Chapter 1

Programs and refinement

1.1 The traditional view

Traditionally, *programs* are collections of detailed instructions to a computer. They are written in a programming *language*, whose form (syntax) and meaning (semantics) are precisely defined. Programs are easy to execute (computers do that), but hard to understand.

The study of methods for making programs is *programming methodology*. We are concerned with methods that take a *specification* of what the computer is to do, and aim to produce a program which will cause the computer to do it. Specifications might be written in English, or in some more mathematical style. They are hard to execute (computers cannot do that in general), but easy to understand — or they should be.

There would be little need for this book if all programs were understandable, or all specifications executable. But alas neither is true — and matters are likely to stay that way.

Specifications *must* be understood, because each is a contract between a programmer and his client. The client relies on the specification for his *use* of the program; the programmer relies on the specification for his *construction* of the program. A complex specification will spawn subspecifications, each defining a component whose construction the programmer may then delegate to his subordinates. That turns the programmer into a client, and his subordinates become programmers; the subspecifications are contracts between him and them. Ultimately, programs are contracts between the lowest-level programmers and the computer.

1.2 A novel view

Our departure from tradition is a small one: we simply banish the distinction between specifications, subspecifications (super-programs?), and programs. To us,

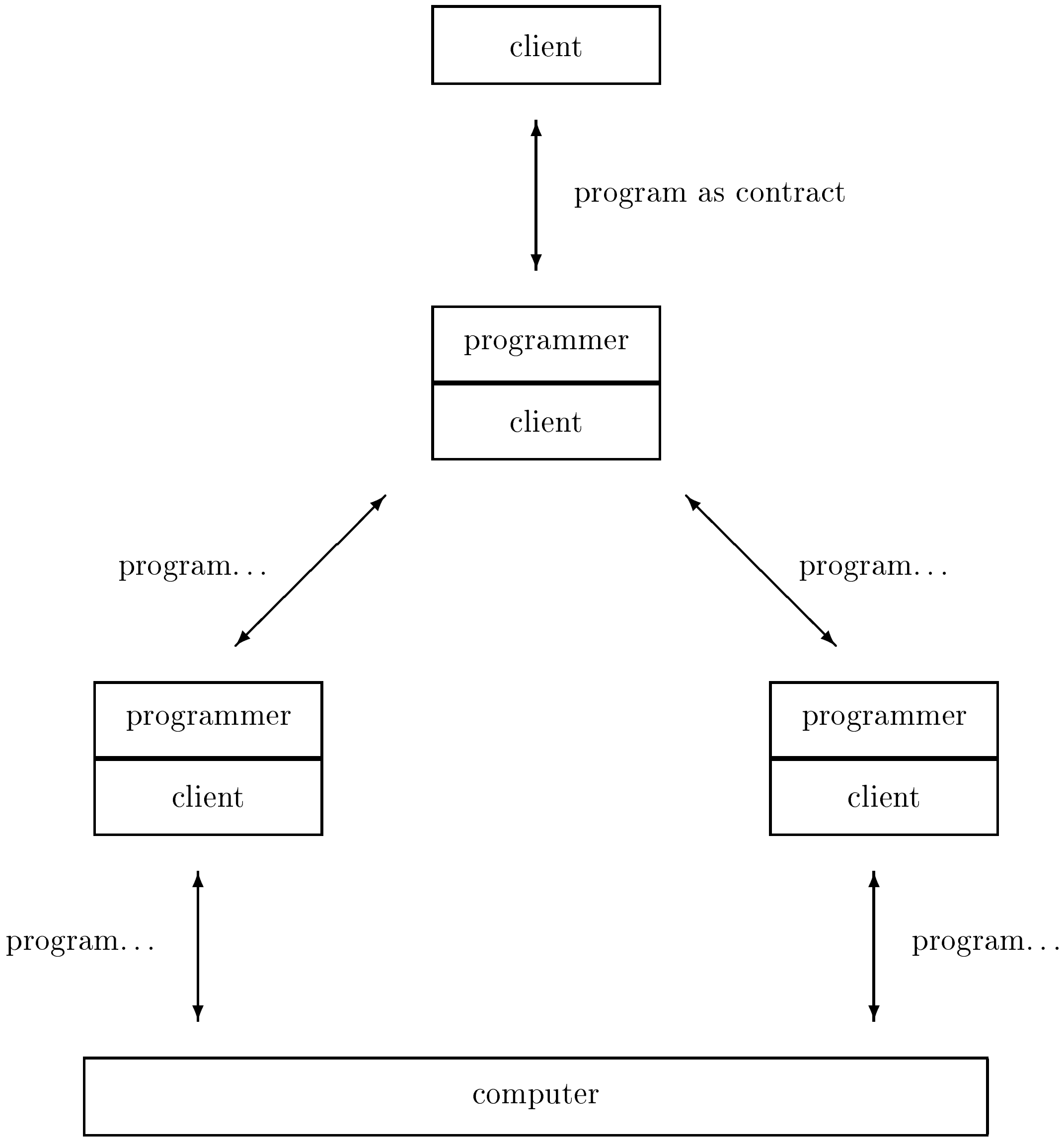

Figure 1.1 A programming hierarchy

they are *all* programs; what we give up is that all programs are directly executable.

What we gain instead is a more uniform approach in which programs play a role at every level. Figure 1.1 illustrates the resulting hierarchy. At the top of the hierarchy we find abstract programs, not necessarily executable. Section 1.4.3 describes some of the constructions they can use.

