PROCEEDINGS OF THE SYMPOSIUM ON
LANGUAGE DESIGN AND PROGRAMMING METHODOLOGY
SYDNEY, 10-11 SEPTEMBER, 1979

THE MODULE : A SYSTEM STRUCTURING FACILITY IN HIGH-LEVEL PROGRAMMING LANGUAGES

Niklaus Wirth
Institut für Informatik
ETH Zürich

ABSTRACT

The key to successful programming is finding the "right" structure of data and program. A programming language concept called module is presented here as a means to partition systems effectively. The module allows to encapsulate local details and to specify explicitly those elements that are part of the interface between modules.

Modules as presented here are part of the language Modula-2. Problems of separate compilation and of splitting a module into definition (interface) and implementation parts are discussed.

* Institut für Informatik
ETH-Zentrum
CH-8092 Zürich

Introduction

The predominant factor influencing the field of computing today is the advent of the microprocessor and of LSI semiconductor technology in general. Depending on one's point of view, it is assessed as positive or negative; in any case it is exciting. I have recently heard the remark that it has set back the practice of programming by some 20 years. Considering the resurgence of the use of rather primitive techniques of assembly and even numeric coding one tends to agree.

Yet I prefer to see recent trends in an opposite perspective. The microprocessor spreads the availability of computing in an unforeseen degree. Hence, it also spreads the need for programming know-how and tools. In fact, it makes adequate programming techniques not only an advantageous convenience but an indispensable asset. Economically it becomes the longer the more penny-wise and pound-foolish to use antiquated programming techniques in order to save a few cheap chips. However, we must not make the mistake of adopting this doctrine to a degree that we ignore possibilities to use modern programming methods and save chips at the same time. This, I think, is the true challenge of computer engineering today.

Programming appears as an activity in many contexts. When I subsequently use the word "programming", I refer to the construction of relatively complex systems for long-time use, rather than the formulation of a small program which is discarded and forgotten after the intended result has been computed. As such, programming is computer engineering.

Computers, i.e. the hardware, are unique among the products of modern technology in many respects, but perhaps their most unique characteristic is their incompleteness. A so-called complete hardware system is an utterly useless gadget without software, i.e. programs. Each program extends the hardware creating that machine which is capable of computing or behaving in the desired manner. Hence programming is constructing that machine, starting from the basis of some given multipurpose elements. These elements may indeed be specific hardware components of a given kind. But the modern view is that they are the constructs which are offered by a programming language. Hence we tend to abstract from a given concrete machine and to work with idealized building blocks. We have recognized that abstraction allows us to suppress (to hide) details that are irrelevant to our ultimate goal, a process that is indispensable in the construction of complex machinery. The division of a computing system into hardware and software was and is the most clear-cut borderline that hides details (of the hardware from the software and vice versa). It appears that we need a large number of such borderlines, and that the one usually given between hard- and software is by no means a good one.

As a side remark, let me point out that the mentioned incompleteness of computers is also the key to their flexibility and universality. As it is, against widespread beliefs, much more costly to realize a given system entirely in hardware than partly with software, the tendency is to implement as hardware only those parts that are very universally applicable. Therefore it is possible to produce components in very large quantities necessary to become cost-effective. As a consequence, the trend is to increase the percentage of a system implemented by software. And _this_ is the reason for steadily increasing software cost and decreasing hardware cost.

I have tried to motivate my view of (system) programming languages as abstract tool sets for the construction of computing machinery. This is in strong contrast to the view of a language as a medium of communication between man and machine. Admittedly this latter view is not only more common but has also coined the term "programming language", which in my opinion is ill-chosen and misleading. "Program notation" would be eminently more appropriate.

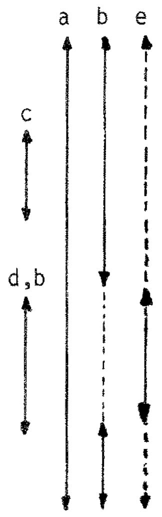
Historical development

Let us now consider in which ways programming languages aid the system designer in his task to develop complex machinery, and in particular to hide details of some part from other parts.

