1984

Modularization: a first draft

Alfs T. Berztiss
University of Pittsburgh, uow@berztiss.edu.au

MODULARIZATION: A FIRST DRAFT

Prof. A.T. Berztiss

Department of Computer Science
University of Pittsburgh

Preprint No. 84-8 May 29, 1984

P.O. Box 1144, WOLLONGONG N.S.W. 2500, AUSTRALIA
tel (042)-270-859
telex AA29022

MODULARIZATION: A FIRST DRAFT

1. Introduction

From the very beginning of electronic computation program size has created problems. They have been both conceptual (How am I to understand this fifty page listing?) and physical (Must I recompile 2500 lines of code *after* every little change?). Software engineering has been our response to these problems. Two primary tools of software engineering are data abstraction and modularization, and our purpose here is to try and make them consistent with each other.

Data abstraction encompasses three concepts: identification of data structures with operations, data independence, and encapsulation. The essence of an algebra are its operations, e.g., union and complementation for the algebra of sets. Similarly we now regard the essence of a data structure to be the set of operations associated with it, e.g., *openstack*, *push*, *pop*, *readtop*, and *empty* in the case of a stack. The interpretation of data structures as algebraic systems can be carried as far as expressing the meaning of the data structure operations in the form of axioms. This formal approach is called algebraic specification.

There may be some doubts regarding the practicality and even practicability of algebraic specification, but less formal approaches to specification of data structures can certainly be made a permanent part of the programmer's craft, and the practical utility of data independence has been well established. What the latter means is that programs that make use of data structures should be unaffected by changes in the implementation of these data structures. In other words, a program should communicate with its data structures only by means of procedure and function calls. This enables the effects of any change in the implementation of a data structure to be confined to the implementation alone; we do not have to examine all programs that make use of our data structure for the effects the change might have on these programs. This helps a lot because we would be sure to forget to examine some such programs, we would be sure to miss a place where a change is needed, we would be sure to make erroneous changes.

Encapsulation carries data independence one step further. Under encapsulation the programmer is *forced* to access all data structure implementations by means of function and procedure calls alone. No other means of access are provided. This approach is also called information hiding. The effect of encapsulation is to make the interface of the encapsulated unit with the rest of the program as small as possible.

The term *module* has received many definitions. Here are a few of the more common ones:

(1) A separately compilable unit.

(2) A separately compilable unit kept within certain size limits, e.g., 50 lines of code.

(3) An encapsuled unit, i.e., a unit that presents a small interface to the rest of the program, but need not be identified with a data type.

(4) An encapsuled unit representing the implementation of a particular data type, e.g., of a symbol table.

Our purpose of wanting to combine data abstraction with modularization makes us pick (4) as our definition of a module. Moreover, because we shall deal with both specifications and implementations of modules, we must arrive at an understanding of what is meant by a specification. We adopt the following definition:

> A specification of a module is a description of the interface between the module and a user program that is sufficiently detailed and precise to enable the user to design the program that is to use the module without any knowledge of how the module is implemented, and to enable the module to be implemented without any knowledge of the programs that are to use the module. The specification should also enable us to establish separately the correctness of the module and of the programs that are to use it.

Throughout we shall see the need for compromise. On the one hand, for purposes of verification of the consistency of a module with its specification, and for reusability of a module in a variety of settings, we would prefer a module to be fairly small. On the other hand, we want to keep down the traffic of data across a module interface, and this objective is easier to achieve with large modules.

2. Principles of modularization

Modules have been around almost as long as programming, but, to begin with, their significance was seen exclusively in separable compilation. The idea that modules are not just subroutines found its first significant realization in the design of Simula [Bi73]; the Simula *class* is what we now understand a module to be.

Simula is an extension of Algol60, and, whereas Fortran provides complex numbers as a built-in data type, Algol60 does not. Our example of a Simula class definition will be the class of complex:

```
class complex(re,im); real re,im;
begin
  ref(complex) procedure add(c); ref(complex) c;
    if c=/=none then add:- new complex(re+c.re, im+c.im);
  ref(complex) procedure sub(c); ref(complex) c;
    if c=/=none then sub:- new complex(re-c.re, im-c.im);
  ref(complex) procedure mult(c); ref(complex) c;
    if c=/=none then mult:- new complex(re*c.re-im*c.im, re*c.im+im*c.re);
  ref(complex) procedure conjugate;
    conjugate:- new complex(re,-im);
  ref(complex) procedure stretch(k); real k;
    stretch:- new complex(k*re,k*im);
  real procedure modulus;
    modulus:= sqrt(re*re+im*im);
  ref(complex) procedure div(c); ref(complex) c;
    begin real m;
      if c=/=none then
      begin
        m:= C.modulus;
        if m≠0.0 then
          div:- mult(c.conjugate).stretch(1/m)
      end
    end div;
end complex;
```

Although the procedures of a class declaration were intended to be the only means of access to objects of a class, proper encapsulation did not become a feature of Simula implementations until 1976 or so [Pa76]. However, already in 1972 Parnas had clarified the principles on which to base encapsulation [Pa72a], and perform the decomposition of systems into modules [Pa72b]. These papers have dated remarkably little.

The first of Parnas' papers enunciates the principles of information hiding, and gives five examples of specifications of modules. The first two specify a stack and a binary tree. The other three specify modules of a program for the construction of KWIC indexes. The second paper takes up the KWIC index program, outlines two different modularizations, and discusses why one of the modularizations is preferable to the other. The task consists of accepting a sequence of text lines that are themselves sequences of text words. The lines are to be subjected to circular shifting in which the first word of each line is repeatedly moved from the beginning to the end of the line. In the example below five circular shifts have been produced by this process:

software module specification with examples
module specification with examples software
specification with examples software module
with examples software module specification
examples software module specification with

The output of the program is to be an alphabetized list of all circular shifts of all input lines.

Parnas starts out by defining what he calls a conventional program decomposition into five modules: input, circular shifting, alphabetization, output, and master control. Then he proposes a second decomposition: line storage, input, circular shifting, alphabetization, output, master control. The only obvious difference between them is the addition of the line storage module to the second decomposition, but this change brings about such differences in the specifications of the other modules that in the end very little besides their names remains unchanged.

Let us examine the first decomposition in detail. The input module is to read data lines and store them in fast memory. Characters are to be packed in machine words, and a special word terminating character is to be inserted after every text word. The output of this module is to be the stored text, and an index of line starting addresses. The circular shifting module is to generate an index that gives the address of the first character of each text word of each line paired to the number of the line in which the word occurs. Entries in this index follow the order in which the words of the text are stored, and the purpose of the alphabetization module is to arrange these entries in alphabetical order of the words. The output module is to use the alphabetical word index and the line index to generate an alphabetical listing of all circular shifts. The master control module is to look after the sequencing among the other modules, produce error messages, etc. The essential feature of this decomposition is its linearity: data enter a pipeline, and the master control module pushes the data along this pipeline.

Under the second decomposition each module typically consists of several procedures. The ensemble of procedures that make up the line storage module could contain the following: (a) function *getchar* that is to have for its value a designated character in a designated word of a designated line, e.g., *getchar*(1,2,3) would return the third character of the second word of the first line; (b) procedure *putchar* that is to insert a given character in a designated position of a designated word in a designated line; (c) function *wordcount* that is to have the number of words in a designated line for its value; (d) function *charcount* that is to return the number of characters in a particular word; and so forth. Here, whenever we speak of a word, we mean a text word.

