2 SEQUENTIAL PROCESSES

This chapter describes the role of abstraction and structure in problem solving, and the nature of computations. It also summarizes the structuring principles of data and sequential programs and gives an example of hierarchal program construction.

2.1. INTRODUCTION

The starting point of a theory of operating systems must be a *sequential process*—a succession of events that occur one at a time. This is the way our machines work; this is the way we think. Present computers are built from a small number of large sequential components: store modules which can access one word at a time, arithmetic units which can perform one addition at a time, and peripherals which can transfer one data block at a time. Programs for these computers are written by human beings who master complexity by dividing their tasks into smaller parts which can be analyzed and solved one at a time.

This chapter is a summary of the basic concepts of sequential programming. I assume you already have an intuitive understanding of many of the problems from your own programming experience. We shall begin by discussing the role of abstraction and structure in problem solving.

2.2. ABSTRACTION AND STRUCTURE

Human beings can think precisely only of simple problems. In our efforts to understand complicated problems, we must concentrate at any moment on a small number of properties that we believe are essential for our present purpose and ignore all other aspects. Our partial descriptions of the world are called *abstractions* or *models.*

One form of abstraction is the use of *names* as abbreviations for more detailed explanations. This is the whole purpose of terminology. It enables us to say "operating system" instead of "a set of manual and automatic procedures that enable a group of people to share a computer installation efficiently."

In programming, we use names to refer to *variables.* This abstraction permits us to ignore their actual values. Names are also used to refer to *programs* and *procedures.* We can, for example, speak of "the editing program." And if we understand *what* editing is, then we can ignore, for the moment, *how* it is done in detail.

Once a problem is understood in terms of a limited number of aspects, we proceed to analyze each aspect separately in more detail. It is often necessary to repeat this process so that the original problem is viewed as a hierarchy of abstractions which are related as components within components at several levels of detail.

In the previous example, when we have defined what an editing program must do, we can proceed to construct it. In doing so, we will discover the need for more elementary editing procedures which can "search," "delete," and "insert" a textstring. And within these procedures, we will probably write other procedures operating on single characters. So we end up with several levels of procedures, one within the other.

As long as our main interest is the properties of a component as a whole, it is considered a *primitive component*, but, when we proceed to observe smaller, related components inside a larger one, the latter is regarded as a *structured component* or *system.* When we wish to make a distinction between a given component and the rest of the world, we refer to the latter as the *environment* of that component.

The environment makes certain assumptions about the properties of a component and vice versa: The editing program assumes that its input consists of a text and some editing commands, and the user expects the program to perform editing as defined in the manual. These assumptions are called the *connections* between the component and its environment.

The set of connections between components at a given level of detail defines the *structure* of the system at that level. The connections between the editing program and its users are defined in the program manual. Inside the editing program, the connections between the components "search," "delete," and "insert" are defined by what these procedures assume about the properties and location of a textstring and by what operations they perform on it.

Abstraction and recognition of structure are used in all intellectual disciplines to present knowledge in forms that are easily understood and remembered. Our concern is the systematic use of abstraction in the design of operating systems. We will try to identify problems which occur in all shared computer installations and define a set of useful components and rules for connecting them into systems.

In the design of large computer programs, the following difficulties must be taken for granted: (1) improved understanding of the problems will change our *goals* in time; (2) technological innovations will eventually change our *tools*; and (3) our intellectual limitations will often cause us to make *errors* in the construction of large systems. These difficulties imply that large systems will be modified during their entire existence by designers and users, and it is essential that we build such systems with this in mind. If it is difficult to understand a large system, it is also difficult to predict the consequences of modifying it. So *reliability* is intimately related to the *simplicity* of structure at all levels.

Figure 2.1(a) shows a complicated system S, consisting of n components $S1$, $S2$, . . . , Sn. In this system, each component depends directly on the behavior of all other components. Suppose an average of p simple steps of reasoning or testing are required to understand the relationship between one component and another and to verify that they are properly connected. Then the connection of a single component to its environment can be verified in $(n - 1)p$ steps, and the complete system requires $n(n - 1)p$ steps.

This can be compared with the system shown in Fig. 2.1(b): There, the connections between the n components are defined by a common set of constraints chosen so that the number of steps q required to verify whether

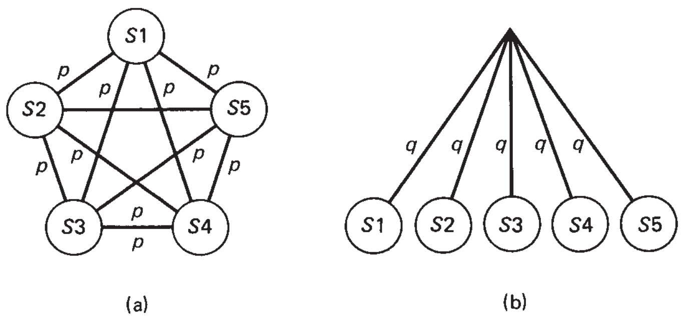

Fig. 2.1 Two examples of system structures.

a component satisfies them is independent of the number of components n. The proof or test effort is now q steps per component and nq steps for the whole system.

The argument is, of course, extreme, but it does drive home the following point: If the intellectual effort required to understand and test a system increases more than linearly with the size of the system, we shall never be able to build reliable systems beyond a certain complexity. Our only hope is to restrict ourselves to simple structures for which the effort of verification is proportional to the number of components.

The importance of *precise documentation* of system structure can hardly be overemphasized. Quite often, a group of designers intend to adopt a simple structure, but they fail to state precisely what the assumptions are at each level of programming, and instead rely on informal, spoken agreements. Inevitably, the result is that each member of the group adds complexity to the structure by making unnecessary or erroneous assumptions about the behavior of components designed by his colleagues. The importance of making assumptions explicit is especially felt when an initial version of a system must be modified; perhaps by a different group of people.

2.3. COMPUTATIONS

In Chapter 1 the word computation was used intuitively to refer to program execution. In the following, this concept and its components data and operations are defined explicitly.

