

Chapter 2

On Programming

We are *not* going to try to ‘teach programming’ in the sense that the reader might understand the term. In this chapter, we are going to make some observations on programs and programming — and on ways in which programmers have tried to ‘guarantee’ that their programs were ‘right’. In Chapter 3, we shall set about the formal derivation of programs from specifications. In some sense, therefore, these chapters are competing. We make no pretence that the competition is fair. For the reader who skips through this chapter, some common elements are repeated in Chapter 3.

Of course, not all programs are written by professional or even trainee or student programmers¹; and not all programmers are Computer Scientists or Software Engineers. Many of those involved in training or examining student programmers will often advocate the use of well-commented, well-structured code which uses meaningful identifiers. Nobody would argue with this, but it is certainly not enough. Merely adhering to these maxims produces programs which look pretty; programming is more properly about the way in which a program (denoted by the program text) is derived, how it is built, how it is constructed. What most people regard as a program — the final text or what it does when it is loaded into a computer and executed — is merely the output of the programming process.

Paradoxically, most good programs look simple (the converse need not be true), but it is how you get the program — and how you know that it does what it is required to do — which is important, not its appearance *per se*.

¹ But by workers in other professions who have decided to, or have been asked to, ‘write a program’. The mere fact that this situation arises frequently confirms the commonly held — but certainly erroneous — belief that programming is easy and can be undertaken by almost anybody. There is more to playing chess (well) than simply knowing the moves.

2.0 Overview.

We start, in Section 2.1, with a discussion about the essential features of procedural programs and procedural programming. This is the kind of programming which is most common (so common that the qualification ‘procedural’ is usually omitted). Other terms are sometimes included (such as object-oriented, or user-centered), but these do not contradict the fact that much of the central code is still procedural and hence all that we have to say is still relevant. This is followed with a brief digression on what some people regard as ‘good’ programming.

Then, in Section 2.3, we start to get a little more technical and introduce the notion of flowcharts and their structure. We then introduce the PDL language, which we use for the code of examples throughout the book. Although we give a reasonably complete description of the syntax (which includes some features that might be unfamiliar to the reader), the semantics — the meaning — is described less formally. There is sufficient information to allow the reader to construct and interpret PDL programs but perhaps not enough for someone to construct a compiler.

We move, gradually, to the important question of the required and actual effects of a program by discussing (in Section 2.5) comments and then the related, but more formal, notion of assertions — executable comments! Assertions can be used to demonstrate program correctness but are expensive to evaluate. (Essentially each assertion is a small program, written in another kind of language, and we shall return to this idea in Chapter 3.) In Section 2.6, we go ‘all technical’ and introduce the concept of program verification. Here, in principle, we can take a program together with a formal specification of what it is supposed to compute, and justify, mathematically, that they fit together in the required way; namely, that for each valid data input value, the result produced by the program (together with the data) satisfies the specification. We only consider the verification of PDL ‘structured programs’ and give the basic verification rules; we do not develop the theory or illustrate its use. The main reason for this is that using assertions, or verification (or indeed testing²) is a retrospective process; we have to build a program and then try to demonstrate that it is ‘right’. And if it is *not* then we are stuck.

We wish to adopt an alternative approach and move in the opposite direction; we want to write a program so that it is ‘correct by construction’. This is the central theme of our book, and hence we go into it in some considerable detail. As a way of ‘breaking the ice’, we discuss the idea briefly in Section 2.7.

² As we shall say many times, for programs which are used with many different inputs, testing is simply not a serious option. Unless you can apply exhaustive test methods (which take more time than any of us have in a single lifetime), testing can only demonstrate program failure, not correctness.

2.1 Procedural Programming

In procedural programming, the programmer indicates, explicitly, how the execution of the program should proceed; or how a collection of procedures should be sequenced, controlled. Most programmers (certainly most ‘occasional’ programmers and indeed many professional programmers) equate all ‘programs’ with ‘procedural programs’ — they are simply not aware of other programming paradigms.

So, we are not teaching programming, in the sense of coding. We assume that the reader is familiar with some (possibly object-oriented) procedural programming language. Here we merely make observations on ‘style’, typical features, and later (cursory remarks on) their verification.

Procedural programs are thought of as being easy to write because they only do simple things — essentially assignments. But, it is the ways in which these simple actions can be combined which cause difficulties in keeping track of the overall effect caused by these actions.

Characteristically, procedural programs work by causing changes in ‘state’. The state (of a computation at a particular place in the program, when execution has reached this point, at this time) can be thought of as the current set of values associated with the ‘variables’ to which the program has access at that point.

The name x might refer to different locations (and possibly different types of values) in different parts of the program. Moreover, a given x might be required to change in value as the execution proceeds, and the value of x at a given place in the program will often be different when the execution passes through the same place on a subsequent occasion.

A lot *can* change, and indeed we require that some values *do* change, but keeping track of these changes, and reasoning about them, can be complicated. It is specifically to avoid (or at least to defer) the complexity of state changes that we shall adopt the LFP³ scheme for program derivation in Chapter 3. It is also for this reason that we do not include much detail when describing the (direct) verification and construction of procedural programs.

In any procedural language (such as our PDL — Program Design Language — in Section 2.4), there are three basic kinds of components: declarations, expressions and commands. Declarations introduce new (local) names which will be associated with entities within the surrounding block. These entities are typically locations — ‘variables’ in colloquial, but erroneous, terminology — in which we may store

³ Logic, Functions and Procedures, or Logical, Functional and Procedural.

values of a given type. This type is given in the declaration. We can also declare functions (see ‘expressions’ below — a function call, here, is little more than a parameterised expression). A block, in many languages delimited by ‘begin ... end’, is really a compound command and consists of an optional list of declarations followed by a list of commands. Declarations perform no computation; they merely introduce entities which can be used in subsequent expressions and commands.

Expressions can be evaluated to compute values, and change nothing (in our language we do not allow ‘unexpected’ side effects). They are well-formed mathematical expressions and as such follow certain syntactic and semantic rules — the rules are those given as type specifications in Chapter 1. The evaluation of an expression gives a value which can be used in a command. Expressions cannot occur in isolation⁴. (In PDL, expressions can be conditional, and again the forms introduced in Chapter 1 are used.) Function calls yield values, and hence may such calls may occur within expressions.

In general terms, commands are language components which have the facility to change the values stored in named locations⁵, to import a value from the input stream into a named location, export the value of an expression (to the output stream), or influence the flow of program control.

Though not regarded as essential, we also admit the existence of labels (which label, identify, commands). Indeed, although very messy, we could regard all commands as a labelled ‘proper’ command followed by a conditional or unconditional “goto” statement⁶ which then directs the program to the labelled command to be executed next. Fortunately we can do much better than this — but flow of control is important and should not be taken for granted or disregarded.

Alternatively, a procedural program could be regarded as a description of journeys which can be taken around a (control) flowchart. The program text is simply a description of that flowchart. The route of this journey is sometimes referred to as the locus of control.

The fundamental building blocks of flowcharts are: one start point, and (for convenience) one stop point, (optionally labelled) 1-in, 1-out rectangles representing computational commands⁷, and diamond shapes with one in path and two out paths (labelled True and False), the diamond containing a Boolean

⁴ There is one exception in the shorthand form: the expression which delivers the result of a function evaluation.

