.

Term 3 amounts to nothing more than a definition of the initial point in time.

Term 4 specifies that the control file will be maintained in every pass through the loop, not only when the operator chooses to interrupt the game. This adds a certain degree of robustness to the operation of the system. If the control file is updated as the last step in the execution of the loop, an equipment or power failure at most points within the loop will still leave the system's files in a consistent state and at most the results of the last function selected will be lost.

Terms 5, 6 and 7 state simply that the values of the control variables fulfill the definitions of these variables.

Term 8 follows from the definition of the variable *curtime* above. The equality highlights the fact that this variable is redundant.

Alternatively, terms 1 and 2 above may be replaced by:

1(a) *ltpres* $\leq$ *ltpmd*(c) $\leq$ *ltpres* + 1, for all c, **and**
2(a) *ltpres* $\leq$ *ltptd* $\leq$ *ltpres* + 1 **and**

The alternative forms of these two terms in the loop invariant represent different answers to the question whether valid decisions may exist for future time periods. Such decisions could, in principle, exist either (a) because the game has been reset to a previous period or (b) because teams were allowed to enter decisions for periods beyond the next one to be calculated. The specifications given in Section 6.10.0 do not explicitly indicate whether or not decisions applicable to future periods are to be considered valid or permitted. Decisions for future periods might be desirable if, for example, the results computed for a particular time period include forecasts of future trends, where such forecasts are based on companies' strategies for the future. In an actual software design project, this question should be discussed with the user or other responsible persons for whom the system is being developed.

In our design, we will consider the more general situation in which decisions may be entered for future periods. The reader should note which parts of the resulting program would be affected by specifying the more restrictive alternate terms 1(a) and 2(a) in the loop invariant.

6.10.4 Initial conditions

The first time period for which decisions are to be entered and calculations performed is time period 1. The specifications of the calculation process and the decision entry functions imply that both result and decision files for the preceding time period must be present. Therefore, at the beginning of the management game, a complete set of files for the time period '0' must exist: a decision file for each company, a trainer decision file, a result file and a control file.

The decision and result files for time period '0' must contain valid data as required for the proper functioning of the subprograms which access these files. That is, the contents of these files must satisfy any conditions (e.g. data invariants) imposed upon the corresponding files for later time periods. In the initial control file all control variables 'last time period ...' will have the value 0. Note that these values satisfy the loop invariant.

6.10.5 The control program

The structure of the control program follows relatively straightforwardly from the specification of the operational cycle (see Section 6.10.0) and, of course, the need to establish the truth of the loop invariant by initializing variables before executing the main loop:

Read control file (control variables *ncomp*, *ltpmd*(*c*) (for all *c*), *ltptd* and *ltpres*)

curtime := *ltpres* + 1
loop
    Display the state of the game (see Section 6.10.5.0 below).
    Display valid functions (see Section 6.10.5.1 below).
    Suggest a function to the operator (see Section 6.10.5.2 below).
    Request choice of function (see Section 6.10.5.3 below).
if selected function = interrupt management game **then exit endif**
    Perform selected function (see Section 6.10.5.4 below).
endloop

6.10.5.0 Display the State of the Game

The values of the following variables are displayed in a suitable format:

curtime
ltpmd(*c*), for $c = 1, 2, \ldots,$ *ncomp*
ltptd
ltpres

6.10.5.1 Display Valid Functions

The following functions are valid choices at the current state of the game:

1 Enter the decisions of one management team or of the trainer for any time period from *curtime* to *ltpmd*(*c*) + 1 or *ltptd* + 1 respectively and inclusive (for the time period *curtime* only if the alternative terms 1(a) and 2(a) are specified in the loop invariant, see Section 6.10.3 above).
2 Calculate and print the results for the time period *curtime* if and only if
    curtime $\leq$ *ltpmd*(*c*), for all *c* **and**
    curtime $\leq$ *ltptd*
3 Reset the game to a previous time period if and only if
    curtime $>$ 1
4 Interrupt the management game.

6.10.5.2 Suggest a Function to the Operator

The calculations for the current time period (*curtime*) may be started only when all required decisions have been entered (see point 2 above). The program should suggest entering the set of decisions which is still needed. If two or more sets of decisions are missing, the program should suggest entering the decisions for the company with the lowest identification number. If no decisions are missing, the program should suggest calculating and printing the results for the time period *curtime*.

6.10.5.3 *Request Choice of Function*

This subprogram asks the operator to indicate which function should be executed next. The selected function must be permitted at the current state of the game (see 'Display Valid Functions' above). If an invalid choice is entered, this subprogram displays an appropriate error message and asks again for the operator's selection.

6.10.5.4 *Perform Selected Function*

If the function 'Enter the decisions of one management team' was selected, this subprogram asks the operator for which company. If the response is invalid, an appropriate error message is displayed and the request is repeated.

If the entry of decisions (either of a management team or of the trainer) was selected, then

> the time period *dtime* for which the decisions are to be entered is requested (see Section 6.10.5.1, item 1 for bounds on *dtime*) or set equal to *curtime* if the alternative terms 1(a) and 2(a) are specified in the loop invariant (see Section 6.10.3),
> the decision file for the selected company or for the trainer for the time period min(*dtime*, *ltpmd*(*c*)) or min(*dtime*, *ltptd*) respectively is read into main memory,
> the subprogram 'enter the decisions of one management team' or 'enter the trainer's decisions' is called,
> the decision file for the selected company or for the trainer for the time period *dtime* is written,
> *ltpmd*(*c*) or *ltptd* respectively is set to the value of *dtime*

and the control file is rewritten.

If the function 'calculate and print' was selected, then

> *calctime* is set to the value of *curtime*,
> the subprogram 'calculate and print one time period' is called,
> *ltpres* is set to the value of *curtime*,
> *curtime* is set to the value of *ltpres* + 1 and
> the control file is rewritten.

If the function 'reset the game to a previous time period' was selected, the program asks to which time period the game should be reset. The new time period must be an integer between 1 and *curtime* − 1 inclusive. If the operator chooses a valid new time period, then

> *ltpres* is set to the new time period − 1,
> *curtime* is set to the value of *ltpres* + 1,
> *ltpmd*(*c*), for all *c*, and *ltptd* are set to the value of *curtime* (optional, see the previous remark regarding the alternative loop invariant) and
> the control file is rewritten.

If the operator chooses an invalid new time period, an appropriate error message is displayed; no further function is performed.

Exercise

31 Show that the initialization section of the control program establishes the truth of the loop invariant. What preconditions must be satisfied, if any? How is the truth of such preconditions guaranteed?
32 Derive the bounds stated in Section 6.10.5.1, item 1, on the time period *dtime* for which decisions are to be entered.
33 Show that at least one time period satisfies these bounds, i.e. that

> $curtime \leq ltpmd(c) + 1$, for all c **and**
> $curtime \leq ltptd + 1$

34 Show that *dtime* must be equal to *curtime* if the alternative form of the loop invariant is chosen.
35 Show that requiring *dtime* to be equal to *curtime* is consistent with the less restrictive form of the loop invariant (terms 1 and 2).
36 Explain the reason for the precondition $curtime > 1$ for resetting the game to an earlier time period (see Section 6.10.5.1, item 3).
37 Verify formally that the loop of the control program preserves the loop invariant. Determine the post- and precondition of each statement in the control program. Perform this analysis for both alternative forms of the loop invariant.

Epilogue

Chapter 7

The practice of software engineering tomorrow

So that we may say the door is now opened, for the first time, to a new method fraught with numerous and wonderful results which in future years will command the attention of other minds.

– Galileo Galilei

The empiric school produces dogmas of a more deformed and monstrous nature than the sophistic or theoretic school; not being founded in the light of common notions ..., but in the confined obscurity of a few experiments. ... We could not, however, neglect to caution others against this school, because we already foresee and argue, that if men be hereafter induced ... to apply seriously to experiments ..., there will then be imminent danger from empirics; owing to the premature and forward haste of the understanding, and its jumping or flying to generalities and the principles of things.

– Francis Bacon

Errors, like straws, upon the surface flow;
He who would search for pearls must dive below.

– John Dryden

Thought without learning is dangerous.

– Confucius

Give me ... the engineer; and take your ... relics and miracles.

– Charles Kingsley

7.0 Our point of departure

Correct, error free software – in which one can have confidence from the very outset – has, in practice, remained until today a dream. In other technical and engineering fields such a professional state of affairs is *not* a dream, but has, in fact, been a reality for quite a long time, e.g.

civil engineering,
mechanical engineering,
aircraft construction,
shipbuilding, etc.

These fields were not always professions; earlier, bridges collapsed, ships sank much more frequently than they do now. But in the meantime, the practitioners of these fields have successfully accomplished the transition from a trade or craft to an engineering science. Today, software still fails often; software development has not yet undergone its metamorphosis to software engineering.

Those of us who live and work in buildings, who travel in automobiles and airplanes, who ride over bridges, etc. should be thankful that the designers of those structures verified their designs thoroughly before building them and releasing them for our use. Correspondingly, the users of our software have the right to expect us to verify our designs thoroughly before releasing them for their use. But are we software developers fulfilling these expectations? Are we capable of doing so?

'We build (software) systems like the Wright brothers built airplanes – build the whole thing, push it off the cliff, let it crash, and start all over again.' With these words R. M. Graham described the state of the art of software development in 1968, when the term 'software engineering' first entered into our vocabulary (Naur and Randell 1969, p. 17). The situation is not very different today, as many users of new software packages for microcomputer systems will attest.

More recently C. A. R. Hoare compared software development with recognized engineering disciplines. He pointed out that most setbacks can be attributed to the programmer's mistakes and oversights and that the programmer – unlike the engineer – has no generally applicable mathematical or theoretical foundation for his work (Hoare, 1984). But if, as many believe, programming is 'a tough engineering discipline with a strongly mathematical flavour' (Dijkstra, 1982, p. 273), a foundation comparable to those of other engineering fields is sorely needed for software engineering.

For some years now such a foundation has been under development, but it is being transplanted into practice only slowly. Until now, only a small minority of software developers has mastered this material and applied it in their daily work. There are many reasons for the passive and active resistance to the widespread introduction of such a theoretical foundation for practical work in software development. Lack of awareness of the existence of this body of knowledge, prejudiced attitudes regarding its applicability, inability to understand it, lack of the time needed to learn it, fear of inability to learn it and inadequate prerequisite knowledge all play

a role. In addition, much of the literature on this mathematical and theoretical foundation for software engineering is written in a language and uses terminology more oriented to the scientist advancing the theory than to the engineer interested in applying it. These barriers to placing software development on a sound engineering basis must be overcome in the long run – by supporting and/or threatening present and future software developers in various ways.

7.1 Our goal

Nothing is really changed when we simply attach new terms – such as 'software engineering' – to our old ways of doing things. There are some indications that this approach is already being taken by some, perhaps subconsciously. It is important to realize that renaming alone will not bring about a fundamental improvement. Neither will incremental modifications to current methods. The content and substance of our way of designing software must be changed fundamentally. Our current approach to software development, which is characterized by patchwork and underqualified practitioners, must be replaced by one patterned after that employed in already established engineering fields.

As mentioned in Section 7.0 above, a body of fundamental principles has been developed in recent years which, when applied by a knowledgeable software engineer, makes possible the design and development of error free programs. This material has come to the attention of only relatively few working programmers. In many university level courses in informatics this foundation – if offered at all – is not yet included in the required core curriculum. It must be brought to the attention of many more working software developers and it must become established as a key element in the professional education of all software engineers.

We must also imitate other aspects of the educational processes typical of established engineering fields. In particular, students of software engineering must be willing to study fundamental principles and building blocks in considerable depth and detail. They must not shy away from efforts which appear at first to be rather time consuming, for in the long run these often represent the shortest and quickest path to real mastery of the field. He who is intimidated by the demands of such an intensive approach to learning the field cannot expect to achieve a level of mastery of the material comparable to that achieved by true professionals in other fields. No effortless, automatic path to 'software engineering' exists; it is, therefore, a waste of time to search for one.

Engineering design – the translation of detailed external specifications into a complete and unambiguous description of the internal structure of

the device, system, product, etc. – is characterized by two phases. The first phase is a creative one which is not mechanizable. Guidelines based on theory are useful but insufficient. Subjective insight, experience, practice and even hunches play a role. Different designers with essentially the same knowledge and experience will often create different proposed designs. The second phase is quite different. The proposed design is analyzed systematically, mechanistically and precisely to verify that it fulfills certain criteria, among them the original external specifications. For example, the civil engineer calculates stresses in the structure to verify that the strengths of the materials used will not be exceeded under any environmental conditions to be anticipated. The electrical engineer calculates the voltages and currents at the various points in his circuit to verify that the desired output-signal will be produced in response to the specified input. Tolerances are specifically considered to take account of unavoidable deviations in manufacturing, variations in the characteristics of materials used, etc.

In the case of software engineering, the designer will derive the preconditions for the correctness of the programs in question and will verify that those preconditions will be met. In the process, he will determine preconditions, postconditions and loop invariants at the various intermediate places in his proposed programs as appropriate. In this way, he will verify that the external specifications of his software system will be fulfilled just as engineers in other fields verify their proposed designs.

Properly trained software engineers will be able to achieve impressive results. It should be emphasized, however, that – again, like their colleagues in other engineering fields – they will not be omnipotent magicians. They will not be able to perform miracles and the client who expects them will still be disappointed in the future just as he was in the past (Dijkstra, 1986). In contrast with today's less qualified software developers, the software engineer will be able to determine whether a specification for a proposed system can be reliably fulfilled or not before developmental work is initiated, thus protecting his clients against unpleasant surprises of the sort still all too common today. When the future software engineer states such limitations, the prudent client will recognize this as a strength, a mark of professionalism, instead of as a suggestion of weakness and incompetence as is often the case today.

If the field of software development is to become an engineering field, its practitioners must take a more professional attitude regarding responsibility for the correctness, reliability and safety of their designs. The theoretical foundation for their work mentioned above and presented in this book gives them the technical tools necessary to ensure such correctness. This is not, of course, sufficient. An attitude of professional responsibility must be instilled in them in their initial education and must be supported

throughout their careers by, for example, their professional associations. In addition, it can be expected that society will define and enforce legal liability in one form or another. Some steps in this direction can already be observed in some countries. Present and future software developers must prepare themselves to meet the challenges arising from the assumption of such responsibility.

There are, to be sure, important differences of a basic nature between software engineering and other engineering fields. They derive in the final analysis from the fact that other engineering fields are based on the natural sciences. Their artifacts are composed of physical materials with given properties not under the control of the engineer. Those materials are subject to forces which obey natural laws. Software engineering is not based on the physical sciences; its artifacts are composed of abstract, purely artificial components created entirely and solely by the mind (not even the hands) of man. Those components do not obey natural laws but rather laws defined by man or following directly from his other definitions. This difference has several significant implications.

The engineer in another field must always contend with the possibility that some physical phenomenon not yet known or adequately understood is at work and will render his analysis invalid and his design inadequate. His mathematical models are good, usually excellent, approximations to the real world but can never be considered to be exact. Despite all the scientific and mathematical basis of his field, he works in a world which, in the context of his knowledge, is fickle and will, in principle, always be so. The software engineer, on the other hand, does not have this problem. He defines precisely the components with which he works (e.g. the fundamental programming statements and constructs defined in Chapter 2) and, therefore, knows – or at least can know – all of their relevant properties exactly. He deals only with ideal, perfect objects and therefore his mathematical models of them can be exact. Any appearance of fickleness in the world in which he works is attributable to his incomplete analysis of his systems and models, of the consequences of his own definitions, not to an inherent characteristic of that world.

A well-known and widely publicized example of such a problem in classical engineering was the series of crashes of the Comet jet airliner in the 1950s. Only after extensive, costly investigations was the cause discovered: fatigue in a particular structural element. Fatigue in metals, a physical, metallurgical phenomenon, was not particularly well understood at the time and had obviously not been adequately considered by the aircraft's designers. An earlier example was the Tacoma Narrows bridge. Its designers had not been aware of the potential importance of the load exerted upon the bridge by certain aerodynamic effects. Soon after it was

built, the bridge was twisted apart by aerodynamically induced oscillations during a heavy wind. This also well-publicized collapse had a major and lasting impact on bridge design.

Superficially comparable software collapses are not and will not be attributable to a similar and inherent lack of knowledge about the underlying phenomena. Instead, they are and will continue to be due solely to human mistakes and oversights on the part of software designers who do not make adequate use of the knowledge and information available to them.

Mathematics alone plays the role in software engineering which mathematics and the relevant natural sciences together play in other engineering sciences. This implies that mathematics is even more important to the software engineer than it is to other engineers. The education of software engineers and the practice of software engineering will increasingly reflect this reality.