The input module uses the procedures of the line storage module. Parnas' circular shifting module consists of procedures that are analogous to those of the line storage module. Their purpose is to create an impression that every line of *n* words has been replaced by *n* lines, which are the circular shifts of the original line. For example, *csgetchar*(1,2,3) would return the third character of the second word of the first line of this "expanded" table. Of course, the procedures of the circular shift module could be defined in terms of the procedures of the line storage module. Indeed, they should be so defined because then there would be no need for a separate specification; the definition would be its own specification. The procedures of the alphabetization and output modules could similarly be defined in terms of those of the line storage module. The second decomposition is hierarchical in nature, with the line storage module its foundation.

The chief advantage of the second decomposition is that only the line storage module need be concerned with the physical storage of the text. Suppose we were to change the line storage mode, packing a different number of characters to a machine word, or doing without the text word separator. Under the first decomposition such changes would affect every module. Under the second, only the line storage module would have to be changed. It seems that we still have the pipeline aspect: input done before circular shifting, circular shifting before alphabetization, and alphabetization before output. However, now we can make changes among these modules, for example intermeshing alphabetization and output, without any effect on the remaining modules.

Whereas the first decomposition was based on the notion of a sequence of tasks, the basis for the second decomposition is data. The line storage module is a *data structure*, composed of an object--the line store--and a set of operations. The input module creates the object, circular shifting expands it, alphabetization rearranges it. Because we are dealing with an *abstract* data structure, the expansion and rearrangement may be actual, or, alternatively, the same effect may be achieved by means of indexes. The output module merely demands output lines in alphabetical order of circular shifts. Whether there actually exists an expanded table of circular shifts from which to pick up lines for output, or the text of the circular shifts is generated as and when required by means of indexes is immaterial as regards the output module.

Three interesting features of modularization remain to be discussed. The first relates to multiple representations. The input text has to be stored somewhere in its original form, but a circular shift may be merely an index entry. Therefore the table of inputs and the table of circular shifts can differ in kind. This difference becomes largely immaterial when we make the line our primary data object. Then the tables are lists of lines, and the internal representation of a line should be irrelevant. In practice, however, this is not quite so. For example, an equality test for lines raises nontrivial conceptual problems. Modules provide a convenient way of dealing with multiple representations. We stipulate: when a set of modules subsumes a data type, the data type may have different representations from module to module, but not within a module.

This stipulation has a significant effect on module interfaces. Suppose a line is handed from module to module. If representation of lines differs in the two modules, then the interface should take care of the mapping between representations. The problem is how to make this consistent with data independence as far as practicable. One solution is to introduce special procedures that we call *transformers*. They would be interposed between modules, and their purpose would be to map the output of one module into the form expected as input by the other. Transformers would be implementation dependent, i.e., whenever the form of the output from the implementation of a module would change, all transformers associated with the module would have to be rewritten.

Next let us consider pipelining. We mentioned above that the pipeline concept was the basis for the first decomposition. We should note now that a pipeline is more than just a bunch of modules set end to end, and that there are two distinct kinds of modules. Modules of the first kind are data types, modules of the second kind processes. One way of looking at a process is that it accepts an input, and transforms it into an output. In other words, we are dealing with *objects*. A data type, on the other hand, is a set of *operations*. Some appreciation of this distinction can be seen in the second decomposition, but there has not been total separation. If a total separation were to be made, then we would have a set of data types consisting of operations, and a set of processes expressed in terms of invocations of these operations. The processes would then form a pipeline in which the output of one process becomes the input of the next, and--and this is the important property of a pipeline--a receiving process could start to receive and process data before the generating process had completed its work.

Finally, let us be aware of a certain arbitrariness in all of this. The text line seems the most appropriate data type for our problem. However, in deciding on the operations that will constitute this data type, we should look beyond the immediate problem, imagine other contexts in which the data type might come to be used, and try to make the data type very general. Obviously we cannot imagine all future uses of the data type, and the data type will keep evolving. One way of decreasing the costs of this evolutionary process is to select a set of unchanging primitive operations, and define all other operations in terms of these primitives or of operations previously defined in terms of these primitives. But does not then the distinction between operations and processes become somewhat blurred? Moreover, in a broader context, should not the entire KWIC index program be regarded as a primitive operation? These are difficult questions, but we hope that we shall be giving enough insight into the problems associated with modularization to enable one to tackle such questions.

3. A case study: file update

Consider the customer sales file of a company. We shall distinguish between a master file, which constitutes a substantially complete source of information regarding sales by the company, and a temporary transaction file. Each working day the company makes sales and receives payments. Information regarding these transactions goes into the transaction file, each new transaction record being simply appended to the end of the file. At fixed intervals of time, say at the end of each week, the transaction file is used to update the master file. It has then served its purpose, but the next transaction thereafter begins a new transaction file. This file is built up over the next week and used to update the master file again at the end of that week.

Let us now consider the file update problem in completely general terms. Given a *master file* ordered on unique keys, and a *transaction file* ordered on the time of transaction. The temporal ordering of the transaction file may be achieved explicitly by means of time stamps, or implicitly by the sequential order of the transactions in the file. For a record with key *K* in the master file, there may be zero, one, or more than one record with this key in the transaction file. The transaction file may also contain unmatched records, i.e., records whose keys do not occur in the master file. Moreover, a key may temporarily disappear from the master file during an update. Such would be the case with the sequence of transactions

Update 1 for record with key *K*
Update 2 for record with key *K*
Deletion of record with key *K*
Creation of (new) record with key *K*
Update 3 for record with key *K*

Note here that while Updates 1 and 2 relate to one entity, Update 3 is likely to relate to a totally different entity.

There may be more than one transaction file. Then it becomes essential to use time stamps, and a preliminary to the actual file update would be the creation of a single transaction file, either actually or as an abstract object.

The file update problem has a fairly extensive literature. Clean "modern" solutions have been advanced by Dijkstra [Di76] (attributed to W.H.J. Feijen), and Barry Dwyer [Dw81]. Dwyer points out that earlier solutions put too much emphasis on unmatched records, with the result that creation and deletion of records were regarded as essentially different from regular updates. The complexity that this differentiation creates is avoided by considering the entire *space* of possible key values rather than just the keys that have physical records associated with them. On a conceptual level every member of the key space then has a record associated with it. Some of the records will have physical existence, others will be degenerate. A degenerate record consists of just a status marker, which indicates that the record is degenerate, i.e., that its key has not been assigned to a physical record. Creation of a new physical record is then just another update, one that changes a key status from unassigned to assigned, and vice versa for deletion.

So far the organization of the master file has been left undefined. First note a few different ways of dealing with degenerate records. We could have a separate file entry for *every* record, even a degenerate one. An alternative is to provide a bit map to which the key space maps. Keys of degenerate records would be represented by zeros in the bit map, keys of nondegenerate records by ones. The actual file then consists of nondegenerate records alone. The most common approach is to store just the file of nondegenerate records--the absence of a physical record for a given key is sufficient indication of its degeneracy. Secondly, the file could be random or sequential, and this makes a difference to what happens to unchanged records. In a random file they are strictly left alone; in a sequential file they have to be copied from the old to the new version of the master file. (Note, though, that it is foolhardy to start modifying a random master file without having made a copy first.)