The first step with this goal in mind has been the introduction of subroutines in Fortran. Their purpose had always been proclaimed as avoidance of code replication (and saving the programmer from having to write so much!). But equally important is the fact that a subroutine can be accompanied with objects (variables) that are unknown to other subroutines.

This aspect was clearly recognized by the designers of Algol 60. It resulted in the concepts of locality and block structure. It is noteworthy that a necessary ingredient was the insistence on explicit declaration of objects, a rule that has been heavily criticized by Fortran programmers, but nevertheless has proved to be extremely helpful in discovering mistakes at an early stage by the compiler. Blocks in Algol 60 are properly nested, i.e. each block is fully contained in another one (except of course the outermost block).

```
BEGIN INTEGER a, b ;
     S1 ;
     BEGIN REAL c ;
          S2
     END ;
     S3 ;
     BEGIN BOOLEAN d, b ;
           OWN INTEGER e ;
          S4
     END ;
     S5
END
```



In Algol 60, visibility and existence are intimately connected: Whenever an object is visible, it exists (i.e. is allocated) and vice-versa. There are, however, two exceptions to this intentional rule, one being accidental, the other planned but unsatisfactory. The accidental exception is shown in Fig. 1 by the integer variable b: in statement S4 it is invisible, although it exists, because the identifier b is there taken to denote a local Boolean variable. The intended exception is the concept of <u>own</u> variables. As it proved to be an unsatisfactory solution, we shall not pursue it any further here.

Block structure proved to be an extremely important concept and was adopted by virtually all subsequently designed programming languages, notably PL/I and Pascal.

Yet it became apparent that Algol 60's intriguingly simple rule of "whenever an object exists, it is also visible" was inappropriate in certain situations. Simula was the language that presented a construct which departed from this rule, although the language, based on Algol 60, retained most of its ancestor's characteristics [2] . I refer to the concept of the <u>class</u>. A class is defined similar to a procedure; it specifies a set of local objects, a set of parameters and a statement body. In contrast to the procedure, however, the statement body represents a coroutine, i.e. a process that is executed concurrently with other processes in the realm of abstraction, but in time-interleaved fashion in reality. Hence, a coroutine can be suspended and later resumed. During the period of suspension, its local objects exist, but are invisible. This rule follows very naturally from the premise that one process' local data should not be manipulated by other processes (even when suspended). The intended exception from the rule is provided by the external accessibility of a class' parameters.

The class structure of Simula was inspired by particular requirements of discrete event simulation. From the point of view of programming language theory it represents a feature which combines several distinct concepts, and this conglomeration makes it rather difficult to understand. With respect to Algol, the novel concepts introduced by the Simula class feature are

- concurrency (or quasi-concurrency) of execution
- definition of a class, i.e. of a template of data objects associated with executable statements vs. creation of instances of this class
- distinction between visibility and existence of objects
- access of class instances via pointers, and explicit facilities to manipulate pointers (instead of referenced objects).

The process of disentangling these concepts took considerable time in the history of language development, as it was only slowly recognized that such a separation of issues would add useful flexibility to languages.

Algol W introduced the class idea into a purely sequential language and disconnected the data template from its association with a body of executable statements. This resulted in the so-called record class [4, 8] . The Algol W class declaration defines a template of data (records) without associated statements. However, it retains the rule that instances of this record class are accessed exclusively via explicit pointers, whereas all other data are accessed directly.

The following are examples of Algol W record class declarations:

```
RECORD Node (REFERENCE (Node) left, right)
RECORD Person (STRING name;
               INTEGER age;
               LOGICAL male;
               REFERENCE (Person) father, mother)
```

An instance of the class is generated by use of the class name and argument of the generated element's pointer to a reference variable:

```
REFERENCE (Node) root
REFERENCE (Person) p
root := Node (NIL, NIL)
p := Person ("Albert", 0, TRUE, p1, p2)
```

Pascal went a step further, disentangling also the concepts of access, structure, and instantiation of data: all variables can be either accessed directly or via pointers, be unstructured or structured (i.e. be arrays, records, or sets), be defined as a single instance or by a template of which many instances can be created. The record class, in fact, led to the generalization of the template declaration, called type definition in Pascal [9].