Another implication of the characteristics of software engineering discussed above is that it can be expected to be a less experimental field than many other engineering sciences. There is no need to measure physical constants, to determine empirically the strengths or other properties of materials used, etc. There is less need to investigate experimentally the various phenomena arising from the definitions of the several fundamental components combined by the software engineer to form his systems of programs. Thus, experimentation and testing will undoubtedly play an even smaller role in software engineering than they do in the design phases of other engineering fields.

Another difference between software engineering and traditional engineering fields relates to the mathematical nature of their respective components. Engineers in fields based on the physical sciences employ mainly mathematics dealing with continuous and differentiable functions, for these describe the real physical world most accurately. As one professor of nuclear physics was wont to say, 'Nature abhors sharp corners.' While the software engineer will use such functions and the calculus for some of his analyses (especially of the time complexity, memory complexity, etc. of his algorithms), most of his work will deal with logical expressions, propositions, etc. Consequently, his mathematics will deal mainly with discrete domains and ranges, where the notions of continuity and differentiability are of no meaning – a world in which smooth corners simply do not exist.

Among the implications of this situation is a particularly important one regarding testing. Because physical systems are best described by continuous functions, one can usually reasonably assume that if a system works at all extreme points, it will work at all intermediate points as well. This assumption underlies essentially all testing in traditional engineering work. But since the software engineer's systems cannot typically be adequately described in terms of continuous functions, this assumption underlying

testing is not generally valid. Only exhaustive testing can demonstrate empirically the correctness of a design and such testing is usually impossible. When in principle possible, it is almost always impractical. One is left with thorough mathematical analysis as a replacement for other engineers' testing to demonstrate that a design fulfills its specifications.

The above comments are also indicative of differences between software engineering tomorrow and programming today. Today, the typical software developer does not employ mathematical analysis, e.g. of the type presented in this book, in designing or verifying his programs. (Most of the reasons were listed in section 7.0 above.) Tomorrow's software engineer will do so.

Today's programmer relies heavily on 'testing' to get his designs more or less right. 'Testing' here is a misnomer for 'design by trial and error' or, recalling the Mocsian three week wonders (Baber, 1982, Ch. 0), 'try building it and see if it collapses'. This form of 'testing' will essentially disappear from the design phases of software development. Testing will remain in software engineering only in the same form as it appears in other engineering work – to demonstrate that the external specifications have been fulfilled. But even there, for the reasons discussed above, testing will probably assume a lesser role, often becoming merely a subjective demonstration supplementing the more conclusive and reliable analytical verification.

In summary, the field of software development will be increasingly characterized by an engineering approach. Sometime in the future, the practice of software engineering will be based on a theoretical foundation and will rely even more heavily on mathematical analysis than do already established engineering fields. Like other engineers today, the software engineer will accept professional responsibility for the correctness of his designs and for his technical ability to ensure that correctness. Much of his effort will be devoted to avoiding errors in the first place.

7.2 The transition

The transition from software development as a trade or craft to a more professional engineering field has been discussed for some years already. Progess has been slow but is apparently accelerating. This evolutionary pattern will undoubtedly continue.

There are many vested interests in the status quo and the consequent resistance to change will impede progress. In the course of the step by step development of the field some of these vested interests will, however, be injured from time to time, making life difficult for some practitioners. Some will not survive.

The driving force behind the transition to the professional practice of software engineering will be an increase in the qualifications of the

field's practitioners. Most importantly, the fraction of practitioners with a professional education will increase. The main mechanism by which this change will come about will be natural attrition. A significant and increasing fraction of new entrants to the field will be graduates of university courses in software engineering, informatics and computing science, while relatively few, if any, persons leaving the field will have such a background. The contributions made by the true professionals will be out of proportion to their number. This will, in turn, increase the standards against which non-professional programmers will be measured, inducing them to improve their technical understanding fundamentally or, failing that, causing them to be relegated to positions of lesser importance, e.g. coding technicians with less responsibility. As a result, the influence of the professional software engineers on the development of the field will in time increase and the influence of the non-professionals will decrease. This process will not occur suddenly and discontinuously but rather it will take place slowly but surely over an extended time.

Present software developers can expect to feel competitive pressure especially from recent graduates of university level courses in informatics and software engineering. Their more extensive preparation and better theoretical foundation will enable them to benefit from practical experience rapidly, so that they will soon overtake their non-professional working colleagues with longer experience.

Several alternatives are open to experienced software developers who become threatened by newer, better qualified entrants to the field. Some will try to learn the most important and fundamental new concepts and theory, some of them successfully, some not. Others will seek refuge in management positions, again, some successfully, some not. Still others will yield to the threat and leave the field of software development, some voluntarily, some not. The remainder will stand fast and resist, mostly unsuccessfully in the long term. The excess of demand over supply of software developers will tend to minimize the rate at which practitioners leave the field, but recessions in demand will cause temporary oversupply from time to time. During these periods the less qualified will tend to become squeezed out of the field.

Those non-professionally trained software developers who successfully move into management will be able to take advantage of their experience and put it to good use. They will not be immune, however, to competitive pressure from the more recent professional entrants to the field, but rather will enjoy only a respite. At a later time, the replacement process will continue at the level of technical software management as well.

The software developers most threatened by the process outlined above will be the recent and future non-professional entrants to the field (the

Mocsian 'three week wonders'). In the immediate future, the pressure described above will not be too apparent but it will increase continually throughout their entire careers. Clearly, many of them will not be able to withstand it over the course of a career lasting some 40 years.

Some precursors of changes of the types outlined above can already be discerned. More and more universities and colleges are offering degree programs in informatics, computing science and software engineering. These programs are growing quantitatively and qualitatively. They are popular among prospective students as well as their graduates' employers. These trends can be observed in different countries.

In most cases, software engineering subjects are taught within the department of computing science. No clear separation of such departments into two – science and engineering – is yet apparent, although some observers are convinced that very preliminary indications of such a division can be detected.

Engineering societies are engaging more actively in computing. While their entry into the field was typically based in hardware areas (e.g. electrical engineering and communication), they are devoting ever more attention to software. Again, these developments can be observed in different countries.

Associations of computing specialists are cooperating more closely with established engineering societies. This cooperation takes on various forms such as jointly sponsoring technical conferences, jointly publishing individual issues of their professional journals, seriously considering merging, formally joining a national federation of engineering societies, etc.

7.3 Concluding remarks

The history of established engineering fields strongly suggests that a mathematical and theoretical foundation of the type presented in this book will become fundamental to the education of software developers and to the professional practice of software development. It therefore behooves each software practitioner of today to give serious thought to his own personal transition to tomorrow's world of software development – what role he can realistically expect to play in that world, how he can best acquire the new knowledge and expertise he will need, etc. Some, believing or fearing that they cannot acquire an adequate understanding of the mathematical and theoretical foundation needed by the professional, will reject this material. They should candidly assess their chances of survival in the field, for the status of many working programmers will, especially in the long run, be threatened by the increasing numbers of new, professionally qualified entrants to the field. In time, these professionals will ensure that practical software development will be based on principles of the type

presented in this book; those who are unwilling or unable to acquire an adequate working knowledge of this theoretical foundation will, slowly but surely, be relegated to ancillary positions or squeezed out of the field entirely.

Today, software development appears to be well into an initial phase of transition to an engineering field. There have been many technical changes to date, but a more fundamental change is on the horizon. The preparation of practitioners will constitute the first area of major change. A corresponding change in the practice of the field will follow in due time. Present software developers will have to adapt to these changes by acquiring new, more fundamental knowledge such as that presented in this book, transferring to (temporarily) safe positions in management or involving very specialized technical (as opposed to engineering) skill, accepting less responsible positions as software technicians or allowing themselves to be squeezed out of the field sooner or later.

This book has presented a significant part of the mathematical and theoretical foundation which can enable the software engineer to develop error free programs. Critical prerequisites for applying these concepts successfully are a thorough understanding of them together with fluency in the relevant mathematics – just as in the case of every other engineering field. Mastery of some set of tools and techniques is not enough – it is, in the language of mathematics, a necessary but not a sufficient condition for success.

Much can be achieved with relatively little, not especially advanced mathematical knowledge. Rather, a mathematical way of thinking is the essential element. A program or a part thereof is viewed as a mathematical object – just as we looked upon points, lines, triangles, spheres, numbers, etc. as mathematical objects when we were in school. Theorems about certain properties of the mathematical object in question are formulated and logically proved. In order to do this, certain axioms and assumptions are presumed.

Such an approach is useful not only when applied to existing or proposed programs. Often, substantial parts of a program can be derived from a proof of correctness or a sketch of such a proof.

Probably the greatest problem arising in the application of the concepts presented in this book is formulating the theorems which represent the criteria of 'correctness', i.e. which give meaning to the term 'correct' as applied to a particular program. This problem can be solved only by mastering the language in which these theorems are best and most conveniently expressed – i.e. by mastering the language of mathematics. Again as in the case of other engineering fields, one need not be a mathematician, but one must be able to 'speak mathematics'.

We software developers want to be professionals. But are we? What must we do to become such? It is characteristic of all professionals that they view mistakes as signs of incompetence and take extensive steps to reduce their error rate as much as possible. Most pilots never crash. Most surgeons never kill a patient. Most civil engineers never design a building or a bridge which collapses. Only when most software developers regularly deliver error free programs to their clients will we be able to convince others that we are professionals, software engineers in the true sense of the word.

Appendix 0

Mathematical fundamentals

> This, therefore, is mathematics: she reminds you of the invisible form of the soul; she gives life to her own discoveries; she awakens the mind and purifies the intellect; she brings light to our intrinsic ideas; she abolishes oblivion and ignorance which are ours by birth.
>
> – Proclus

> In the mathematics I can report no deficience, except it be that men do not sufficiently understand the excellent use of the pure mathematics, in that they do remedy and cure many defects in the wit and faculties intellectual.
>
> – Francis Bacon

This appendix reviews those specific mathematical topics upon which the material presented in the body of this book is based. For the most part, this consists of definitions of the relevant mathematical terms. Many of these definitions are presented in two forms: an informal explanation or description of the concept involved and a formal mathematical definition. The informal explanation serves usually to motivate the introduction of the term and to convey a general, subjective understanding of the concept in question. The formal definition, in rigorous mathematical language, makes the term precise.

This appendix does not purport to treat these subjects thoroughly. Rather, it is intended to (a) refresh the reader's memory in certain key areas and (b) permit him to extend his knowledge to a limited extent into areas which are new to him. For more information on these subjects, the reader is referred to the many mathematical texts on analysis, functional analysis and algebra. Additional material on topics covered in this appendix can often be found in introductory chapters or appendices in text books on other mathematical subjects, e.g. groups, linear algebra, measure theory, probability theory, etc. The more advanced reader will also find mathematical

texts on logic, predicate calculus and related subjects of interest and relevance.

The main topics covered below are sets, sequences, functions and Boolean algebra.

A0.0 Sets

A0.0.0 Basic definitions

A *set* is a collection of distinguishable objects. Any one of the objects comprising a particular set is called an *element* or a *member* of that set.

By distinguishable we mean that each element in a set is different from every other one. In other words, one element cannot appear more than once in a particular set. If a set is specified by enumerating its elements and the same element is listed more than once, it is included only once in the set.

A set may contain a finite number of elements, infinitely many (an 'unbounded number' of) elements or no elements. A set containing no elements is called the *empty* or *null* set. A set containing exactly one element is sometimes called a *singleton* set.

Example 1: The set {1, 2, 3} is a set containing three elements, the numbers 1, 2 and 3.

Example 2: The set {1, 2, 3, 4, ...} contains infinitely many elements. This set is the set of *natural numbers*; each of its elements is called a natural number.

A particular set is defined by specifying, directly or indirectly, which objects it contains, i.e. by specifying its elements. When defining a set, one should be careful that the purported definition is logically consistent. Particularly when the definition of a set contains references to the set being defined, must one beware of ambiguities and paradoxes. Such recursive references may be explicit or implied by such terms as 'all sets', 'any set', etc. The mathematical literature contains a number of famous paradoxes illustrating logical difficulties in this area (see, for example, Kline, 1980, pp. 204 ff.).

A simple way of defining a set is to list all of its elements, as in the two examples above. When defining a set in this way, the order in which the elements are listed is of no consequence. Another way to define a set is to state a condition; every object (element) which satisfies the condition is a member of the set and any potential element which does not satisfy the

condition is excluded from the set. Such a definition of a set is usually written in the following form:

$$S = \{x \mid C(x)\}$$

meaning that the set S contains every x which satisfies the condition C – and no other elements.

The *magnitude* or size of a set is the number of elements it contains. If, for example, the set A is defined to be the set {true, false}, then the magnitude of A (usually written $|A|$) is 2, i.e. $|A| = 2$.

It sometimes happens that one set contains all the elements of another set. In such a case, we say that the latter set is a *subset* of the first set. More precisely,

Definition A0.0: If A and B are sets and if every element of A is also an element of B, then the set A is a *subset* of B.

If A is a subset of B, one sometimes says that B is a *superset* of A.

If the set A is empty, then by convention the above definition of subsets is interpreted in such a way that A, the empty set, is a subset of any set B.

Two sets which contain exactly the same elements are, by the above definition, subsets of each other. This provides an appropriate and convenient condition for a definition of equality of sets:

Definition A0.1: The sets A and B are *equal* if A is a subset of B and B is a subset of A.

When the set A is a subset of set B, but not vice versa, then the set A is 'smaller' than the set B in the sense that B contains one or more elements which are not elements of A. In such a case, we say that A is a *proper subset* of B.

A0.0.1 Combination of sets

It is useful to combine sets in certain ways. One such combination forms a set by merging or lumping all the elements in two or more given sets together. More formally:

Definition A0.2: The *union* of two given sets A and B is the set consisting of those elements contained in either of the given sets (and only of those elements). Symbolically,

$$A \cup B = \{x \mid x \text{ is an element of } A \text{ or } x \text{ is an element of } B\}$$

The connective 'or' in the above condition is to be interpreted so that an x which is an element of both A and B satisfies the condition; i.e. such an element is a member of the union of A and B.

The above definition can be extended in a straightforward and obvious way for more than two sets. The union of any number of sets contains every element which is contained in any one (or more) of the given sets.

The phrase 'is an element of' occurs frequently in such expressions and formulae. It is, therefore, often shortened to 'is in' or simply 'in'.

Another commonly useful combination of sets is the set containing only those elements which belong to *all* of the given sets:

Definition A0.3: The *intersection* of two given sets A and B is the set consisting of those elements contained in both of the given sets (and only of those elements). Symbolically,

$$A \cap B = \{x \mid x \text{ in } A \text{ and } x \text{ in } B\}$$

The difference of two sets is the set of elements contained in one set but not the other:

Definition A0.4: Given two sets A and B, the *difference*

$$A - B = \{x \mid x \text{ in } A \text{ and } x \text{ not in } B\}$$

For a variety of reasons, it is often desirable to take one element from one set and a second element from another set and consider the pair of elements selected, keeping track of which element came from which set. The set of all such pairs of elements is, in such cases, also of interest. We formally define such a structure as follows.

Definition A0.5: The *cartesian product* of the sets A and B is the set of all ordered pairs, the first element of which is in A and the second, in B. Symbolically,

$$A \times B = \{(x, y) \mid x \text{ in } A \text{ and } y \text{ in } B\}$$

The word 'ordered' in the above definition expresses the restriction that, for example, the pair (1, 2) is not to be considered the same as the pair (2, 1). By maintaining this distinction, we keep track of which element came from which set as required by the informal definition. This distinction is still made when the two given sets are equal.

The above definition can be generalized to the case of more than two given sets. The cartesian product of n sets is the set of all ordered n-tuples, the first element of which is in the first set, the second element of which is in the second set, etc.

Example 3: If $A = \{1, 2, 3\}$ and $B = \{\text{true}, \text{false}\}$, then the cartesian product of A and B is the set

$$A \times B = \{(1, \text{true}), (1, \text{false}), (2, \text{true}), (2, \text{false}), (3, \text{true}), (3, \text{false})\}$$

and the cartesian product $B \times A$ is the set

$$B \times A = \{(\text{true}, 1), (\text{false}, 1), (\text{true}, 2), (\text{false}, 2), (\text{true}, 3), (\text{false}, 3)\}.$$

Note that $A \times B \neq B \times A$. Note also that $|A \times B| = |B \times A| = |A| * |B|$.

A0.1 Relations, functions and expressions

A0.1.0 Relations

It is often meaningful and convenient to associate one or more elements of one set with one or more elements of a second set. If desired, the two sets may be equal. Such an association, or *relation*, can be specified by listing all pairs of elements which are to be associated with one another.

Example 4: Consider a set P of persons and a set L of languages. We will relate (associate) with each person (element) in P the language or languages in L which that person speaks. If the set of persons is

$$P = \{\text{Dominique, James, Giovanna, Carlos}\}$$

and the set of languages is

$$L = \{\text{French, English, Italian, Spanish}\}$$

then the relation between the elements of these sets might be

$$R = \{(\text{Dominique, French}), (\text{Dominique, English}), (\text{James, English}), (\text{Giovanna, Italian}), (\text{Giovanna, French}), (\text{Carlos, Spanish}), (\text{Carlos, French}), (\text{Carlos, English})\}$$

Note that the above relation R is a subset of the cartesian product $P \times L$.