⁵ They may not *always* cause a change. For example, when executing the sequence ‘ $x \leftarrow 1$; $x \leftarrow 1$ ’, the second statement never causes a state change; it replaces the ‘value of x ’ with ‘the value of x ’, which is 1.

⁶ ‘Statement’ is simply another name for ‘command’.

⁷ ‘Leaning’ parallelograms are also often used to indicate input/output commands. We shall ignore them since we presume that all input precedes proper processing and output comes last.

expression. These facilitate the switching of control flow between two alternatives. Graphically, these are typically as depicted in Figure 2.1.

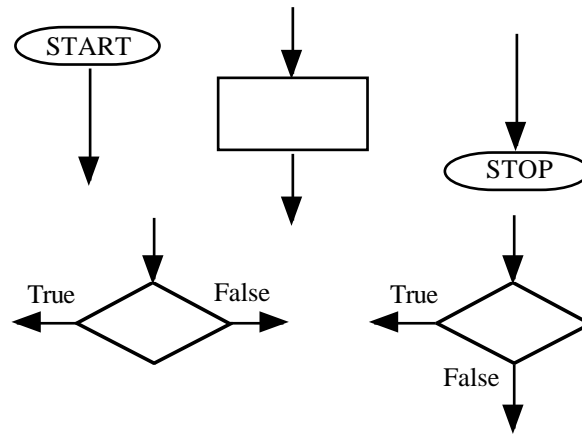


Figure 2.1

More problematic is how components are put together. We take one START node, one STOP node and as many computation rectangles and test diamonds as required. The rectangles can be inscribed with a description of a computational step (a command) and the diamonds with a Boolean expression. So far, so good. Now join up the arrows, but preserve the number of inward and outward arrows shown in the illustration. The only other ways of linking arrows is by means of joining points, at which several (but usually two) incoming arrows link up with one outgoing arrow, such as in Figure 2.2.

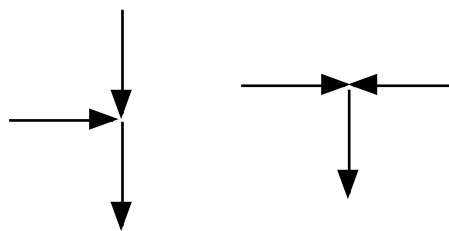


Figure 2.2

Absolutely *any* configuration is possible. We certainly are not going to give an example. Try it for yourself and see exactly why we can end up with what have been called ‘spaghetti’ programs.

Execution of a flowchart program is as follows. Control starts at the START node and follows the arrows until we reach STOP. Upon encountering a rectangle, we carry out the enclosed command (usually resulting in a state change) and follow the ‘out’ arrow. On reaching a test, we evaluate the enclosed Boolean expression and follow either the True arrow or the False arrow depending on the result of that evaluation. Yes, it sounds easy — after all, computers can only perform simple tasks (but very many, very quickly).

A procedural program (or, more properly, the text of such a program) is simply a representation of one such flowchart.

By virtue of having only a single way in and a single way out, the entire flowchart can be regarded as another higher-level ‘rectangle’. Adding extra entry points and extra (abnormal?) exit points will usually complicate any description of what the program actually does. It is for the same reason, at a lower level, that the computational rectangles are restricted to having unique entry and exit points. We will say more about program structure in Section 2.3.

Other features commonly found in procedural language include recursive expressions (achieved by means of recursive functions) and blocks which can be named (and parameterised) to give procedures which may or may not be recursive. There are also array types, but these are not central to *our* exposition and will only be used in certain (small) sections.

2.2 ‘Good’ Programming

Today, most programming is done in so-called high-level languages. One characteristic of such languages is that they allow the programmer to devise and use ‘long’ names. [But it may not be easy to invent enough names which are meaningful, distinct, and not too long. Some language implementations have in the past permitted the use of very long names but then only took notice of the first ‘*n*’ characters — not so clever!]. They also have English⁸ keywords that are fixed in the language and are supposed to convey the appropriate semantic meaning. The upshot of these possibilities is that the program text can be made more ‘readable’, less cryptic. In the reverse direction, there is the desire to make the written form of the program reasonably compact. This is the same argument as applied to specifications. They should not be needlessly verbose: otherwise, you cannot see what is there because there is too much ‘noise’.

⁸ We expect that the reader will encounter these situations, but, for example, French would be perfectly acceptable for a French reader or writer of programs.

So, for instance, in place of

“evaluate E and store the result in L ”

we may write

“put E in L ”

or

“ $E \rightarrow L$ ” — yes there are languages where assignment statements are strictly from left to right.

rather than

“ $L := E$ ” — the ‘:=’ combination being an accident of (the lack of) technology.

We shall use

“ $L \leftarrow E$ ”. — location L is given the value of expression E .

Within the body of this text⁹, ‘=’ means ‘equals’ — the predicate, the test — and nothing else. It certainly has nothing to do with the assignment command *per se*.

Making the program text more readable reduces the need for comments. Certainly comments that merely describe adjacent commands in a program are a waste of time and effort (but see Section 2.5).

[Similar principles can be used when using systems-programming languages. These look like high-level languages but also have lower level operations.]

2.3 Structuring and (Control) Flowcharts

Any 1-in, 1-out block can be thought of as a state-to-state assignment. Any part of a flowchart having this 1-in, 1-out property can be regarded as a logical sub-program and can, for documentation purposes and to aid reasoning, be drawn separately as a stand-alone flowchart and referenced — in the appropriate position — in the ‘main’ flowchart by means of a labelled (named) rectangle as shown in Figure 2.3.

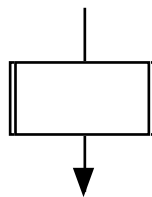


Figure 2.3

⁹ Except for brief segments where we try to draw similarities (or familiarity) with more traditional manipulations.

In so-called structured programs, only certain ways of combining components into higher-level 1-in, 1-out ‘structures’ are used. These assist in breaking down the overall problem/solution into fewer pieces, the computational elements of which are again 1-in, 1-out. Moreover, the meanings of the components are ‘easily’ related logically to the combination. These are not independent, nor are they exhaustive. The major ones used are shown, in flowchart form, in Figure 2.4. At the top left we have the conditional construct “if ... then ... else ... fi”. The test is performed and either the left or right fork taken, depending on whether the result is True or False. In the top right of Figure 2.4, we have the often forgotten sequence construct. This is represented by the “;” operator, the “go - on” symbol. Here we simply execute the upper command first and then the lower one. Notice that we use the semi-colon as a *separator*, not a terminator. In the lower two diagrams in Figure 2.4, we have two kinds of loop configurations. On the left is a “while .. do ... od” loop in which the test is performed and, if - and *only* if - the result is True, the body of the loop is performed and we return (hopefully with a changed state) to the beginning of the loop and execute the construct again. The loop exits when the test evaluation yields False. [Remember that the evaluation of a test — and indeed any expression — causes no change in state.] On the right is a “repeat ... until...” construct. Here we start by executing the enclosed command and then we evaluate the test. We loop back to the start, with the current state, if the result from the test is False.