The gradual process of disentangling these distinct concepts led to one definitely remarkable discovery. C.A.R. Hoare and P. Brinch Hansen recognised that the association of data and procedure declaration as embodied in the class concept would be an ideal frame for promoting the idea of program design with various levels of abstraction. If the class body is regarded as an initialization statement of local data instead of as a coroutine, and if the procedures are made globally accessible instead of local to the coroutine body, and if the associated variables are local, i.e. hidden from the environment of the class declaration, then such a structure would be an ideal tool for representing abstract objects in terms of more concrete details. This was the origin of the idea of the abstract data type.

In fact the class concept with hidden variables and visible (public) procedures was inspired by considerations of multiprogramming needs. The grouping of those variables that are accessed by several concurrent processes together with the procedures that embody these accesses created the concept of a so-called monitor [1, 5]. It enables a compiler to automatically insert interlocks (among the processes) at the appropriate places in the program, namely at all entries and exits of a monitor's procedures (instead of upon each individual reference to a shared variable). Such interlocks are a sufficient means to guarantee the absence of unintended process interference (mutual exclusion), which in turn is a sufficient condition to assert the integrity of the shared data.

```
TYPE Stack =
     CLASS  i : INTEGER ;
           el : ARRAY [1..N] OF INTEGER ;
           PROCEDURE push (x : INTEGER) ;
                BEGIN i := i + 1 ; el[i] := x END ;
           PROCEDURE pop : INTEGER ;
                BEGIN pop := el[i] ; i := i - 1 END ;
           PROCEDURE  nonempty : BOOLEAN ;
                BEGIN nonempty := i > 0 END ;
     BEGIN (* initialize *) i := 0
     END
```

Once postulated, the concept of the abstract data type led to a considerable amount of research activities. Only a few matured into implemented systems [6, 7]. It was soon recognized that a truly useful facility would have to offer parameters to type definitions, whose actual values could vary from one instantiation to another. This generalization,however, has considerable implications on the complexity of implementation. At this time, the question about the practical usefulness of the abstract data type to the programmer is still not settled, and factual experience with this concept is largely missing.

```
TYPE Stack (n : INTEGER ; T : TYPE) =
     CLASS i : INTEGER ;
           el : ARRAY [1..n] OF T ;
           PROCEDURE push (x : T) ;
                BEGIN (* assume i < n *)
                     i := i + 1 ; el[i] := x
                END ;
     ...
     END
```

A careful observer will notice that in spite of the enthusiasm with which the abstract data type was received, there are relatively few successful applications. They are therefore widely used as typical examples (e.g. the stack and the queue). This is in strong contrast to the general data and statement structures earlier introduced by high-level languages. The reason for this lack of proven applications may lie in the current lack of good implementations of data type facilities; perhaps it also points out that it is inherently difficult to discover suitable abstract data types for a given problem.

Modules

In Modula [10], we proposed another, similar construct for information hiding, namely the so-called module. The guiding idea was once again the separation of issues. Whereas the class of Concurrent Pascal (and Portal) combines both a facility for information hiding (i.e. a class declaration establishes a scope of identifiers) and for defining a template of data (i.e. a type) for possible instantiation, the module represents a facility serving the first purpose only. The module concept is oriented towards the need of separating large programs into partitions with relatively "thin" connections, and to make the elements establishing this connection more evident in the program text.

The module is effectively a bracket around a group of (type, variable, procedure, etc.) declarations establishing a scope of identifiers. In the first instance, we may regard this bracket as an impenetrable wall, objects declared outside of the module are invisible inside it, and those declared inside are invisible outside. This wall is punctured selectively by two lists of identifiers: the import list contains those identifiers defined outside which are to be visible inside too, and the export list contains the identifiers defined inside that are to be visible outside. This scheme is of a very attractive conceptual simplicity. As a consequence, it is relatively easy to find appropriate applications and to develop a methodology (a style?) for its usage. Implementation is quite manageable, although not trivial.