The last observation in the above example is a general one and provides the motivation for the following definition.

Definition A0.6: A *relation* between (or on) two given sets A and B is a subset of the cartesian product $A \times B$.

The above definition can be generalized to more than two given sets in the obvious manner.

In the above example, some persons speak more than one language and some languages are spoken by more than one person. In some special but mathematically interesting cases, a less general situation prevails.

Example 5: Consider the set P of persons in the above example and their ages. The corresponding relation Ra between P and Y, the set of possible ages in years (non-negative integers), might be

$$Ra = \{(\text{Dominique, 25}), (\text{James, 15}), (\text{Giovanna, 45}), (\text{Carlos, 45})\}$$

The set Ra is clearly a subset of $P \times Y$, so it fulfills the definition of a relation. Associated with each person is only one age. Thus, the relation is unique in the direction from a person to an age. It is not unique in the other direction, however. Two people, Giovanna and Carlos, are 45 years old.

A0.1.1 Functions

Relations which are unique in the above sense in at least one direction arise in important situations in mathematical analyses. The term *function* is applied to such relations.

Definition A0.7: A *function* F on a set A to a set B is a relation between A and B which exhibits the following property: If $(a, b1)$ is an element of F and $(a, b2)$ is an element of F, then $b1 = b2$.

In other words, the first element of an ordered pair uniquely determines the second element of that pair. Any particular element in A appears as a left element in an ordered pair in F at most once. The function (relation) F associates with any a in A at most one b in B. A function from the set A to the set B can be viewed as a rule for transforming or converting an element of A into a unique element of B.

In the above definition the sets A and B may be equal. No modification of the above statements is needed.

The notion of a function as a rule suggests the commonly used notation $F(a)$ to represent that element of B which is associated with the element a in A. The element a of the set A is called the *argument* of the function. The element $F(a)$ of the set B is the *value* of the function obtained by applying F to the argument a.

The set A may be the cartesian product of other sets (e.g. $A1$, $A2$, etc.) in which case the function is said to have several arguments, each of which

is an element of $A1$, $A2$, etc. respectively. If, for example, $A = A1 \times A2 \times A3$, then an element of A could be written $(a1, a2, a3)$, where $a1$ is an element of $A1$; $a2$, of $A2$ and $a3$, of $A3$. The corresponding value of the function would be written $F(a1, a2, a3)$.

The term *mapping* is a synonym for function. The terms *operator* and *operation* are often used as synonyms for function. They are also sometimes used as synonyms for relation, especially in cases in which the relation defines a function in an obvious or straightforward way.

A function F on the set A to the set B is a subset of the cartesian product $A \times B$. Therefore, if (a, b) is an element of F, then a must be an element of A. But the reverse statement is not necessarily true, that is, for any particular a in A there need not be an element (a, b) in F. In other words, the function F does not necessarily transform every a in A into an element in B. Similarly, the function F need not cover B entirely, that is, associate every element in B with some element in A.

Example 6: Consider the set $\mathbb{N}$ of natural numbers ($\mathbb{N} = \{1, 2, 3, \ldots\}$) and the square root function on $\mathbb{N}$ to $\mathbb{N}$. Expressed as a subset of $\mathbb{N} \times \mathbb{N}$, this function is

$$\{(1, 1), (4, 2), (9, 3), (16, 4), (25, 5), \ldots\}$$

Note that many elements of $\mathbb{N}$ have no square root in the natural numbers, e.g. 2, 3, 5, etc.; i.e. this square root function does not map every element of $\mathbb{N}$ into some value.

We will often want to talk about the subset of A consisting of those elements which F does transform into an element of B. This subset is called the *domain* of the function F. We will also want to talk about the subset of B consisting of those elements into which F transforms elements of A. This subset is called the *range* of F. These sets can be defined more formally as follows.

Definition A0.8: Let F be a function on the set A to the set B. The *domain* D of F is the set

$$D = \{a \mid a \text{ is in } A \text{ and there exists some } b \text{ in } B \text{ such that } (a, b) \text{ is in } F\}$$

Definition A0.9: Let F be a function on the set A to the set B. The *range* R of F is the set

$$R = \{b \mid b \text{ is in } B \text{ and there exists some } a \text{ in } A \text{ such that } (a, b) \text{ is in } F\}$$

Clearly, the domain D as defined above is a subset of A and the range R is a subset of B. One says that F is a function *from* its domain D *onto* its range R.

Example 7: Referring to the last example above, the domain of the square root function on $\mathbb{N}$, the set of natural numbers, is

$$D = \{1, 4, 9, 16, 25, \ldots\}$$

Its range is $\mathbb{N}$.

Example 8: Consider again the set $\mathbb{N}$ of natural numbers. Addition is a function (an operation) from $\mathbb{N} \times \mathbb{N}$ to $\mathbb{N}$. Some of the elements of this function are

$$((1, 1), 2), ((1, 2), 3), ((3, 6), 9)$$

Using the functional notational form specified above, the above elements of the function can be specified as follows:

$$+(1, 1) = 2$$
$$+(1, 2) = 3$$
$$+(3, 6) = 9$$

For such commonly occurring functions the well-known *infix* notation is more frequently used:

$$(1 + 1) = 2$$
$$(1 + 2) = 3$$
$$(3 + 6) = 9$$

One should be able to think in terms of either of these notational forms with equal facility since both are frequently used. In particular, one should always be aware of the fact that they have the same meaning.

Example 9: Consider the set $\mathbb{R}$ of real numbers. The operator $<$ is a relation between $\mathbb{R}$ and $\mathbb{R}$. Some of the elements of this relation are

$$(1, 1.5), (1, 2), (-4, 2), (0, 5)$$

The following elements of $\mathbb{R} \times \mathbb{R}$ are not elements of the relation $<$:

$$(1, 0)\ (2, 1), (2, -2)\ (3, 3)$$

Example 10: Consider the relation $<$ between $\mathbb{R}$ and $\mathbb{R}$ in the last example above. We define a function F from $\mathbb{R} \times \mathbb{R}$ to {true, false} as follows. For any elements r and s in $\mathbb{R}$,

$F(r, s)$ = true, if (r, s) is an element of the relation $<$,
= false, otherwise

Thus, the relational operator $<$ can be viewed as a relation between $\mathbb{R}$ and $\mathbb{R}$ or as a function from $\mathbb{R} \times \mathbb{R}$ to {true, false}. Semantically, these two mathematically distinct views are equivalent.

The function $<$ is usually written in infix notational form. For example,

$(1 < 1.5)$ = true
$(1 < 2)$ = true
$(1 < 0)$ = false
$(2 < 1)$ = false
$(3 < 3)$ = false

The domain of the function $<$ is $\mathbb{R} \times \mathbb{R}$. Its range is {true, false}.

A0.1.2 Expressions

An *expression* in mathematics is a rule for obtaining a value (element of some set) from a given value or values (element or elements of some set or sets). In other words, it is a function. An expression is written as a combination of functions and their arguments. In principle, the initial, intermediate and final values may be elements of any sets consistent with their usage. Typically, variable names appear in an expression in place of the values which they represent. When applying the rule represented by the expression, one first replaces the variable names by the current values of those variables and then evaluates the expression. The result is the value of the function defined by the expression for the given arguments (values of the variables appearing in the expression).

Example 11: Consider the expression $((2*x + y) < z)$. Applying the expression (functional rule) to the argument values 1, 2 and 8 for x, y and z respectively, one obtains

$$((2*x + y) < z) = ((2*1 + 2) < 8) = (4 < 8) = \text{true}$$

Applying this expression to the argument values 6, 7 and 3 for x, y and z respectively, one obtains

$$((2*x + y) < z) = ((2*6 + 7) < 3) = (19 < 3) = \text{false}$$

This expression is a function from $\mathbb{R} \times \mathbb{R} \times \mathbb{R}$ (its domain) to {true, false} (its range).

A0.1.3 Extending the formally defined domain of a function

As pointed out above, the domain of a function on a set A to a set B may be a proper subset of A; i.e. the function may not be defined for all elements of A. It is sometimes convenient to extend the formal definition of such a function to all elements of A or even of a superset of A. One simple and often useful way of doing this is the following.

Let F be a function with domain D and range R. Let D' be a superset of D. We define a new function F' with domain D' as follows. For every a in D'

$F'(a)$ = $F(a)$, if a in D
= 'undef', otherwise

The element 'undef' is a new element not in D or R. We allow this new element 'undef' to be in the set D'; in fact, it is often desirable that D' include it. The range of F' is R, the range of F, augmented by the new element. Expressed differently, the range of F' is the union of R and the singleton set {undef}.

Sometimes it is appropriate to modify the above definition of F' for some values of a in D' but not in D (i.e. for some a in $(D' - D)$). A typical example would arise when extending the definition of multiplication; one might want to define the extended product (0 * undef) to be 0 instead of 'undef'. We will encounter other examples of such an extension in Section A0.3.2, Boolean functions on an extended domain, below.

Note that the element called 'undef' above is itself a well-defined element. This well-defined value of the extended function F' is associated with arguments for which the original function F was not defined, hence the name 'undef'. By extending the domain of a function in this way, one can shorten and simplify some logical arguments and proofs, which would otherwise have to distinguish often between subsets within and outside the domains of the unextended functions. Computer programs in which expressions are based on such extended functions are often logically and structurally simpler than those in which expressions are based on unextended functions.

A0.1.4 Images and preimages under a function

A function maps an element of its domain into an element of its range. We sometimes wish to talk about that subset of the range consisting of those elements into which the function maps elements of a given set. Such a subset of the range is called the *image* of the given set. Conversely, we sometimes wish to talk about that subset of the domain whose elements

are mapped into elements of a given set. Such a subset of the domain is called the *preimage* or *inverse image* of the given set.

More precisely and formally, we define these terms as follows.

Definition A0.10: Let the function F with domain D and the set A be given. The *image* of A under F, written $F(A)$, is the set

$$F(A) = \{F(a) \mid a \text{ is an element of } A \text{ and of } D\}$$

The above definition of $F(A)$ may be written in the following alternative but equivalent form:

$$F(A) = \{F(a) \mid a \text{ in } A \cap D\}$$

Definition A0.11: Let the function F with domain D and range R and the set B be given. The *preimage* (*inverse image*) of B under F is the set

$$F^{-1}(B) = \{a \mid a \text{ in } D \text{ and } F(a) \text{ in } B\}$$

A preimage is always a subset of the domain of the function in question. Similarly, an image is always a subset of the range. The range is the image of the domain. The domain is the preimage of the range.

A0.2 Sequences

A *sequence* is an ordered, linear (i.e. not branching) arrangement of objects. A sequence differs from a set in that (a) the order in which the members of the sequence appear is significant and (b) two or more of the individual members of a sequence may be equal.

Example 12: [1, 3, 2, 1] is a sequence of four integers. The integer 1 appears twice in the sequence. The first *member* (or *term* or *element*) of the sequence is 1; the second, 3; the third, 2; and the fourth, 1.

A sequence may be empty, it may contain a limited number of terms or it may contain infinitely many terms. The number of terms in a sequence is called its *length*.

Formally, the word 'sequence' may be defined in different ways. Perhaps the mathematically simplest definition is the following.

Definition A0.12: A *sequence* is a function from $\mathbb{N}$, the set of natural numbers, or a subset thereof, to any set B.

Usually, the domain of the function is a set of consecutive integers beginning with 1. The starting point is not critical, however, and some text books define a sequence as a function from the set of non-negative integers (i.e. including 0). It is convenient, but not necessary, if the integers are consecutive.

The length of the sequence is the magnitude of the domain of the function.

Example 13: The function (subset of $\mathbb{N} \times B$) corresponding to the last example above is as follows:

$$\{(1, 1), (2, 3), (3, 2), (4, 1)\}$$

Since the left elements in the above ordered pairs begin with 1 and increase by 1 in each pair, they are redundant. Therefore, listing only the second elements in the pairs in order specifies the sequence completely: [1, 3, 2, 1]. This is the usual notational form for a sequence.

In functional notation, $S(1) = 1$, $S(2) = 3$, $S(3) = 2$ and $S(4) = 1$. The domain of the function is, in this example, the set $\{1, 2, 3, 4\}$. The range is the set $\{1, 2, 3\}$. The domain and range are each subsets of $\mathbb{N}$. The set B is any superset of the range, e.g. $\mathbb{N}$.

Notice that the equation $S(1) = 1$ is just another way of writing 'the first member of the sequence is 1', one of the statements in the last example above. Similarly, $S(2) = 3$ means 'the second member of the sequence is 3', etc.

We often need to talk about rearranged sequences, i.e. a sequence formed by reordering the members of a previously specified sequence. Such a rearrangement of a sequence is called a *permutation* of the sequence.

Example 14: Some of the permutations of the sequence [1, 3, 2, 1] are

[3, 1, 2, 1], [1, 1, 2, 3], [3, 2, 1, 1]

and the original sequence [1, 3, 2, 1] itself.

It will be easier to state a formal definition of a permutation if we consider a special type of function first. Remember that a function was defined as a special type of relation; it maps each element of its domain into a unique element of its range. If the relation is unique in the other direction also, i.e. if the function maps only one element of the domain into any one element of the range, the function is called *one-to-one*. One could say that the relation is a function in both directions.

Definition A0.13: A function F is called *one-to-one* if it exhibits the following property: If $(a1, b)$ is an element of F and $(a2, b)$ is an element of F, then $a1 = a2$.

Note the symmetry between this definition of one-to-one and the formal definition of a function in Section A0.1.1, Functions, above. A function associates an element of the domain with only one element of the range. A one-to-one function associates an element of the range with only one element of the domain.

The second sentence of the formal definition of one-to-one above can be rewritten in functional notation more concisely. The required property of a one-to-one function then becomes: If $F(a1) = F(a2)$, then $a1 = a2$. This notation has the disadvantage, however, that it hides the symmetry to the definition of a function.

If a one-to-one function has the same domain and range, one can think of it as rearranging (or permuting) an ordering (sequence) of the elements of that set. This suggests the following formal definition.

Definition A0.14: A *permutation* is a one-to-one function whose domain and range are equal.

If a function S is a sequence (see the formal definition of a sequence above), and P is a permutation with suitable domain and range, then the composed function $S(P(\cdot))$ is a sequence which is a rearrangement of the sequence S. A suitable domain and range of P in this sense is the domain of S.

Example 15: Consider the above example of a sequence, [1, 3, 2, 1]. In functional notation, the sequence can be written as follows:

$$S(1) = 1,\ S(2) = 3,\ S(3) = 2,\ S(4) = 1$$

The domain of S is the set $\{1, 2, 3, 4\}$. Its range is the set $\{1, 2, 3\}$.

Consider the function P defined as follows:

$$P(1) = 2,\ P(2) = 1,\ P(3) = 3,\ P(4) = 4$$

Note that this function fulfills the definition of a permutation (it is one-to-one and its domain and range are equal). Note also that its domain and range are equal to the domain of S. The composition of the functions S and P gives rise to a sequence S' which is a permutation of S:

$$S'(1) = S(P(1)) = S(2) = 3$$
$$S'(2) = S(P(2)) = S(1) = 1$$

$$S'(3) = S(P(3)) = S(3) = 2$$
$$S'(4) = S(P(4)) = S(4) = 1$$

or, in sequential notation, [3, 1, 2, 1]. This is the first rearrangement of [1, 3, 2, 1] given in the previous example above of a permutation.

The other rearrangements given in the previous example can be generated by selecting other permutation functions for P, e.g. $P(1) = 1$, $P(2) = 4$, $P(3) = 3$, $P(4) = 2$, which permutes (rearranges) the original sequence [1, 3, 2, 1] to [1, 1, 2, 3].

A permutation P with $P(i) = j$ moves the jth term in the original sequence to become the ith term in the new sequence. Expressed more concisely, the permutation P moves the $P(i)$th term in the original sequence to the ith place in the new sequence.

The last sentence in the above paragraph implies the requirement that a suitable function for rearranging a sequence must be one-to-one. For if the function P were not one-to-one, there might exist unequal i and j with $P(i) = P(j)$. Then the $P(i)$th $(= P(j)$th) term in the original sequence would become both the ith and the jth terms in the new sequence, thus duplicating the term from the original sequence. This possibility is clearly inconsistent with our subjective understanding of 'rearranging'. It can be eliminated only by requiring that the rearranging function P be one-to-one. Its usage in the definition of the permuted sequence S' above necessitates that its domain and range be equal, i.e. that it be a permutation as defined formally above.

A0.3 Boolean algebra

In various logical arguments, e.g. in proofs, as well as in preconditions, postconditions, loop invariants, etc. expressions often arise which evaluate to either *true* or *false*. Such logical expressions are then often combined in larger expressions. While the meanings of many such combinations are subjectively clear, we must, to be precise, explicitly define those combining mechanisms which we propose to use.