At the bottom of the hierarchy we find executable programs, which we call *code*. The constructions used there are typical of imperative languages: assignment commands, sequential composition, alternation, iteration, and recursion. Section

cost £10 (paperback)	$\sqsubseteq$	cost £10 (paperback), £20 (hardback)
220V outlet	$\sqsubseteq$	220/110V outlet
safe working load 1000kg	$\sqsubseteq$	safe working load 2000kg
splash-proof	$\sqsubseteq$	water-resistant to 50m
needs at least 4Mb	$\sqsubseteq$	needs at least 2Mb

Figure 1.2 Informal examples of refinement

1.5.2 describes some code.

In the middle, we find programs in which both abstract and executable constructs appear. They contain too much detail for convenient comprehension, but still too much abstraction to be executed. We meet those later, in our case study chapters.

1.3 Programs as contracts: refinement

A program has two roles: it describes what one person wants, and what another person (or computer) must do. With respect to any particular program, we distinguish the *client* and the *programmer*. Remember that a single person can be both: a systems analyst is a programmer with respect to his firm's clients, but a client with respect to his own programming team.

When a contract is made by negotiation between a client and a programmer, each party has primarily his own interests at heart. The client wants the program to do more: to be more accurate, to apply in more situations, to operate more quickly. The programmer wants more freedom in making the program: more leeway in the selection and presentation of results, more information about the situations in which the program is to run, more access to cheap and standard implementation techniques. Their aims are complementary, and the result is always a compromise.

We take the client's point of view in describing the negotiation: if program $prog2$ is better than program $prog1$, *for the client*, we write $prog1 \sqsubseteq prog2$. That relation $\sqsubseteq$, between programs, is called *refinement*: we say that $prog2$ refines $prog1$. In Figure 1.2 are some examples of refinement from more familiar settings. Figure 1.3 illustrates the role of $\sqsubseteq$ in contract negotiation.