2.3.1. Data and Operations

The exchange of facts or ideas among human beings by speech is based on mutual agreement on the meaning of certain sounds and combinations of sounds. Other conventions enable us to express the same ideas by means of text and pictures, holes in punched cards, polarity of magnetized media, and modulated electromagnetic waves. In short, we communicate by means of physical phenomena chosen by us to represent certain aspects of our world. These physical representations of our abstractions are called *data*, and the meanings we assign to them are called their *information*.

Data are used to transmit information between human beings, to store information for future use, and to derive new information by manipulating the data according to certain rules. Our most important tool for the manipulation of data is the digital computer.

A datum stored inside a computer can only assume a finite set of values called its *type*. Primitive types are defined by *enumeration* of their values, for example:

type *boolean* = (*false*, *true*)

or by definition of their *range*:

type *integer* = −8388608. .8388607

Structured types are defined in terms of primitive types, as is explained later in this chapter.

The rules of data manipulation are called operations. An *operation* maps a finite set of data, called its *input*, into a finite set of data, called its *output*. Once initiated, an operation is executed to completion within a finite time. These assumptions imply that the output of an operation is a *time-independent function* of its input, or, to put it differently: An operation always delivers the same output values when it is applied to a given set of input values.

An operation can be defined by *enumeration* of its output values for all possible combinations of its input values. This set of values must be finite since an operation only involves a finite set of data with finite ranges. But in practice, enumeration is only useful for extremely simple operations such as the addition of two decimal digits. As soon as we extend this method of definition to the addition of two decimal numbers of, say, 10 digits each, it requires enumeration of 10^{20} triples $(x, y, x + y)$!

A more realistic method is to define an operation by a *computational rule* involving a finite sequence of simpler operations. This is precisely the way we define the addition of numbers. But like other abstractions, computational rules are useful intellectual tools only as long as they remain simple.

The most powerful method of defining an operation is by *assertions* about the type of its variables and the relationships between their values before and after the execution of the operation. These relationships are expressed by statements of the following kind: If the assertion P is true before initiation of the operation Q, then the assertion R will be true on its completion. I will use the notation

"P" Q "R"

to define such relationships.

As an example, the effect of an operation *sort* which orders the n elements of an integer array A in a non-decreasing sequence can be defined as follows:

```
"A: array 1. .n of integer"
  sort(A);
"for all i, j: 1. .n (i ≤ j implies A(i) ≤ A(j))"
```

Formal assertions also have limitations: They tend to be as large as the programs they refer to. This is evident from the simple examples that have been published (Hoare, 1971a).

The whole purpose of abstraction is *abbreviation.* You can often help the reader of your programs much more by a short, informal statement that appeals to a common background of more rigorous definition. If your reader knows what a Fibonacci number is, then why write

"*F*: **array** 0. .*n* **of** *integer* & *j*: 0. .*n* &
for all *i*: 0. .*j* (*F*(*i*) = **if** *i* < 2 **then** *i* **else** *F*(*i* - 2) + *F*(*i* - 1))"

when the following will do

"*F*(0) *to F*(*j*) *are the first j* + 1 *Fibonacci numbers*"

Definition by formal or informal assertion is an abstraction which enables us to concentrate on what an operation does and ignore the details of how it is carried out. The *type* concept is an abstraction which permits us to ignore the actual values of variables and state that an operation has the effect defined for all values of the given types.

2.3.2. Processes

Data and operations are the primitive components of computations. More precisely, a *computation* is a finite set of operations applied to a finite set of data in an attempt to solve a problem. If a computation solves

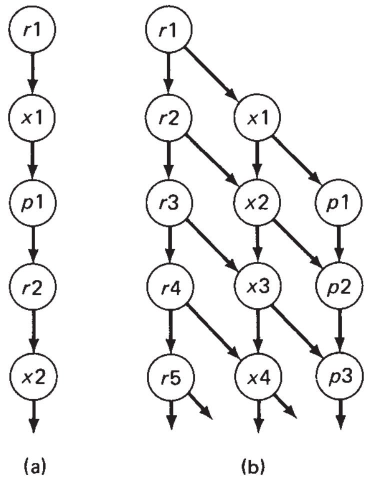

Fig. 2.2 A precedence graph of (a) a sequential and (b) a concurrent computation.

the given problem, it is also called an *algorithm*. But it is possible that a computation is meaningless in the sense that it does not solve the problem it was intended to solve.

The operations of a computation must be carried out in a certain order of precedence to ensure that the results of some operations can be used by others. The simplest possible precedence rule is the execution of operations in strict sequential order, one at a time. This type of computation is called a *sequential process*. It consists of a set of operations which are *totally ordered* in time.

Figure 2.2(a) shows a *precedence graph* of the process performed by an operating system that schedules user computations one at a time. Each node represents an instance of one of the following operations: *r* (read user request), *x* (execute user computation), and *p* (print user results). The directed branches represent precedence of operations. In this case:

$r1$ precedes $x1$,
$x1$ precedes $p1$,
$p1$ precedes $r2$,
$r2$ precedes $x2$,
. . . . .

Most of our computational problems require only a *partial ordering* of operations in time: Some operations must be carried out before others, but some of them can also be carried out concurrently. This is illustrated in Fig. 2.2(b) by a precedence graph of a spooling system (see Chapter 1, Fig. 1.3). Here the precedence rules are:

$r1$ precedes $x1$ and $r2$,
$x1$ precedes $p1$ and $x2$,
$p1$ precedes $p2$,
$r2$ precedes $x2$ and $r3$,
. . . .

Partial ordering makes *concurrent execution* of some operations possible. In Fig. 2.2(b), the executions of the following operations may overlap each other in time:

$r2$ and $x1$,
$r3$ and $x2$ and $p1$,
. . . . .

In order to understand concurrent computations, it is often helpful to try to partition them into a number of sequential processes which can be analyzed separately. This decomposition can usually be made in several ways, depending on what your purpose is.