A suitable logical algebra has been developed and is commonly known under the name of 'Boolean' algebra. Strictly speaking, Boolean algebra is defined in mathematics in a somewhat more general way than we will pursue here. Our approach and definitions are consistent with the more general mathematical definitions. They correspond more closely with the application of Boolean algebra in, for example, electrical engineering – in particular, in the analysis of switching circuits and comparable digital circuitry.

Basically, we define a structure consisting of a set and several operations on that set (and on cartesian products of that set with itself). The structure resembles, but is simpler than, that of arithmetic (the set of numbers and the arithmetic operations of addition, multiplication, etc.).

A0.3.0 Basic definitions

In the following definitions, the set $\mathbb{B}$ is the set {true, false}. Several functions are defined by listing all elements in the domain and specifying for each the value of the function.

Definition A0.15: The function **and** from the set $\mathbb{B} \times \mathbb{B}$ to the set $\mathbb{B}$ is defined as follows:

false **and** false = false
false **and** true = false
true **and** false = false
true **and** true = true

Infix, rather than functional, notation is normally used for this function.

Definition A0.16: The function **or** from the set $\mathbb{B} \times \mathbb{B}$ to the set $\mathbb{B}$ is defined as follows:

false **or** false = false
false **or** true = true
true **or** false = true
true **or** true = true

Infix, rather than functional, notation is normally used for this function also.

Definition A0.17: The function **not** from the set $\mathbb{B}$ to the set $\mathbb{B}$ is defined as follows:

not false = true
not true = false

Prefix notation (like negation in arithmetic), rather than functional notation, is normally used for this function.

The implication function is often used in logical argumentation and proofs. The statement 'x implies y', written $(x \Rightarrow y)$, means that if the statement (Boolean variable or expression) x is true, then the Boolean expression y must or should also be true. If x is false, then nothing is said about y, i.e.

it may be either true or false. This suggests the following formal definition of the implication function:

Definition A0.18: The *implication* function from the set $\mathbb{B} \times \mathbb{B}$ to the set $\mathbb{B}$ is defined as follows:

false $\Rightarrow$ false = true
false $\Rightarrow$ true = true
true $\Rightarrow$ false = false
true $\Rightarrow$ true = true

A0.3.1 Some useful theorems

The following theorems, lemmata, etc. facilitate the manipulation of Boolean expressions.

Theorem A0.0: The **and** function is commutative, that is, (x **and** y) = (y **and** x) for all x and y.

Proof: The above definition of the **and** function is symmetric, i.e. the function's value is not changed by interchanging the two arguments.

Alternatively, this theorem can be proved by listing all four possible combinations of values for x and y and noting that in each case the two expressions have the same value:

x	y	x **and** y	y **and** x
false	false	false	false
false	true	false	false
true	false	false	false
true	true	true	true

■

Theorem A0.1: The **or** function is commutative, that is, (x **or** y) = (y **or** x) for all x and y.

Proof: The proof of this theorem is analogous to that of theorem A0.0. ■

Theorem A0.2: The **and** function is associative, that is,

$$((x \textbf{ and } y) \textbf{ and } z) = (x \textbf{ and } (y \textbf{ and } z))$$

for all x, y and z.

Proof: Note that the value of the **and** function is true if and only if both arguments are true. Thus, ((x **and** y) **and** z) is true if and only if x, y and

z are all true. Similarly, (x **and** (y **and** z)) is true only under the same circumstances. In any other case, both expressions are false; i.e. the two expressions have, in every case, the same value.

Alternatively, this theorem can be proved by listing all eight possible combinations of values for x, y and z and noting that in each case the two expressions have the same value:

x	y	z	((x **and** y) **and** z)	(x **and** (y **and** z))
false	false	false	false	false
false	false	true	false	false
false	true	false	false	false
false	true	true	false	false
true	false	false	false	false
true	false	true	false	false
true	true	false	false	false
true	true	true	true	true

■

Theorem A0.3: The **or** function is associative, that is,

($(x$ **or** $y)$ **or** $z) = (x$ **or** (y **or** $z)$)

for all x, y and z.

Proof: The proof of this theorem is analogous to that of theorem A0.2. ■

Because the **and** function is associative, we may write (x **and** y **and** z) for either ((x **and** y) **and** z) or (x **and** (y **and** z)) without ambiguity, their being equal. Similarly, we may write (x **or** y **or** z) for either of the equal expressions ((x **or** y) **or** z) or (x **or** (y **or** z)).

Theorem A0.4: The **and** function is distributive over the **or** function, i.e.

(x **and** (y **or** z)) = ((x **and** y) **or** (x **and** z))

for all x, y and z.

Proof: The expression on the left is true if and only if x is true and y and z are not both false. The expression on the right is true under exactly the same conditions. Otherwise, both are false; i.e. they are always equal.

Alternatively, this theorem can be proved by listing all eight possible combinations of values for x, y and z and noting that in each case the two expressions have the same value:

x	y	z	(x **and** (y **or** z))	((x **and** y) **or** (x **and** z))
false	false	false	false	false
false	false	true	false	false
false	true	false	false	false
false	true	true	false	false
true	false	false	false	false
true	false	true	true	true
true	true	false	true	true
true	true	true	true	true

■

Thus, the **and** function can be compared with multiplication and the **or** function, with addition. Largely for this reason, the **and** infix operator by convention takes precedence over the **or** operator. That is, (x **and** y **or** z) means ((x **and** y) **or** z).

Because the main laws of arithmetic upon which most algebraic manipulation is based are also satisfied by the **and** and **or** functions of Boolean algebra, Boolean expressions may be manipulated in much the same way as arithmetic algebraic expressions. Boolean algebra satisfies several additional laws, lemmata, etc. also; as a result, Boolean expressions may often be simplified in still other ways.

Unlike the addition and multiplication operators in arithmetic, however, Boolean algebra satisfies the symmetric distributive law also. Before stating the corresponding theorem, we state the following lemmata, which we will use in the proof of the second distributive law.

Lemma A0.0: (x **and** x) = x for all x.

Proof: The argument x can take on only two values, true and false. In each case, the truth of the thesis of this lemma follows directly from the definition of the **and** function. ■

Lemma A0.1: (x **or** (x **and** y)) = x for all x and y.

Proof: If x is true, the expression on the left is true regardless of the value of y, so the equation is satisfied. If x is false, (x **and** y) is false, regardless of the value of y, and the equation is also satisfied. ■

Theorem A0.5: (The second distributive law) The **or** function is distributive over the **and** function, i.e.

(x **or** (y **and** z)) = ((x **or** y) **and** (x **or** z))

for all x, y and z.

Proof: We begin by applying the first distributive law above to the expression to the right of the equals sign:

$$
\begin{aligned}
& (x \textbf{ or } y) \textbf{ and } (x \textbf{ or } z) \\
= \; & ((x \textbf{ or } y) \textbf{ and } x) \textbf{ or } ((x \textbf{ or } y) \textbf{ and } z) && \text{first distributive law} \\
= \; & ((x \textbf{ and } (x \textbf{ or } y)) \textbf{ or } (z \textbf{ and } (x \textbf{ or } y)) && \text{commutative law for } \textbf{and} \\
= \; & ((x \textbf{ and } x) \textbf{ or } (x \textbf{ and } y)) \textbf{ or } ((z \textbf{ and } x) \textbf{ or } (z \textbf{ and } y)) \\
& && \text{first distributive law} \\
= \; & x \textbf{ or } (x \textbf{ and } y) \textbf{ or } (x \textbf{ and } z) \textbf{ or } (y \textbf{ and } z) \\
& && \text{lemma A0.0 and commutative and associative laws} \\
= \; & x \textbf{ or } (y \textbf{ and } z) && \text{lemma A0.1 (applied twice)} \; \blacksquare
\end{aligned}
$$

The following additional facts are frequently of use in simplifying Boolean expressions.

Lemma A0.2: **not** (**not** x) = x for all x.

Proof: This fact follows directly from the definition of the **not** function. ■

Theorem A0.6: **not** (x **and** y) = (**not** x) **or** (**not** y) for all x and y.

Proof: This theorem may be proved easily by listing all four possible combinations of values for x and y and verifying that for each, the above equality applies. ■

Theorem A0.7: **not** (x **or** y) = (**not** x) **and** (**not** y) for all x and y.

Proof: Applying theorem A0.6 to the expressions (**not** x) and (**not** y) yields

not ((**not** x) **and** (**not** y)) = (**not** (**not** x)) **or** (**not** (**not** y))

Applying lemma A0.2,

not ((**not** x) **and** (**not** y)) = x **or** y

Negating both sides, one obtains

not (**not** ((**not** x) **and** (**not** y))) = **not** (x **or** y)

Applying lemma A0.2 again, this reduces to

(**not** x) **and** (**not** y) = **not** (x **or** y) ■

Alternatively, this theorem may be proved in the same way as theorem A0.6, i.e. by listing all four possible combinations of values for x and y and verifying that for each, the thesis of this theorem is satisfied.

Note the symmetry between **and** and **or** and the dual roles which they play in the above two theorems:

not (x **and** y) = (**not** x) **or** (**not** y)
not (x **or** y) = (**not** x) **and** (**not** y)

The following almost trivial but quite useful observations follow directly from the definitions of the **and** and **or** functions. For the sake of completeness, we repeat here a lemma stated earlier as well as the corresponding statement for the **or** function.

x **and** false = false
x **and** x = x
x **and** true = x

x **or** false = x
x **or** x = x
x **or** true = true

The implication function ($\Rightarrow$) can be expressed in terms of **and**, **or** and **not** in different ways. The most useful equivalent expressions are

$$
\begin{aligned}
(x \Rightarrow y) &= (\textbf{not}(x \textbf{ and not } y)) \\
&= ((\textbf{not } x) \textbf{ or } y) \\
&= ((\textbf{not } y) \Rightarrow (\textbf{not } x))
\end{aligned}
$$

The reader should verify that these three equations are true for all four possible combinations of values for x and y.

A0.3.2 Boolean functions on an extended domain

Consider the expression

($i \leq n$) **and** ($a < b(i)$)

where i and n are integers and a and $b(\cdot)$ are real numbers, $b(\cdot)$ being a member of the array b. The $\leq$ and $<$ functions are from $\mathbb{R} \times \mathbb{R}$ to $\mathbb{B}$. The **and** function is a function from $\mathbb{B} \times \mathbb{B}$ to $\mathbb{B}$.

If the value of i is greater than the value of n, the value of the above expression is false, regardless of the values of a and $b(i)$. But if the value of $b(i)$ is not defined in this case, for example because the value of i is outside of the subscript range for the array b, then the arguments of the $<$ function are not in its defined domain and the value of the $<$ function

is not defined. Thus, in turn, one of the arguments of the **and** function is not within its domain. As a result, the value of the **and** function is, strictly speaking, also undefined.

Such situations arise often in connection with Boolean expressions in computer programs. Some computing systems will evaluate the above expression as false in the situation described, while others will determine it to be undefined, issue a corresponding error message and terminate execution of the program abnormally. No universally recognized standard exists for the interpretation of such expressions in computing systems; the convention used is left up to the system's designers and implementors.

The need to consider explicitly such cases in which arguments are outside of the domain of such functions, each and every time they arise, would frequently lead to unnecessarily long and cumbersome proofs and logical arguments. It is sometimes simpler to extend the defined domain of these functions to include a new element representing situations which are undefined under the more restrictive definitions of the functions. This has been discussed already in section A0.1.3.

There is no single correct way to extend the definitions of the Boolean functions in this way. Any one of several approaches can be taken and each has certain advantages and disadvantages. Three commonly encountered extensions are defined and discussed below.

Common among all three approaches is that the set $\mathbb{B}$ = {true, false} is augmented by a third element. In the following, the set $\mathbb{B}e$ is defined to be

$$\mathbb{B}e = \{\text{true, false, undef}\}$$

Convention 1: The general idea motivating this definitional convention is 'once undefined, always undefined'. If any argument is undefined, the value of the function will be undefined. Once an undefined value arises, it propagates throughout all higher order functions in an expression. The general form of the definition of the function on the extended domain was given in Section A0.1.3. Applying it, we obtain the following definitions of the extended **and**, **or**, **not** and < functions. Strictly speaking, we are defining new functions and should assign new names to them. But because the new functions are so intimately related to the old ones, we will reuse the old names.

The **and** and **or** functions are from $\mathbb{B}e \times \mathbb{B}e$ to $\mathbb{B}e$. The **not** function is from $\mathbb{B}e$ to $\mathbb{B}e$:

undef **and** undef = undef
undef **and** false = undef
undef **and** true = undef

false **and** undef = undef
false **and** false = false
false **and** true = false
true **and** undef = undef
true **and** false = false
true **and** true = true

undef **or** undef = undef
undef **or** false = undef
undef **or** true = undef
false **or** undef = undef
false **or** false = false
false **or** true = true
true **or** undef = undef
true **or** false = true
true **or** true = true

not undef = undef
not false = true
not true = false

The extended function < is defined in terms of the unextended < function (with domain $\mathbb{R} \times \mathbb{R}$ and range $\mathbb{B}$) as follows. For all r and s in the set $\mathbb{R} \cup$ {undef}

$(r < s) = (r < s)$, if r in $\mathbb{R}$ and s in $\mathbb{R}$
= undef, otherwise.

These definitions have the advantage that the new functions **and** and **or** are, like their predecessors, commutative and associative. Similarly, both distributive laws apply. Many other – but not all – properties stated earlier in Section A0.3.1 are satisfied by the extended functions as defined under this convention.

The disadvantage of this convention is that it is unduly restrictive and does not properly reflect two useful interpretations of 'undef'. These form the basis for conventions 2 and 3 below.

Convention 2: One scheme for evaluating an expression employs the following logic. A number of implemented computing systems utilize this scheme, so it is of some practical interest. The expression is evaluated from left to right. As soon as the value of a function is uniquely determined, subsequent, redundant parts (if any) of the expression are skipped over and ignored. If they would have given rise to an undefined value, this fact is never discovered. If an undefined value is encountered, an undefined result is returned and the execution of the program is terminated abnormally. The

restricted (not extended) definitions of the various functions serve as the basis for the evaluation procedure.

In effect this scheme defines the extended versions of the **and** and **or** functions as follows. Differences between this convention and convention 1 above are marked with an asterisk.

undef **and** undef = undef
undef **and** false = undef
undef **and** true = undef
false **and** undef = false*
false **and** false = false
false **and** true = false
true **and** undef = undef
true **and** false = false
true **and** true = true

undef **or** undef = undef
undef **or** false = undef
undef **or** true = undef
false **or** undef = undef
false **or** false = false
false **or** true = true
true **or** undef = true*
true **or** false = true
true **or** true = true

This convention restores the validity of the identities in normal (unextended) Boolean algebra (see Section A0.3.1) which were violated in convention 1:

false **and** x = false
true **or** x = true

However, the **and** and **or** functions are no longer commutative. This is an unfortunate result, for the commutative property facilitates manipulating and simplifying Boolean expressions.

Restoring the commutative property leads to the following convention.

Convention 3: The value undef is viewed, loosely speaking, as an undetermined choice between true and false. If the value of a function (according to its restricted, i.e. not extended definition) is the same regardless of whether a particular argument is true or false, then that same value is taken as the value of the function when the argument in question is undef. Otherwise, a value of undef for an argument leads to a value of undef for the function. This convention, in effect, applies the basic idea of convention 2 symmetrically.

The extended **and** and **or** functions are then defined as follows. Differences between this convention and either convention 1 or convention 2 above are marked with an asterisk.

undef **and** undef = undef
undef **and** false = false*
undef **and** true = undef
false **and** undef = false*
false **and** false = false
false **and** true = false
true **and** undef = undef
true **and** false = false
true **and** true = true

undef **or** undef = undef
undef **or** false = undef
undef **or** true = true*
false **or** undef = undef
false **or** false = false
false **or** true = true
true **or** undef = true*
true **or** false = true
true **or** true = true

In accordance with the general idea underlying this convention 3 (see remarks above), we extend the definition of the implication function to the domain $\mathbb{B}e \times \mathbb{B}e$ as follows:

undef $\Rightarrow$ undef = undef
undef $\Rightarrow$ false = undef
undef $\Rightarrow$ true = true
false $\Rightarrow$ undef = true
false $\Rightarrow$ false = true
false $\Rightarrow$ true = true
true $\Rightarrow$ undef = undef
true $\Rightarrow$ false = false
true $\Rightarrow$ true = true

Under this convention, the definition of the extended **not** function is the same as under convention 1 above.

The extended **and** and **or** functions as defined above are commutative. In fact, all of the properties of the unextended functions stated in Section A0.3.1 are satisfied by the extended functions defined here (convention 3). Verifying this statement is a good exercise for the reader. Not surprisingly,

however, there are other identities satisfied by the unextended functions defined in Section A0.3.0, but not by the extended functions defined here.