The ‘while’ loop is also called a pre-check loop because the test comes before the body; the body of the loop may not be executed at all. Executed zero times. On the other hand, the ‘repeat’ loop — a post-check loop because the test comes after the body — always executes the body at least once.

Notice that by default flow is from top to bottom and left to right, and therefore many arrowheads can be omitted. But put them in if you feel that any confusion might arise.

Using only these ways of combining commands and tests within a program means that there is no technical need for *gotos*, but labels can still be used to assist in documentation (comments etc.).

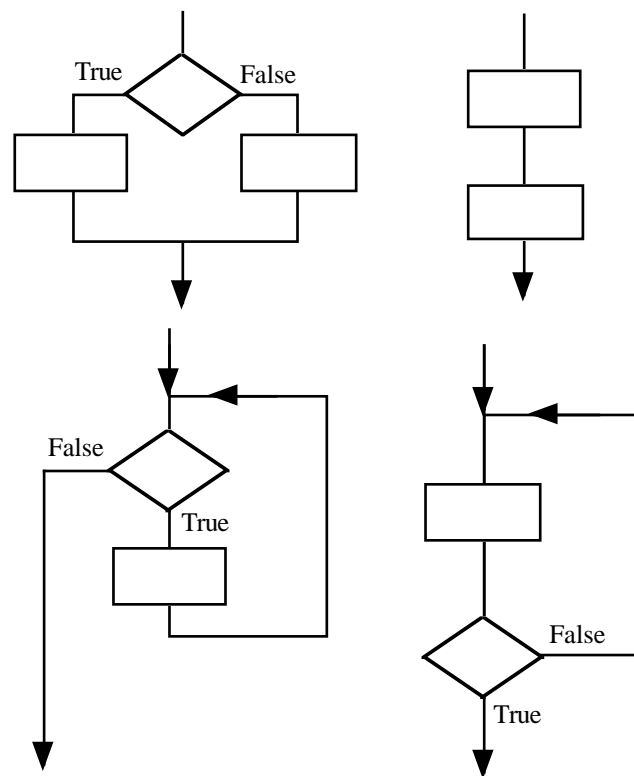


Figure 2.4

2.4 PDL Overview

The language we use to express implementation designs is PDL, which stands for Program Design Language. It is like many actual languages but is not intended to be exactly the same as *any* of them. Indeed, you may also regard it as a pseudo-code which then needs to be translated into a locally available (supported and implemented) language.

PDL has deterministic control (it does not have, as is commonly seen in similar texts, guarded commands, which are loop structures with the possibility of different execution sequences from a given initial state).

We shall follow the common practice of using the notation $A[i]$ to represent the i^{th} location of the array A , with A subscripted by i . This is the location, offset by an amount associated¹⁰ with i , relative to the location of (the array base address) A . Some languages make this more explicit, and perhaps more confusing but technically correct, by using the notation for a function call; i.e., $A.i$ or $A(i)$.

We do not give a fully formal description of PDL. We do not intend that it necessarily be implemented, and hence such a definition is not required¹¹. Instead, we describe its syntactic structures in a variant of BNF¹² which avoids the need for more new symbols.

Our style of presenting the syntax of PDL uses capitals to name syntactic classes. The occurrence of such a name as part of a definition must be expanded using one of the possible alternatives in *its* definition. Symbols used as part of the language (and which therefore might appear within the text of a program) are delimited by double quotation marks. The lowest level of detail is omitted. Explanations of related semantic notions are given in *italic* text.

The syntax is recursive, and hence the syntactic classes have no natural ordering. We use an ordering that approximates to ‘bottom up’; that is, we describe the smaller units first and then assemble them into larger units and ultimately an entire program. Remember, this description is precisely that — a description. In an attempt to keep the description readable and hence useful, some details have been omitted.

¹⁰ The precise details depend on the layout of the computer store and the type of data held in the array.

¹¹ We shun the temptation to include huge amounts of material quite tangential to the main theme of our text. It is certainly true that programming languages do need formal specifications, which may be thought of as specifications of compilers, and compilers *are* programs. But they are very special programs and best specified using techniques other than the general ones discussed elsewhere in this book.

¹² Backus Naur (or Backus Normal) Form, with which we assume the reader has at least a passing acquaintance. The more traditional form of BNF is used in the Appendix.

IDENT	an identifier
is	<i>a meaningful(?) name consisting of a string of letters, digits and the symbol “_”, starting with a letter</i>
IDENTS	a list of IDENTs
is	
	IDENT “,” IDENTs
or	
	IDENT
EXP	an expression
is	<i>any well-formed expression using the types defined in Chapter 1</i>
or	
	IDENT “(”EXPS “)” a function call
or	
	“if” EXP “then” EXP “else” EXP “fi” a conditional expression
	<i>the first expression is of type Boolean; if it yields True, then the middle expression delivers the result, if False then the third expression gives the result. (Here and elsewhere, there is no specific connection between the different occurrences of the same class name, such as ‘EXP’.)</i>
or	
	EXP “where:” EXP <i>the second EXP yields a Boolean value — see later in this section.</i>
or	
	“if” EXP “then” EXP
	“else_if” EXP “then” EXP “else” EXP “fi”
	<i>one contracted form of nested conditional expressions</i>
or	
	CAST “:” EXP
	<i>The result from the EXP is ‘coerced’ into the type indicated by the CAST. Refer to Section 1.4.10</i>
CAST	
is	
	TYPE

TYPE	type
	<i>here undefined but includes all the type indicators given in Chapter 1</i>
TYPES	
is	TYPE “,” TYPES
or	TYPE
EXPS	an expression list
is	EXP “,” EXPS
	<i>the “,” is a list separator</i>
or	EXP
DEC	a declaration ¹³
is	“var” IDENTS “:” TYPE
	a ‘variable’ ¹⁴ declaration
or	IDENT “(” IDENTS “)” “≡” EXP
	a function declaration
or	IDENT “(” IDENTS “)” “≡” “(” EXP “)”
	<i>alternative form</i>
or	IDENT “(” IDENTS “)” “≡” BLOCK
	a function / procedure declaration
	<i>a form of function declaration in which each flow through the BLOCK terminates in an EXP, implicitly or explicitly assigned to the ‘result’. Without a ‘result’, this is a procedure</i>
or	IDENT “≡” BLOCK ¹⁵
	a procedure declaration <i>a procedure with no parameters</i>
or	“var” IDENT “[” TYPES “]” “:” TYPE
	an array declaration <i>the TYPES are index types and must be subranges. TYPE is the type of the data held in the array.</i>

¹³ Other contracted forms are also allowed by way of ‘syntactic sugar’.

¹⁴ These are names of constant locations, but the contents can be changed. Hence they are often described as ‘variables’.

¹⁵ Or a statement other than a ‘goto’ statement.