If you wish to design a spooling system, then you will probably partition the computation in Fig. 2.2(b) into three sequential processes of a cyclical nature each in control of a physical resource:

reader process:	$r1; r2; r3; \ldots$
scheduler process:	$x1; x2; x3; \ldots$
printer process:	$p1; p2; p3; \ldots$

These processes can proceed simultaneously with independent speeds, except during short intervals when they must exchange data: The scheduler process must receive user requests from the reader process, and the printer process must be informed by the scheduler process of where user results are stored. Processes which cooperate in this manner are called *loosely connected processes.*

On the other hand, if you are a user, it makes more sense to recognize the following processes in Fig. 2.2(b):

job 1: $r1; x1; p1;$
job 2: $r2; x2; p2;$
job 3: $r3; x3; p3;$
. . . . . .

Both decompositions are useful for a particular purpose, but each of them also obscures certain facts about the original computation. From the first decomposition it is not evident that the reader, scheduler, and printer processes execute a stream of jobs; the second decomposition hides the fact that the jobs share the same reader, processor, and printer. A decomposition of a computation into a set of processes is a partial description or an abstraction of that computation. And how we choose our abstractions depends on our present purpose.

The abstractions chosen above illustrate a general principle: In a successful decomposition, the connections between components are much weaker than the connections inside components. It is the loose connections or infrequent interactions between the processes above which make it possible for us to study them separately and consider their interactions only at a few, well-defined points.

One of the recurrent themes of this book is *process interaction.* It is a direct consequence of the sharing of a computer. Processes can interact for various reasons: (1) because they *exchange data,* such as the reader, scheduler, and printer processes; (2) because they *share physical resources,* such as job1, job2, job3, and so on; or (3) because interaction *simplifies our understanding* and verification of the correctness of a computation—this is a strong point in favor of sequential computations.

Concurrent computations permit better utilization of a computer installation because timing constraints among physical components are reduced to a minimum. But since the order of operations in time is not completely specified, the output of a concurrent computation may be a time-dependent function of its input unless special precautions are taken. This makes it impossible to reproduce erroneous computations in order to locate and correct observed errors. In contrast, the output of a sequential process can always be reproduced when its input is known. This property of sequential processes along with their extreme simplicity makes them important components for the construction of concurrent computations.

The main obstacles to the utilization of concurrency in computer installations are economy and human imagination. Sequential processes can be carried out cheaply by repeated use of simple equipment; concurrent computations require duplicated equipment.

Human beings find it very difficult to comprehend the combined effect of activities which evolve simultaneously with independent speeds. Those who have studied the history of nations in school, one by one—American history, French history, and so on—recall how difficult it was to remember, in connection with a crucial time of transition in one country, what happened in other countries at the same time. The insight into our historical background can be greatly improved by presenting history as a sequence of stages and by discussing the situation in several countries at each stage—but then the student finds it equally difficult to remember the continuous history of a single nation!

It is hard to avoid the conclusion that we understand concurrent events by looking at sequential subsets of them. This would mean that, even though technological improvements may eventually make a high degree of concurrency possible in our computations, we shall still attempt to partition our problems conceptually into a moderate number of sequential activities which can be programmed separately and then connected loosely for concurrent execution.

In contrast, our understanding of a sequential process is independent of its actual speed of execution. All that matters is that operations are carried out one at a time with finite speed and that certain relations hold between the data before and after each operation.

2.3.3. Computers and Programs

The idea of defining complicated computations rigorously implies the use of a formal language to describe primitive data types and operations as well as combinations of them. A formal description of a computation is called a *program*, and the language in which it is expressed is called a *programming language*.

Programs can be used to communicate algorithms among human beings,

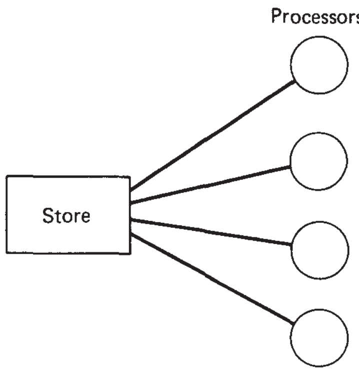

Fig. 2.3 A model of a computer installation.

but, in general, we write programs to solve problems on computers. We will indeed define a *computer installation* as a physical system capable of carrying out computations by interpreting programs.

Figure 2.3 shows a model of a computer installation. It consists of a store and one or more processors. The *store* is a physical component in which data and programs can be retained for future use. It is divided into a finite set of primitive components called *locations*. Each location can store any one of a finite set of data values.

A *processor* is a physical component which can carry out a sequential process defined by a program. During its execution, a program is stored as a sequence of data called *instructions*. An instruction consists of four components defining an operation, its input and output, and a successor instruction. Data used to identify store locations are called *addresses*.

The processors can work concurrently and share the common store. Some of the processors are called *terminals* or *peripheral devices*; they are dedicated to the transfer of data between the environment and the store. Other processors are called *central processors*; they operate mainly on stored data. For our purposes, the distinction between peripheral devices and central processors is not fundamental; it merely reflects various degrees of specialization.

The rest of this chapter is a discussion of the fundamental abstraction, sequential processes. It summarizes methods of structuring data and sequential programs, and serves as a presentation of the programming language used throughout the text, a *subset* of the language *Pascal*, created by Wirth. The algorithmic statements of Pascal are based on the principles and notation of *Algol 60*. But the data structures of Pascal are much more general than those of Algol 60.

Pascal permits *hierarchal structuring* of data and program, extensive *error checking* at compile time, and production of *efficient machine code* on present computers. It combines the clarity needed for teaching the subject with the efficiency required for designing operating systems. If you are familiar with Algol 60, you will find it quite natural to adopt Pascal as your programming tool.

The following is a brief and very informal description of a subset of Pascal with the emphasis on the language features that make it different from Algol 60. Although my summary of Pascal is sufficient for understanding the rest of the book, I recommend that you study the official Pascal report, which is written in clear, informal prose (Wirth, 1971a).

I have taken a few minor liberties with the Pascal notation. They are not mentioned explicitly because my subject is not rigorous language definition, but operating system principles.