Except where specifically stated otherwise, references in this book to extended functions are based on convention 3 rather than convention 1 or convention 2.

A0.4 Series notation

One often needs to write an expression consisting of the sum (or product, **and**, **or**, etc.) of a variable number of terms, the number of terms being determined by the value(s) of one or more variables. For example, expressions comparable to

$$x(1) + x(2) + \ldots + x(n)$$

arise often in mathematics. This expression is usually written more concisely

$$\sum_{i=1}^{n} x(i)$$

The variable i is called the *running variable*. The value of n is the *final value* of the running variable and here 1 is the *initial value* of the running variable. The initial and final values of the running variable need not be constants or variables as here; they may be given by expressions.

Note that

$$[\sum_{i=a}^{b} x(i)] + [\sum_{i=b+1}^{c} x(i)] = [\sum_{i=a}^{c} x(i)]$$

The meaning of the above definition is clear when the final value of the running variable is greater than or equal to its initial value. If the final value is less than the initial value, however, the meaning of the above notation is not yet clear. Setting $b = a - 1$ in the above equation, one obtains

$$[\sum_{i=a}^{a-1} x(i)] + [\sum_{i=a}^{c} x(i)] = [\sum_{i=a}^{c} x(i)]$$

which suggests that an appropriate definition of the Σ notation should provide that

$$\sum_{i=a}^{a-1} \ldots = 0$$

for any value of a and any form of the expression within the Σ operator. The above equation can also be used to deduce an appropriate definition

for those cases in which the final value of the running variable is 2, 3, ..., etc. less than the initial value. In the case of the Σ operator, this is not, however, normally done.

This notational form may be easily extended to any associative function and is particularly appropriate for ones for which the infix notation is usual. In this book, this series notation is employed frequently for the **and** and **or** functions.

Definition A0.19: If a and b represent expressions, then

$$\begin{aligned} \mathbf{and}_{i=a}{}^{b}\ \exp(i) &= \exp(a)\ \mathbf{and}\ \exp(a+1)\ \mathbf{and}\ \ldots\ \exp(b), \\ &\quad \text{if the values of } a \text{ and } b \text{ are integers with } a \leq b \\ &= \text{true, if the values of } a \text{ and } b \text{ are integers with} \\ &\quad a > b \\ &= \text{undef, otherwise.} \end{aligned}$$

Above, i is an internal ('running') variable and exp is an expression with values in $\mathbb{B}$e (i.e. exp is a function to $\mathbb{B}$e).

The value of the series is the special value 'undef' (see Section A0.3.2, Boolean functions on an extended domain) if, for example, the value of a or b is undef or is a fraction. The value of the series can also be undef, of course, if one of the terms exp(i) is undef.

Note that a constant or a variable is a special, simple form of an expression, so that the above definition allows the initial and/or final value of the running variable to be given by a constant or a variable.

The series notation for the **or** function is defined analogously:

Definition A0.20: If a and b represent expressions, then

$$\begin{aligned} \mathbf{or}_{i=a}{}^{b}\ \exp(i) &= \exp(a)\ \mathbf{or}\ \exp(a+1)\ \mathbf{or}\ \ldots\ \exp(b), \\ &\quad \text{if the values of } a \text{ and } b \text{ are integers with } a \leq b \\ &= \text{false, if the values of } a \text{ and } b \text{ are integers with } a > b \\ &= \text{undef, otherwise.} \end{aligned}$$

Above, i is an internal variable and exp is an expression as in the preceding definition above.

Note that the value of the empty **and** series is true, while the value of the empty **or** series is false. This convention is most sensible because (true **and** x) = x but (false **or** x) = x.

When an **and** series is **and**ed to another term, the preceding **and** may be dropped, e.g.

$$x\ \mathbf{and}\ [\mathbf{and}_{i=a}{}^{b}\ \exp(i)]$$

may be written more concisely

$$x \ \mathbf{and}_{i=a}^{\ b} \exp(i)$$

Similarly,

$$x \ \mathbf{or}_{i=a}^{\ b} \exp(i)$$

means

$$x \ \mathbf{or} \ [\mathbf{or}_{i=a}^{\ b} \exp(i)]$$

The two definitions above do not, strictly speaking, allow for unending series. It is meaningful to consider 'infinity' to be an integer in the sense of the above definitions, especially as a permitted value for the final value of the running variable, thus providing for an unending series. In such cases, the application of the definitions of the **and** and **or** functions given in Section A0.3 does not terminate and hence does not determine the value of these functions applied to an unbounded number of arguments.

We therefore generalize the definitions of the extended **and** and **or** functions given in convention 3 of Section A0.3.2 to allow for infinitely many arguments.

Definition A0.21: The value of the **and** function applied to finitely or infinitely many arguments is

true, if and only if all arguments are true,
false, if and only if any (one or more) arguments are false and
undef, otherwise.

If the **and** function is applied to no arguments, its value is true.

Definition A0.22: The value of the **or** function applied to finitely or infinitely many arguments is

false, if and only if all arguments are false,
true, if and only if any (one or more) arguments are true and
undef, otherwise.

If the **or** function is applied to no arguments, its value is false.

It is usually easy to ensure or verify that the values of the expressions a and b, the limits of the above series, are integers. In this case, the following identities apply:

$$\begin{aligned} \mathbf{and}_{i=a}^{\ b} \exp(i) &= (a > b) \ \mathbf{or} \ [(a \leq b) \ \mathbf{and}_{i=a}^{\ b} \exp(i)] \\ &= (a > b) \ \mathbf{or} \ [\mathbf{and}_{i=a}^{\ b} \exp(i)] \end{aligned}$$

The comparable expression for the **or** series is

$$\mathbf{or}_{i=a}^{\ b} \exp(i) = (a \leq b) \ \mathbf{and} \ [\mathbf{or}_{i=a}^{\ b} \exp(i)]$$

Expressions can sometimes be simplified after substituting one of these alternative forms for the ordinary **and** or **or** series notation.

In logical expressions, the **and** series notation can usually be substituted for the *for all* construct. Similarly, the **or** series notation can usually be substituted for the *there exists* construct. These possibilities are utilized frequently in this book.

Appendix 1

Solutions to the exercises

In the subjects we propose to investigate, our inquiries should be directed, not to what others have thought, nor to what we ourselves conjecture, but to what we can clearly and perspicuously behold and with certainty deduce; for knowledge is not won in any other way.
– René Descartes

To doubt everything or to believe everything are two equally convenient solutions; both dispense with the necessity of reflection.
– Jules Henri Poincaré

But answer came there none –
And this was scarcely odd, because
They'd eaten every one.
– Lewis Carroll

I was gratified to be able to answer promptly, and I did. I said I didn't know.
– Mark Twain

Below the reader will find solutions or sketches of the solutions to the exercises posed in the text in the body of this book.

Chapter 0: Komputema Simio, the computing monkey of Moc

1 The report prepared by Komputema Simio by executing the given instruction list upon the given initial data sheet will be:

Stress calculation,
beam loaded at one point, supported at each end
Load = 1000
at 2 kulongs from the left end
Beam length = 6 kulongs

maximum shear = 666.667
maximum moment = 1333.333
End of calculation, this is.

2 The final contents of the other data sheet will be as follows:

name	set	value
s	num	333.333
m	num	0
smax	num	666.667
mmax	num	1333.333
P	num	1000
a	num	2
L	num	6
x	num	6.1
ix	num	0.1

Chapter 2: Basic semantics of computer programs and programming constructs

1 An expression is a composition of functions. Its domain consists of those data environments

in which the values of the referenced variables are defined and are in the respective domains of the functions referencing them and
for which all intermediate function values are in the domains of the functions applied to them.

When interpreting the above in a specific case, one must duly consider whether the functions as implemented in the actual system in question are defined on extended domains or not and if so, the precise definitions of the extensions (see Appendix 0, Section A0.1.3, Extending the formally defined domain of a function, and Section A0.3.2, Boolean functions on an extended domain).

Consider, for example, the expression $(x * (y/z))$. This expression can be written in the equivalent form mult$(x, \text{div}(y, z))$. A typical implementation (e.g. involving floating point arithmetic) would be based on the arithmetic operations (or an approximation thereto) applied to values in a large but finite, and therefore bounded, subset F of the set $\mathbb{R}$ of real numbers. If the functions as implemented are not defined on extended domains, then the domain of the above expression would consist of those data environments d in $\mathbb{D}$ which contain variables named x, y and z and for which

valvar(y, d) is in F,
valvar (z, d) is in (F − {0}),
the quotient [valvar(y, d)/valvar(z, d)] is in F (i.e. its attempted computation does not give rise to an 'overflow error'),
valvar(x, d) is in F and
the product of valvar(x, d) and the above quotient is in F (i.e. its attempted computation does not cause 'overflow').

If the values of x, y and z in the data environment d were 2, 9 and 3 respectively, d would be in the domain of the above expression in any practically useful system. If the values of x, y and z in d were 1, 2 and 0 respectively, d would not typically be in the domain of the expression.

If the functions mult, div and/or valvar are defined on extended domains, then the domain of the expression above will encompass correspondingly more data environments (elements of $\mathbb{D}$). Depending upon the particular extensions, the domain of the expression may even be $\mathbb{D}$ in its entirety.

2 The equivalence of the statements 1 and 2 follows directly from the definition of 'weaker'. To prove that statement 1 implies statement 3, consider any element $d2$ for which valexp($E2$, $d2$) = true. By the definition of $E2^*$, $d2$ is in $E2^*$. Because $E2^*$ is a subset of $E1^*$, $d2$ is also in $E1^*$. By the definition of $E1^*$, valexp($E1$, $d2$) = true, so that statement 3 is true. To show that statement 3 implies statement 1, consider any $d2$ in $E2^*$. By the definition of $E2^*$, valexp($E2$, $d2$) = true. By statement 3, valexp($E1$, $d2$) is also true. By the definition of $E1^*$, d2 is in $E1^*$, so $E1^*$ must be a superset of $E2^*$.

3 The domain of a simple assignment statement $x := E$ consists of those data environments d in $\mathbb{D}$ for which

d is in the domain of the expression E (see question 1),
a variable named x exists in d and
valexp(E, d) is an element of the set associated with the first such named variable in d.

If the variable receiving a new value is a subscripted variable (e.g. $x(s)$), then the second statement above must be extended accordingly (e.g. d must be in the domain of the expression s and a variable named x(valexp(s, d)) must exist in d).

4 The domain of a multiple assignment statement

$$(x1, x2, \ldots) := (E1, E2, \ldots)$$

consists of those data environments d in $\mathbb{D}$

which are in the intersection of the domains of the simple assignment statements $x1 := E1$, $x2 := E2$, ... (see question 3 above) and

in which the values of all expressions associated with a variable referenced more than once in the replacement list to the left of the := symbol are equal (if any variable is so referenced).

5 If Bt is the subset of $\mathbb{D}$ on which the conditional expression B is defined and has the value true, if Bf is the subset of $\mathbb{D}$ on which B is defined and false, if $D1$ is the domain of the statement $S1$ and if $D2$ is the domain of $S2$, then the domain of the statement **if** B **then** $S1$ **else** $S2$ **endif** is

$$(Bt \cap D1) \cup (Bf \cap D2)$$

Note that the set ($Bt \cup Bf$) is the (unextended) domain of B.

6 The domain of the sequence ($S1$, $S2$) of statements is that subset of $\mathbb{D}$ which $S1$ maps into the domain of $S2$. The domain of $S2$ is, of course, that subset of $\mathbb{D}$ upon which $S2$ is defined, i.e. which $S2$ maps into $\mathbb{D}$. In other words, the domain of ($S1$, $S2$) is the preimage of $\mathbb{D}$ under ($S1$, $S2$) or, in functional notation,

$$S1^{-1}\ (S2^{-1}(\mathbb{D}))$$

7 Define W_i as the subset of $\mathbb{D}$ upon which the loop **while** B **do** S **endwhile** terminates after exactly i executions of S, where i is any non-negative integer. Then it follows from the definition of the **while** loop that

$$W_0 = \textbf{not}\ B$$

and that for positive i

$$W_i = B\ \textbf{and}\ S^{-1}\ (W_{i-1})$$

The domain of the **while** loop is the subset of $\mathbb{D}$ upon which the **while** loop terminates after finitely many executions of S, i.e. the union of W_i over all non-negative integers i.

8 The equivalence of definitions 2.11 and 2.12 follows from the definition of a sequence of program statements (see Section 2.1.2). Similarly, the equivalence of definitions 2.11 and 2.13 follows from the definition of the **if** statement (see Section 2.1.1). Definition 2.14 follows from a repetitive application of definition 2.11 while definition 2.11 can be derived from definition 2.14 in a corresponding manner.

9 The domain of a multiple declaration statement

declare ($x1$, $S1$, $E1$), ($x2$, $S2$, $E2$), ...

consists of those data environments d in $\mathbb{D}$

which are in the intersection of the domains of the expressions $E1$, $E2$, etc. (see question 1) and

for which valexp($E1$, d) is an element of the set $S1$, valexp($E2$, d) is in $S2$, etc.

If the name of a variable being declared is subscripted (e.g. $x(s)$), then the first statement above must be extended accordingly (e.g. d must also be in the intersection of the domains of all such subscript expressions appearing in the declaration statement).

10 The domain of a release statement consists of those data environments d in $\mathbb{D}$ containing a variable with the name appearing in the release statement. If that name is subscripted, then d must also, of course, be in the domain of the subscript expression.

11 The domain of an assignment statement A^* on $\mathbb{D}^*$ consists of those computational histories d^* in $\mathbb{D}^*$ which are of positive, finite length and for which the data environment last(d^*) is in the domain of A on $\mathbb{D}$.

12 The domain of the statement **if** B **then** $S1$ **else** $S2$ **endif** on $\mathbb{D}^*$ consists of those computational histories d^* in $\mathbb{D}^*$ which are of positive, finite length and for which either

last(d^*) is in Bt and [last(d^*)] is in the domain of $S1^*$ on $\mathbb{D}^*$ or
last(d*) is in Bf and [last(d^*)] is in the domain of $S2^*$ on $\mathbb{D}^*$.

Bt and Bf are as defined in the solution to question 5.

Note that a computational history d^* is in the domain of any statement S^* if and only if the computational history [last(d^*)] is in the domain of S^* (on $\mathbb{D}^*$). This is not equivalent to the proposition that last(d^*) is in the domain of S on $\mathbb{D}$. $S^*(d^*)$ may, for example, be an unending (and well defined) sequence, in which case S(last(d^*)) is not defined; in other words, last(d^*) is not in the domain of S on $\mathbb{D}$. If last(d^*) is in the domain of S on $\mathbb{D}$, then [last(d^*)] and therefore d^* also are in the domain of S^* on $\mathbb{D}^*$, but the converse is not true; i.e. total correctness on $\mathbb{D}$ is a stronger property than total correctness on $\mathbb{D}^*$.

If one defines

Bt^* as the set of those d^* in $\mathbb{D}^*$ for which B(last(d^*)) = true,
Bf^* as the set of those d^* in $\mathbb{D}^*$ for which B(last(d^*)) = false,
$D1^*$ as the domain of $S1^*$ on $\mathbb{D}^*$ and
$D2^*$ as the domain of $S2^*$ on $\mathbb{D}^*$

then the domain of the **if** statement above on $\mathbb{D}^*$ is

$$(Bt^* \cap D1^*) \cup (Bf^* \cap D2^*)$$

cf. solution to question 5.

13 The domain of a sequence ($S1$, $S2$)* of statements on $\mathbb{D}^*$ consists of those computational histories d^* in $\mathbb{D}^*$ which are of positive, finite length and for which either

last(d^*) is in the domain of $S1$ on $\mathbb{D}$ and $S1^*(d^*)$ is in the domain of $S2^*$ on $\mathbb{D}^*$ or
last(d^*) is not in the domain of $S1$ on $\mathbb{D}$ but d^* is in the domain of $S1^*$ on $\mathbb{D}^*$.

In the former case, d^* is in the domain of $S1^*$ on $\mathbb{D}^*$ and $S1^*(d^*)$ is a finite sequence of data environments. In the latter case, $S1^*(d^*)$ is an unending, well-defined sequence.

14 The domain of the loop **while** B **do** S **endwhile**, called W below, on $\mathbb{D}^*$ consists of those computational histories d^* in $\mathbb{D}^*$ which are of positive, finite length and for which either

1 last(d^*) is in the domain of W on $\mathbb{D}$ (see question 7),
2 last(d^*) is not in the domain of W on $\mathbb{D}$, but there exists a positive integer n such that all of the following conditions are met:
last(d^*) is in the domain of S^{n-1} on $\mathbb{D}$,
S^j(last(d^*)) is in the domain of B and $B(S^j(\text{last}(d^*))) = \text{true}$ for every integer j in the interval $0 \leq j < n$ and
d^* is in the domain of $(S^*)^n$ on $\mathbb{D}^*$ and $(S^*)^n(d^*)$ is an unending sequence

or

3 last(d^*) is not in the domain of W on $\mathbb{D}$ and no such n as specified above exists, but for every non-negative integer i both of the following conditions are satisfied:
last(d^*) is in the domain of S^i on $\mathbb{D}$ and
S^i(last(d^*)) is in the domain of B and $B(S^i(\text{last}(d^*))) = \text{true}$.

