Optimization Across Module Boundaries

A.H.J. Sale†

Languages which provide separate compilation features through the module or package concept (such as Modula-2 and Ada) have some little-known problems with maintaining integrity and also achieving optimizations that cross the module boundaries. The paper addresses these questions using Modula-2 as the example language, and proposes a number of different methods of constructing Modula-2 processors (systems that run Modula-2 programs) which have properties different from the conventional structure of such processors.

Keywords and Phrases: **optimization, modules, Modula-2, opaque export.**
CR Category: **D.3.4.**

1. THE MODULE STRUCTURE OF MODULA-2

The programming language Modula-2 (Wirth, 1985) permits programs to be constructed from modules. Each module is considered to be a relatively self-contained program fragment and communication between modules is confined to a set of facilities which is defined by import and export declarations.

Modules may arise in the construction of a program as a convenient way to fragment it into parts which can be written by separate programming groups. They can also arise as a consequence of implementing a data abstraction, when they provide a means of hiding the details of the data abstraction from the user of the module (for example in Sale, 1986). In this form modules are frequently collected into a library and may be used by a number of programs. The module concept also contains the procedure library as a special case.

A simple local module is shown below as an example for users not familiar with Modula-2. The module is useful for reassuring screen watchers that progress is being made during a long computation by making regular calls to the procedure Tick. After an appropriate number of calls to the Tick procedure a character is written to the output, preceded by an EOL (end-of-line) if the line is filled. The procedure Set allows the user to specify the character to be used, the line length and the number of ticks to be accumulated without visible effect (by default ".", 50 and 10 respectively). All the remaining identifiers of the module are inaccessible from outside, since only the procedure identifiers Set and Tick are exported.

There are three kinds of conceptual modules in Modula-2: *program modules*, *local modules* and *separate modules*. The meaning of these terms is defined in the following paragraphs.

```
MODULE Reassure;

(*  Every identifier required from the surrounding
    scope must be imported. *)
IMPORT CHAR, CARDINAL, Write, EOL;

EXPORT QUALIFIED Set, Tick, Positive;

TYPE
    Positive = [ 1 .. 200 ];

VAR
    (*  Protected values which control the module
        behaviour. *)
    Mark : CHAR;
    LineLength : Positive;
    SilentTicks : Positive;
    (*  Variables to hold the state of the ticking. *)
    Ticks : CARDINAL;
    Marks : CARDINAL;

    PROCEDURE Set
        (ch : CHAR; length : Positive; quiet : Positive);
    BEGIN
        Mark := ch;
        LineLength := length;
        SilentTicks := quiet;
        Ticks := 0;
        Marks := 0;
    END Set;

    PROCEDURE Tick;
    BEGIN
        Ticks := Ticks + 1;
        IF Ticks = SilentTicks THEN
            IF Marks = LineLength THEN
                Marks := 0;
                Write(EOL);
            END;
            Ticks := 0;
            Marks := Marks + 1;
            Write(Mark);
        END;
    END Tick;

BEGIN
    (*  Initialization sequence, setting defaults. *)
    Set(".", 50, 10);
END Reassure;
```


† *Department of Information Science, University of Tasmania, GPO Box 252C, Hobart 7001.* *Manuscript received July, 1986; revised October, 1986.*