As the module allows grouping of data and procedure declarations and making only selected elements of this group externally visible, it may also serve to define "abstract data types" that can be instantiated in arbitrary numbers.

As an example, the following Concurrent Pascal class declaration

```
     TYPE C  = CLASS
                  VAR x, y : T ;
                  PROCEDURE p ;
                       BEGIN ... x ... y ... END p ;
                  PROCEDURE q ;
                       BEGIN ... x ... y ... END q
               END
```

can be expressed by a module as

```
     MODULE M ;
          IMPORT T ;
          EXPORT p, q, C ;
          TYPE C = RECORD x, y : T END ;
          PROCEDURE p (c : C) ;
               BEGIN ... c.x ... c.y ... END p ;
          PROCEDURE q (c : C) ;
               BEGIN ... c.x ... c.y ... END q
     END M
```

For creating two instances of the class (type) C we declare "VAR a, b : C" in Concurrent Pascal as well as in Modula. In order to invoke a procedure applied to a class object, we write p(a), q(b) in a conventional manner, instead of a.p, b.q as in Concurrent Pascal.

This analogy between module and class hides one important ingredient of the abstract data type concept that must be mentioned for the sake of fairness and clarity. A crucial idea of the abstract data type is that each instance is governed by some condition (called the type's invariant) which holds for the hidden data at all times (more specifically: whenever control is outside the procedures defined within the class). This invariant is established as soon as an instance is created by some initialization body of the class declaration (not shown above). In the solution using a module, this initialization would have to be performed by an additional, explicit procedure declared within M. Its call might be forgotten by a programmer - missing initialization is indeed a common error - whereas for the class instance it is automatically implied by its declaration.

The sacrifice of this benefit perhaps shows that in choosing the module in place of the class,formal verification techniques have not been foremost on our mind. Instead, our motivation was to provide a facility whose purpose is unique and easy to understand. Experience with programming in Modula - designing compilers and other large programs - has justified the presence of the module facility in a very convincing manner. Most of us would never wish to design another large program without an encapsulation feature such as the module. A revealing fact was also that in the average only a few percent of the modules occurring in a program serve in the sense of an abstract data type as template with several instances. More important is the possibility to nest modules. The distinct asset, however, is the simple but general rule about import and export, the freedom to select objects for import and export, and to have the same rule govern procedures as well as variables, types, and other objects.

In spite of these virtues, the module as described so far has its shortcomings. We have tried to overcome some of them in Modula-2 [11] and during these efforts came to the conclusion that the belief in the existence of an ideal solution satisfying all needs is not only too optimistic but mistaken.

Let me nevertheless present a few points of controversy and point out our choices for Modula-2, a language oriented towards system programming.

Primarily if programs are to be composed from collections of existing modules, i.e. if the writer of a module may not know the environment in which his modules will be embedded, the described solution is unsatisfactory. For, if two inner modules export, by coincidence, the same identifiers, a conflict arises:

```
MODULE M ;
     MODULE M0 ;
        EXPORT x ;
        VAR x : CARDINAL ;
           ...
     END M0 ;
     MODULE M1 ;
        EXPORT x ;
        VAR x : BOOLEAN ;
     END M1 ;
     (* at this point x denotes two different variables *)
END M.
```

In order to disambiguate the above situation, a qualification of x is introduced. In the above example, x must be exported in so-called qualified mode. In this case (see below), no naming conflict arises, because the Cardinal variable x is to be referred to as M0.x, and the Boolean as M1.x.

Qualified export has its drawbacks too, however, and in our experience it is used almost exclusively in the case of utility modules only. Typically, modules are named by relatively long identifiers. They have to be used whenever an object exported in qualified mode is referenced. This results in a "heavy" and cumbersome notation.

In Modula-2, we have eased this situation by providing a facility with an effect similar to that of a WITH statement for records.