2.4. DATA STRUCTURES

2.4.1. Primitive Data Types

Constants are denoted by numbers or identifiers. A definition of the form:

const *a*1 = *c*1, *a*2 = *c*2, . . . , *ak* = *ck*;

introduces the identifiers *a*1, *a*2, . . . , *ak* as synonyms of the constants *c*1, *c*2, . . . , *ck*, for example:

const *e* = 2.718281828;

Variables are introduced by declarations of the form:

var *v*1, *v*2, . . . , *vk*: <type>;

which associates the identifiers *v*1, *v*2, . . . , *vk* with a data type.

A *data type* is the set of values which can be assumed by a variable. A type can be defined either directly in the declaration of a variable or separately in a type definition which associates an identifier *T* with the type:

type *T* = <type> ;
var *v*1, *v*2, . . . , *vk*: *T*;

A *primitive type* is a finite, ordered set of values. The primitive types:

boolean
integer
real

are predefined for a given computer.

Other primitive types can be defined by *enumeration* of a set of successive values:

$$(a1, a2, \ldots, ak)$$

denoted by identifiers $a1, a2, \ldots, ak$, for example:

```
type name of month =
(January, February, March, April, May, June, July,
August, September, October, November, December);
```

A primitive type can also be defined as a *range* within another primitive type:

cmin. .cmax

where *cmin* and *cmax* are constants denoting the minimum and maximum values in the range, for example:

```
type number of day = 1. .31;
var payday: number of day;

var summer month: June. .August;
```

The first example is a variable *payday*, which can assume the values 1 to 31 (a subrange of the standard type *integer*). The second example is a variable *summer month*, which can assume the values *June* to *August* (a subrange of the type *name of month* defined previously).

The set of values which can be assumed by a variable v of a primitive type T can be *generated* by means of the standard functions:

min(T) *max*(T) *succ*(v) *pred*(v)

in ascending order:

```
v:= min(T);
while v ≠ max(T) do v:= succ(v);
```

or in descending order:

```
v:= max(T);
while v ≠ min(T) do v:= pred(v);
```

2.4.2. Structured Data Types

Structured types are defined in terms of primitive types or in terms of other structured types using the connection rules of arrays and records.

The type definition:

array *D* **of** *R*

defines a data structure consisting of a fixed number of components of type *R*. Each component of an array variable *v* is selected by an index expression *E* of type *D*:

v(*E*)

Examples:

```
type table = array 1. .20 of integer;
var A: table; i: 1. .20;
. . . A(i) . . .

var length of month:
    array name of month of number of day;
. . . length of month (February) . . .
```

The type definition:

record *f*1: *T*1; *f*2: *T*2; . . . ; *fk*: *Tk* **end**

defines a data structure consisting of a fixed number of components of types *T*1, *T*2, . . . , *Tk*. The components are selected by identifiers *f*1, *f*2, . . . , *fk*. A component *fj* within a record variable *v* is denoted:

v . *fj*

Example:

```
type date = record
                day: number of day;
                month: number of month;
                year: 0. .2000;
            end
var birthday: date;
. . . birthday . month . . .
```

Good programmers do not confine themselves exclusively to the structures defined by an available programming language. They invent notations for abstractions that are ideally suited to their present purpose. But they will choose abstractions which later can be represented efficiently in terms of the standard features of their language.

I will occasionally do the same and postulate extensions to Pascal which help me to stress essential concepts and, for the moment, ignore trivial details.

As one example, I will assume that one can declare a variable *s* consisting of a *sequence* of components of type *T*:

var *s*: **sequence of** *T*

Initially, the sequence is empty. The value of a variable *t* of type *T* can be appended to or removed from the sequence *s* by means of the standard procedures

put(*t*, *s*)   *get*(*t*, *s*)

The components are removed in the order in which they are appended to the sequence. In other words, a sequence is a first-in, first-out store.

The boolean function

empty(*s*)

defines whether or not the sequence *s* is empty.

The implementation of sequences by means of arrays will be explained in Chapter 3. For a more detailed discussion of the representation of various data structures see Knuth (1969).

2.5. PROGRAM STRUCTURES

2.5.1. Primitive Statements

Operations and combinations of them are described by statements. The *primitive statements* are exit statements, assignment statements, and procedure statements.

The *exit statement*

exit *L*

is a restricted form of a *go to* statement. It causes a jump to the end of a compound statement *labeled L*:

label L **begin** . . . **exit** L; . . . **end**

This use of jumps is an efficient way of leaving a compound statement in the exceptional cases when a solution to a problem is found earlier than expected, or when no solution exists. But in contrast to the unstructured *go to* statement, the exit statement simplifies the verification of program correctness.

Suppose an assertion P holds before a compound statement Q is executed. There are now two cases to consider: Either Q is executed to completion, in which case an assertion R is known to hold; or an exit is made from Q when an exceptional case S holds. So the effect of statement Q is the following:

"P" Q "R **or** S"

The *assignment statement*

$v := E$

assigns the value of an *expression* E to a variable v. The expression must be of the same type as v. Expressions consist of operators and functions applied to constants, variables, and other expressions. The *operators* are:

arithmetic:	$+$	$-$	$*$	$/$	**mod**	
relational:	$=$	$\neq$	$<$	$>$	$\leqslant$	$\geqslant$
boolean:	&	**or**	**not**			

The *function designator*

$F(a1, a2, \ldots, ak)$

causes the evaluation of a function F with the actual parameters $a1$, $a2, \ldots, ak$.

The *procedure statement*

$P(a1, a2, \ldots, ak)$

causes the execution of a procedure P with the actual parameters $a1$, $a2, \ldots, ak$.

The *actual parameters* of functions and procedures can be variables and expressions. Expressions used as parameters are evaluated before a function or procedure is executed.

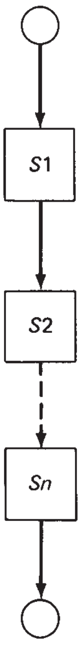

Fig. 2.4 A compound statement.

2.5.2. Structured Statements

Structured statements are formed by connecting primitive statements or other structured statements according to the following rules:

(1) *Concatenation* of statements $S1$, $S2$, . . . , Sn into a compound statement:

label L **begin** $S1; S2;$ . . . $; Sn$ **end**

begin $S1; S2;$ . . . $; Sn$ **end**

(See Fig. 2.4).