In case 1 above, the loop terminates and the computational history generated is of finite length. In case 2, the nth execution of S results in an unending sequence of data environments. In case 3, each application of S appends a finite sequence to the computational history, but the condition B is always true so the loop never terminates.

15 The equivalence of the four definitions given in Section 2.2.3 can be shown in the same manner as the equivalence of the four corresponding definitions given in Section 2.1.3; see question 8.

16 In general, lemma 2.4 will be true only if each execution of the body

S of the loop increases the length of the computational history by appending a non-empty sequence of data environments. This can be guaranteed in all cases only if the execution of an empty body S is defined accordingly, e.g. so that it duplicates the last data environment in the previously developed computational history. This is an important reason for the definition selected later in Section 2.2.7.

If either of the situations postulated in the lemma occurs, one can show that the computational history generated is an unending sequence by applying definition 2.26 straightforwardly. To show the converse, note that definition 2.26 can lead to an unending computational history only if either (a) $B(\text{last}(d0^*))$ = true and the application of S leads to an unending sequence (see the definition of the effect of the sequence of statements given in Section 2.2.2) or (b) the recursion in definition 2.26 never ceases because $B(\text{last}(\cdot))$ is always true.

17 The conjecture is true only for those statements or constructs S which append exactly one data environment to the computational history. The simple (not compound) statements – assignment, declaration and release – are the most important examples of statements for which the conjecture is true. In these cases, the truth of the conjecture follows directly from the definitions of these statements' functions on $\mathbb{D}^*$.

Only in special situations will the **if** and **while** constructs satisfy the conjecture. Depending upon the definition selected for the null statement (see Section 2.2.7), it will or will not satisfy the conjecture.

Counterexamples for the conjecture are a sequence of two or more typical statements and **while** loops with non-empty bodies which are executed many times. These constructs all append many data environments to the computational history and therefore cannot satisfy the conjecture.

The following variation of the conjecture is generally true: For any statement or construct S and for any finite, non-empty computational history $d0^*$ in $\mathbb{D}^*$, $\text{last}(S^*(d0^*)) = S(\text{last}(d0^*))$. Here, equality is to be interpreted in the usual way, i.e. if either side of the equation is defined, then both sides are defined and equal. If either side is undefined, then both sides are undefined, e.g. if $S^*(d0^*)$ is an infinite sequence, then it has no last term; in this case, $S(\text{last}(d0^*))$ is undefined.

18 The domain of a declaration statement D^* on $\mathbb{D}^*$ consists of those computational histories d^* in $\mathbb{D}^*$ which are of positive, finite length and for which the data environment $\text{last}(d^*)$ is in the domain of D on $\mathbb{D}$. Note that this pattern applies to all simple (non-compound) statements; cf. the assignment statement (see question 11).

19 The domain of a release statement R^* on $\mathbb{D}^*$ consists of those computational histories d^* in $\mathbb{D}^*$, which are of positive, finite length and for which the data environment last (d^*) is in the domain of R on $\mathbb{D}$. See also questions 11 and 18.

Chapter 3: Proof rules for the individual programming constructs

1 The first part of theorem 3.0 can be proved as follows. Consider a data environment d in the intersection of $Q1$ and $Q2$. The data environment d is in $Q1$. Because $\{Q1\}\ S\ \{P1\}$, $S(d)$ (if defined) is in $P1$. Similarly, $S(d)$ (if defined) is in $P2$. Thus $S(d)$ (if defined) is in the intersection of $P1$ and $P2$, so the intersection of $Q1$ and $Q2$ satisfies the definition of a precondition of the intersection of $P1$ and $P2$ under S. The second part of theorem 3.0 is proved analogously.

To prove theorem 3.1, the same approach can be employed. The argument is changed slightly: replace '$S(d)$ (if defined)' above by '$S(d)$ is defined and'.

To prove theorem 3.2, one begins by noting that ($Q1$ **and** $Q2$) and ($Q1$ **or** $Q2$) are preconditions by theorem 3.0. We need only to show that they are complete. Consider any data environment $d1$ in the intersection of $P1$ and $P2$ and its preimage $S^{-1}(\{d1\})$ (which may be empty). Because $d1$ is in $P1$ and $Q1$ is a complete precondition of $P1$, any element d in the preimage of $\{d1\}$ is in $Q1$. Similarly, d is also in $Q2$. It is, therefore, in the intersection of $Q1$ and $Q2$, from which it follows that this intersection is a complete precondition. The second part of theorem 3.2 is proved analogously.

2 The equivalence of statements (a), (b) and (c) follows directly from the definitions of a precondition (definition 3.0) and of an image and preimage of a set under a function (see Appendix 0, Section A0.1.4, Images and preimages under a function).

3 Let Qs and Qc be any strict and complete preconditions respectively of a given postcondition P under S. The weakest strict precondition is clearly a superset of any strict precondition. By the definition of a complete precondition (definition 3.2), it is a superset of the weakest strict precondition. Thus, Qs is a subset of $wsp(P, S)$ and $wsp(P, S)$ is a subset of Qc. If a precondition Q is both a strict and a complete precondition, it is both a subset and a superset of $wsp(P, S)$. It must, therefore, be $wsp(P, S)$. Conversely, $wsp(P, S)$ is both a strict and a complete precondition of P under S. As stated in the text in Chapter 3, the preimage of P under S is $wsp(P, S)$. Thus the three given statements are equivalent.

4 By the definition of a complete precondition, it is a superset of $wsp(P, S)$. But $wsp(P, S)$ satisfies the definition of a complete precondition, so it is the strongest complete precondition.

5 When two or more variables with the same name are declared in a multiple declaration statement, the thesis of the lemma must be altered accordingly. The equations for accessible variables are as stated, whereby only the first variable in any group of identically named variables being

declared is taken into consideration. The other variables in such a group become concealed immediately and are taken into account among the equations for concealed variables. The exponents $n+1$ must be replaced by $n+m$, where m is the number of variables with the corresponding name which are declared in the multiple declaration statement.

Consider the example given in the statement of the problem. The resulting data environment $d1$ is as follows:

$$d1 = D(d0) = [(x, S1, E1(d0)), (x, S2, E2(d0)), (w, S3, E3(d0)), (z, S4, v4), (x, S5, v5)]$$

The thesis of the lemma must be restated as follows. For accessible variables, we have

$$\text{valvar}(x, d1) = \text{valexp}(E1, d0)$$
$$\text{valvar}(w, d1) = \text{valexp}(E3, d0)$$

and

$$\text{valvar}(z, d1) = \text{valvar}(z, d0) \cdot$$

For concealed variables, the following equations apply:

$$\text{valvar}(x, (\textbf{release } x)^1 (d1)) = \text{valexp}(E2, d0)$$

and for all integers $n \geq 0$ (only $n = 0$ is of relevance in this example)

$$\text{valvar}(x, (\textbf{release } x)^{n+2}(d1)) = \text{valvar}(x, (\textbf{release } x)^n(d0))$$

6 In view of the fact that the stated postconditions refer only to values of accessible variables, the theorem for the assignment statement (Section 3.0.1) can be applied. Only the first declaration of each variable name is of relevance, because it is this variable which is accessible in the new data environment. The preconditions are, therefore, as follows.

$\{P1(E1, z)\}\ D\ \{P1(x, z)\}$ completely
$\{P2(E3, E1, z)\}\ D\ \{P2(w, x, z)\}$ completely

Chapter 6: The construction of correct programs

1 The loop invariant is:

$$\begin{aligned}
&first \leq a \leq il \\
&\textbf{and } ih \leq b \leq last \\
&\textbf{and } [(il \leq ih) \textbf{ or } (ih = a - 1 \textbf{ and } il = b + 1)] \\
&\textbf{and}_{i=first}{}^{a-1}\ k(i) < skey \\
&\textbf{and}_{i=il}{}^{ih}\ k(i) = skey \\
&\textbf{and}_{i=b+1}{}^{last}\ k(i) > skey
\end{aligned}$$

The term $(il \leq ih)$ in the third line above represents case 1; the remainder of that line, case 2.

2 One can prove that the body of the loop preserves the truth of the loop invariant in the same general way as was used in Section 5.8.0. Alternatively but equivalently, one can determine the precondition of each part of the loop invariant under the body of the loop and show that the loop invariant and the **while** condition imply that precondition. The fact that the array is in non-descending order will be required in several places in this proof. In this way, this condition becomes part of the precondition of the subprogram.

3 Derive a precondition of the loop invariant under the initialization; this becomes a precondition of the entire subprogram. Show that the loop invariant and the negation of the **while** condition together imply the postcondition. Finally, show that the loop terminates (hint: show that the size of the unknown region(s) is reduced by at least 1 during each execution of the body of the loop) and that each individual statement will execute (i.e. that all referenced variables are defined, that their values are within the required ranges, etc.).

4 Theorem: Let $d0$ be a data environment containing variables named *first* and *last* with integer values such that $first - 1 \leq last$. Furthermore, let $d0$ contain variables named $k(first)$, $k(first + 1)$, ..., $k(last)$ and *skey* with values in a common linearly ordered set and with $k(first) \leq k(first + 1) \leq \ldots \leq k(last)$. Then the result of executing the subprogram designed in Section 6.2 upon $d0$ will be the data environment

$$d1 = [(ih, \mathbb{Z}, \cdot), (il, \mathbb{Z}, \cdot)] \ \&\ d0$$

satisfying the postcondition

$$\begin{aligned}
&first \leq il \\
&\textbf{and } il - 1 \leq ih \\
&\textbf{and } ih \leq last \\
&\textbf{and}_{j=first}{}^{il-1}\ k(j) < skey \\
&\textbf{and}_{j=il}{}^{ih}\ k(j) = skey \\
&\textbf{and}_{j=ih+1}{}^{last}\ k(j) > skey
\end{aligned}$$

Proof (sketch): The main parts of the proof are outlined in the answers to the questions 1 through 3 above. In addition, one must trace the effects of the declaration and release statements outside the loop in order to demonstrate that $d1$ has the structure stated in the thesis of the theorem. ■

5 It is frequently useful to trace the effect of each statement in the body of the loop upon the diagrammatic form of the loop invariant. In this way the designer can often verify the correctness of his initial design easily and quickly, if not completely rigorously. This procedure is illustrated below.

If he discovers no errors during this informal procedure, he should then

prove the correctness of his design rigorously. The insight gained by tracing the development of the diagrammatic form of the loop invariant often helps him in constructing the complete, formal proof.

Before the body of the loop is executed, the loop invariant and the **while** condition are true:

$\|il$	$\|k$	$eq\|$	$gr\|$	ig'
$x(i)<key$	$x(i)?$	$x(i)=key$	$x(i)>key$	

if $x(eq) < x(gr)$
then

$\|il$	$\|k$	$eq\|$	$gr\|$	$ig\|$
$x(i)<key$	$x(i)?$	$<$ $x(i)=key$	$x(i)>key$	

$x(eq) :=: x(k)$

$\|il$	$\|k$	$eq\|$	$gr\|$	$ig\|$
$x(i)<key$	$<$ $x(i)?$	$?$ $x(i)=key$	$x(i)>key$	

$k := k + 1$

$\|il$	$\|k$	$eq\|$	$gr\|$	$ig\|$
$x(i)<key$	$x(i)?$	$x(i)=key$	$x(i)>key$	

else
if $x(eq) = x(gr)$
then

$\|il$	$\|k$	$eq\|$	$gr\|$	$ig\|$
$x(i)<key$	$x(i)?$	$=$ $x(i)=key$	$x(i)>key$	

$eq := eq - 1$

$\|il$	$\|k$	$eq\|$	$gr\|$	$ig\|$
$x(i)<key$	$x(i)?$	$x(i)=key$	$x(i)>key$	

else $\{x(eq)>x(gr)\}$

$\|il$	$\|k$	$eq\|$	$gr\|$	$ig\|$
$x(i)<key$	$x(i)?$	$>$ $x(i)=key$	$x(i)>key$	

$x(eq) :=: x(gr)$

$\|il$	$\|k$	$eq\|$	$gr\|$	$ig\|$
$x(i)<key$	$x(i)?$	$=$ $x(i)=key$	$>$ $x(i)>key$	

$gr := gr - 1$
$eq := eq - 1$

$\|il$	$\|k$	$eq\|$	$gr\|$	$ig\|$
$x(i)<key$	$x(i)?$	$x(i)=key$	$x(i)>key$	

endif
endif

Thus after the body of the loop has been executed, one of the following diagrams applies. Either

$\|il$	$\|k$	$eq\|$	$gr\|$	$ig\|$
$x(i)<key$	$x(i)?$	$x(i)=key$	$x(i)>key$	

or

$\|il$	$\|k$	$eq\|$	$gr\|$	$ig\|$
$x(i)<key$	$x(i)?$	$x(i)=key$	$x(i)>key$	

or

$\|il$	$\|k$	$eq\|$	$gr\|$	$ig\|$
$x(i)<key$	$x(i)?$	$x(i)=key$	$x(i)>key$	

all of which represent the loop invariant but with different positions of the boundaries of the several regions. Thus, the body of the loop preserves the truth of the loop invariant. Note that in each case, the size of the unknown region is decreased.

6 If $x(i) < x(gr)$, then the values $x(i)$ and $x(k)$ should be exchanged. If $x(i) = x(gr)$, then $x(i)$ and $x(eq)$ should be exchanged. If $x(i) > x(gr)$, then two exchanges are required in order to bring the original $x(i)$ into position gr, the original $x(gr)$ into position eq and the original $x(eq)$ into position i.

7 Theorem: Let $d0$ be a data environment containing the variables il and ig with integer values satisfying the inequality $il \leq ig$. Furthermore, let $d0$ contain the subscripted variables $x(il)$, $x(il + 1)$, ..., $x(ig)$ with values in a common, linearly ordered set. Then the result of executing the subprogram '*partition*' upon $d0$ will be the data environment

$$d1 = [(eq, \mathbb{Z}, \cdot), (gr, \mathbb{Z}, \cdot)] \ \& \ d0'$$

where $d0'$ is equal to $d0$ except for the values of $x(il)$, $x(il + 1)$, ..., $x(ig)$. The resulting data environment $d1$ will satisfy the postcondition stated in the text of Section 6.3.

Proof (sketch): This theorem can be proved in the same manner as the theorem given in the answer to question 4. ■

8 Theorem: Let $d0$ be a data environment containing the variables il and ig with integer values satisfying the inequality $il - 1 \leq ig$. Furthermore, let $d0$ contain the subscripted variables $x(il)$, $x(il + 1)$, ..., $x(ig)$ with values

in a common, linearly ordered set. Then the result of executing the subprogram '*sort*' upon $d0$ will be a data environment $d1$ which is equal to $d0$ except that the values of $x(il)$, $x(il + 1)$, ..., $x(ig)$ will have been permuted so that in $d1$ they are sorted, i.e.

$$x(il) \leq x(il + 1) \leq \ldots \leq x(ig)$$

Proof (sketch): Prove the theorem by induction on the number of elements in the array to be sorted ($ig - il + 1$, called n below). If this number is 0 or 1, $il \geq ig$ and the subprogram does nothing; $d1 = d0$ and the thesis of the theorem is satisfied.