Our objective is to design a file updating program that is as independent of the organization of the master file as we can possibly make it. Let us start with an adaptation of Dwyer's description of the file update:

1. Sort transaction file on keys, and on transaction time for records having the same key.
2. Open the files.
3. While there remain keys to process do:
   - 3a. Get the next key.
   - 3b. Get the master record for this key.
   - 3c. While there remain transactions to process for this key, process the transactions, updating the master record. (If the record is

degenerate to begin with, the update creates a new physical record; if a record is nondegenerate, certain fields of it are changed; deletion changes the status of the record to degenerate.)

3d. Insert the updated record in the new master file.

4. Close files and halt.

Most of the activity is in Step 3c, and Dwyer has ensured that this step is independent of the file organization. The file organization would determine the form of Steps 3b and 3d. The form of Step 3a would also be determined by the file organization. Conceptually the "next key" of this step relates to the entire key space, but in practice we would adapt our interpretation to the file organization actually in effect. For a random file the key space would be restricted to the keys in the transaction file alone, for a sequential file to the union of these keys and the keys of the nondegenerate master records.

Levy [Le82] takes this a step further by defining an abstract data object for the problem. His object is an abstract file, which is a collection of grouped records, with one group for each key value. Records are typed: each group contains at most one record of type M (Master), and transaction records of types I (Insert), D (Delete), or C (Change). Because the order of transactions matters (e.g., DCCI makes no sense, because an attempt is made to change a deleted record), records in a group must be ordered. Levy proposes two modules: an input module, corresponding more or less to Steps 3a and 3b above, and an update module corresponding to Steps 3c and 3d. The input module consists of the following operations:

input-open	initialize
new-group	predicate indicating start of a new group
get-record	get next record
input-close	finalize

The output module comprises:

update-open	initialize
start-group	start a new group
insert	add a record for the current group
change	change the record
delete	delete the record
update-close	finalize

In terms of these operations the update program is extremely simple:

```
program update;
   input-open;
   update-open;
   while not eof do
   begin
      get-record(rec);
      if new-group then
         start-group;
      case rec.type of
      M,I:      insert(rec);
      C:        change(rec);
      D:        delete(rec);
      end;
   end;
   input-close;
   update-close
end.
```

Coding of the procedures is to be found in [Le82]. Levy regards the old master and the transaction file as a single composite object, and at first this rather daring approach seems very promising. However, further analysis shows that here we have a case of carrying abstraction both too far and not far enough. On the one hand, because the abstractions are problem-specific, nothing could be salvaged for a different file processing application. On the other, the design embodies the assumption of a sequential file organization. If the master file were in fact random, work would have to be expended on modifications that could easily have been avoided.

A similar approach is taken by Logrippo and Skuce [Lo83]. They view the file update problem as two cooperating sequential processes. The first, which they call *merge*, takes several input files (the old master file, and one or more transaction files), and merges them into a single abstract file that is sorted by key, and for each key contains a sequence of zero or one record of each of the types *master*, *insert*, *change*, *delete* (in this order). It is not at all clear why there cannot be more than one record of type *change*, or, indeed, why a sequence such as *master*, *change*, *change*, *delete*, *insert*, *change*, say, should not be permissible. The second process, called *update*, converts the sequence of records with the same key into a single new master record. Our criticism of Levy's approach holds here as well.

Let us now turn to a different aspect of the file update problem. This is its potential for parallelism. Without attempting to develop a parallel algorithm for the file update, we can still exploit one obvious opportunity for overlapping execution. This is the sorting of the transaction file. It is well known that sorting of n records is an $O(n \log n)$ process, but we also know that some sorting algorithms can produce output well before the sorting process has run to completion. Thus heapsort, after the initial heap creation phase, which is an $O(n)$ process, delivers sorted items at $O(\log n)$ intervals. Unfortunately heapsort is not a stable sort, i.e., it does not maintain the relative order of records sharing the same key, and this makes it unsuitable for our application here. Note that the approaches in which the old master and the transaction file are considered as a single abstract file lend themselves very well to an overlap of the creation of this abstract file and the transformation of the abstract file into the new master file.

We shall see that the easiest approach to the file update program is to use three abstract data types, one each for the old master, the transactions, and the new updated master. In Section 5 we shall present a program based on these three types, which will be designed in such a way that the possible overlap of the sorting of the transaction file and the actual file update will become a mere implementation detail. First, however, we should examine data abstraction in greater detail.

4. Specification

Let us again note that data abstraction is concerned with identification of data structures with operations, data independence, and encapsulation. We shall now look at different ways of specifying the operations that are the essence of a data structure. We want formal specifications as a blueprint for implementations, and as a device that permits us to reason about data structures. Let us begin with the well known and often used examples of a *stack* and a *queue*.

Our first exhibit is an *operational* or *abstract model* specification of a stack of integers, taken from [Be83], in which we have made use (hopefully correct) of the syntax of Alphard [Sh81]:

ALPHARD SPECIFICATION OF A STACK

```
Form Istack(n:Integer) =
    Requires n > 0
    Let Istack = ⟨...x_i...⟩
    Invariant 0 ≤ Length(Istack) ≤ n
    Initially Istack = Nullseq
    Function
        Push(s:Istack,x:Integer)
        Pre 0 ≤ Lengths(s') < n Post s = s' ˜x
        Pop(s:Istack)
        Pre 0 < Length(s') ≤ n Post s = Leader(s')
        Read(s:Istack) Returns (x:Integer)
        Pre 0 < Length(s') ≤ n Post x = Last(s')
        Empty(s:Istack) Returns (b:Boolean)
        Post b = (s'=Nullseq)


        (the implementation part follows)


Endform
```

This specification relies on an underlying domain of sequences: the symbol $\langle ... x_i ... \rangle$ stands for a prototypical sequence, and the functions *Length*, *Leader*, *Last*, concatenation ˜, equality =, and the special function *Nullseq* that returns an empty sequence belong to the data type of sequence. The semantics of the operations are expressed by the pre- and postconditions that follow the domain specifications of the operations. In these conditions the primed symbol *x′* stands for the value of the formal parameter *x* at the beginning of the operation, and *x* for its value at the end. Similar specifications have been proposed by Hoare [Ho72] and King [Ki78]. The former deals primarily with correctness, the latter with implementation.

The specification of one data type (the stack) in terms of another (the sequence) can be regarded as an encroachment on the freedom of action of the implementor. The implementor is forced into using sequences, but the limitations imposed by this constraint are not as serious as they may at first appear. After all, sequences can be implemented in a variety of ways. Nevertheless, a totally "neutral" approach has come to be advocated as an alternative to operational specifications. This is *algebraic* specification, in which the set of axioms that provide the operations of a data type with meaning is self contained.