DECS	declarations
is	
DEC “;” DECS	
or	
DEC	
STMT	a statement
is	
IDENT “:” STMT	<i>the identifier is a label</i>
or	
BLOCK	
or	
STMT “ ” STMT	parallel execution
or	
“skip”	<i>the skip command, change nothing</i>
or	
IDENT “←” EXP	assignment, <i>evaluate the EXP and pass the value to the location associated with IDENT</i>
or	
IDENT “[” EXPS “]” “←” EXP	assignment to an array element
or	
< IDENTS > “←” < EXPS >	parallel assignment. <i>The lists are of equal length and of corresponding types. All the EXPS are evaluated. Then the values are deposited in the locations named by the corresponding IDENTs.</i>
or	
“if” EXP “then” STMT “else” STMT “fi”	conditional statement
or	
“if” EXP “then” STMT “fi”	<i>contracted form, presumes “else skip”</i>
or	
“if” EXP “then” STMT “else_if” EXP “then” STMT “else” STMT “fi”	<i>one contracted version of one form of the nested conditional statement.</i>
or	
“while” EXP “do” STMT “od”	‘while’ loop ¹⁶ , EXP is Boolean
or	
“repeat” STMT “until” EXP	‘repeat’ loop, EXP is Boolean
or	
“goto” IDENT	IDENT is a label in the same BLOCK

¹⁶ We could also have ‘for’ commands as syntactic sugar for certain ‘while’ loops.

or	[“result ←”] EXP	<i>delivers the result of a function evaluation</i>
		[“result ←”] <i>denotes optionality</i>
or	IDENT “(” EXPS “)”	procedure call
or	IDENT	procedure call <i>with no parameters</i>
BLOCK ¹⁷		
is	“begin” DECS “;” STMTS “end”	
or	“begin” STMTS “end”	
STMTS		
is	STMT “;” STMTS	
or	STMT	
PROG		a program
is	BLOCK	

We also have comments and assertions. These may be placed between statements.

COMMENT

*any sequence of characters (other than quotation marks)
delimited by “ and ”.*

ASSERTION

	“\$” EXP “\$”	
or		<i>where EXP is of type Boolean</i>
	“\$” “assert” EXP “\$”	

These syntax rules can be used either in the generation of programs or the analysis of strings (as part of the process of determining whether you have a valid program). They can also be regarded as rewrite rules, albeit in a context different from that found throughout the rest of this book.

2.4.1 “Let” and “Where”. Conventionally, and for very good practical reasons, most procedural languages require that all named entities, functions etc., be declared before use. This is the ‘let’ style of presentation, even if the word ‘let’ is not used

¹⁷ And “(...)” can be used in place of “begin ... end” as delimiters.

explicitly. This is available in PDL, but for use in intermediate forms we also have the ‘where’ style, which allows the use of incomplete (or general) expressions that are then completed (or restricted) by quoting additional Boolean information, definitions or specifications.

For example:

$$x + f(y) \quad \text{where: } x = a + b \quad \text{and} \quad f \triangleq (\lambda x:X)(x + 3)$$

which means $a + b + y + 3$

2.4.2 Scope and Parameters. ‘Variables’ referenced within a block but not declared within that block are those declared within the smallest surrounding block. Within function and procedure calls, that block is a block surrounding the call rather than the declaration. Notice also that parameters are passed by value and are therefore constants, which cannot be changed within a function or procedure.

2.5 Comments and Assertions

Programs written in modern programming languages should be very nearly self-documenting and hence there is less need for comments. Nevertheless, comments, if up-to-date and related to the code, can be very useful. However, they *may* not have anything to do with the code. They may be out of date, or just plain wrong.

Comments are sometimes written at the same time as the code and hence are likely to be related to the code (in a meaningful and relevant way). When code is modified, since the comments are ignored by the compiler and need not be changed in order to make the code work, it is probable that the comments are *not* modified. Hence the comments, even if helpful once, may not always be so. The fewer the comments, the easier it is to ensure that they *are* up-to-date.

Of course, they are only there for the benefit of the human reader; they are totally ignored by the compiler.

Instead of comments, we could use **assertions** — executable comments. If the evaluation of an assertion (the body of an assertion, is a Boolean expression) gives the value False, then the program is aborted (halted) preferably with some indication of where the failure took place. If the assertion yields True, indicating that some property you thought ought to hold (at that point in the program) was actually true, and execution of the program continues, albeit at some computational cost. Using the pre-condition¹⁸ as an assertion immediately after the input phase reflects the assumptions made about acceptable input values¹⁹.

¹⁸ Part of the specification of the function which the program is supposed to compute.

¹⁹ Together with the data type constraints imposed by the read command is as far as we can go to cope with robustness.

Similarly, if we keep a copy of the input values (and that means making no subsequent changes to the relevant ‘variables’), then, just before the output phase, we can include the post-condition as another assertion. This would then check that the answer which we had computed was in fact an acceptable answer for the given input.

This sounds fine, and it is much easier than doing lots of technical work with specifications and programs, but evaluating assertions can be very expensive and take huge amounts of time and other (machine) resources. Moreover, placing assertions between every pair of commands and including all information that the programmer thinks *may* be of relevance would make the text completely unreadable. Of course, the assertions which are True immediately before a certain command are logically related to those that are True immediately afterwards, and hence not everything need be repeated.

To see how these assertions are connected mathematically, we consider the three basic computational components of a flowchart. These are depicted (with assertions) in Figure 2.5.

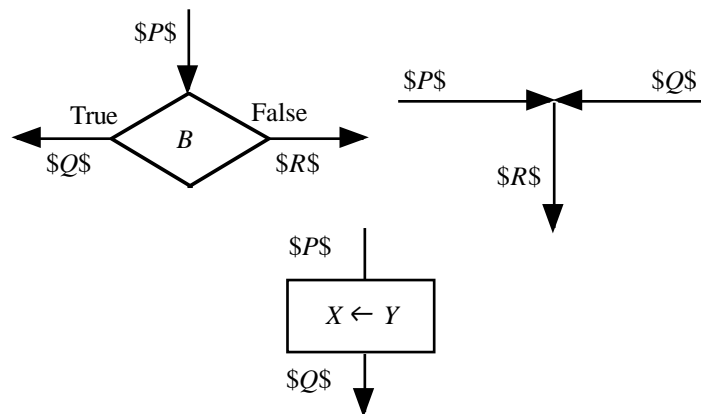


Figure 2.5

The three kinds of component are tests, joins and assignments. The first two are often forgotten because nothing seems to be happening, but they are crucially important. In Figure 2.5 the assertions (annotations?) are Boolean expressions but we shall also treat P , Q and R as being synonymous with labels at the positions indicated on the segments of the flowchart.

In the test flowchart, any facts within P are clearly also within Q and R since we have done nothing to change that information(?). Here we assume that the test B was useful within the program and hence that either outcome of the test was

possible. If this is not so, then we could simply always go from P to Q , or always go from P to R , and remove the test. By similar reasoning, Q and R should be different; otherwise it makes no sense to ‘follow’ them with different program segments. So what is the difference? The difference is that at Q we know that the value of B is True and, if we get to R then the value of B is False. Hence we can write

$$Q \blacklozenge P \wedge B \quad \text{and} \quad R \blacklozenge P \wedge \neg B$$

These are the strongest assertions which can be used in positions Q and R . We may not wish to retain all this information; we may simply not need it, and hence some of the information could be discarded without compromising further reasoning. However, in all cases we *do* know the following:

$$P \wedge B \Rightarrow Q \quad \text{and} \quad P \wedge \neg B \Rightarrow R$$

This helps in appreciating the inter-relationships between the assertions associated with the join flowchart. Here, we have:

$$P \Rightarrow R \quad \text{and} \quad Q \Rightarrow R$$

These implications always hold. The (logically) strongest assertion we can use for R is

$$R \Leftrightarrow P \vee Q$$

— we simply do not know which branch we followed on our way to R .