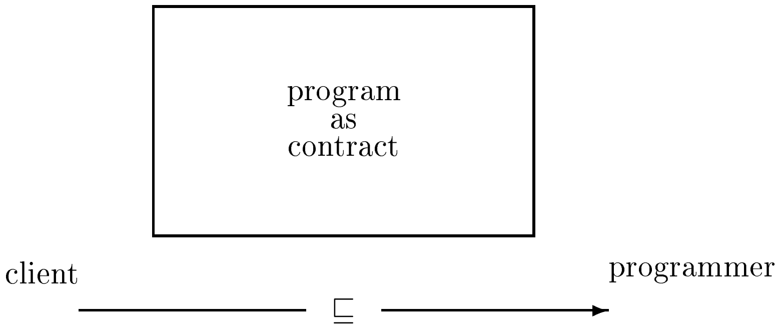

Figure 1.3 Refinement in contract negotiation

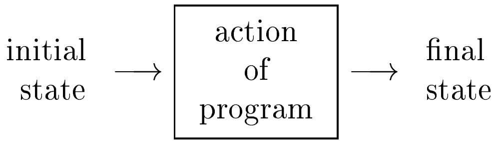

Figure 1.4 The imperative view of programming

1.4 Abstract programs

1.4.1 Initial and final states

Any program takes a computer from an initial state to a final state. That view, illustrated in Figure 1.4, is called *imperative*, and it is appropriate for most computer languages in general use today. We suppose that the data on which the computer is to operate (the input) are placed in the initial state; the results of the computation (the output) are found subsequently, in the final state.

The *state* of a computer is a collection of named values. The names are called *variables*, and the *values* are taken from ordinary mathematical discourse: natural numbers, integers, real numbers, characters, etc. A state *maps* variables to their values.

For the rest of this chapter, let our variables be just x, y, and z; and let their values be real numbers, from the set $\mathbb{R}$. We shall see later that the set $\mathbb{R}$ is the *type* of x, y, and z. Here are two states that differ in the value of y only:

$$\mathit{state0}: \quad \begin{array}{|c|r|} \hline x & 2 \\ y & 17 \\ z & 3 \\ \hline \end{array} \qquad\qquad \mathit{state1}: \quad \begin{array}{|c|r|} \hline x & 2 \\ y & \sqrt{2} \\ z & 3 \\ \hline \end{array} \; .$$

An imperative program is used to take *state0* initially to *state1* finally.

1.4.2 Descriptions of states

Pictures of state mappings, however, are not very useful for our development of programs: they say both too much and too little. They say too much because the value of every variable must be given exactly, even the variables in which we might not be interested. They say too little because each gives only *one* state, and to understand a program completely we need to know its behaviour on *all* initial states. That is far too many pictures...

We describe states rather than draw them. A formula *describes* a state if it is made true by the mappings in the state. And we say that a state *satisfies* a formula if that formula describes it. Thus each of the formulae $x = 2$, $x + z < y$, and $z \neq 4$ describes *state0*. This formula describes *state0* exactly:

$$x = 2 \wedge y = 17 \wedge z = 3 \ .$$

The formula $y^2 = x$, for example, describes *state1*.

As an extreme case, the formula true describes *all* states (because it is true in all states). Similarly, the formula false describes *no* states (because it is true in no states).

We use the predicate calculus as the language of our formulae. It includes the usual equations (like $x = y$) and relations ($z < 17$), the logical connectives $\wedge$ (and), $\vee$ (or), $\neg$ (not), and $\Rightarrow$ (implies); and it has the familiar quantifiers $\forall$ (for all) and $\exists$ (there exists).

1.4.3 Specifications

The *specification* is the principal feature of abstract programs. Its *precondition* describes the initial states; its *postcondition* describes the final states; and its *frame* lists the variables whose values may change. If a computer could execute it, this would be the effect:

> *If* the initial state satisfies the precondition *then* change only the variables listed in the frame so that the resulting final state satisfies the postcondition.

It is deliberate that a specification can leave some possibilities unconstrained. If the initial state does not satisfy the precondition, we do not know what will happen; one possibility is that the computer does not deliver any final state at all — that is, it fails to terminate. If there are several possible final states satisfying the postcondition, we do not know which will be chosen (nondeterminism).