Consider the following example:

```
MODULE M ;
     MODULE M0 ;
          EXPORT QUALIFIED x, y ; ...
     END M0 ;
     MODULE M1 ;
          EXPORT QUALIFIED x, y ; ...
     END M1 ;
     MODULE M2 ;
          FROM M0 IMPORT x ;
          FROM M1 IMPORT y ;
          ... x ... y ...
     END M2 ;
     ... M0.x ... M1.y ...
END M.
```

An import list of the form

FROM M IMPORT x

will allow reference by unqualified x, although x has been exported from M in qualified mode.

The simple export rules indicated above have consequences that are undesirable in some situations. If a module is employed to implement an abstract data type as indicated above, then external visibility of the record field identifiers x and y is definitely undesirable. A strict interpretation of the given rules, however, exports x and y automatically together with C. Here we are confronted with the dilemma of transparent vs. obscure export of types. In Modula, we had decided for obscure export - to cater for the view of abstract data types - in Modula-2 we opted for transparent export, which not only allows for a simple export rule, but is highly desirable in many practical cases.

```
MODULE M ;
     MODULE M1 ;
          EXPORT A, R, P ;
          TYPE A = ARRAY 0..9 OF CHAR ;
          TYPE R = RECORD x, y : CARDINAL END ;
          PROCEDURE P (a : A ; r : R) ;
             ...
          END P ;
          ...
     END M1 ;
     VAR a0, a1 : A ; r0, r1 : R ;
BEGIN ...
     (* assume obscure type export *)
     a0[i] := a1[j] ;
     (* illegal because it is unknown here that a0, a1 are arrays *)
     r0.x := r1.y ;
     (* illegal because it is unknown here that r0, r1 are records;
     x, y are  here unknown identifiers *)
     P(a0, r1) (* legal call *)
END M.
```

Our experience has been that transparent export is rather the normal case, and obscure export is desired, if the module is designed in the sense of an abstract data type. Modula-2 offers transparent type export as default, and obscure export in special cases (see below).

Separate modules

Large systems are built in layers with a hierarchical structure. The very large number of elementary components asks for a fair number of logical layers, each being described by a set of abstractions and by conditions governing them. It would indeed be surprising, if the same constructs would be optimally suited for the decomposition on all levels of the hierarchy. But of course, a careful designer has a natural tendency not to introduce different concepts and mechanisms for each level; in fact, one should like to use the same concept for all layers.

We feel that a certain compromise may yield the best results, and recognize a primary distinction of requirements between the level on which components are designed and implemented by different people, and the levels below it. If different people are involved, the incentive to specify thin interfaces, and to nail them by binding agreements, is very much more pronounced, than if the parts are designed, combined, and tested by one person. In the former case, it appears desirable to separate the specifications of a program from the specification of the interface, i.e. of those items that are exported and imported. Such a separation has the decisive advantage that the interface specification can be widely distributed, whereas the actual program is kept private to its implementor. Such a scheme is used in MESA [3].

This scheme is particularly useful if a compiler is designed to check for the consistency of the individual program parts. It must verify that a program is consistent with its specified interface, and that users of a module M are consistent with M's interface. Only in this way can it be determined whether or not changes in a program P can be implemented without changes in any of the users of P; the condition is that P's interface remains unchanged.

At this point, we have quiescently abandoned the view of a system defined by a program written on one piece of paper. Instead, it is specified by many such pieces (we call them modules) designed and developed by different people. For several reasons, it is desirable to be able to compile these modules separately. This facility, belonging entirely to the realm of technical realization should, however, not influence the conceptual design of the language. A program, being a piece of text, must be understood without resort to explanations of its execution and preparation. If it consists of several pieces, it must be regarded as a concatenation of these pieces, and the language rule must cover this case.