Now for explicit computation steps, as indicated by rectangles within flowcharts. First notice that we have written an assignment command to represent the computational process between P and Q . Any command²⁰ can be regarded as an assignment, albeit, in general, a parallel multiple assignment (in which n expressions are evaluated and then placed in the corresponding named locations), with the possibility of conditional expressions and/or recursive function calls. So, in restricting consideration to an assignment, we lose nothing.

Obviously, with complex expressions, logical reasoning is more complicated; we shall content ourselves with simple illustrations to introduce the necessary relationships.

The situation depicted in Figure 2.5 can be written as

$$\begin{array}{c} \$ P \$ \\ X \leftarrow Y \\ \$ Q \$ \end{array}$$

²⁰ Other than a ‘goto’ command.

If we need Q to be True, what can we say about P ? That is, what do we need to be True at P in order to guarantee that Q will be True after we have executed the assignment statement? Suppose either that P is initially absent or that it is present but tells us nothing (it provides no information, no reason for halting the program), in which case it is logically, identically, True.

Example 2.1

$$\begin{array}{l} \$ P? \$ \\ z \leftarrow x + 9 \\ \$ z > 0 \$ \end{array}$$

For P we could have $x + 9 > 0$ since the final value of z is equal to $x + 9$ and hence the properties of z must be equivalent to the properties of the expression $x + 9$. In fact, here we can say more:

$$\begin{array}{l} \$ x + 9 > 0 \$ \\ z \leftarrow x + 9 \\ \$ z = x + 9 \wedge z > 0 \$ \end{array}$$

□

Of course, it gets rather involved if values change (and we often need them to) as in

$$\begin{array}{l} \$ x + 9 > 0 \$ \\ x \leftarrow x + 9 \\ \$ x > 0 \$ \end{array}$$

In general, if we have

$$\begin{array}{l} \$ P \$ \\ X \leftarrow Y \\ \$ Q \$ \end{array}$$

then, given Q , we have that

$$P \Rightarrow Q[X \leftarrow Y]$$

That is, given P (is True) we can deduce the expression Q in which X has been replaced by Y . P must logically include $Q[X \leftarrow Y]$.

Of course, the coincidental use of this notation is not accidental. If P initially tells us nothing (so it is identically True) then we can have $P \Leftrightarrow Q[X \leftarrow Y]$ and hence

$$\begin{array}{l} \$ Q[X \leftarrow Y] \$ \\ X \leftarrow Y \\ \$ Q \$ \end{array}$$

Example 2.2 We can extend this idea in a natural way; for instance

$$\begin{aligned} & \$ (x + 9) * 2 - 1 > 0 \$ \\ & x \leftarrow x + 9; \\ & \$ (x * 2) - 1 > 0 \$ \\ & x \leftarrow x * 2 - 1 \\ & \$ x > 0 \$ \end{aligned}$$

Consequently, we can remove the intermediate working and simply write

$$\begin{aligned} & \$ (x + 9) * 2 - 1 > 0 \$ \\ & x \leftarrow x + 9; \\ & x \leftarrow x * 2 - 1 \\ & \$ x > 0 \$ \end{aligned}$$

□

Working backwards from the final required predicate, the post-condition, we can²¹ — in principle — move through a finite sequence of statements (each equivalent to an assignment statement) with the aim of obtaining an initial assertion (which logically follows, using ‘ \Rightarrow ’, from the post-condition).

But there *are* problems. Very few procedural programs are totally sequential but involve some repetition, such as loops, a possibility which we consider below.

More serious is the possibility of including an inappropriate assignment which effectively destroys information, making it impossible to complete the calculation. [If a program is wrong, there is no way that assertions can be inserted which prove it correct!] Moreover, an assertion is applicable at one specific point in the program code (although that point may be revisited on numerous occasions), but a specification, in particular the post-condition part of a specification, refers to the input *and* output of a (sub-) calculation. Hence, to use assertions to enforce/affirm/check adherence to a specification requires that we keep a copy of the original input values to the corresponding segment of code.

We need to address the possibility of state changes (which are fundamental to the philosophy of procedural languages) and slip from a classical function, f , to a (computational) operation²², F . To illustrate this, consider the pair of assignment commands

$$y \leftarrow f(x)$$

and

$$\langle x', y' \rangle \leftarrow F(\langle x, y \rangle)$$

²¹ There is an extra technicality involved here, which means that we may need to refer to an earlier ‘state’.

²² These are akin to commands in programming languages, not traditional mathematical operations, which are merely alternative syntactic forms for common functions.

For these to represent the same action, we need to interpret the second as

$$y' \leftarrow f(x)$$

and

$$x' \leftarrow x$$

assuming that x and y are different.

or, more generally,

$$\sigma' \leftarrow F(\sigma)$$

where σ (lower case Greek letter sigma) denotes the state, which can be thought of as the n -tuple of (allowed/accessible) locations, and σ' (and, as required, σ_1 , σ_2 etc.) represents the same n -tuple at another point in the program. Remember that we are usually dealing with changes in value, and hence σ may represent different values even within the *same* command.

So, instead of

$$\sigma' \leftarrow F(\sigma)$$

we would usually write (in PDL)

$$\sigma \leftarrow F(\sigma)$$

which represents the context shown in Figure 2.6.

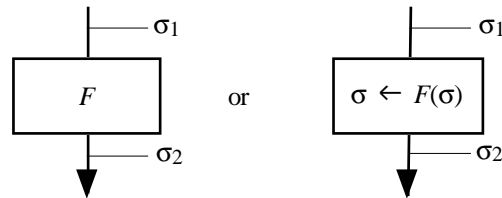


Figure 2.6

Here, the state σ_2 can be calculated from σ_1 using

$$\sigma_2 \blacktriangleleft F(\sigma_1) \quad \text{derived from} \quad \sigma_2 \triangleq F(\sigma_1)$$

Having set up the necessary notation, we can explain the basic concept of correctness. Suppose we had program P , which was intended to compute some function F specified using pre- F and post- F . The program (or program segment) P is linked to states σ_1 and σ_2 as in Figure 2.7.

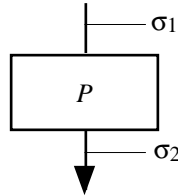


Figure 2.7

For P to correctly compute the specified function, F , we must be able to demonstrate the truth of the implication:

$$\begin{array}{l} \text{pre-}F(\sigma_1) \\ \Rightarrow \\ \text{post-}F(\sigma_1, \sigma_2) \quad \text{where } \sigma_2 = P(\sigma_1) \end{array}$$

Therefore, in terms of assertions, we need

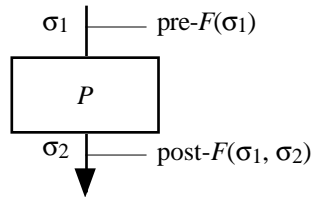


Figure 2.8

To implement such (post-) assertions we need to allocate extra storage to save the value of σ_1 for future use. Clearly, the problem of extra storage and its management can get out of hand, particularly when we try to decompose P into smaller parts more closely resembling primitive (rather than multiple) assignments. However, this is more manageable when we insist on the discipline of structured programming, as in the next section.