Let us specify an unbounded queue of integers in algebraic terms, again taking our example from [Be83]:

ALGEBRAIC STACK SPECIFICATION

```
Type Istack
Declare
       New :                   → Istack
      Push : Istack × Integer  → Istack
       Pop : Istack            → Istack
      Read : Istack            → Integer ∪ {error}
     Empty : Istack            → Boolean
For All s ∈ Istack, i ∈ Integer Let
                 Empty(New) = True
         Empty(Push(s,i)) = False
                   Pop(New) = New
           Pop(Push(s,i)) = s
                  Read(New) = error
          Read(Push(s,i)) = i
End Istack
```

The advantage of algebraic specification derives from its mathematical origins. Mathematics is a truly hierarchical science in which new results are derived from previously established knowledge by precisely prescribed methods. Consequently questions of consistency (is our set of axioms without contradictions?) and completeness (is there an interpretation for every syntactically legal composition of operations?) can be addressed with comparative ease in the algebraic framework. On the other hand, algebraic specification of some conceptually simple data types, e.g., the traversible stack [Ma77,Ka79], turns out to be a very difficult task. It has been said, see e.g. [Fl79], that algebraic specifications are well suited for program verification, but that for the verification of implementation correctness the abstract model approach is better. Actually the issue is not as clear cut. Algebraic specification deals with *values* rather than *objects*. For example, the stack [4 3 7 2 is not regarded as an object, but as the value of the functional composition

Qpush (*Qpush* (*Qpush* (*Qpush* (*Qnew*,4),3),7),2)

An operation transforms one value into another; it does not change the state of an object. The implications of this are far reaching. For example, assignment, which associates names with objects, has no place in this scheme of things. Even equality presents problems. Flon and Misra [Fl79] consider two deques, one generated by additions at the head, the other by additions at the tail, and in such a way that if these deques were regarded as objects, they would be indistinguishable. However, an otherwise adequate algebraic specification (consisting of 15 axioms) is incapable of dealing with equality, and has to be augmented with an explicit definition of equality. Bounds present another problem. How is one to specify a bounded stack or queue in terms of values? The concept of *size* makes sense only when we consider an object as a composition of more basic objects (or we deal with measurable physical objects). In light of this the usefulness of the algebraic approach as regards correctness of programs appears to be limited to *functional* programming.

A number of attempts have therefore been made to divorce the abstract model approach from excessive implementation dependence, beginning with the paper of Flon and Misra, and finding a particularly interesting formulation in Claybrook's work [Cl79,Cl82]. Claybrook recognizes a composite data object as precisely that, an aggregate of component objects. This aggregate is viewed as having a *logical structure*, which consists of relationships between the components of an object of the type being defined, a description of the state of such an object, and invariant assertions regarding the above.

For example, in the case of the stack, there is just one relationship, *ontopof*, which is binary and relates elements stored in the stack. Here the state consists of the set S, which is the set of elements stored in the stack, and a relation R, which is the instance of relationship *ontopof* pertaining to the elements in S. An invariant assertion proclaims *ontopof* to be a *linear* relationship. Operations *push* and *pop* are now defined as state changes. Thus the value of *push*(s,e) is a new state of stack s. If the stack was previously empty, S now consists of the single element e, but R still remains empty. If the stack was not empty, then S becomes the union of the original S with e, and R the union of the original R with the ordered pair $\langle e, readtop(s)\rangle$, where the value of *readtop*(s) is the topmost element of the original stack.

The language for writing the data type specifications provides a rather extensive collection of types of relationships for use in expressing invariant assertions [Cl79]. A sample: acyclic, reflexive, symmetric, partially ordered, totally ordered, linear, tree, forest.

Once a data object is regarded as a structured aggregate of data objects of a lower type, the problem of accessing the elements of the lower type in some specified order becomes a valid concern. To take an example that could arise in practice, suppose we have a binary search tree of integers, and all values in the tree are to be scaled by subtraction of the smallest value from every other value. This makes every value non-negative, with the smallest value becoming zero.

Under algebraic specification the standard approach to a traversal of a binary tree is to generate a queue of the integers stored in the binary tree. For our example one would generate a queue of the integers corresponding to a preorder traversal, and take the first element off the queue. One would "unravel" the binary tree, make the necessary changes, and "splice" the tree back together. A recursive applicative program for this is given below. For a more detailed explanation of this program see [Be83].

```
Scale(b,T) is
   if not Empty(b) then Make (Scale(Left(b),T),
                              Data(b) − Qread(Inord(T)),
                              Scale(Right(b),T))
                   else b;
```

Our concern is that we want to be able to combine a traversal with other processing activities. This can be achieved in two ways. One is to have a single procedure for carrying out the entire traversal, and, on getting to each node in turn in accordance with the traversal discipline, to call a procedure that does the processing of the node. This does not work when two traversals are to be carried out at the same time. One algorithm for the strong components of a digraph is based on intermeshed preorder and postorder traversals of a tree [Be80a]. One could define a composite "prepost" traversal, but this would be problem specific, and one of the principles of modularization is to avoid problem specific operations as far as possible.

The other approach is to call a traversal procedure *n* times, where *n* is the number of elements in the structure. Each time the procedure returns either the element or a pointer to the element, and the elements are processed in the calling program. This incremental mode presents interesting implementation problems. In essence, we want a procedure that can halt execution and return to the main program at an arbitrary point in the procedure, and resume execution from this point on the next call. The difficulty lies in having to preserve the state in which the procedure was when the last return from it was made. In a language such as Pascal this has to be achieved by means of global variables, and the procedure cannot be recursive. Here we have an excellent example of the usefulness of coroutines, and of recursive coroutines at that.

Procedure *preorder* exemplifies this mode in Pascal terms. The first call is made with *index* pointing to the root of the binary tree to be traversed, and the Boolean variable *done* having value true. The first execution of *preorder* leaves the value of *index* unchanged, but in subsequent executions its value is changed from a pointer to a particular node to a pointer to the node that follows this node under preorder. As part of the *n*th execution of the procedure the value of *done* becomes true. This signals the completion of the traversal. No test is made for the binary tree being empty. Note that traversal is driven by a stack. Consequently a stack has to be declared in the calling program, and this stack is ferried back and forth between the calling program and procedure *preorder*.

```
procedure preorder(var index: ptype; var done: boolean;
                           var stack: stackhead);
begin
  if done then
    begin
      openstack(stack);
      done:= false
    end
  else
    begin
      if index↑.right<>nil then push(stack,index↑.right);
      index:= index↑.left;
      if (index=nil) and not empty(stack) then
        index:= pop(stack)
    end;
  done:= (index↑.left=nil) and (index↑.right=nil) and empty(stack)
end;
```

In terms of *preorder* and an analogous procedure *inorder* our scaling procedure becomes

```
procedure scale(tree: ptype);
var stack: stackhead;
    scaler: datatype;
begin
  done:= true;
  inorder(tree,done,stack);
```

```
    scaler:= tree↑.datum;
    done:= true;
    repeat
      preorder(tree,done,stack);
      tree↑.datum:= tree↑.datum - scaler
    until done
  end;
```

Note here that the inorder traversal is broken off after reaching the first element in the sequence, at which *point the stack* is *unlikely to* to *be* empty. It is important that *openstack* be so designed that, on being called from *preorder*, it could cope with this situation.