In Modula-2 we regard the concatenated modules as able to export into and to import from a vacuous environment. Typically, such a concatenation consists of one module representing the so-called main program and a set of modules with data and procedures used by the primary module. The primary module imports only, the others export and may import. From the foregoing it follows that only the secondary modules have reason to be specified in two parts, namely as an interface specification and a program implementation. We have decided that they must be specified in two parts, and therefore obtain three different species of modules:

- The "normal" module. It can be nested. If it occurs as a separate piece of text, it cannot export and is regarded as a main program.
- The definition module. It specifies the interface of a module, in particular all objects that are exported. (Modula-2 allows qualified export only in this case.)
- The implementation module. It belongs to the definition module carrying the same name. It contains the bodies of the procedures whose headings are listed in the definition module, and possibly declarations of further objects that are local, i.e. not exported.

Definition and implementation modules occur as pairs and as separate pieces of text, i.e. they cannot be nested in some other module. The implementation module is assumed to import all objects defined in its associated definition module.

```
DEFINITION MODULE IO ;
     EXPORT put, get ;
     PROCEDURE put (ch : CHAR) ;
     PROCEDURE get ( ) : CHAR ;
END IO.

IMPLEMENTATION MODULE IO ;
     VAR inbuf, outbuf : ARRAY 0..15 OF CHAR ;
     PROCEDURE put (ch : CHAR) ;
          BEGIN (* body of put *)  END put ;
     PROCEDURE get ( ) : CHAR ;
          BEGIN (* body of get *)  END get ;
BEGIN     (* initialization of data *)
END IO.
```

```
MODULE Main ;
     FROM IO IMPORT put, get ;
     VAR ch : CHAR ;
BEGIN
     ... put (ch) ... ch := get ( ) ...
END Main.
```

The separation into definition and implementation modules yields an unexpected benefit. It solves the problem of transparent vs. obscure export of types in a most natural manner: If a type is fully specified in the definition module, this signals transparent export. Obscure export is achieved by listing merely the identifier in the definition module and by "hiding" the full declaration within the implementation module.

```
DEFINITION MODULE M ;
     EXPORT T0, T1, ... ;
     TYPE T0 ;  (* obscure type export *)
     TYPE T1 = ARRAY 0..15 OF CHAR ;  (* transparent export *)
END M.

IMPLEMENTATION MODULE M;
     TYPE T0 = POINTER TO
                   RECORD x, y : CARDINAL ;
                           ...
                   END ;
     ...
END M.
```

<u>The use of the module facility</u>

It is a well-known truth that any new facility requires not only an appropriate formulation and specification, but also a "theory and practice" of its use. Although well-motivated examples for the principle of information hiding have been circulating for some time, and although they appeared to demonstrate convincingly where classes and modules would be useful, we have experienced that it is by no means an easy task to determine the best modularization of a program in development. This is particularly the case for the larger partitions, except perhaps when a hierarchical structure of procedure layers is fairly evident, such as e.g. in the case of input/output utilities, file handlers, and drivers.

In general, a module is established whenever a collection of procedures share a set of variables that is accessed exclusively by these procedures. (These variables are typically _own_ in the sense of Algol 60). We have gradually shifted, however, from the view of a module being a collection of procedures sharing common variables towards the view of a data structure whose access is restricted by having to occur via a small set of operators. Hence, the data rather than the procedures move into the foreground of attention. A module is typically characterized by the _data_ it contains (and perhaps hides) rather than by its set of exported procedures.

Often - in the case of larger partitions - these data are structured in quite complex manners. In particular, structures with several different kinds of elements occur. It therefore is inappropriate to characterize a module as a type. Instead, the exported procedures often access and alter data elements of several types. This fact may explain partly the earlier mentioned lack of practical success of the class structure which effectively exports a single type only. The module does not suffer from this restrictiveness; it allows arbitrary definition of many data types and structures, and export of all of them if desired.

In essence, then, the recognition of a useful modularization of the involved data structure is the key to finding the appropriate decomposition of a program into modules. As any experienced programmer knows, this is the hardest task; often one discovers the right data structures only while developing the entire program. Without experience and foresight, one is condemned to restructure a program's module decomposition from time to time. The obvious alternative is, of course, not to bother to improve the program.