(2) *Selection* of one of a set of statements by means of a boolean expression B:

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

if B **then** S

or by means of an expression E of a primitive type T:

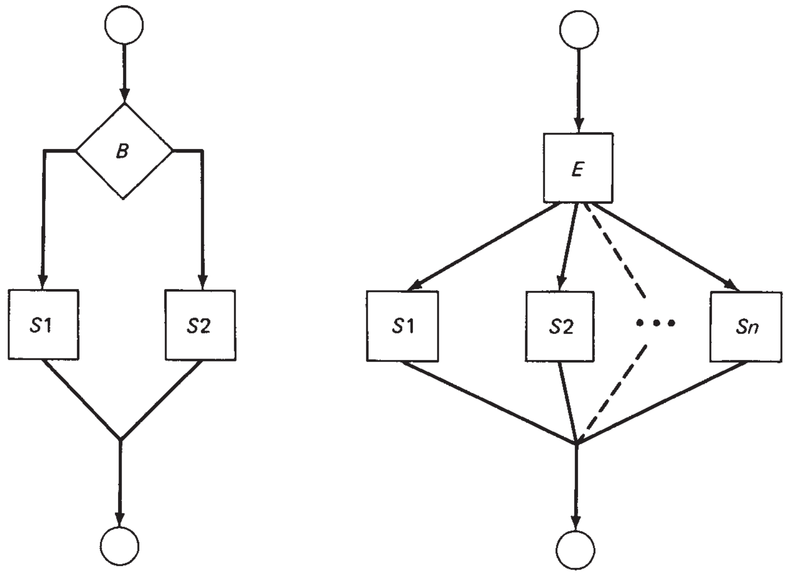

Fig. 2.5 The **if** and **case** statements.

```
type T = (c1, c2, . . . , cn);
. . .
case E of
c1: S1; c2: S2; . . . cn: Sn;
end
```

If $E = cj$ then statement Sj is executed (See Fig. 2.5).

(3) *Repetition* of a statement S while a boolean expression B remains true:

```
while B do S
```

or repetition of a sequence of statements $S1$, $S2$, . . . , Sn until a boolean expression B becomes true:

```
repeat S1; S2; . . . ; Sn until B
```

(See Fig. 2.6).

Another possibility is to repeat a statement S with a succession of primitive values assigned to a control variable v:

```
for v:= Emin to Emax do S
```

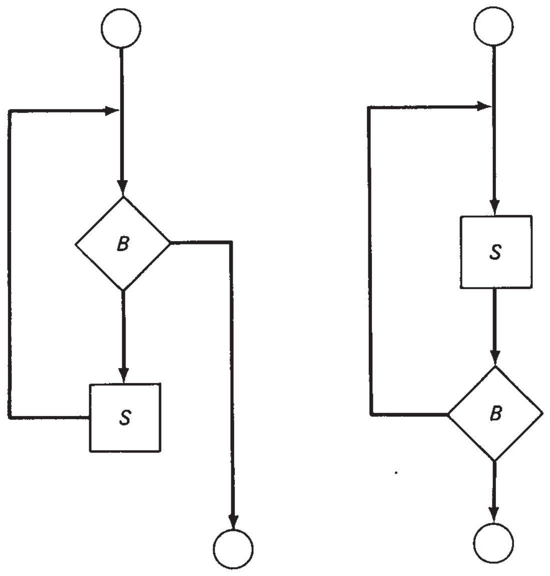

Fig. 2.6 The while and repeat statements.

(4) *Recursion* of a procedure *P* which calls itself:

procedure *P*(. . .);
begin . . . *P*; . . . **end**

(See Fig. 2.7.)

Notice that the analytical effort required to understand the effect of these structures is proportional to the number of component statements. For example, in the analysis of an *if* statement, we must first prove that a certain assertion *R* will be true after execution of statement *S*1, provided assertions *B* and *P* hold before *S*1 is initiated, and similarly for *S*2:

"B & P" $S1$ "R"   "**not** B & P" $S2$ "R"

From this we infer that

"P" **if** B **then** $S1$ **else** $S2$ "R"

The repetition statements are understood by mathematical induction. From

"B & P" S "P"

we infer that

"P" **while** B **do** S "**not** B & P"

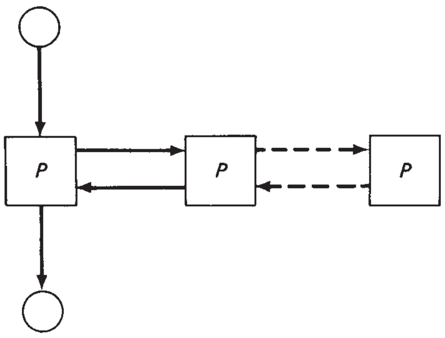

Fig. 2.7 A recursive procedure statement.

The aim is to find an *invariant P*—an assertion which is true before the iteration is started and remains true after each execution of the statment *S*.

For records, the following structured statement can be used:

with *v* **do** *S*

It enables the statement *S* to refer to the components of a record variable *v* by their identifiers *f*1, *f*2, . . . , *fk* without qualifying them with the record identifier *v*, for example:

```
with birthday do
begin day:= 19; month:= April; year:= 1938 end
```

Finally, we have the important abstraction of assigning a name to a sequence of statements *S*1, *S*2, . . . , *Sn* by means of *procedure* and *function declarations* of the form:

```
procedure P(p1; p2; . . . ; pk);
< local declarations >
begin S1; S2; . . . ; Sn end

function F(p1; p2; . . . ; pk): <result type> ;
< local declarations >
begin S1; S2; . . . ; Sn end
```

where *p*1, *p*2, . . . , *pk* are declarations of *formal parameters.* The declaration *pj* of a constant or variable parameter *vj* of type *Tj* has the form:

const *vj*: *Tj*   **var** *vj*: *Tj*

The preceding specifier can be omitted for constant parameters.

A function *F* computes a value that must be of a primitive type. At least one of its statements $S1, S2, \ldots, Sn$ must assign a result value to the function identifier F

$$F := E$$

where E is an expression of the result type.