Here is a specification which assigns to y the square root of x, provided x lies between 0 and 9:

precondition	$0 \leq x \leq 9$
postcondition	$y^2 = x$
frame	y

$$y{:}\ \left[0 \leq x \leq 9\ ,\ y^2 = x\right] . \qquad (1.1)$$

x: [true , $y^2 = x$]	Make x the square of y.
y: [$x \geq 0$, $y^2 = x$]	Make y a square root of x, provided x is not negative.
e: [$s \neq \{\}$, $e \in s$]	Make e an element of the set s, provided s is non-empty.
x: [$b^2 \geq 4ac$, $ax^2 + bx + c = 0$]	Make x a solution to the quadratic equation, provided the discriminant is non-negative.

Figure 1.5 Example specifications

On the right is the same specification written more compactly: in general, for precondition *pre*, postcondition *post*, and frame w, the compact form is

$$w\colon [\mathit{pre}\ ,\ \mathit{post}]\ .$$

Specification (1.1) leaves some possibilities unconstrained: we do not know what would happen if it were applied to an initial state which mapped x to 10, because the precondition would not be true. Even when the precondition is true, there is still some uncertainty: applied to *state0*, it will produce either *state1* (above) or *state2* below — but beforehand we do not know which it will be.

state2:

x	2
y	$-\sqrt{2}$
z	3

Figure 1.5 lists some other examples of specifications. Notice how in those examples (especially the quadratic) the use of formulae allows a specification to capture the intention without necessarily giving the method. It can say 'what' without having to say 'how'.

1.4.4 Refinement of specifications

A specification is improved (for the client) by *strengthening* its postcondition, so that the new postcondition implies the old: *if* a book is available in paperback and hardback, *then* it is available in paperback at least. Requiring the square root to be non-negative is another example: *if* you have a non-negative square root, *then* you at least have a square root. So (1.2) refines (1.1):

$$y\colon \left[0 \leq x \leq 9\ ,\ y^2 = x \wedge y \geq 0\right]. \tag{1.2}$$

Specification (1.2) is better for the customer because he knows more about the final state: he can *depend* on $y \geq 0$, if he wants to, and he could not before.

In general, we have the following law of refinement, of which (1.1) $\sqsubseteq$ (1.2) is an example:

Law 1.1 <u>strengthen postcondition</u> If $post' \Rrightarrow post$, then

$$w\colon [pre\ ,\ post] \quad \sqsubseteq \quad w\colon [pre\ ,\ post']\,.$$

□

(For now, read '$\Rrightarrow$' as implication; it is defined in Chapter 2.) The requirement $post' \Rrightarrow post$ must hold whenever the law is used, and it is called the *proviso*. The symbol □ indicates the end of a law (or similar), and the resumption of normal text.

A different kind of improvement is gained by *weakening* a precondition, so that the old precondition implies the new: *if* at least 4Mb is required, *then* certainly at least 2Mb is required. Requiring our square root program to operate for *any* non-negative x is another example: if x is non-negative and no greater than 9, then x is still non-negative. So (1.3) refines (1.2):

$$y\colon \left[0 \leq x\ ,\ y^2 = x \wedge y \geq 0\right]. \tag{1.3}$$

Specification (1.3) is better than (1.2) because it works even when $x > 9$.

The general law for preconditions is the following:

Law 1.2 <u>weaken precondition</u> If $pre \Rrightarrow pre'$, then

$$w\colon [pre\ ,\ post] \quad \sqsubseteq \quad w\colon [pre'\ ,\ post]\,.$$

□

Note that it too has a proviso.

1.5 Executable programs

1.5.1 Code

Specifications are written in a language (predicate calculus) whose meaning is known precisely. They are unambiguous, and very convenient because they are so expressive. Why not build a computer that executes them?

The simple answer to that question is 'it's impossible'. It can be proved that no computer, as the term is presently understood, can be built which could execute all specifications.