5. Iterators and their synchronization

Let us look at the strong component algorithm of the preceding section in some detail. It is based on a tree representation of a digraph, and goes as follows. Traverse the tree under preorder until a terminal node is reached. Then switch to postorder and traverse the tree under postorder while every node that is reached has already been visited under preorder, switch back into preorder until again a terminal node is reached, switch to postorder, and so forth until both traversals have been completed. In the preorder phase an Action A is performed at every node reached; in the postorder phase an Action B is performed at each node. Actually it is more convenient to use a Knuth transform of the tree. This is a binary tree. The preorder sequence of the general tree is given by preorder traversal of the transform, but the postorder sequence by inorder traversal of the transform. Criteria for switching from traversal to traversal: (1) switch from preorder to inorder when the current node has no left child; (2) "switch from inorder to preorder when the current node has a right child.

We propose that access to the different nodes in the order determined by the particular traversal discipline be provided by *iterators*. An iterator delivers at every invocation the next element of a traversal sequence. Iterators are provided by a number of programming languages, such as Alphard [Sh81] and CLU [Li81], but there they are coupled to a for-loop. This prevents intermeshing of traversals. A generalization of the for-loop by means of a construct called *controlled iteration* permits synchronization of iterators [Be80b], but this construct is incompatible with the very important criterion of simplicity in the design of programming languages. Iterators are also provided by Icon [Gr81], where they are not tied to a for-loop, but the synchronization issue has not been a concern in the design of Icon.

Our solution to the synchronization problem is to associate a set of states with an iterator. State-transition as a programming tool has been investigated by a number of authors (see, e.g., [At79,Ju80,He82], but synchronization based on states seems to be new. For our example we need to define a set of states for the preorder and inorder iterators. Actually we shall use the same set of states in both instances, namely T (terminal node), B (internal node with both children), L (internal node with just a left child), R (internal node with just a right child), D (traversal completed, but the iterator continues to deliver the last item of the traversal sequence to avoid problems with undefined values). Note that we have tried to make the state set independent of the particular application. The fact that the iterator remains active after a traversal has been completed requires explicit opening and closing of iterators.

We also need to allow for the possibility of a structure being subjected simultaneously to several traversals of the same kind. Consequently the definition of an iterator should be separated from declarations of particular instances of this iterator. Further, the nature of the structures from which iterators deliver inputs to a process should be no concern of this process. For example, if we are to merge two sorted input streams to produce a single sorted list, all that matters from the point of view of the merge process is that elements are delivered from the source structures in ascending order of their keys. It makes no difference whether the structures are binary trees, or linear arrays, or one is a binary tree and the other a linear linked list. An instance of an iterator should be an interface between a data structure and a process that ensures total independence of one from the other.

In this framework, assuming *presequence* and *insequence* to have been declared as instances of the iterators *preorder* and *inorder*, respectively, we get the following schematic program for the strong components:

```
open presequence;
open insequence;
repeat
  repeat
    presequence(T,Anode,prestate);
    Action A with Anode
  until (prestate=R) or (prestate=T);
  repeat
    insequence(T,Bnode,instate);
    if instate<>D then Action B with Bnode
  until (instate<>T) and (instate<>L)
until instate=D;
close insequence;
close presequence;
```

Let us now define the iterator *preorder*. In technical terms, an iterator is a semicoroutine [Wa71]. We shall use a syntax that is a slight extension of Pascal. The iterator interrupts execution and returns to the calling program on reaching the deliver-statement. On the next entry to the iterator execution resumes with the statement that follows the deliver-statement. The deliver-statement also indicates the value that the iterator returns. Here the iterator returns a pointer to a node.

```
iterator preorder(binarytree: pointertype; var item: pointertype; var state: statetype);
var stack: stackrecord;
   node: pointertype;
begin
  if binarytree=nil then
    begin item:= nil;
          state:= D
    end
  else
  begin
    openstack(stack);
    push(stack,binarytree);
    repeat
      node:= pop(stack);
      repeat
        if node↑.right<>nil then
          begin
            push(stack,node↑.right);
            if node↑.left<>nil then
              state:= B
            else
              state:= R
          end
        else if node↑.left<>nil then
          state:= L
        else
          state:= T;
        repeat (* stay in this loop after state becomes D *)
          deliver item:= node;
          if (node↑.left=nil) and empty(stack) then
            state:= D
        until state<>D;
        node:= node↑.left;
      until node=nil
    until false (* this is a do-forever *)
  end
end;
```

In terms of iterators procedure *scale* becomes

```
procedure scale(tree: pointertype);
var insequence: inorder;
    presequence: preorder;
    instate,prestate: statetype;
    node: pointertype;
    scaler: datatype;
begin
  open insequence;
  insequence(tree,node,instate);
  scaler:= node↑.datum;
  close insequence;
  open presequence;
  presequence(tree,node,prestate);
  while prestate<>D do
  begin
    node↑.datum:= node↑.datum - scaler;
    presequence(tree,node,prestate);
  end;
  close presequence
end;
```

Let us now *return to the file update.* *We propose two input iterators,* one for the old master, the other for the transaction file (or files), and name the instances of the iterators to be used in our program *nextmaster* and *nexttrans*, respectively. Both iterators *return* records. There are two state indicators for the transaction iterator: *transtype*, with values {I,C,D}, indicates whether the transaction is an insertion (I), change (C), or deletion (D); *keytype*, with values {F,N,D}, indicates whether the transaction is the first of a set sharing the same key (F), or the last item of the iteration sequence has been delivered (D, in which case the iterator continues delivering this last item), or it is neither of the above (N).

The state set of the master file iterator is {M,N,D}, and these states indicate whether the key of the record being delivered matches the supplied key (M), or does not match the supplied key (N), or all records of the master file have already been delivered (D). The program as displayed below has been designed for a sequential master file. If the master is a random file, then two groups of four lines of code have to be removed (they are marked with asterisks), and the master file iterator has to be rewritten. The transaction iterator generates a sorted input stream (possibly from more than one transaction file). The generation of the input stream can well proceed in parallel with the actual update, but then we need a mechanism for waits in case the iterator cannot produce the input stream at as fast a rate as it is consumed.

```
open nextmaster;
open nexttrans;
nexttrans(transaction,T,transtype,keytype);
repeat
  nextmaster(master,R,T.key,state);
  while R.key<T.key do                           (***)
  begin putrecord(newmaster,R);                  (***)
        nextmaster(master,R,T.key,state)         (***)
  end;                                           (***)
  if state=M then
    newrecord:= R
  else if transtype<>I then
    ERROR CONDITION;
  transferswitch:= true;
  repeat
    case transtype of
    I:    begin newrecord:= T;
                transferswitch:= true
          end;
    C:    make changes to newrecord;
    D:    begin do deletion bookkeeping;
                transferswitch:= false
          end
    end;
    nexttrans(transaction,T,transtype,keytype);
  until (keytype=F) or (keytype=D);
  if transferswitch then putrecord(newmaster,newrecord)
until keytype=D;
while state<>D do                                (***)
begin nextmaster(master,R,T.key,state);          (***)
      putrecord(newmaster,R)                     (***)
end;                                             (***)
close nexttrans;
close nextmaster;
```

We indicated earlier that here we are dealing with three data types. We could just as well decide to regard *master*, *newmaster*, and *transaction* as three objects belonging to a single data type with which we would associate our two iterators and the procedure *putrecord*. This, however, would reduce flexibility. Whereas *master* and *newmaster* correspond to single physical files, *transaction* is an abstract object that could correspond to more than one file.