The function statements must not assign values to actual parameters and non-local variables. This rule simplifies program verification as will be explained in Chapter 3.

The *declarations* of identifiers which are *local* to a procedure or function are written *before* the *begin* symbol of the statement part. They are written in the following order:

```
const <constant definitions>
type <type definitions>
var <variable declarations>
<local procedure and function declarations>
```

A *program* consists of a declaration part and a compound statement.

Finally, I should mention that *comments* are enclosed in quotes:

"This is a comment"

This concludes the presentation of the Pascal subset.

2.6. PROGRAM CONSTRUCTION

We design programs the same way we solve other complex problems: by step-wise analysis and refinement. In this section, I give an example of hierarchal program construction.

2.6.1. The Banker's Algorithm

The example chosen is a *resource sharing* problem first described and solved by Dijkstra (1965). Although the problem involves concurrent processes, it is used here as an example of sequential programming.

An operating system shares a set of resources among a number of concurrent processes. The resources are *equivalent* in the sense that when a process makes a request for one of them, it is irrelevant which one is chosen. Examples of equivalent resources are peripheral devices of the same type and store pages of equal size.

When a resource has been allocated to a process, it is occupied until the process releases it again. When concurrent processes share resources in this manner, there is a danger that they may end up in a *deadlock*, a state in which two or more processes are waiting indefinitely for an event that will never happen.

Suppose we have 5 units of a certain resource, and we are in a state where 2 units are allocated to a process *P* and 1 unit to another process *Q*. But both processes need two more units to run to completion. If we are lucky, one of them, say *Q*, will acquire the last two units, run to completion, and release all three of its units in time to satisfy further requests from *P*. But it is also possible that *P* and *Q* both will acquire one of the last two units and then (since there are no more) will decide to wait until another unit becomes available. Now they are deadlocked: *P* cannot continue until *Q* releases a unit; *Q* cannot continue until *P* releases a unit; and each of them expects the other to resolve the conflict.

The deadlock could have been prevented by allocating all units needed by *P* (or *Q*) at the same time rather than one by one. This policy would have forced *P* and *Q* to run at different times, and as we have seen in Chapter 1, this is often the most efficient way of using the resources. But for the moment, we will try to solve the problem without this restriction.

Let me define the problem more precisely in Dijkstra's terminology: A *banker* wishes to share a fixed *capital* of *florins* among a fixed number of *customers*. Each customer specifies in advance his maximum *need* for florins. The banker will accept a customer if his need does not exceed the capital.

During a customer's *transactions*, he can only borrow or return florins one by one. It may sometimes be necessary for a customer to wait before he can borrow another florin, but the banker guarantees that the waiting time will always be finite. The current loan of a customer can never exceed his maximum need.

If the banker is able to satisfy the maximum need of a customer, then the customer guarantees that he will complete his transactions and repay his loan within a finite time.

The current situation is *safe* if it is possible for the banker to enable all his present customers to complete their transactions within a finite time; otherwise, it is *unsafe*.

We wish to find an algorithm which can determine whether the banker's current situation is safe or unsafe. If the banker has such an algorithm, he can use it in a safe situation to decide whether a customer who wants to borrow another florin should be given one immediately or told to wait. The banker makes this decision by pretending to grant the florin and then observing whether this leads to a safe situation or not.

The situation of a customer is characterized by his current *loan* and his further *claim* where

$$claim = need - loan$$

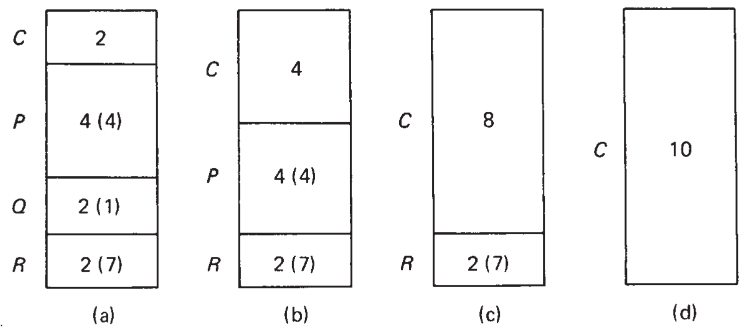

Fig. 2.8 The banker's algorithm: a safe situation.

The situation of the banker is characterized by his original *capital* and his current amount of *cash* where

cash = capital - sum of loans

The algorithm that determines whether the overall situation is safe or not is quite simple. Let me illustrate it by an example: Fig. 2.8 shows a situation in which three customers, *P*, *Q*, and *R*, share a capital of 10 florins. Their combined need is 20 florins. In the current situation, Fig. 2.8(a), customer *Q* has a loan of 2 florins and a claim of 1 florin; this is denoted 2 (1). For *P* and *R* the loans and claims are 4 (4) and 2 (7) respectively. So the available cash *C* at the moment is 10 - 4 - 2 - 2 = 2.

The algorithm examines the customers one by one, looking for one who has a claim not exceeding the cash. In Fig. 2.8(a), customer *Q* has a claim of 1. Since the cash is 2, customer *Q* will be able in this situation to complete his transactions and return his current loan of 2 florins to the banker.

After the departure of customer *Q*, the situation will be the one shown in Fig. 2.8(b). The algorithm now scans the remaining customers and compares their claims with the increased cash of 4 florins. It is now possible to satisfy customer *P* completely.

This leads to the situation in Fig. 2.8(c) in which customer *R* can complete his transactions. So finally, in Fig. 2.8(d) the banker has regained his capital of 10 florins. Consequently, the original state Fig. 2.8(a) was safe.

It is possible to go from the safe state in Fig. 2.8(a) to an unsafe situation, such as the one in Fig. 2.9(a). Here, the banker has granted a request from customer *R* for another florin. In this new situation, customer *Q* can be satisfied. But this leads us to the situation in Fig. 2.9(b) in which we are stuck: neither *P* nor *R* can complete their transactions.