So, we can relate assertions within programs to the specifications of the functions/operations they are intended to compute. We have seen a simple example of how an intermediate assertion can be removed once it has been used to establish the logical link between earlier and later ones.

Verification (the next section) may thus be seen as removing the need for assertions, and hence because, for valid input, the output is correct, the post-condition (which is usually very expensive to calculate/evaluate) does not need to

be included explicitly. All predicates used in the verification proof obligations (in Section 2.6) may be ‘implemented’ as assertions (and, by some, regarded as more practical than theoretical)²³ but at considerable cost and risk.

We make one final observation on assertions. Very many true but irrelevant pieces of information can be included within acceptable assertions. These add nothing to our understanding of the computational processes and can be very distracting and misleading. They also make the evaluation of assertions more complex and time consuming to compute. Therefore, to be informative to the reader, assertions should be as concise as possible, whilst providing adequate information to enable the logical links with the specifications to be established.

2.6 Verification of Procedural Programs

Verification of a program is the process of justifying that it is correct with respect to (wrt) its specification.

This means that

“for every valid input, the program runs to completion
and delivers an acceptable (correct) result”.

Using the formal notation introduced in Chapter 1, we can express this requirement as a theorem, the correctness theorem:

f is a correct implementation (of its specification) if

$$(\forall x:X)(\text{pre-}f(x) \Rightarrow \text{post-}f(x, f(x)))$$

where

$$\begin{array}{ll} \text{pre-}f(x) \triangleq \dots & \text{the test for valid input } x \\ \text{post-}f(x,y) \triangleq \dots & \text{the test that } y \text{ is a valid output for input } x \end{array}$$

Another way to look at verification is to regard it as justification for the removal of *all*²⁴ assertions (because, for valid input, the output is correct, so the usually very expensive post-condition does not need to be checked/evaluated)

²³ This is, of course, not true. The only difference is that failure in verification prevents a program being ‘delivered’ and used; failure in the evaluation of a run-time assertion would cause the program to fail/abort/halt at run time, and this may be catastrophic.

²⁴ With the possible exception of that associated with the pre-condition to address the problem of the robustness of a program.

Reasoning (constructing useful assertions and hence eventually including the pre- and post-conditions) with an arbitrary flowchart program can be very hard. Fortunately, Structured Programming comes to the rescue.

Identifying states by σ_1 , etc., as in Section 2.5, we give the required logical relationships between various points (positions, ‘line segments’) in the flowcharts of non-atomic structured program components.

Note that here we are not concerned with how a program is created but presume that a program has (somehow) been written and the task is to verify that it satisfies a given specification — retrospectively!

The structure of a ‘structured program’ is defined so that any 1-in, 1-out segment is either a single, atomic, command or can be decomposed into smaller components using one of the following forms²⁵, which we have already met:

- (1) Sequencing $P; Q$
- (2) Alternation (choice) if b then P
 else Q
 fi
- (3) Iteration while b
 do P od

For each of these we attach ‘state markers’ and then quote the so-called ‘proof obligations’ which when discharged (i.e., proven to be True) guarantee correctness of the overall combination, provided that its proper sub-components are correct (relative to their own specifications).

2.6.1 Sequencing

We refer to the flowchart in Figure 2.9.

P is a correct implementation of (the specification of) F_1

$$\text{if } (\forall \sigma_1) \quad \text{pre-}F_1(\sigma_1) \Rightarrow \text{post-}F_1(\sigma_1, \sigma_2) \\ \text{where: } \sigma_2 = P(\sigma_1)$$

So, taking the $(\forall \sigma_1)$ as implicit,

$$\text{pre-}F_1(\sigma_1) \Rightarrow \text{post-}F_1(\sigma_1, P(\sigma_1))$$

²⁵ Others are possible, but they can be derived from the three given here; they are syntactic sugar.

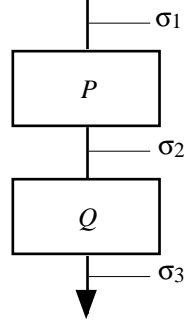


Figure 2.9

Similarly, Q is correct wrt F_2 if

$$\text{pre-}F_2(\sigma_2) \Rightarrow \text{post-}F_2(\sigma_2, Q(\sigma_2))$$

We require that the sequential combination $P; Q$ be correct wrt F ,

i.e.,
$$\text{pre-}F(\sigma_1) \Rightarrow \text{post-}F(\sigma_1, Q \circ P(\sigma_1)) \quad \text{see}^{26}$$

Instead of trying to tackle this directly (which could be quite difficult since we may not know explicitly the details of P and Q , but merely that they are correct implementations of F_1 and F_2 respectively), we consider the inter-relationships between states and take our lead from the flowchart.

The logical links (rules) are:

$$D_1: \quad \text{pre-}F(\sigma_1) \Rightarrow \text{pre-}F_1(\sigma_1)$$

$$D_2: \quad \text{pre-}F_1(\sigma_1) \wedge \text{post-}F_1(\sigma_1, \sigma_2) \Rightarrow \text{pre-}F_2(\sigma_2)$$

$$R_1: \quad \text{pre-}F(\sigma_1) \wedge \text{post-}F_1(\sigma_1, \sigma_2) \wedge \text{post-}F_2(\sigma_2, \sigma_3) \\ \Rightarrow \text{post-}F(\sigma_1, \sigma_3)$$

Notice that these rules do not mention P and Q explicitly; we only need to know that they satisfy their respective specifications. These rules together justify that $P; Q$ satisfies F .

D_1 and D_2 are domain (or data) rules. D_1 says that we can start to execute P if we can start to execute $P; Q$.

²⁶ Recall the change in order, which is necessary so as to fit with the ‘function of a function’ notation.

D_2 says that, after executing P , the state reached (σ_2) is suitable for input to Q .

R_2 is a range (or result) rule. This says that with a suitable initial state (σ_1) we can first derive σ_2 and then σ_3 , which is a correct output for $P;Q$ relative to the initial state σ_1 and (the specification of) F .

Already²⁷ we know something about how σ_2 is related to σ_3 (in such a way that Q always works correctly), but working forward is more difficult.

□

2.6.2 Alternation

For this, see Figure 2.10.

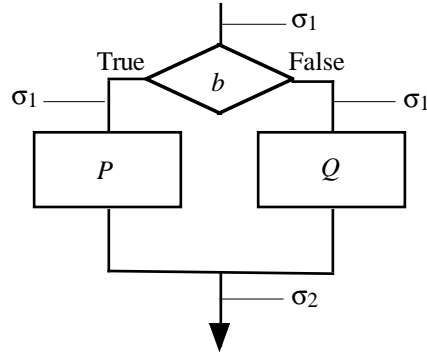


Figure 2.10

With the previous assumptions about P and Q , we now consider the requirements for the construction:

if b then P else Q fi satisfies the specification G .