The problem is that our formulae are written in too powerful a language: we can say too much with them. We could use a weaker language — but the weaker the language, the more involved become the constructions that we need to say what we must. And the more involved they are, the more likely it is that the client and programmer will misunderstand each other *at the very beginning*, before the design, modularisation, coding, integration, debugging... The enterprise would be doomed before it had started.

Our approach is to have it both ways. The programming language includes specifications, for their expressive power, but has also constructions which are designed to be executable. We call them *code*.

1.5.2 The assignment command

The notation $w := E$ is an *assignment command*, and is our first example of code. It changes the state so that the variable w is mapped to the value E, and all other variables are left unchanged. Assignment commands form the basis of imperative programming languages; they are easy to execute provided the expression E is constructed from constants and operators that the programming language provides (such as 0, 1, 2, $+$, $-$, $\times$ and $\div$). Below we give a law of refinement for assignments.

Law 1.3 assignment If $pre \Rrightarrow post[w \backslash E]$, then

$$w, x\colon [pre\ ,\ post] \quad \sqsubseteq \quad w := E\ .$$

□

The formula $post[w \backslash E]$ is obtained by replacing in $post$ all occurrences of w by E. (Such substitutions are explained more fully in Section A.2.1.) Note that the frame w, x can include variables x not assigned to in the code: the frame says that x *may* change, not that it must.

Law 1.3 allows a programmer to implement a specification $w\colon [pre\ ,\ post]$ by code $w := E$ that is better from the client's point of view — it is better because where *post* may have allowed several final values for w, now it allows only one, and the customer knows exactly which one it will be. In quite a different sense, it is better for the programmer too: a computer can execute it. Laws like *assignment* 1.3 are the programmer's tools; his expertise is in deciding which to use.

In a programming language with a positive square root operator $\sqrt{}$ (and assuming infinite precision arithmetic), Law 1.3 gives us this refinement immediately:

$$y\colon \left[0 \le x \le 9\ ,\ y^2 = x \wedge y \ge 0\right] \quad \sqsubseteq \quad y := \sqrt{x}\ .$$

That is because we can prove the relevant proviso:

$$\begin{array}{ll} & 0 \le x \le 9 \\ \Rrightarrow & \text{“}\sqrt{x}\text{ well-defined in given range”} \\ & (\sqrt{x})^2 = x \\ \equiv & \text{“definition of }\sqrt{}\text{”} \end{array}$$

$\longleftarrow$ increasing clarity

programs $\boxed{\text{code}}$

increasing executability $\longrightarrow$

Figure 1.6 Code is the language of *executable* programs

$$
\begin{array}{ll}
 & (\sqrt{x})^2 = x \wedge \sqrt{x} \geq 0 \\
\equiv & (y^2 = x \wedge y \geq 0)[y \backslash \sqrt{x}] \ .
\end{array}
$$

(For now, read '$\equiv$' as equivalence; it is defined in Chapter 2.)

If $\sqrt{}$ is not provided by the programming language, then more complex — but still executable — coding features must be used. And that is the usual case. The overall refinement of specification (eventually) to code is made in steps, each introducing a little more executability, or a little more efficiency. A sequence results, each element refined by the next:

$$spec \sqsubseteq mixed_0 \sqsubseteq \cdots \sqsubseteq mixed_n \sqsubseteq code \ .$$

We see later that the intermediate steps $mixed_i$ mix specifications and code together. Indeed, in such sequences it is difficult to say when specifying stops and coding begins, and there is nothing to be gained by trying. We call *all* of them programs, whether executable or not, and we reserve the term 'specification' for frame and predicate pairs $w\colon [pre \ , \ post]$ only; they can be an entire program (like *spec*), or parts of programs (occurring in $mixed_i$). We use the term 'code' for programs written entirely in our executable language (which we define shortly). Usually, an assignment command is code; a specification is not.