If the banker's algorithm finds that a situation is unsafe, this does not necessarily mean that a deadlock will occur—only that it might occur. But if the situation is safe, it is always possible to prevent a deadlock. Notice

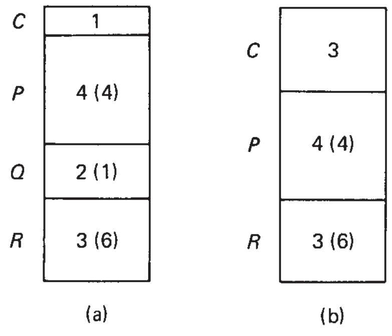

Fig. 2.9 The banker's algorithm: an unsafe situation.

that the banker prevents deadlocks in the same way as we originally proposed: by serving the customers one at a time; but the banker only does so in situations where it is strictly necessary.

We will now program the banker's algorithm for the general case considered by Habermann (1969), in which the banker's capital consists of several *currencies*: florins, dollars, pounds, and so on.

2.6.2. A Hierarchal Solution

The first version of the banker's algorithm is trivial:

```
type S = ?

function safe(current state: S): boolean;
```

It consists of a boolean function *safe* with a parameter defining the *current state*. The details of the function and the type of its parameter are as yet unknown.

The first refinement (Algorithm 2.1) expresses in some detail what the

ALGORITHM 2.1 The Banker's Algorithm

```
type S = ?

function safe(current state: S): boolean;
var state: S;
begin
  state:= current state;
  complete transactions(state);
  safe:= all transactions completed(state);
end
```

function *safe* does: It simulates the completion of customer transactions as far as possible. If all transactions can be completed, the current state is safe.

In the second refinement (Algorithm 2.2), the state is decomposed into: (1) an array defining the *claim* and *loan* of each customer and whether or not that customer's transactions have been *completed*; and (2) two components defining the *capital* and *cash* of the banker. The exact representation of currencies (type *C*) is still undefined.

The procedure *complete transactions* is now defined in some detail. It examines the transactions of one customer at a time: If they have not already been completed and completion is possible, the procedure simulates the return of the customer's loan to the banker. This continues until no more transactions can be completed.

ALGORITHM 2.2 The Banker's Algorithm (cont.)

```
type S = record
           transactions: array B of
                         record
                           claim, loan: C;
                           completed: boolean;
                         end
           capital, cash: C;
         end
  B = 1. .number of customers;
  C = ?

procedure complete transactions(var state: S);
var customer: B; progress: boolean;
begin
  with state do
  repeat
    progress:= false;
    for every customer do
    with transactions(customer) do
    if not completed then
    if completion possible(claim, cash) then
    begin
      return loan(loan, cash);
      completed:= true;
      progress:= true;
    end
  until not progress;
end
```

The statement

for every *customer* **do** . . .

is equivalent to

for *customer*:= *min*(*B*) **to** *max*(*B*) **do** . . .

Algorithm 2.3 shows that the current state is safe if the banker eventually can get his original capital back.

ALGORITHM 2.3 The Banker's Algorithm (cont.)

```
function all transactions completed(state: S): boolean;
begin
  with state do
  all transactions completed:= capital = cash;
end
```

In the third refinement (Algorithm 2.4), we define the representation of a set of currencies as an array of integers and write the details of the function *completion possible*, which shows that the transactions of a single customer can be completed if his claim of each currency does not exceed the available cash.

ALGORITHM 2.4 The Banker's Algorithm (cont.)

```
type C = array D of integer;
     D = 1. .number of currencies;

function completion possible(claim, cash: C): boolean;
var currency: D;
label no
begin
  for every currency do
  if claim(currency) > cash(currency) then
  begin
    completion possible:= false;
    exit no;
  end
  completion possible:= true;
end
```

Algorithm 2.5 shows the final details of the banker's algorithm.

ALGORITHM 2.5 The Banker's Algorithm (cont.)

```
procedure return loan(var loan, cash: C);
var currency: D;
begin
  for every currency do
  cash(currency):= cash(currency) + loan(currency);
end
```

To be honest, I do not construct my first version of any program in the orderly manner described here. Nor do mathematicians construct proofs in the way in which they present them in textbooks. We must all experiment with a problem by trial and error until we understand it intuitively and have rejected one or more incorrect solutions. But it is important that the final result be so well-structured that it can be described in a step-wise hierarchal manner. This greatly simplifies the effort required for other people to understand the solution.

2.6.3. Conclusion

I have constructed a non-trivial program (Algorithm 2.6) step by step in a hierarchal manner. At each level of programming, the problem is described in terms of a small number of variables and operations on these variables.

ALGORITHM 2.6 The Complete Banker's Algorithm

```
type S = record
           transactions: array B of
                         record
                           claim, loan: C;
                           completed: boolean;
                         end
           capital, cash: C;
         end
     B = 1. .number of customers;
     C = array D of integer;
     D = 1. .number of currencies;

function safe(current state: S): boolean;
var state: S;

  procedure complete transactions(var state: S);
  var  customer: B; progress: boolean;
```

```
        function completion possible(claim, cash: C): boolean;
        var currency: D;
        label no
        begin
          for every currency do
          if claim(currency) > cash(currency) then
          begin
            completion possible:= false;
            exit no;
          end
          completion possible:= true;
        end

        procedure return loan(var loan, cash: C);
        var currency: D;
        begin
          for every currency do
          cash(currency):= cash(currency) + loan(currency);
        end

      begin
        with state do
        repeat
          progress:= false;
          for every customer do
          with transactions(customer) do
          if not completed then
          if completion possible(claim, cash) then
          begin
            return loan(loan, cash);
            completed:= true;
            progress:= true;
          end
        until not progress;
      end

      function all transactions completed(state: S): boolean;
      begin
        with state do
        all transactions completed:= capital = cash;
      end

    begin
      state:= current state;
      complete transactions(state);
      safe:= all transactions completed(state);
    end
```

At the most abstract level, the program consists of a single variable, *current state*, and a single operation, *safe*. If this operation had been available as a machine instruction, our problem would have been solved. Since this was not the case, we wrote another program (Algorithm 2.1), which can solve the problem on a simpler machine using the operations *complete transactions* and *all transactions completed*.