Using the same kind of diagrammatic/logical reasoning, we have

$$D_1: \quad \text{pre-}G(\sigma_1) \wedge b(\sigma_1) \Rightarrow \text{pre-}F_1(\sigma_1)$$

$$D_2: \quad \text{pre-}G(\sigma_1) \wedge \neg b(\sigma_1) \Rightarrow \text{pre-}F_2(\sigma_1)$$

$$R_1: \quad \text{pre-}G(\sigma_1) \wedge b(\sigma_1) \wedge \text{post-}F_1(\sigma_1, \sigma_2) \Rightarrow \text{post-}G(\sigma_1, \sigma_2)$$

$$R_2: \quad \text{pre-}G(\sigma_1) \wedge \neg b(\sigma_1) \wedge \text{post-}F_2(\sigma_1, \sigma_2) \Rightarrow \text{post-}G(\sigma_1, \sigma_2)$$

²⁷ See Section 2.5.

Here the D rules check that σ_1 , the initial state, is a valid input for the combination, together with b being either True or False ensure that P or Q can be executed correctly wrt their own specifications. The R rules then check that the results / changes produced by P and Q , respectively, fit with those required by G . \square

2.6.3 Iteration. This is the most complex construct. For total correctness²⁸ a ‘while’ loop can pass through a finite but unbounded number of states²⁹.

We first deal with partial correctness. By this we mean that, *if* we get to the end, the result is acceptable, but The reasoning here is similar to that used for the previous constructs but with two extra (seemingly unproductive but logically useful) predicates which provide a link between one iteration and the next. The implications can be deduced by the use of recursion together with sequencing and alternation, but we shall go straight to the rules.

We want to use the construct

while b do P od to implement H .

Again we appeal to a flowchart with some named states; see Figure 2.11.

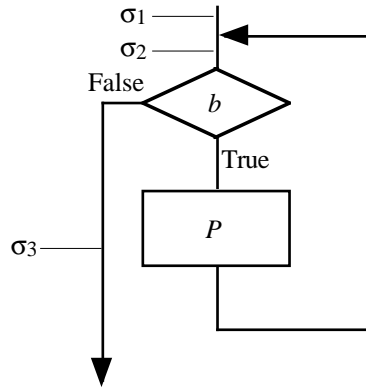


Figure 2.11

Notice that although σ_1 and σ_3 are visited only once, σ_2 may be visited many times. Here is where *invar* is evaluated.

²⁸ Here ‘total’ correctness stresses what we have implicitly taken for granted — that the evaluation of an expression or command terminates, concludes, in a finite number of steps.

²⁹ Each loop will exit legally in n steps, for some $n \in \mathbb{P}$, but we can always construct a case where we will need more than n steps.

Now, ‘out of the hat’ we postulate the predicates:

$invar: \text{state} \rightarrow \mathbb{B}$
 and
 $to\text{-}end: \text{state} \times \text{state} \rightarrow \mathbb{B}.$

The rules which allow us to join the three marked places and the computational/test components are:

- $D_1: \quad pre\text{-}H(\sigma_1) \Rightarrow invar(\sigma_1) \quad \text{enter the loop}$
 $(\quad pre\text{-}H(\sigma_1) \Rightarrow invar(\sigma_2) \quad \text{but no calculation takes place,}$
 $\quad \quad \quad \text{so on the first time through } \sigma_1 = \sigma_2 \quad)$
- $D_2: \quad invar(\sigma_2) \wedge b(\sigma_2) \Rightarrow pre\text{-}F_1(\sigma_2)$
 $\quad \quad \quad \text{remember that } P \text{ satisfies } F_1$
- $D_3: \quad pre\text{-}F_1(\sigma_2) \wedge post\text{-}F_1(\sigma_2, \sigma_2') \Rightarrow invar(\sigma_2')$
 $\quad \quad \quad \text{round again, } \sigma_2' \text{ is } \sigma_2 \text{ on the next iteration.}$
- $R_1: \quad invar(\sigma_2) \wedge \neg b(\sigma_2) \Rightarrow to\text{-}end(\sigma_2, \sigma_3)$
 $\quad \quad \quad \text{no change has taken place here so, at this stage}$
 $\quad \quad \quad \text{of the iteration we could write}$
 $\quad \quad \quad invar(\sigma_2) \wedge \neg b(\sigma_2) \Rightarrow to\text{-}end(\sigma_2, \sigma_2)$
- $R_2: \quad pre\text{-}F_1(\sigma_2) \wedge post\text{-}F_1(\sigma_2, \sigma_2') \wedge to\text{-}end(\sigma_2', \sigma_3) \Rightarrow to\text{-}end(\sigma_2, \sigma_3)$
 $\quad \quad \quad \text{link the final state to a } \sigma_2 \text{ state which is one}$
 $\quad \quad \quad \text{iteration further back, if there is one.}$
- $R_3: \quad pre\text{-}H(\sigma_1) \wedge to\text{-}end(\sigma_1, \sigma_3) \Rightarrow post\text{-}H(\sigma_1, \sigma_3)$
 $\quad \quad \quad \text{link back to initial state, } \sigma_1.$

The justification that this does prove partial correctness of the loop follows from ‘joining up’ segments of any execution path through the flowchart. For each rule, track it through the relevant part of the flowchart (Figure 2.11). We shall not attempt an inductive proof that the rules are suitable; In Chapter Four you will see that these rules are not necessary. They are included here for completeness and because it is via such rules that the science of verification was developed³⁰.

³⁰ The thesis of this book is that we should construct a program to be correct, not that we write a program and then, retrospectively, (try to) prove it to be correct.

$$term: \text{state} \rightarrow \mathbb{P} \quad (\text{at the } \sigma_2 \text{ point in the flowchart})$$
$$\begin{array}{ll}
T_1: & \text{pre-}H(\sigma_1) \Rightarrow \text{term}(\sigma_1) \geq 0 \quad \text{initial value} \\
T_2: & \text{invar}(\sigma_2) \wedge (\text{term}(\sigma_1) > 0) \Rightarrow b(\sigma_2) \\
T_3: & \text{invar}(\sigma_2) \wedge (\text{term}(\sigma_1) = 0) \Rightarrow \neg b(\sigma_2) \\
& \quad \quad \quad (\text{so } (\text{term}(\sigma) = 0) \Leftrightarrow \neg b(\sigma_2), \\
& \quad \quad \quad \text{term is a measure of how close we} \\
& \quad \quad \quad \text{are to the exit condition}) \\
T_4: & \text{pre-}F_1(\sigma_2) \wedge \text{post-}F_1(\sigma_2, \sigma_2') \Rightarrow \text{term}(\sigma_2') < \text{term}(\sigma_2)
\end{array}$$

The analogue of *term* in the recursive situation will be seen as very important (Section 3.1). It cannot be avoided, but once it has been successfully handled, it will guarantee that derived loops terminate automatically.

$$\langle x, y, \dots \rangle \leftarrow C(\langle x, y, \dots \rangle) \quad \text{where } C \text{ denotes a computation,} \\ \text{a function.}$$
$$\sigma \leftarrow C(\sigma)$$

whence
$$\begin{array}{l} \$ P(\sigma) \$ \\ \sigma \leftarrow C(\sigma) \\ \$ P(\sigma) \wedge \sigma' = C(\sigma) \$ \end{array}$$

where σ is the state before the assignment and σ' is the state immediately after its execution.