If $n > 1$, i.e. $il < ig$, assume that the subprogram sorts an array containing fewer than n elements correctly. Show that the precondition of the subprogram '*partition*' (i.e. the hypothesis of its correctness theorem, see question 7) is satisfied. The 'equal' region created by '*partition*' will not be empty, so the 'less than' region will contain fewer than n elements, as will the 'greater region'. Show that when '*sort*' is first called recursively, its precondition is satisfied for sorting the 'less than' region returned by '*partition*'. Similarly, show that the second recursive call to '*sort*' will sort the 'greater than' region. Show that $x(eq) \leq x(eq + 1)$ and $x(gr) \leq x(gr + 1)$. Combining these inequalities with the fact that each of the three regions is also sorted, conclude that the entire array is sorted. Finally, verify that all variables declared in '*sort*' and its subsidiary subprograms are released, so that the resulting data environment has the specified structure. ■

9 Define the several terms appearing in I and E as follows:

$T1$: $pred = endvalue$
$T2$: $f = succ$
$T3$: $p(pred) = succ$
$T4$: $e(pred) < skey$
$T5$: $succ = endvalue$
$T6$: $skey \leq e(succ)$

Then

$$I = (T1 \textbf{ and } T2) \textbf{ or } (\textbf{not } T1 \textbf{ and } T3 \textbf{ and } T4)$$

and

$$E = T5 \textbf{ or } T6$$

or, equivalently,

$$E = T5 \textbf{ or } (\textbf{not } T5 \textbf{ and } T6)$$

Therefore,

$$I \textbf{ and } E = (T1 \textbf{ and } T2 \textbf{ and } T5) \textbf{ or}$$

$$(T1 \textbf{ and } T2 \textbf{ and not } T5 \textbf{ and } T6) \textbf{ or}$$
$$(\textbf{not } T1 \textbf{ and } T3 \textbf{ and } T4 \textbf{ and } T5) \textbf{ or}$$
$$(\textbf{not } T1 \textbf{ and } T3 \textbf{ and } T4 \textbf{ and not } T5 \textbf{ and } T6)$$

or

$$pred = endvalue \textbf{ and } succ = endvalue \textbf{ and } f = succ \textbf{ or}$$

$$pred = endvalue \textbf{ and } succ \neq endvalue \textbf{ and } f = succ$$
$$\textbf{and } skey \leq e(succ) \textbf{ or}$$

$$pred \neq endvalue \textbf{ and } succ = endvalue \textbf{ and } p(pred) = succ$$
$$\textbf{and } e(pred) < skey \textbf{ or}$$

$$pred \neq endvalue \textbf{ and } succ \neq endvalue \textbf{ and } p(pred) = succ$$
$$\textbf{and } e(pred) < skey \leq e(succ)$$

which is equivalent to the final version of the postcondition given in the text.

10(a) Upon termination of the loop, the mth element in the list satisfies the postcondition, i.e. is the first element in the list which is greater than or equal to *skey*. If $m = n + 1$, all elements in the list (if any) are less than *skey*. During execution of the loop, the first $m - 1$ elements in the list have been found to be less than *skey*.

(b) The postcondition can be rewritten in the form

$$1 \leq m \leq n + 1$$
$$\textbf{and } [(m < 2) \textbf{ or } (m \geq 2) \textbf{ and}_{i=1}{}^{m-1}\ e(p^{i-1}(f)) < skey]$$
$$\textbf{and } [(m > n) \textbf{ or } (m \leq n) \textbf{ and}_{i=m}{}^{n}\ skey \leq e(p^{i-1}(f))]$$

(see Appendix 0, Section A0.4, Series notation).

Because the list is ordered, the above form of the postcondition can be simplified to

$$1 \leq m \leq n + 1$$
$$\textbf{and } [(m = 1) \textbf{ or } (m \neq 1) \textbf{ and } e(p^{m-2}(f)) < skey]$$
$$\textbf{and } [(m = n + 1) \textbf{ or } (m \neq n + 1) \textbf{ and } skey \leq e(p^{m-1}(f))]$$

The variable *succ* represents $p^{m-1}(f)$; *pred*, $p^{m-2}(f)$. Thus, $succ = p(pred)$ when $m \geq 2$. The situation $pred = endvalue$ corresponds to $m = 1$; $succ = endvalue$, to $m = n + 1$. The first inequality in the above postcondition states that an actual element under consideration must be in the list in question; this condition is not expressed explicitly in the postcondition in the text in Section 6.5.1. Leaving it aside, the above postcondition can be rewritten in the form

$$(m = 1) \textbf{ and } (m = n + 1) \textbf{ or}$$
$$(m = 1) \textbf{ and } (m \neq n + 1) \textbf{ and } skey \leq e(p^{m-1}(f)) \textbf{ or}$$

$(m \neq 1)$ **and** $(m = n + 1)$ **and** $e(p^{m-2}(f)) < skey$ **or**
$(m \neq 1)$ **and** $(m \neq n + 1)$ **and** $e(p^{m-2}(f)) < skey \leq e(p^{m-1}(f))$

which can be transformed into

pred = *endvalue* **and** *succ* = *endvalue* **or**
pred = *endvalue* **and** *succ* ≠ *endvalue* **and** *skey* ≤ *e*(*succ*) **or**
pred ≠ *endvalue* **and** *succ* = *endvalue* **and** *e*(*pred*) < *skey* **or**
pred ≠ *endvalue* **and** *succ* ≠ *endvalue* **and** *e*(*pred*) < *skey* ≤ *e*(*succ*)

In the last transformation, information regarding the relationship between *succ* and *pred* was lost; reintroducing it in the appropriate form for each of the four cases, we obtain

pred = *endvalue* **and** *succ* = *endvalue* **and** *succ* = *f* **or**
pred = *endvalue* **and** *succ* ≠ *endvalue* **and** *succ* = *f*
and *skey* ≤ *e*(*succ*) **or**
pred ≠ *endvalue* **and** *succ* = *endvalue* **and** *succ* = *p*(*pred*)
and *e*(*pred*) < *skey* **or**
pred ≠ *endvalue* **and** *succ* ≠ *endvalue* **and** *succ* = *p*(*pred*)
and *e*(*pred*) < *skey* ≤ *e*(*succ*)

which is equivalent to the final version of the postcondition stated in Section 6.5.1.

(c) The loop scans the list, element by element, from the first to the last element. The second term in the postcondition pertains to previous elements in the list; the last term, to succeeding elements. It seems natural, therefore, to interpret the second term in the postcondition as pertaining to the elements already scanned. This suggests defining the loop invariant as the first two terms of the postcondition and the termination condition, as the third term. The second form of the postcondition stated in the solution to part (b) above is probably the most appropriate for our purposes and leads to

$1 \leq m \leq n + 1$
and $[(m = 1)$ **or** $(m \neq 1)$ **and** $e(p^{m-2}(f)) < skey]$

for the loop invariant and

$[(m = n + 1)$ **or** $(m \neq n + 1)$ **and** $skey \leq e(p^{m-1}(f))]$

or, more simply and equivalently,

$m = n + 1$ **or** $skey \leq e(p^{m-1}(f))$

for the termination condition. Negating the termination condition, we obtain

$m \neq n + 1$ **and** $e(p^{m-1}(f)) < skey$

for the **while** condition.

The program is, then

```
m := 1
while m ≠ n + 1 and e(p^(m-1)(f)) < skey do
    m := m + 1
endwhile
```

(d) The program developed in part (c) above is not suitable for execution on most systems. The notation for $p(p(p(\ldots)))$ used above is not typically implemented. Furthermore, n is typically not known and the value of m is seldom of interest in the context of the application.

One can, however, convert this version of the program into a more practically useful one quite easily. We begin by defining the variable *succ* as the pointer to the current element of the list, i.e.

$succ = p^{m-1}(f)$

which permits rewriting the **while** condition in a form which is syntactically correct in most actual computing systems. The effect of increasing m in the above version of this program can be achieved by assigning $p(succ)$ as the new value of *succ*, thereby stepping to the next element in the list. The initialization in effect assigns the value of f to *succ*. The first term in the **while** condition detects the end of the list; this is equivalent to comparing *succ* with *endvalue*. Modifying the above version of the program accordingly, we obtain

```
succ := f
while succ ≠ endvalue and e(succ) < skey do
    succ := p(succ)
endwhile
```

The version of the program developed in the text of Section 6.5.1 calculated an additional variable, *pred*, which points to the element preceding the one to which *succ* points. This goal can be achieved simply by assigning to *pred* the value of *succ* immediately before *succ* is modified to point to the next element in the list. Adding *pred* to the initialization to reflect the convention that *endvalue* 'points' to f, the 'predecessor' of the first element in the list, and declaring the variables *pred* and *succ*, the program becomes

```
declare (pred, subscripts, endvalue)
declare (succ, subscripts, f)
while succ ≠ endvalue and e(succ) < skey do
    pred := succ
    succ := p(succ)
endwhile
```

which is equivalent to the final version of the body of the procedure '*locate*' developed in Section 6.5.1.

Note the relative simplicity of this somewhat more abstract (some programmers would say less practical) approach, even though the first version of the program developed was not usable in that form.

11 The procedure '*delete*' is to delete an element of the list whose value is equal to *skey*. The postcondition of the procedure '*locate*' does not guarantee equality, however, but only that the located element is either greater than or equal to *skey*. (There may be no element in the list whose value is equal to *skey*.) The **if** condition in question is required to distinguish between these two possibilities and to ensure that the element is deleted only if its value is equal to *skey*.

12 (Sketch of proof) Because the original list was ordered, the sublist ending with the element to which *pred* points is ordered, as is the sublist beginning with the element to which *succ* points. The postcondition of the procedure '*locate*' ensures that $e(pred) < skey = e(pnew) \leq e(succ)$. Together, these facts imply that the modified list is ordered.

13 (Sketch of proof) Show that the program invariant is a postcondition of the procedure 'open file' and that it is both a pre- and postcondition of the other two procedures. The propositions then follow in a straightforward manner. The theorem at the end of procedure 'position file' follows from the loop invariant stated in the proposition before the **while** statement and the negation of the **while** condition.

The various pre- and postconditions can be proved formally if one replaces '**read** *x*' by '*line* := *line* + 1, *x* := *file*(*line*)' and then applies the proof rules presented in Chapter 3. The **open** and **rewind** statements should be replaced by statements initializing the variable *line*.

14 The analysis numbers, including the zero marking the end of a sample group, are read and processed as a single group in the subprogram which transmits them to the analyzer. The standard file position should, therefore, be outside of this data group, i.e. either immediately before or immediately after a sample number.

The designer probably felt that it would be logically simpler if the file were positioned immediately before the next data item which would normally be required for transmission to the analyzer. Perhaps he also suspected that the logic for detecting and handling an end of file condition would be simpler if a file position immediately after the last possible data element in the file were a standard position (see below).

If a position immediately before a sample number had been chosen for the program invariant, the structure of the subprograms would be slightly different: the sample number would be read near the beginning, rather than near the end, of the two larger procedures and it would not be read at all

in the procedure 'open file'. Slightly more complex logic would be required in the **while** loop in the procedure 'position file' to handle the end of file condition. Also, the designer would have to specify the meaning of the variable *pos* in the program invariant more precisely (whether it is the position number of the previous or next sample group), as this point is not so obvious with the alternative invariant. These might have been other reasons for the designer's choice for the standard file position.

15 The obvious choice for a postcondition of the procedure 'print footer' is *lineno* = *llp*. After the **while** loop terminates, it must, therefore, be true that *lineno* = *lld*. The **while** loop itself (more precisely, its termination condition) ensures only that *lineno* ≥ *lld*. The required postcondition of the **while** loop will be satisfied only if *lineno* ≤ *lld* is a loop invariant and, therefore, only if the loop initialization establishes this condition. Since there is no initialization of the loop in the procedure 'print footer', the calling subprogram must ensure that *lineno* ≤ *lld*; i.e. *lineno* ≤ *lld* is a precondition of the procedure 'print footer'.

Initially, *lineno* = *llp* and *pageno* < *firstpage*. If *lld* < *llp*, then *lineno* > *lld*, violating the precondition of the procedure 'print footer'. Thus, the **if** condition is required in the procedure 'new page'.

16 A precondition of the procedure 'print footer' is *lineno* ≤ *lld*. Its postcondition is *lineno* = *llp*. (See question 15 above.)

A precondition of the procedure 'print header' is *lineno* = *llp*. Its postcondition is *lineno* = *llh* **and** *pageno* ≥ *firstpage*.

In addition, term *I*1 of the program invariant is both a precondition and a postcondition of these subprograms.

17 The **if** condition is required in the procedure 'terminate printing' for the reasons given in the solution to question 15. Viewed differently, the procedure 'print footer' should be called at the end of printing a report if and only if the report is not empty. If the report is not empty, *pageno* ≥ *firstpage*. Conversely, if *pageno* ≥ *firstpage*, then it follows from the program invariant that *lineno* > *llh*, which means that data has been printed on the report; i.e. the report is not empty if and only if *pageno* ≥ *firstpage*, so this is an appropriate condition for the **if** statement in question.

Specifying the procedure's precondition itself, i.e. *lineno* ≤ *lld*, as the **if** condition would, of course, lead to correct operation. The condition *lineno* < *llp* would also be correct.

When such alternatives are available, the designer should apply the following criteria in making his choice: readability of the resulting program, the ease with which it can be understood and the logical simplicity of its proof of correctness. Although these criteria are distinct, an alternative which satisfies one usually satisfies the other as well. These goals rarely conflict with one another.

18 A modification of the program invariant (replace term *I*2 by *pageno* ≥ *firstpage* **and** *llh* ≤ *lineno* ≤ *lld*) is a loop invariant of the **while** loop in the procedure 'print a data group'. The following proposition is also a loop invariant: The values of the array variables *dataline*(*first*), *dataline*(*first* + 1), ..., *dataline*(*i*) have been printed **and** *first* − 1 ≤ *i* ≤ *last*. More precisely,

$$\textbf{and}_{j=0}^{i-first}\ report(pageno,\ lineno - j) = dataline(i - j)$$
and *first* − 1 ≤ *i* ≤ *last*

where *report*(*pageno*, *lineno*) is a variable whose value is the data printed on the corresponding page and line of the report (see Section 4.1.1, Display output).

19 This is a moderately detailed but straightforward task. Use the various invariants, preconditions and postconditions stated in the text and in the answers to the questions 15 through 18.

20 Operationally, the assignment statement *lineno* := *llp* in the procedure 'print footer' is redundant. This procedure is called only in the procedures 'new page' and 'terminate printing'. After it is called in 'new page', the procedure 'print header' is called, for whose operation the initial value of the variable *lineno* is irrelevant. After 'terminate printing' has executed, the values of the various printing control variables are of no operational significance.

21 The postcondition is term *I*1 of the program invariant **and** *lineno* = *llp* **and** *pageno* ≥ *firstpage* − 1. The paper will be positioned so that the next line transmitted to the printer will be printed on the first line of a new page. Note that this latter condition is the initial condition required by our subprograms and established in the procedure 'initialize printing'. Thus it is an invariant of the report printing system.

22 One can verify that the two specified conditions are pre- and postconditions of the two procedures by tracing their effects in a progressive analysis. Alternatively, a rigorous retrogressive analysis can be performed.

23 Apply the retrogressive proof rule for the assignment statement to the statement which increments the variable *online*. The precondition so derived is part of the precondition of the procedure.

24 The **if** conditions on both calls to the procedure 'close current line' ensure that *online* > 0 whenever the statement 'terminate the current printed line' is executed.

25 Replace every 'print' command with a corresponding assignment (or declaration) statement of the form *report*(*pageno*, *lineno* + 1, *groupno*) := .... Show that the condition 0 < *groupno* ≤ *N* is true whenever such a statement is executed. Alternatively, show that *online* ≤ *N* whenever the statement 'terminate the current printed line' is executed.

26 Only the last line of the report can contain fewer than *N* data groups.

A line is terminated only in the procedure 'close current line'. This procedure is called from two places. Only one of these calls can be executed with *online* < *N*. That call is executed at the end of the report, immediately before terminating the report.

27 In Section 6.7 a program invariant was specified for the report printing system. That program invariant must be extended for our purposes in Section 6.8 to the following condition, which is a precondition and a postcondition of the procedure 'output a data group'. From the structure of this procedure, it follows that this program invariant must be a postcondition (but not necessarily a precondition) of the procedure 'close current line' also.

*I*1: The report has been printed to and including
line number *lineno* on page number *pageno* completely (i.e. all of these lines have been terminated) and

data group number *online* on line number *lineno* + 1 on page number *pageno*. (This line has not yet been terminated.)
Subsequent data groups have not been printed.

and

*I*2: [(*pageno* = *firstpage* − 1 **and** *llp* = *lineno* **and** *online* = 0)
or (*online* = 0 **and** *llh* < *lineno* ≤ *lld*)
or (0 < *online* < *N* **and** *llh* ≤ *lineno* < *lld*)]

The last two lines in *I*2 state that data has been printed on the current page.

The postcondition of the procedure 'close current line' is *online* = 0 **and** the above program invariant. In order to ensure the truth of the postcondition after execution of this procedure and that the correctness criteria of the report are not violated by the execution of this procedure, the following precondition must be satisfied upon calling 'close current line'.

llh ≤ *lineno* < *lld* **and** *online* ≤ *N*

This precondition will always be satisfied. Its truth follows from the program invariant as a precondition of the procedure 'output a data group' together with the test for page overflow in that procedure. In the case of the call to 'close current line' at the end of the report, the program invariant and the **if** condition on the call statement together imply the above precondition.

28 The condition is superfluous. The condition *lineno* ≥ *lld* is sufficient. This latter condition and the program invariant (see the solution to question 27 above) together imply *online* = 0. The designer undoubtedly included it to make it clear to the reader of the procedure that the subprogram 'new page' would never be called if the current line were partially printed and unterminated.