This program in turn was rewritten for a still simpler machine (Algorithms 2.2 and 2.3). The refinement of previous solutions was repeated until the level of detail required by the available machine was reached.

So the design of a program involves the construction of a series of *programming layers*, which gradually transform the data structures and operations of an ideal, non-existing machine into those of an existing machine. The non-existing machines, which are simulated by program, are called *virtual machines* to distinguish them from the physical machine.

It was mentioned in Section 1.1.3 that every program simulates a virtual machine that is more ideal than an existing machine for a particular purpose: A machine that can execute the banker's algorithm is, of course, ideally suited to the banker's purpose. We now see that the construction of a *large program* involves the simulation of a *hierarchy of virtual machines*. Figure 2.10 illustrates the virtual and physical machines on which the banker's algorithm is executed.

At each level of programming, some operations are accepted as *primitives* in the sense that it is known *what* they do as a whole, but the details of *how* it is done are unknown and irrelevant at that level. Consequently, it is only meaningful at each level to describe the effect of

Machine	Operations	Data types
1	*safe*	*S*
2	*complete transactions* *all transactions completed*	*S*
3	*completion possible* *return loan*	*C*
4	Pascal statements	*D*
5	Machine language	Machine types
6	Instruction execution cycle	Registers

Fig. 2.10 The banker's algorithm viewed as a hierarchy of virtual and physical machines.

the program at discrete points in time before and after the execution of each primitive. At these points, the *state* of the sequential process is defined by *assertions* about the relationships between its variables, for example:

"*all transactions completed* $\equiv$ *capital* = *cash*"

Since each primitive operation causes a *transition* from one state to another, a *sequential process* can also be defined as a *succession of states* in time.

As we proceed from detailed to more abstract levels of programming, some concepts become irrelevant and can be ignored. In Algorithm 2.2, we must consider assertions about the local variable, *progress*, as representing distinct states during the execution of the partial algorithm. But at the level of programming where Algorithm 2.2 is accepted as a primitive, *complete transactions*, the intermediate states necessary to implement it are completely irrelevant.

So a state is a partial description of a computation just like the concepts sequential process and operation. A precise definition of these *abstractions* depends on the level of detail desired by the observer. A user may recognize many intermediate states in his computation, but for the operating system in control of its execution, the computation has only a few relevant states, such as "waiting" or "running."

In order to *test* the correctness of a program, we must run it through all its relevant states at least once by supplying it with appropriate input and observing its output. I remarked in Section 2.3.1 that the definition of operations by enumeration of all possible data values is impractical except in extremely simple cases. The same argument leads to the conclusion that exhaustive testing of operations for all possible input values is out of the question.

If we were to test the addition of two decimal numbers of 10 digits each exhaustively, it would require 10^{20} executions of a program loop of, say 10 μsec, or, all in all, $3 * 10^7$ years. The only way to reduce this time is to use our knowledge of the internal *structure* of the adder. If we know that it consists of 10 identical components, each capable of adding two digits and a carry, we also know that it is sufficient to test each component separately with $10 * 10 * 2$ combinations of input digits. This insight immediately reduces the number of test cases to 2000 and brings the total test time down to only 20 msec.

Returning to the problem of testing the correctness of a program such as the banker's algorithm, we must accept that such a test is impossible at a level where it is only understood as a primitive: *safe*. We would have to exhaust all combinations of various currencies and customers in every possible state! But if we take advantage of the layered structure of our programs (see Fig. 2.10), we can start at the machine level and demonstrate once and for all that the machine instructions work correctly. This proof can then be appealed to at higher programming levels independent of the actual data values involved. At the next level of programming, it is proved once and for all that the compiler transforms Pascal statements correctly into machine instructions. And when we reach the levels at which the banker's algorithm is programmed, it is again possible to test each level separately, starting with Algorithm 2.5 and working towards Algorithm 2.1.

In this chapter, I have stressed the need for simplicity in programming. I cannot accept the viewpoint that the construction of programs with a pleasant structure is an academic exercise that is irrelevant or impractical to use in real life. Simplicity of structure is not just an aesthetic pursuit—It is the key to survival in programming! Large systems can only be fully understood and tested if they can be studied in small, simple parts at many levels of detail.

2.7. LITERATURE

This chapter has briefly summarized concepts which are recognized and understood, at least intuitively, by most programmers.

The role of hierarchal structure in biological, physical, social, and conceptual systems is discussed with deep insight by Simon (1962).

In the book by Minsky (1967) you will find an excellent and simple presentation of the essential aspects of sequential machines and algorithms. Horning and Randell (1972) have analyzed the concept "sequential process" from a more formal point of view.

Hopefully, this book will make you appreciate the Pascal language. It is defined concisely in the report by Wirth (1971a).

A subject which has only been very superficially mentioned here is correctness proofs of algorithms. It was suggested independently by Naur and Floyd and further developed by Hoare (1969).

The practice of designing programs as a sequence of clearly separated layers is due to Dijkstra (1971a). He has successfully used it for the construction of an entire operating system (1968). Eventually, this constructive approach to programming may change the field from a hazardous application of clever tricks into a mature engineering discipline.

DIJKSTRA, E. W., "The structure of THE multiprogramming system," *Comm. ACM 11*, 5, pp. 341-46, May *1968*.

DIJKSTRA, E. W., *A short introduction to the art of programming*, Technological University, Eindhoven, The Netherlands, Aug. *1971a*.

HOARE, C. A. R., "An axiomatic basis for computer programming," *Comm. ACM 12*, 10, pp. 576-83, Oct. *1969*.

HORNING, J. J. and RANDELL, B., "Process structuring," University of Newcastle upon Tyne, England, *1972.*

MINSKY, M. L., *Computation: finite and infinite machines*, Prentice-Hall Inc., Englewood Cliffs, New Jersey, *1967.*

SIMON, H. A., "The architecture of complexity," *Proc. American Philosophical Society 106*, **6**, pp. 468-82, *1962.*

WIRTH, N., "The programming language Pascal," *Acta Informatica 1*, **1**, pp. 35-63, *1971a.*