But is the assignment correct? What is P ?

Essentially, both these questions relate to a specification; let's suppose it is [pre, post]. Using assertions, we could require that

$$\begin{array}{l} \$ \text{assert pre}(\sigma) \$ \\ \sigma \leftarrow C(\sigma) \\ \$ \text{assert post}(\sigma, C(\sigma)) \$ \end{array}$$

If σ represents *all* the data to which the program has access, then this works perfectly well and we can use these relationships (again) but now we write them as proof obligations.

The assignment $\sigma \leftarrow C(\sigma)$

is a correct implementation of the specification J if

$D_1: \quad \text{pre-}J(\sigma) \Rightarrow \text{pre-}C(\sigma)$

$R_1: \quad \text{pre-}J(\sigma) \wedge \text{post-}C(\sigma, \sigma') \Rightarrow \text{post-}J(\sigma, \sigma')$

Often, by design, the calculation, C , can always be evaluated, and hence $\text{pre-}C(\sigma)$ is True and therefore D_1 holds. Moreover, R_1 can often be simplified to

$$\text{pre-}J(\sigma) \Rightarrow \text{post-}J(\sigma, C(\sigma))$$

So, for example, if J is $[x > 0, y' > x + 3]$ and we have the assignment

$$y \leftarrow x + 6 \quad \text{where: } x, y: \mathbb{Z}$$

Namely

$$\langle x, y \rangle \leftarrow \langle x, x + 6 \rangle$$

so

$$\langle x', y' \rangle = \langle x, x + 6 \rangle$$

then, since the validity of $x > 0$ implies that x must have some valid \mathbb{Z} value,

$$D_1 \blacklozenge \text{True}$$

and

$$\begin{aligned} R_1 & \blacklozenge \text{pre-}J(\sigma) \Rightarrow \text{post-}J(\sigma, C(\sigma)) \\ & \blacklozenge \text{pre-}J(\sigma) \Rightarrow \text{post-}J(\langle x, y \rangle, \langle x', y' \rangle) \\ & \blacklozenge x > 0 \Rightarrow y' > x + 3 \\ & \blacklozenge x > 0 \Rightarrow x + 6 > x + 3 \\ & \blacklozenge x > 0 \Rightarrow 6 > 3 \\ & \blacklozenge x > 0 \Rightarrow \text{True} \\ & \blacklozenge \text{True} \end{aligned}$$

So, that is all we want to say about verification. Theorists would say that this is all we need, but finding feasible predicates and then showing that they work — by discharging the proof obligations given above — is very time consuming, even with a tame theorem prover to do all the calculations, hopefully in an error-free way. And what if one of the required obligations is False? There is no routine way of correcting an error even when we have located it. The actual problem may be elsewhere in the program, not where it has caused a problem.

Can we do anything about this? Can we apply the basic theory in a different way? Yes, as we will show in the next section.

2.7 Program Derivation

Just as we usually assume ‘correctness’ of computer hardware when we run a program, we must also presume that the language support systems are also ‘correct’. We are therefore more properly concerned with contextual correctness (or relative correctness), so running the hardware system, the run-time system (etc.), and the user program together, in parallel, works in accordance with the specification. Of course, the only component within this parallel combination on which we have any influence is our own program. If any of the other components is ‘wrong’ in some way, there is nothing that we can do to fix the error. We can only try to ensure that our contribution is error-free.

Given a program, we could attempt to argue its correctness using assertions. But, apart from the assertions which express the pre- and post-conditions (as supplied within the given specification), other, intermediate, *assertions* can be difficult to find. And once we have found them and used them to substantiate adequate logical connections between the program components, since they could be expensive to evaluate, we may remove³¹ them or convert them into comments.

³¹ Adequate (minimal) assertions can be deleted, or transformed (using \blacklozenge) but not otherwise changed.

An alternative approach is to start with the specification and use it to *create* a program design. One way to view this process is to begin with a single unknown command (one which is totally lacking in detail), as in Figure 2.12, and try to ‘fill it in’.

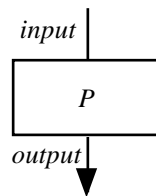


Figure 2.12

A general technique for the construction of program P is to ‘divide and conquer’ the problem and build it from smaller ‘sub-programs’ which solve ‘parts’ of the problem; but in a way that is consistent with the logical requirements within the specification.

Two (of the many) ways in which this could be done are by ‘vertical decomposition’ and by ‘horizontal decomposition’. These can be illustrated as in Figures 2.13 and 2.14

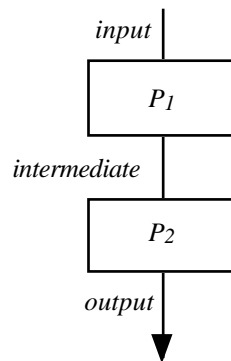


Figure 2.13 Vertical Decomposition

In vertical decomposition, P_1 takes the input and produces an intermediate result (or state), which then acts as input to P_2 . Execution of P_2 then delivers the final result. This can be written as:

$$\text{output} \blacklozenge P_2(P_1(\text{input}))$$

which fits exactly with the concept of a ‘function of a function’. But, again, determining a suitable intermediate *state* is not always easy.

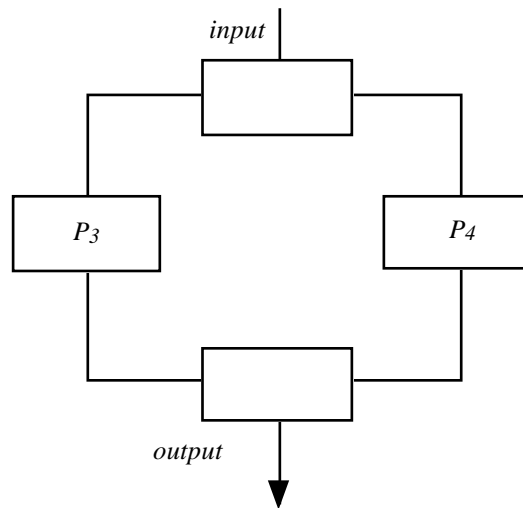


Figure 2.14 Horizontal Decomposition (i)

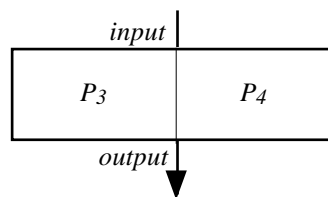


Figure 2.15 Horizontal Decomposition (ii)

In horizontal³² decomposition, the input is divided up (somehow), each part is processed separately, and the two (or more) intermediate results from P_3 and P_4 are combined to give the overall result, from P .

³² Of course, drawing the diagrams so that the flow is left to right would mean that the use of alternative (opposite?) terminology would seem ‘natural’. Be careful. As in many situations, the terminology is *not* ‘standard’.

Much of the material presented in Chapter 3 concerns variants of horizontal (de-) composition. In practice, most program derivations involve a mix of the two approaches, applying vertical decomposition whenever an opportunity (usually a clear case of ‘function of a function’) presents itself.



<http://www.springer.com/978-1-85233-820-6>

Constructing Correct Software

Cooke, D.J.

2005, XXI, 509 p. 100 illus., Softcover

ISBN: 978-1-85233-820-6