Finally note that legal sequences of transactions can be described by a state transition diagram:

$$\begin{tikzcd} I \arrow[r, bend left] \arrow[rr, bend left=50] & D \arrow[l, bend left] & C \arrow[l] \arrow[loop right] \end{tikzcd}$$

Given a group of transaction records for the same key. If there exists a master record with this key, then the first transaction has to be of type C or D; if no master record exists, the first transaction has to be of type I. Our program above checks that we start off correctly. After that, only the state transitions indicated by the diagram are valid, and a validity check would have to be built into the transaction iterator.

6. Case study: two-way merge

Let us now consider another fairly complicated example of modularization. Our context will be two-way merge as a basis of external sorting. The primary activity in external sorting is the merging of sorted files to produce larger sorted files. We shall therefore first examine an *internal* sorting procedure based on merging, called *merge sort*. Merge sort relates to efficient implementation of set operations, which gives it additional interest. Set operations, such as union and intersection, take much longer when the operations are carried out on sets with unordered elements than when the elements are ordered. Assume that vector *A* contains the *n* elements of *setA*, and that vector *B* contains the *m* elements of *setB*. Algorithm 1 is a procedure that merges elements of *A* and *B* into *C*. We shall use the ideas developed in Section 5, but programming will be in conventional Pascal throughout.

ALGORITHM 1. *A procedure for merging* integers stored in order of magnitude in *A* and *B*. Arrays *A* and *B* are assumed to be of the same type, with subscript range [1..n], and all elements except the first *Atop* of *A* and the first *Btop* of *B* assumed to contain the value −maxint (it is assumed that *setA* and *setB* cannot legitimately contain an element having this value). The result is returned in array *C*, which is of the same type as *A* and *B*. In *C* all elements except the first *Atop*+*Btop* are set to −maxint by the procedure. Availability is assumed of *errorprocedure*, which handles the case of *Atop*+*Btop* exceeding n.

```
procedure merge(var A,B,C: setarray);
var i,j,k,Atop,Btop,index: integer;
  function size(var sourceset: setarray): integer;
  var top: integer;
      alldone: boolean;
  begin
    top:= 0;
    repeat
      if top<n then
        alldone:= sourceset[top+1]=-maxint
```

```
            else
               alldone:= true;
            if not alldone then top:= top+1
         until alldone;
         size:= top
      end;
   begin
      Atop:= size(A);
      Btop:= size(B);
      if Atop+Btop>n then errorprocedure
      else if Atop=n then C:= A
      else if Btop=n then C:= B
      else
      begin
         i:= 1; j:= 1; k:= 1;
         repeat
            if A[i]<B[j] then
               begin C[k]:= A[i];
                     k:= k+1;
                     i:= i+1
               end
            else
               begin C[k]:= B[j];
                     k:= k+1;
                     j:= j+1;
               end;
         until (i>Atop) or (j>Btop);
         if i>Atop then
            for index:= j to Btop do
            begin C[k]:= B[index];
                  k:= k+1
            end
         else
            for index:= i to Atop do
            begin C[k]:= A[index];
                  k:= k+1
            end;
         for index:= k to n do
            C[index]:= -maxint
      end
   end;
```

The double traversal of arrays *A* and *B*, first to count the elements in the sets, then to do the actual merging, may seem a duplication of effort, and hence a waste of program execution time. This is not so. In any other design the equivalent of the test *Atop+Btop*>n of Algorithm 1 is spread throughput the procedure, and in fact takes more time. Moreover, the procedure becomes more difficult to understand.

Suppose now that the input to the merge is to be from magnetic tapes (or magnetic disk). Suppose we have four tape drives at our disposal, named *A*, *B*, *C*, *D*, and suppose further that the file to be sorted resides on tape *C*, with keys c_1, c_2, ..., c_k. For example, the keys of the input file could be

```
C:  19   1  26  43  92  87  88  26  17  34  69
```

The first stage is to distribute the records to tapes *A* and *B*:

```
i:= 1;
outputswitch:= true;
while i<=k do
begin
   if outputswitch then
      output($c_i$,A)
   else
      output($c_i$,B);
   i:= i+1;
   if i<=k then
      if $c_i<c_{i-1}$ then
         outputswitch:= not outputswitch
end;
```

At the end of the initial distribution we have

```
A:  19  87  88  17  34  69
B:   1  26  43  92  26
```

If tape *B* were now empty, we would be finished. Note that we are taking advantage of the natural order that may already exist in the input file. In our case we can distinguish five *runs*, i.e. blocks of keys that are already in order, namely (19), (1,26,43,92), (87,88), (26), (17,34,69). After the initial distribution there are two runs on *A* and two on *B*.

The next stage is to merge records from tapes *A* and *B* onto tapes *C* and *D*. The program for this task is too complicated to be written without a thorough preliminary analysis. We shall regard this analysis as a case study in program modularization.

For generality we shall regard *A* and *B* as *input streams*, which could be tape files, arrays holding a known number of elements, or arrays in which the convention of Algorithm 1 is followed. We propose that at any one time the input stream will be in one of a small number of states. Four states are sufficient: F(inished), L(ast), H(old), and N(ormal). The input stream is in state F when all records from it have been used up, or it was empty to begin with. It is in state L when the last record of a run is being looked at. State H is rather special. Suppose an output run is being generated by merging input runs from the two input streams, that all records from the first input stream have already been transferred into the output, but that transfer of the run from the other input stream has not yet been completed. The first input stream is then in state H. An input stream is in state N when it is not in one of the other states.

Table 1 shows all combinations of states in which the pair of input streams, *A* and *B*, can be found, and the action that is to be followed in each instance. If the state pair is LL, LN, NL, or NN, the transfer into the output stream can be from either input stream, depending on which stream holds the record with the smallest key value. When the state pair is LH, LF, HL, or FL, then the record that is now transferred into the output stream closes off an output run, and the next run will be built up as part of the other output stream. We can now build up a gigantic case statement to parallel the actions of Table 1. This we enclose in a procedure *moverecord*:

```
procedure moverecord;
var r: mergerecord;
begin
  case stateA of
  L: case stateB of
     L: if K(A)<K(B) then
             begin getrecord(A,r,stateA);
                   if stateA<>F then stateA:= H
             end
           else
             begin getrecord(B,r,stateB);
                   if stateB<>F then stateB:= H
             end;
     H: begin getrecord(A,r,stateA);
              stateB:= N;
              outswitch:= true
           end;
```