All that is summarised in Figure 1.6. *Program development* moves, via a series of refinement steps, within the outer box (of programs) towards the inner box (of programs that are code as well).

Developing programs, in that sense, is the main topic of this book.

1.6 Mixed programs

Mixed programs occur during development. They contain both abstract and executable constructs, linked by program constructors like *sequential composition*, here defined informally:

> The effect of the program *prog*1; *prog*2 is the effect of *prog*1 followed by the effect of *prog*2.

For example, the mixed program

$$x := 9;$$
$$y \colon [\mathsf{true} \ , \ y^2 = x]$$

sets x to 9 and y either to 3 or to -3.

Constructors like sequential composition are introduced by their own laws, which are the subject of later chapters.

1.7 Infeasible programs

Suppose Specification (1.3) is refined still further (by Law 1.2) to

$$y \colon \left[\mathsf{true} \ , \ y^2 = x \wedge y \geq 0\right] . \qquad (1.4)$$

Here, the programmer has given up too much. His job in Specification (1.4) is impossible, as we can easily show: since x, y are in $\mathbb{R}$ there are some allowed initial values of x for which *no* final value of y will do. Such a specification is *infeasible.*

Infeasible specifications cannot be refined by *any* code; and so agreeing on a contract containing an infeasible specification means eventual disappointment for the client, and possible financial ruin for the programmer. For that reason it is important to be able to check for feasibility:

Definition 1.4 feasibility The specification $w \colon [pre \ , \ post]$ is *feasible* iff

$$pre \quad \Rrightarrow \quad (\exists\, w : T \cdot post) \ ,$$

where T is the type[1] of the variables w.

□

The right-hand side is read "there exists a w of type T such that *post*". The symbol $\exists$ is further discussed in Chapter 2, and types are the subject of Chapter 6.

Applying Definition 1.4 to Specification (1.4), the programmer tries to prove

$$\mathsf{true} \quad \Rrightarrow \quad \left(\exists\, y : \mathbb{R} \cdot y^2 = x \wedge y \geq 0\right)$$

under the assumption that x is of type $\mathbb{R}$. But he cannot: the right-hand side is equivalent to $x \geq 0$, which is not implied by true. Hence (1.4) is infeasible. For historical reasons, infeasible programs are sometimes called *miracles.*

Remember that code is designed to be executable: all code, therefore, is feasible. (See Exercise 1.17.)

[1]In Chapter 6 the notion of type will be generalised to include so-called 'local invariants', and then a more comprehensive definition (6.5) of feasibility will be appropriate. It does not concern us now, but must be borne in mind if ever referring to the definition above once local invariants have been introduced.

1.8 Some common idioms

We can compact our new notations somewhat by taking advantage of common idioms. Often the precondition of a specification is just true, for example, indicating termination in all circumstances. In that case we just omit it:

Abbreviation 1.5 <u>default precondition</u>

$$w\colon [post] \;\widehat{=}\; w\colon [\mathsf{true}\ ,\ post]\ .$$

□

The symbol '$\widehat{=}$' indicates a definition, and is used in preference to equality when the left-hand side is newly introduced. (Colon ':' is used instead of membership '$\in$' in the same circumstances.)

When the frame is empty and the postcondition is true, we have a command that either fails to terminate (because its precondition is false), or terminates but changes nothing (because the frame is empty). We call those *assumptions*, and they are related to the practice of 'annotating' a program with formulae that are supposed to hold at various points. With that in mind we have

Abbreviation 1.6 <u>assumption</u>

$$\{pre\} \;\widehat{=}\; \colon [pre\ ,\ \mathsf{true}]\ .$$

□

As a special case (to strengthen further the resemblance to annotating programs), we allow the semicolon that would normally indicate sequential composition to be omitted if it follows an assumption. Thus the program

$$\{0 \le x \le 9\}\ y := \sqrt{x} \qquad (1.5)$$

sets y to the non-negative square root of x provided x lies between 0 and 9 inclusive. If x does not fall within that range, then $\{0 \le x \le 9\}$ aborts, effectively aborting the whole of (1.5).