But I must draw attention to the fact that - owing to the explicit modules - the inappropriateness of a program's structure is much more obvious than it is (or was) when written in a language without modules. Programming with a module facility is indeed more difficult, as it forces the programmer to reason from the start more carefully about a program's structure. However, the benefit is that such programs are - once designed - easier to reason about, to change, to document, and to understand. For system programs, which in general are expected to have a long life span, the gain in "maintainability" is well worth a heavier investment in the effort of their design. In many cases, finding the right structure initially is not merely a benefit, but simply a matter of success or failure.

The module also appeared as an aid towards a solution of another longstanding problem typical of systems programming languages. I refer to the use of low-level, perhaps machine-dependent facilities in a high-level language with program-defined data types. Whereas machine-dependent facilities such as special operators and data types like WORD or BYTE are virtually indispensable in, for example, a disk driver, we strongly wish to hide them from most program parts. The module provides exactly this hiding capability, and therefore emerges as the means to make all levels of a system expressible in one and the same language, without being restrictive in low-level modules, nor being too "permissive" in high-level modules.

In Modula-2, we have postulated a (machine-dependent) pseudo-module, called SYSTEM, that is typically imported in low-level modules only.It contains data types such as ADDRESS, WORD, and procedures corresponding to specialized machine instructions. Moreover, it also contains the primitives used for coroutine handling, from which typically a process scheduler could be programmed. We call this a pseudo module, because its components must be known to the compiler, which acts according to special rules (e.g. generates the specialized instructions instead of regular procedure calls).

Of importance besides the availability of the system module are the additional compatibility rules of these system types with other types. For example, the type ADDRESS is said to be compatible with all pointer types. This holds for both assignment and actual-formal parameter substitution. Compatibility also exists between the types WORD and INTEGER (or CARDINAL). Thus it becomes possible to perform arithmetic operation on pointers, but only in the low-level module, where the pointers are typed as addresses (such as e.g. in a storage manager, see below).

```
MODULE Storage ;
     FROM SYSTEM IMPORT ADDRESS ;
     EXPORT new ;
     VAR LastAdr : ADDRESS ;
     PROCEDURE new (VAR a : ADDRESS ; size : CARDINAL) ;
          BEGIN a := LastAdr ;
               LastAdr := LastAdr + size
          END new ;
BEGIN LastAdr := 0
END Storage.

MODULE Main ;
     IMPORT Storage ;
     TYPE T = ... ;
     VAR p : POINTER TO T ;
BEGIN ...
     Storage.new (p, SIZE(T)) ...
END Main.
```

(Note: The decomposition of the module Storage into definition and implementation parts is not shown here).

Exception handling

When a programmer is experienced in the use of a structured language with sufficiently flexible control statements (such as IF, WHILE, etc.), the GO TO statement will appear as quite dispensable to him. The introduction of modules, however, in particular of separate modules, reintroduces the need for a jump. The major need arises from the handling of exceptional cases, i.e. of exit jumps from procedures. Although this case might be handled by the passing of additional output parameters, the exceptional exit jump is desirable, because parameters would cause additional, unacceptable overhead upon each call. A regular GO TO statement is inadequate, however, if the point of resumption is unknown in the module where the exceptional condition arises.

For this purpose, Modula-2 provides a feature called <u>exception</u>. It is worth emphasizing that the need for it is mainly a consequence of the module facility. We distinguish between three constructs, called exception declaration, exception call, and exception handler. The following example illustrates their use:

```
MODULE M ;
     EXCEPTION ex ;     (* declaration *)
     PROCEDURE p ;
          BEGIN ...
               ex  (* transfer to S1, which is invisible here *)
          END p ;
     PROCEDURE q ;
          BEGIN ... p ; ...
          WHEN ex DO S1  (* exception handler *)
          END q ;
BEGIN ... q ; ...
     ex ; (* exception call, handled by S2 *)
               ...
WHEN ex DO S2
END M.
```

When an exception is called, control exits the called procedures up to the first one which provides a corresponding handler. This handler is executed, whereupon the procedure is terminated and execution resumes at the point of its call. The handlers (WHEN ...) occur at the end of procedure bodies. They can be regarded like procedures; however, when called, the search follows the dynamic history of procedure activations instead of the static, nested scopes of identifier visibility.