TABLE 1
STATE-ACTION TABLE FOR A TWO-WAY MERGE

AB	Action
LL	If input from *A*, change its state to H or F; if input from *B*, change its state to H or F
LH	Input from *A*: change its state to N, F, or L (arises if next run consists of just one record); change state of *B* to N; switch to other output stream
LN	If input from *A*, change its state to H or F; if input from *B*, keep its state unchanged, or change it to L
LF	Input from *A*: change its state to N, F, or L; switch to other output stream
HL	Input from *B*: change its state to N, F, or L; change state of *A* to N; switch to other output stream
HH	Cannot arise
HN	Input from *B*: keep its state unchanged, or change it to L
HF	Cannot arise
NL	If input from *A*, keep its state unchanged, or change it to L; if input from *B*, change its state to H or F
NH	Input from *A*: keep its state unchanged, or change it to L
NN	If input from *A*, keep its state unchanged or change it to L; if input from *B*, keep its state unchanged or change it to L
NF	Input from *A*: keep its state unchanged, or change it to L
FL	Input from *B*: change its state to N, F, or L; switch to other output stream
FH	Cannot arise
FN	Input from *B*: keep its state unchanged, or change it to L
FF	HALT

```
        N: if K(A)<K(B) then
              begin getrecord(A,r,stateA);
                    if stateA<>F then stateA:= H
              end
           else
              getrecord(B,r,stateB);
        F: begin getrecord(A,r,stateA);
                 outswitch:= true
           end
        end;
    H: case stateB of
        L: begin getrecord(B,r,stateB);
                 stateA:= N;
                 outswitch:= true
           end;
        N: getrecord(B,r,stateB)
        end;
    N: case stateB of
        L: if K(A)<K(B) then
              getrecord(A,r,stateA)
           else
              begin getrecord(B,r,stateB);
                    if stateB<>F then stateB:= H
              end;
        H: getrecord(A,r,stateA);
        N: if K(A)<K(B) then
              getrecord(A,r,stateA)
           else
              getrecord(B,r,stateB);
        F: getrecord(A,r,stateA)
        end;
    F: case stateB of
        L: begin getrecord(B,r,stateB);
                 outswitch:= true
           end;
        N: getrecord(B,r,stateB)
        end
    end;
    if outC then
       putrecord(C,r)
    else
       putrecord(D,r);
    if outswitch then
```

```
        begin outC:= not outC;
              outswitch:= false
        end
      end;
```

Procedure *moverecord* is completely independent of the form of the files that are to be processed. The interface to the files consists of procedures *getrecord* and *putrecord*, and function *K*. These routines provide access to the files, are therefore dependent on the actual form of the files, and a new set has to be written each time the representation of the files is changed.

To provide an example, we shall consider a file in which the main field is *stream*, an array of m records. This is the actual file. In addition to the array, the record holds fields *size*, which tells how many of the *m* elements of *stream* are actually occupied by the file, and *index*, which indicates the record of the file that is currently being accessed. An appropriate set of declarations:

```
const m = .....;
type keytype = integer;
     mergerecord = record
                     key: integer
                   end;
     mergestream = record
                     size,
                     index: integer;
                     stream: array[1..m] of mergerecord
                   end;
     statetype = (L,H,N,F);
```

Procedure *moverecord* calls procedures *getrecord* and *putrecord*, and function *K*. Procedure *getrecord* returns the record to which *index* points, and increments *index*. It also sets the state indicator of the file to F, L, or N. We have made *size* and *index* part of our file representation, but not the state indicator. This is so because its setting does not depend entirely on the file by itself. The indicator can have the fourth value H, and this setting depends in part on the current state of the other file. Procedure *putrecord* increments the value of *size*, and assigns the record that is one of its arguments to the location in *stream* defined by the value of *size*. Function *K* returns the key of the record indicated by *index*. In our example a record consists of just the one field *key*.

In addition to these routines we need procedure *open*, which initializes *index* of a non-empty file to 1, and also initializes the state indicator, procedure *ready*, which sets *size* to zero, i.e., turns a file into an empty file, and procedure *close*, which does some final processing after a file has received all its data. In our example *close* is a vacuous procedure, but not in general. For example, under the convention of Algorithm 1 it would fill in the unused part of the array with −maxint. Procedures *open* and *ready* correspond to the Pascal file procedures *reset* and *rewrite*. Indeed, if our streams were Pascal files, then procedure *ready* would consist of just a call to *rewrite*.

Routines *getrecord* and *K* invoke procedure *errorprocedure* when the value of *index* exceeds that of *size*. This procedure has been left undefined.

```
procedure getrecord(var a: mergestream; var item: mergerecord; var state: statetype);
begin
   if a.index>a.size then
      errorprocedure
   else with a do
   begin item:= stream[index];
         index:= index+1;
         if index>size then
            state:= F
         else if index=size then
            state:= L
         else if stream[index].key>stream[index+1].key then
            state:= L
         else
            state:= N
   end
end;

procedure putrecord(var a: mergestream; var item: mergerecord);
begin
   with a do
   begin size:= size+1;
         stream[size]:= item
   end
end;
```

```
function K(var a: mergestream): keytype;
begin
  with a do
  begin if index>size then
           errorprocedure
        else
           K:= stream[index].key
  end
end;

procedure open(var a: mergestream; var state: statetype);
begin
  with a do
  begin if size=0 then
           state:= F
        else
           begin if size=1 then
                    state:= L
                 else if stream[1].key>stream[2].key then
                    state:= L
                 else
                    state:= N;
                 index:= 1
           end
  end
end;

procedure ready(var a: mergestream);
begin
  a.size:= 0
end;

procedure close(var a: mergestream);
begin
end;
```

We are now ready to put the bits and pieces together into the procedure *twowaymerge*:

```
procedure twowaymerge(var A,B,C,D: mergestream);
var outC,outswitch: boolean;
    stateA,stateB: statetype;
  procedure getrecord ......
    ... ... ... ... ...
  procedure putrecord ......
    ... ... ... ... ...
  function K ......
    ... ... ... ... ...
  procedure open ......
    ... ... ... ... ...
  procedure ready ......
    ... ... ... ... ...
  procedure close ......
    ... ... ... ... ...
  procedure moverecord ......
    ... ... ... ... ...
    ... ... ... ... ...
begin
  open(A,stateA); open(B,stateB); ready(C); ready(D);
  outswitch:= false;
  outC:= true;
  while (stateA<>F) or (stateB<>F) do
    moverecord;
  close(C); close(D)
end;
```

Returning to the sorting example we started with, recall that the initial distribution stage has distributed records to streams *A* and *B*. The next stage is to merge records from *A* and *B* onto *C* and *D*. If now *D* is empty, we have finished. Otherwise records are merged from *C* and *D* back onto *A* and *B*. If *B* is now empty, we have finished; otherwise the next merge is again from *A* and *B* onto *C* and *D*, and so forth.