The similarity between (1.5) and (1.2) is not accidental, and suggests some further correspondences. One is

Law 1.7 <u>simple specification</u> Provided E contains no w,

$$w := E \quad = \quad w\colon [w = E]\ .$$

If w and E are lists, then the formula $w = E$ means the equating of corresponding elements of the lists.
□

Law 1.7 together with a law relating assumptions and preconditions will allow us to show that (1.5) and (1.2) are equal. The law is

Law 1.8 absorb assumption An assumption before a specification can be absorbed directly into its precondition.

$$\{pre'\}\ w\colon [pre\ ,\ post] \quad = \quad w\colon [pre' \wedge pre\ ,\ post]\ .$$

□

Law 1.8 highlights the fact that 'aborting now' (at $\{pre'\}$) is the same as 'aborting later' (at $[pre' \wedge \cdots$) in imperative programs. (See Exercise 1.6.)

With those new laws, the proof of equality runs as follows:

$$\{0 \le x \le 9\}\ y := \sqrt{x}$$
$=$ "*simple specification* 1.7"
$$\{0 \le x \le 9\}\ y\colon [y = \sqrt{x}]$$
$=$ "*absorb assumption* 1.8"
$$y\colon [0 \le x \le 9\ ,\ y = \sqrt{x}]$$
$=$ "rewrite postcondition"
$$y\colon [0 \le x \le 9\ ,\ y^2 = x \wedge y \ge 0]\ .$$

With the comment "rewrite postcondition" we are merely relying on the fact that for reals x and y

$$y^2 = x \wedge y \ge 0 \quad \equiv \quad y = \sqrt{x}\ .$$

1.9 Extreme programs

We finish with some specification pathology. From the client's point of view, the worst specification of all is

$$w\colon [\mathsf{false}\ ,\ \mathsf{true}]\ .$$

It is *never* guaranteed to terminate (precondition false); and even when it does, it has complete freedom in its setting of the variables (postcondition true). As a contract, it allows any refinement at all: infinite loops, programs setting w to arbitrary values — even programs that change variables other than w. We call it **abort**.

Slightly better is the program that always terminates, but guarantees no particular result:

$$w\colon [\mathsf{true}\ ,\ \mathsf{true}]\ .$$

It can be refined to any *terminating* program that changes only w; we can imagine that it just chooses w at random. We call it **choose** w.

Better still is the program which always terminates, changing nothing. Its frame is empty:

$$\colon [\mathsf{true}\ ,\ \mathsf{true}]\ .$$

We call it **skip**.

Best of all is the infeasible program that always terminates and establishes the impossible false:

$$w\colon [\mathsf{true}\ ,\ \mathsf{false}]\,.$$

No computer can execute that program; no contract based on it could ever be met. We call it **magic**.

Most of the above are seldom written deliberately in programs (**skip** is the exception). But we need their names if only to reason about them.

1.10 Exercises

Exercises marked with ♡ are answered in Appendix B.

Ex. 1.1 ♡ 'The programmer's job is to take specifications, via a sequence of refinement steps, to code. Hence the more refined the client's requirements, the fewer refinement steps remain for the programmer to do, and the easier his job.'

The above argument suggests the *opposite* of Figure 1.3. Where is the error?

Ex. 1.2 ♡ Recall Specification (1.1) from p.5. Write a new specification that finds a square root y of x if x is non-negative but no more than 9, and sets y to 0 if x is negative.

Ex. 1.3 Revise your answer to Exercise 1.2 so that when x is negative initially the specification does not choose any particular final value for y, but still terminates.

Ex. 1.4 ♡ Which of these refinements are valid? (Use *strengthen postcondition* 1.1 and *weaken precondition* 1.2).