Implementation

A pleasant property of the module concept as described here is that it does not present any major problems to implementors. In essence, its influence is restricted primarily to the mechanisms for symbol table access in the compiler. In effect, the module structure is invisible in the code ultimately generated, as it only concerns the scopes (visibility ranges) of objects, but not the semantics of a program.

Large modules specified as separate entities should be separately compilable. This requirement complicates matters to a considerable extent; it does not merely require the generation of relocatable instead of absolute code. The emphasis is on separate compilation in contrast to independent compilation. Thereby we understand that upon compilation, syntax and type consistency checks are performed regardless of whether or not a program is presented in its entirety or in separate pieces. As a consequence, a compiler processing a module M must have access to information about all modules imported by M. Acutally, this information can be restricted to the items exported and, in order to expedite matters, can be stored in a compiled form. A compiler now does not only generate an object (code) file and a listing, but additionally (in the case of a definition module) a symbol table file describing all exported objects. It not only reads from a source file, but also inputs compiled symbol table files of all modules it imports.

This implies that a module M0 being used by M1 must be compiled prior to compilation of M1. Circular references are thereby prohibited. Fortunately, the split into definition and implementation parts solves this problem, if perhaps not fully, then at least for the practically relevant cases. A definition module is compiled - resulting in a symbol table file - before its corresponding implementation module is compiled. If two modules both contain references to the other, then these imports belong to the implementation parts. Hence, both definition parts can be compiled separately (in any sequence), whereupon both implementation parts are processed, yielding the actual code files.

An important consideration is that a compiler should have to consult only the symbol table files of directly imported modules. For example, if M2 imports M1 which in turn imports M0, then the table of M1 should contain processed information on those selected objects of M0 that are referenced in M1. If this were not the case, M2 would have to consult the tables of both M1 and M0. In practical cases, this might quickly lead to an unacceptably large volume of tables to be loaded for modules on higher levels.

Although the complications induced on a compiler by a scheme of separate compilation is by no means minor, a separate compilation facility for modules is virtually indispensable for the construction of large systems. The simple solution of replacing separate by independent compilation is unacceptable, as it would eliminate type consistency checking across module boundaries, thereby giving away one of the most effective assets of a structured language.

References

[1] P. Brinch Hansen. The programming language Concurrent Pascal. IEEE Trans. Software Eng., 1, 2, 199-207 (1975)

[2] O.J. Dahl, K. Nygaard. Simula - An Algol-based simulation language. Comm. ACM 9, 9, 671-678 (Sept. 1966)

[3] Ch. M. Geschke, J.H. Morris, E.H. Satterthwaite. Early experience with Mesa. Comm. ACM, 20, 8, 540-553 (Aug. 1977)

[4] C.A. R. Hoare. Record handling; in Programming Languages, F. Genuys, Ed., London and New York, 1968. (pp. 291-347)

[5] C.A. R. Hoare. Monitors: An operating system structuring concept. Comm. ACM, 17, 10, 549-557 (1974)

[6] H. Lienhard. The real-time programming language PORTAL. R. Schild. Parallel processes in PORTAL, exemplified in a group project. Landis & Gyr Review 25, 2, 2-16 (1978)

[7] B. Liskov et al. CLU Reference Manual. Computation Structures Group Memo 161. MIT Lab. for Comp. Sci. July 1978

[8] N. Wirth and C.A. R. Hoare. A contribution to the development of Algol. Comm. ACM, 9, 6, 413-432 (June 1966)

[9] N. Wirth. The programming language Pascal. Acta Informatica 1, 35-63 (1971).

[10] N. Wirth. Modula: A language for modular multiprogramming. Software - Practice and Experience, 7, 3-35 (1977)

[11] N. Wirth. Modula-2. Tech. Report 27, Institut für Informatik ETH. Zürich, Dec. 1978.