29 The loop invariant is: Player S wishes to remove W matches.
30 First, extend the loop invariant: Player S now wishes to remove W matches after having previously entered E invalid choices in succession (in one turn). Insert a new statement before the **loop** statement initializing E to 0. Replace that part of the loop between **loop** and **if** ... **exit** by

```
if E ≥ emax
then Inform player S that the computer is making a valid decision on
     his behalf.
     W := a valid choice (e.g. 1)
else if N = 1
     then Inform player S that he must remove the last match.
     W := 1
     else Ask player S how many matches he wishes to remove (W).
     endif
endif
```

and insert a statement before **endloop** which increases E by 1. The variable *emax* above is a constant parameter whose value should be some positive integer. The program as extended above will assign a valid choice to W after a player has entered *emax* invalid decisions in succession.
31 Rewrite the initialization as a sequence of declaration statements and apply the retrogressive proof rule. It will then be seen that the initialization establishes the truth of the loop invariant provided that the control file contains appropriate values and the corresponding files exist and contain valid data (see Section 6.10.4, Initial conditions). These conditions must be established by this system's developers, who must create these files in some way, e.g. by manually entering the data via a file editing program.
32 If $dtime \leq ltpres$, changing a decision file for time period *dtime* would have the effect of invalidating the contents of the result file for time period *dtime* (see the definition of *ltpres* in Section 6.10.3). It would, therefore, be necessary to change (reduce) the value of *ltpres*. Such action would constitute an implicit reversion of the game to an earlier time period. While in principle correct, this could be misleading and confusing to the trainer and participants; it would probably be better to require the trainer to reset the game explicitly to a previous time period if he really wishes to do so. We require, therefore, that $dtime \geq ltpres + 1 = curtime$.

In order to enter decisions (e.g. of the trainer) for time period *dtime*, a corresponding decision file must exist for at least the previous time period; i.e. $dtime - 1 \leq ltptd$. Combining with the lower bound on *dtime* derived in the paragraph above, we obtain

$$curtime \leq dtime \leq ltptd + 1$$

For each management team, the corresponding inequality applies.
33 The second term in the loop invariant is $ltpres \leq ltptd$. Therefore, $ltpres + 1 \leq ltptd + 1$. But $curtime = ltpres + 1$, so it follows that $curtime \leq ltptd + 1$. Therefore, $dtime = curtime$ always satisfies the bounds derived for *dtime* in the solution to question 32 above.
34 If the alternative terms 1(a) and 2(a) are specified in the loop invariant, then part of the postcondition of the decision entry subprogram is $ltptd \leq ltpres + 1 = curtime$. This subprogram contains the assignment statement $ltptd := dtime$. Applying the retrogressive proof rule to that assignment statement, we obtain $dtime \leq curtime$ as a precondition. (The same argument applies in the case of management team decisions). Combining this upper bound on *dtime* with the lower bound derived in the solution to question 32 above, it follows that $dtime = curtime$ when the alternative terms 1(a) and 2(a) are specified in the loop invariant.
35 Requiring *dtime* to be equal to *curtime* will cause *ltptd* or *ltpmd*(*c*) to be set to $curtime = ltpres + 1 > ltpres$ after the corresponding set of decisions are entered; i.e. after a set of decisions has been entered, $ltpres < ltptd$ or $ltpres < ltpmd(c)$, which is clearly consistent with term 1 or term 2 respectively of the loop invariant.
36 If $curtime \leq 1$, then $ltpres = curtime - 1 \leq 0$. The loop invariant requires that $ltpres \geq 0$, so it must be true that $ltpres = 0$. *ltpres* cannot be reduced any further, i.e. the game cannot be reset to a truly earlier period without violating the loop invariant.
37 (Sketch of the proof's structure) Rewrite the program using only the fundamental statements defined in Chapter 2. In particular, express all file operations as declaration or assignment statements involving variables representing data stored in files. Use one array variable with appropriate subscripts to represent all decision variables in one file. Restate the loop invariant accordingly. Then apply the retrogressive proof rules to show that the loop invariant is a precondition of the loop invariant (as a postcondition) with respect to the body of the loop – i.e. to prove that the body of the loop preserves the truth of the loop invariant. Note that a detailed proof is required only for those segments of the program which modify one or more of the variables referenced in the loop invariant.

Bibliography

If one wants to make progress in mathematics, one should study the masters and not the pupils.
– Niels Henrik Abel

Lernolibron oni devas ne tralegi, sed tralerni.
– Ludoviko Lazaro Zamenhof

A'h-mose (ca. 1700*BC*) 'The Rhind Papyrus', excerpted and commented in Newman (q.v.), Vol. 1, Ch. 2, pp. 169–78.

Aho, Alfred V., Hopcroft, John E. and Ullman, Jeffrey D. (1974) *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, Massachusetts.

Alagic, Suad and Arbib, Michael A. (1978) *The Design of Well-Structured and Correct Programs*, Springer-Verlag, New York.

Anderson, Robert B. (1979) *Proving Programs Correct*, John Wiley & Sons, New York.

Arbib, Michael A. (1969) *Theories of Abstract Automata*, Prentice-Hall, Englewood Cliffs, N. J.

Baber, Robert Laurence (1982) *Software Reflected: The Socially Responsible Programming of Our Computers*, North-Holland, Amsterdam.

Baber, Robert Laurence (1985) 'I/O statements in higher programming languages: unnecessary and undesirable', *IEEE Computer*, **18**, 6, p. 112, June.

Baber, Robert Laurence (1986) *Softwarereflexionen: Ideen und Konzepte für die Praxis*, Springer-Verlag, Berlin.

Backhouse, Roland C. (1986) *Program Construction and Verification*, Prentice-Hall International, Englewood Cliffs, N. J.

Bakker, Jacobus Willem de (1980) *Mathematical Theory of Program Correctness*, Prentice-Hall International, Englewood Cliffs, N. J.

Bauer, Friedrich L. and Wössner, Hans (1984) *Algorithmische Sprache und Programmentwicklung*, Springer-Verlag, Berlin.

Beckman, Frank S. (1980) *Mathematical Foundations of Programming*, Addison-Wesley, Reading, Massachusetts.

Boyer, Robert S. and Moore, J. Strother (eds.) (1981) *The Correctness Problem in Computer Science*, Academic Press, London.

Brady, J. M. (1977) *The Theory of Computer Science: A Programming Approach*, Chapman and Hall, London.

Brooks, Frederick P. Jr. (1979) *The Mythical Man-Month*, Addison-Wesley, Reading, Massachusetts.

Brooks, Frederick P. Jr. (1986) 'No Silver Bullet – Essence and Accidents of Software Engineering', Information Processing 86, *Proceedings of the IFIP 10th World Congress*, Dublin, Ireland, September 1–5, pp. 1069–76, edited by H.-J. Kugler, North-Holland, Amsterdam.

Buxton, J. N. and Randell, Brian (eds.) (1970) *Software Engineering Techniques, Report on a conference sponsored by the NATO Science Committee, Rome, Italy, 27th to 31st October 1969*, NATO Science Committee, Brussels.

Dahl, Ole-Johan, Dijkstra, Edsger W. and Hoare, Charles Antony Richard (1972) *Structured Programming*, Academic Press, London.

Dal Cin, Mario, Lutz, Joachim and Risse, Thomas (1984) *Programmierung in Modula-2*, B. G. Teubner, Stuttgart.

De Millo, Richard A., Lipton, Richard J. and Perlis, Alan J. (1979) 'Social Processes and Proofs of Theorems and Programs', *Communications of the ACM*, **22**, 5, pp. 271–80, May.

Denvir, Tim (1986) *Introduction to Discrete Mathematics for Software Engineering*, Macmillan Education, Basingstoke.

Dijkstra, Edsger W. (1968) 'The Structure of the 'THE'-Multiprogramming System', *Communications of the ACM*, **11**, 5, pp. 341–6, May.

Dijkstra, Edsger W. (1976) *A Discipline of Programming*, Prentice-Hall, Englewood Cliffs, N. J.

Dijkstra, Edsger W. (1982) *Selected Writings on Computing: A Personal Perspective*, Springer-Verlag, New York.

Dijkstra, Edsger W. (1986) *Science fiction and science reality in computing*, EWD 952 (semipublicly distributed by the author).

Fehr, E. (1982) 'Funktionale Programmierung', *Informatik-Spektrum*, **5**, 3, pp. 194–6, September, Springer-Verlag, Berlin.

Filman, Robert E. and Friedman, Daniel P. (1984) *Coordinated Computing: Tools and Techniques for Dsitributed Software*, McGraw-Hill, New York.

Floyd, Robert W. (1967) 'Assigning Meanings to Programs', *Proceedings of the Symposium of Applied Mathematics*, **19**, pp. 19–32, American Mathematical Society, Providence, R. I.

Foley, M. and Hoare, Charles Antony Richard (1971) 'Proof of a recursive program: Quicksort', *The Computer Journal*, **14**, 4, pp. 391–5, November.

Goos, Gerhard and Hartmanis, Juris (eds.) (1981) 'The Programming Language Ada Reference Manual', *Lecture Notes in Computer Science*, **106**, Springer-Verlag, Berlin.

Gries, David (ed.) (1978) *Programming Methodology*, Springer-Verlag, New York.

Gries, David (1981) *The Science of Programming*, Springer-Verlag, New York.

Guillemin, Ernst A. (1953) *Introductory Circuit Theory*, John Wiley & Sons, New York.

Hehner, Eric C. R. (1984) *The Logic of Programming*, Prentice-Hall International, Englewood Cliffs, N. J.

Herzog, Otthein, Reisig, Wolfgang and Valk, Rüdiger (1984) 'Petri-Netze: ein Abriß ihrer Grundlagen und Anwendungen', *Informatik-Spektrum*, **7**, 1, pp. 20–7, Februar, Springer-Verlag, Berlin.

Hoare, Charles Antony Richard (1969) 'An Axiomatic Basis for Computer Programming', *Communications of the ACM*, **12**, 10, pp. 576–80, 583, October.

Hoare, Charles Antony Richard (1981) 'A Calculus of Total Correctness for Communicating Processes', Technical Monograph PRG-23, Oxford University Computing Laboratory, April.

Hoare, Charles Antony Richard (1984) 'Programming: Sorcery or Science?', *IEEE Software*, **1**, 2, pp. 5–16, April.

Hoare, Charles Antony Richard (1985) *Communicating Sequential Processes*, Prentice-Hall International, Englewood Cliffs, N. J.

Jacobs, Dean and Gries, David (1985) 'General Correctness: A Unification of Partial and Total Correctness', *Acta Informatica*, **22**, 1, pp. 67–83, April.

Jones, Cliff B. (1980) *Software Development: A Rigorous Approach*, Prentice-Hall International, Englewood Cliffs, N. J.

Jones, Cliff B. (1986) *Systematic Software Development Using VDM*, Prentice-Hall International, Englewood Cliffs, N. J.

Kfoury, A. J., Moll, Robert N. and Arbib, Michael A. (1982) *A Programming Approach to Computability*, Springer-Verlag, New York.

Kley, Adolf (1986) 'Dynamische Systeme und kommunizierende Prozesse – eine Analogiebetrachtung', *Informatik-Spektrum*, **9**, 1, pp. 29–38, Februar, Springer-Verlag, Berlin.

Kline, Morris (1980) *Mathematics: The Loss of Certainty*, Oxford University Press, New York.

Knuth, Donald Ervin (1969) *The Art of Computer Programming, Volume 2, Seminumerical Algorithms*, Addison-Wesley, Reading, Massachusetts.

Knuth, Donald Ervin (1973) *The Art of Computer Programming, Volume 3, Sorting and Searching*, Addison-Wesley, Reading, Massachusetts.

Knuth, Donald Ervin (1978) *The Art of Computer Programming, Volume 1, Fundamental Algorithms*, 2nd edition, Addison-Wesley, Reading, Massachusetts.

Lewis, Harry R. and Papadimitriou, Christos H. (1981) *Elements of the Theory of Computation*, Prentice-Hall, Inc., Englewood Cliffs, N. J.

Linger, Richard C., Mills, Harlan D. and Witt, Bernard I. (1979) *Structured Programming: Theory and Practice*, Addison-Wesley, Reading, Massachusetts.

Loeckx, Jacques and Sieber, Kurt (1984) *The Foundations of Program Verification*, B. G. Teubner, Stuttgart, and John Wiley & Sons, Chichester.

Manna, Zohar (1974) *Mathematical Theory of Computation*, McGraw-Hill Kogakusha, Ltd., Tokyo.

McGowan, Clement L. and Kelly, John R. (1975) *Top-Down Structured Programming Techniques*, Petrocelli/Charter, New York.

Meriam, J. L. (1951) *Mechanics Part I, Statics*, John Wiley & Sons, New York.

Mills, Harlan D. (1980) 'How to Make Exceptional Performance Dependable and Manageable in Software Engineering', *Proceedings of the 4th Computer Software and Applications Conference*, October 27–31, pp. 19–23, IEEE.

Mills, Harlan D. (1986) 'Structured Programming: Retrospect and Prospect', *IEEE Software*, **3**, 6, pp. 58–66, November.

Milner, Robin (1986) 'Process Constructors and Interpretations', Information Processing 86, *Proceedings of the IFIP 10th World Congress*, Dublin, Ireland, September 1–5, pp. 507–14, edited by H.-J. Kugler, North-Holland, Amsterdam.

Myers, Glenford J. (1975) *Reliable Software through Composite Design*, Van Nostrand Reinhold, New York.

Myers, Glenford J. (1979) *The Art of Software Testing*, John Wiley & Sons, New York.

Naur, Peter (editor) (1962) *Revised Report on the Algorithmic Language Algol 60*, Regnecentralen, Copenhagen.

Naur, Peter and Randell, Brian (eds.) (1969) *Software Engineering, Report on a conference sponsored by the NATO Science Committee, Garmisch, Germany, 7–11 October 1968*, NATO Scientific Affairs Division, Brussels.

Newman, James R. (1956) *The World of Mathematics*, Volumes 1–4, Simon and Schuster, New York.

Parnas, David Lorge, Clements, Paul C. and Weiss, David M. (1985) 'The Modular Structure of Complex Systems', *IEEE Transactions on Software Engineering*, **11**, 3, pp. 259–66, March.

Reynolds, John, C. (1979) 'Reasoning about Arrays', *Communications of the ACM*, **22**, 5, pp. 290–9, May.

Royden, H. L. (1970) *Real Analysis*, 2nd edition, Collier-Macmillan, London.

Schnorr, Claus P. (1974) *Rekursive Funktionen und ihre Komplexität*, B. G. Teubner, Stuttgart.

Snowdon, Charles T., Brown, Charles H. and Petersen, Michael R. (eds.) (1982) *Primate Communication*, Cambridge University Press, Cambridge.

Sommerville, Ian (1985) *Software Engineering*, 2nd edition, Addison-Wesley, Wokingham.

Turski, Władysław M. (1978) 'Software Engineering – Some Principles and Problems'. In D. Gries (ed.) *Programming Methodology* (q.v.), pp. 29–36.

Turski, Władysław M. (1986) 'And No Philosophers' Stone, Either', Information Processing 86, *Proceedings of the IFIP 10th World Congress*, Dublin, Ireland, September 1–5, pp. 1077–80, edited by H.-J. Kugler, North-Holland, Amsterdam.

Wand, Mitchell (1980) *Induction, Recursion, and Programming*, North Holland, New York.

Wirth, Niklaus (1976) *Algorithms + Data Structures = Programs*, Prentice-Hall, Inc., Englewood Cliffs, N. J.

Witt, Jan (1982) 'Weakest Precondition', *Informatik-Spektrum*, **5**, 1, pp. 48–50, Februar, Springer-Verlag, Berlin.

Wulf, William A., Shaw, Mary, Hilfinger, Paul N. and Flon, Lawrence (1981) *Fundamental Structures of Computer Science*, Addison-Wesley, Reading, Massachusetts.

Author and Subject Index

THE SPINE OF SOFTWARE
Designing Provably Correct Software: Theory and Practice
Robert L. Baber
Software Engineering Consultant

'Software engineers and those who would become such' will find this book's original approach to the semantics of computer programs of considerable value in their chosen field. The book familiarizes software designers and developers with practically applicable results of research in the theory of proving programs correct. Robert L. Baber's approach is practical while his mathematical treatment is rigorous. Many examples illustrate the application of the theory to different types of design problems arising in actual practice.

A body of fundamental principles underlying computing science and software engineering has been developed in recent years. These principles provide guidelines for the design process and enable the software engineer to verify systematically and precisely important characteristics of his proposed design, just as, traditionally, engineers have achieved reliable, error free designs.

About the author
Robert L. Baber received Master of Science degrees in electrical engineering and in industrial management from the Massachusetts Institute of Technology. He is an expert on the design of information systems for management, business and technical applications and has advised clients in various countries on these topics. He has developed and instructs internationally a seminar on designing provably correct software, and is the author of numerous publications. Born in the USA, he now resides in the Federal Republic of Germany.

Contents

ISBN 0-471-91474-6

JOHN WILEY & SONS
Chichester · New York · Brisbane · Toronto · Singapore

BABER
THE SPINE OF SOFTWARE
DESIGNING PROVABLY CORRECT SOFTWARE
THEORY AND PRACTICE
WILEY