1. $x\colon [x \geq 0] \sqsubseteq ? x\colon [x = 0]$
2. $x\colon [x \geq 0\ ,\ \mathsf{true}] \sqsubseteq ? x\colon [x = 0\ ,\ \mathsf{true}]$
3. $x\colon [x \geq 0\ ,\ x = 0] \sqsubseteq ? x\colon [x = 0\ ,\ x \geq 0]$
4. $x\colon [x = 0\ ,\ x \geq 0] \sqsubseteq ? x\colon [x \geq 0\ ,\ x = 0]$
5. $y\colon [x > 0\ ,\ x > y \geq 0] \sqsubseteq ? y\colon [x > y \geq 0]$
6. $y\colon [x > y \geq 0] \sqsubseteq ? y\colon [y = 0]$
7. $y\colon [x > 0\ ,\ x > y \geq 0] \sqsubseteq ? y\colon [y = 0]$

Ex. 1.5 What refinement relations hold between Specification (1.1) and those in Exercises 1.2 and 1.3? (You cannot yet prove them.)

Ex. 1.6 Use *absorb assumption* 1.8 to show that

$$\{pre'\}\ w\colon [pre\ ,\ post] \quad = \quad \{pre\}\ w\colon [pre'\ ,\ post]\ .$$

Ex. 1.7 ♡ Give an informal argument to show that *contracting the frame* is a refinement; that is, argue that

$$w, x\colon [pre\ ,\ post] \quad \sqsubseteq \quad w\colon [pre\ ,\ post]\,.$$

(That refinement appears later as *contract frame* 5.4.)

Ex. 1.8 ♡ Prove that your answer to Exercise 1.2 is feasible.

Ex. 1.9 Prove that your answer to Exercise 1.3 is feasible.

Ex. 1.10 ♡ Show that the following is *not* feasible:

$$y\colon \left[x \geq 0\ ,\ y^2 = x \wedge y > 0\right].$$

Explain informally why it is not.

Ex. 1.11 ♡ Describe informally the program

$$\colon [\mathsf{false}\ ,\ \mathsf{false}]\,.$$

Is it **magic**? (Is it feasible?) Is it **skip**? (Can it change w?) Is it **abort**? (Is it ever guaranteed to terminate?)

Ex. 1.12 ♡ What is the effect of adding to a program an assumption that might not be true during execution? Are there circumstances in which that has no effect at all?

Ex. 1.13 ♡ Is it a refinement to strengthen or to weaken assumptions?

Ex. 1.14 ♡ Prove this equality:

Law 1.9 merge assumptions

$$\{pre'\}\ \{pre\} \quad = \quad \{pre' \wedge pre\}\ .$$

□

Ex. 1.15 Show that assumptions can be removed from a program unconditionally, as expressed in this law:

> *Law 1.10* remove assumption Any assumption is refined by **skip**.
>
> $$\{pre\} \quad \sqsubseteq \quad \mathbf{skip} \; .$$
>
> □

Does that mean that assumptions are code?

Ex. 1.16 Show that neither *strengthen postcondition* 1.1 nor *weaken precondition* 1.2 can refine an infeasible specification to a feasible one.

Ex. 1.17 ♡ 'Infeasible specifications cannot be refined by any code' (p.10). From that, show that all code is feasible.

Ex. 1.18 ♡ Show that anything refines **abort**: that is, that

$$w\colon [\mathsf{false}\ ,\ \mathsf{true}] \quad \sqsubseteq \quad w\colon [pre\ ,\ post]\ ,$$

for *any* formulae *pre* and *post*.

Ex. 1.19 Show that **magic** refines anything: that is, that

$$w\colon [pre\ ,\ post] \quad \sqsubseteq \quad w\colon [\mathsf{true}\ ,\ \mathsf{false}]\ ,$$

for *any* formulae *pre* and *post*.