Procedure *twowaymerge* can be used for the initial distribution as well. In terms of this procedure a complete merging program takes the following form:

```
D.size := 0;
twowaymerge(C,D,A,B);
directionswitch:= true;
repeat
   if directionswitch then
      begin twowaymerge(A,B,C,D);
            done:= (D.size=0);
      end
   else
      begin twowaymerge(C,D,A,B);
            done:= (B.size=0);
      end
   directionswitch:= not directionswitch
until done;
```

7. Modules in Ada

Let us express the type complex of Section 2 as an Ada [US83] *package*. Packages can provide true encapsulation, and in our example we shall completely separate use from implementation by making the latter "private". This is not just of academic interest. A complex number can be implemented as a record of two reals, but it can just as easily be implemented as a two-element array of reals. Programs that make use of type complex should be unaffected by the substitution of one implementation for another. This is achieved by hiding everything regarding the implementation from the view of the user. Such implementation independence does, however, mean that we require to augment the set of operations defined in Section 2. In particular, we need a constructor function that builds a complex number from two reals, and two extractors, for the real and imaginary parts, respectively.

```
package complex_numbers is
   type complex is private;
   function "+" (a,b: complex) return complex;
   function "-" (a,b: complex) return complex;
   function "*" (a,b: complex) return complex;
   function conjugate (a: complex) return complex;
   function stretch (c: real; a: complex) return complex;
   function modulus (a: complex) return real;
   function "/" (a,b: complex) return complex;
   function build (c,d: real) return complex;
   function re_part (a: complex) return real;
   function im_part (a: complex) return real;
private
   type complex is
      record
         re,im: real;
      end record;
end;

package body complex_numbers is
```

```
    function "+" (a,b: complex) return complex is
    begin
      return (a.re+b.re, a.im+b.im);
    end "+";
    ... ... ...
    ... ... ...
    function build(c,d: real) return complex is
    begin
      return (c,d);
    end build;
    ... ... ...
    ... ... ...
  end complex_numbers;
```

The first part of this program text is the package "specification", which is the interface between the package and the program that uses it. This is the visible component. It differs from the specifications we discussed in Section 4 in a significant way--it specifies the types of the arguments and the results of the operations, but does not define their meaning. The second component, the package body, provides the operations with meaning, but, since this is an implementation, it is not an *independent* specification in the sense of Section 4 either.

For another example of an Ada package let us take the stack. Again we shall make sure that the only access to the data is by the stack operations, and, to ensure this, shall make the stack a private type. We shall go even further. For unqualified private types assignment and equality tests are still available; for *limited* private types even these facilities are left to the devices of the implementor.

```
package stacks is
  type stack is limited private
  procedure openstack (s: in out stack);
  procedure push (s: in out stack; x: item);
  procedure pop (s: in out stack);
  function readtop (s: stack) return item;
  function empty (s: stack) return boolean;
private
  bound: constant:= 100;
  type stack is
    record
      s: array(1..bound) of item;
      top: integer range 0..bound;
    end record;
end;

package body stacks is
   procedure openstack (s: in out stack) is
   begin
      s.top:= 0;
   end openstack;
   procedure push (s: in out stack; x: item) is
   begin
      s.top:= s.top+1;
      s.s(top):= x;
   end push;
   ... ... ...
   ... ... ...
end stacks;
```

One of the problems with a language such as Pascal is that if we wished to use the same stack module for storing data of different types, we would have to make a separate copy of the text of the module for each data type. Ada has a *generic* definition mechanism that avoids this difficulty. We would start the package specification with

```
generic
   bound: posinteger;
   type item is private
package stacks is
```

and remove the statement

```
bound: constant:= 100;
```

Then, to create a stack of integers of size 150, say, we *instantiate* the generic package as follows:

```
declare
   package int_stack is new stack(150,integer);
   use int_stack;
   s: stack;
```

8. Bibliography

At79 Atkinson, L.V., Pascal scalars as state indicators. *Software--Practice and Experience* **9**, 427-431 (1979).

Be80a Berztiss, A.T., Depth-first K-trees and critical path analysis. *Acta Inf.*, **13**, 325-346 (1980).

Be80b Berztiss, A.T., Data abstraction, controlled iteration, and communicating processes. *Proc. ACM Annual Conf. 1980*. ACM, New York, 1980, pp.197-203.

Be83 Berztiss, A.T., and Thatte, S., Specification and implementation of abstract data types. *Advances in Comput.* **22**, 295-353 (1983).

Bi73 Birtwistle, G.M., Dahl, O.-J., Myhrhaug, B., and Nygaard, K., *Simula Begin*. Studentlitteratur, Lund, 1973.

Cl79 Claybrook, B.G., Cleaveland, J.C., and Criscione, D., Logical structure specification and data type definition. *Proc. ACM Annual Conf. 1979*. ACM, New York, 1979, pp.203-211.

Cl82 Claybrook, B.G., A specification method for specifying data and procedural abstractions. *IEEE Trans. Sofware Eng.* **SE-8**, 449-459 (1982).

Da68 Dahl, O.-J., Myhrhaug, B., and Nygaard, K., The Simula67 common base language. Norwegian Computing Centre Report, Oslo, 1968.

Di76 Dijkstra, E.W., *A Discipline of Programming*. Prentice-Hall, Englewood Cliffs, New Jersey, 1976.

Dw81 Dwyer, B., One more time--how to update a master file. *CACM* **24**, 3-8 (1981).

Fl79 Flon, L., and Misra, J., A unified approach to the specification and verification of abstract data types. *Proc. Conf. Spec. Reliable Software 1979*. IEEE Computer Society, New York, 1979.

Gr81 Griswold, R.E., Hanson, D.R., and Korb, J.T., Generators in Icon. *ACM Trans. Program. Lang. Syst.* **3**, 144-161 (1981).

He82 Hext, J., and Hirst, S., The formal treatment of state transition tables--a tutorial. *Austral. Computer J.* **14**, 1-6 (1982).

Ho72 Hoare, C.A.R., Proof of correctness of data representations. *Acta Inf.*, **1**, 271-281 (1972).

Ju80 Juliff, P., Program control by state transition tables. *Austral. Computer J.* **12**, 146-152 (1980).

Ka79 Kapur, D., Specifications of Majster's traversable stack and Veloso's traversable stack. *SIGPLAN Notices* **14**, no.5, 46-53 (May 1979).

Ki78 King, P.R., On the specification and design of abstract data types. In *Constructing Quality Software* (P.G. Hibbard and S.A. Schuman, eds.), North-Holland, Amsterdam, 1978, pp.449-470.

Le82 Levy, M.R., Modularity and the sequential file update problem. *CACM* **25**, 362-367 (1982).

Li81 Liskov, B., *et al.*, *CLU Reference Manual*. Lect. Notes Comput. Sci. No. 114, Springer-Verlag, Berlin, 1981.

Lo83 Logrippo, L., and Skuce, D.R., File structures, program structures, and attributed grammars. *IEEE Trans. Sofware Eng.* **SE-9**, 260-266 (1983).

Ma77 Majster, M., Limits on algebraic specification of abstract data types. *SIGPLAN Notices* **12**, *no.*10, 37-42 (Oct.1977).

Pa72a Parnas, D.L., A technique for software module specification with examples. *CACM* **15**, 330-336 (1972).

Pa72b Parnas, D.L., On the criteria to be used in decomposing systems into modules. *CACM* **15**, 1053-1058 (1972).

Pa76 Palme, J., New feature for module protection in Simula. *SIGPLAN Notices* **11**, no.5, 59-62 (May 1976).

Sh81 Shaw, M., ed., *Alphard: Form and Content*. Springer-Verlag, New York, 1981.

US83 U.S. Department of Defense, *Reference Manual for the Ada Programming Language*. Springer-Verlag, New York, 1983.

Wa71 Wang, A., and Dahl, O.-J., Coroutine sequencing in a block structured environment. *BIT* **11**, 425-449 (1971).