A *separate module* is a unit whose activation record has the lifetime of the whole program execution. Unfortunately the syntax and defining documents of Modula-2 describe two components of separate modules and name these *definition modules* and *implementation modules*. This terminology is extremely confusing, since a definition module and its corresponding implementation module are simply two parts of a single module. In this paper these terms will be avoided as far as possible and these two parts will be referred to as the *interface* and the *implementation* of a separate module. The previous example can be rewritten as a separate module with its interface and implementation:

```
DEFINITION MODULE Reassure2;
(*  Interface. *)

TYPE
    Positive = [ 1 .. 200 ];

    PROCEDURE Set
        (ch : CHAR; length : Positive; quiet : Positive);

    PROCEDURE Tick;

END Reassure2.

IMPLEMENTATION MODULE Reassure2;
(*  Implementation. *)

(*  Import from a standard library module for
    input and output. *)
FROM InOut IMPORT Write, EOL;

VAR
    (*  Protected values which control the module
        behaviour. *)
    Mark : CHAR;
    LineLength : Positive;
    SilentTicks : Positive;
    (*  Variables to hold the state of the ticking. *)
    Ticks : CARDINAL;
    Marks : CARDINAL;

    PROCEDURE Set
        (ch : CHAR; length : Positive; quiet : Positive);
    BEGIN
        Mark := ch;
        LineLength := length;
        SilentTicks := quiet;
        Ticks := 0;
        Marks := 0;
    END Set;

    PROCEDURE Tick;
    BEGIN
        Ticks := Ticks + 1;
        IF Ticks = SilentTicks THEN
            IF Marks = LineLength THEN
                Marks := 0;
                Write(EOL);
            END;
            Ticks := 0;
            Marks := Marks + 1;
            Write(Mark);
        END;
    END Tick;

BEGIN
    (*  Initialization sequence, setting defaults. *)
    Set(".", 50, 10);
END Reassure2.
```

The interface is intended to be visible to programmers using the module because it defines the facilities offered by the module and the manner in which they are to be accessed. All identifiers declared in the interface are automatically exported according to the revised rules of Modula-2 (Wirth, 1985). The implementation defines the actual implementation of the facilities and it is possible to deny a programmer access to the source text of this component.

A program module may be regarded as the implementation part of a separate module with an omitted interface. Every program contains one and only one program module. The program module may import facilities provided by other separate modules. Other distinguishing features of a program module are given below:

— Since it has no interface, and therefore no ability to export, no identifier declared in it can be imported by any other module.
— Its initialization sequence is executed after the initialization sequences of all other separate modules, and is conventionally regarded as the 'main body' of the program.

Local modules are textually declared within another module. All the identifiers and implementation structure of a local module are visible to a programmer who has access to the text of the enclosing module. However, unlike a procedure, a requirement to explicitly export and import identifiers across the module boundary allows the programmer to control access to the facilities provided by the module, and thereby use the controlled access to guarantee correctness, or at least to produce a higher degree of confidence regarding correctness.

The relationship between program modules, separate modules and local modules can be illustrated by a more complex example. The program whose module structure is shown schematically in Figure 1 is composed of a program module P and three separate modules A, B and C. The program module P imports facilities exported by the interfaces (**Int**) of modules A and B, while both the interface and the implementation (**Imp**) of A import facilities exported by the interface of C. The program module P contains two local modules (**Loc**) and the implementation

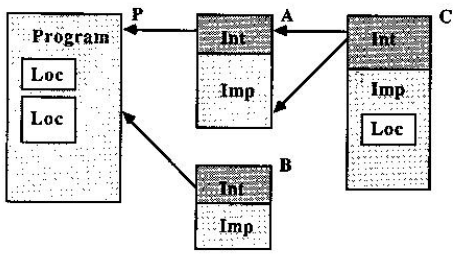

Figure 1. A program.

of C contains one local module. Export and import of identifiers across local module borders are explicitly controlled by export and import declarations which are internal to the enclosing module.

2. FEATURES OF COMMON IMPLEMENTATIONS

Modula-2 provides for 'separate compilation', a feature whose absence many regarded as a deficiency in the design of Pascal. As a consequence, a *Modula-2 processor* (a system for elaborating the meaning of a Modula-2 program when composed with input data) must have facilities for handling the partial processing of the components of the program. The source texts of these components are called *compilation units*. The ability to part-process a program gives rise to a number of difficulties and insecurities, as explained later. Most Modula-2 processors are constructed with three major components: a compiler, a linker and a run-time system. These components are designed to comply with the following conditions:

— All the source texts (program modules, interfaces and implementations) which belong to a program must be successfully processed by a compiler. The compiler checks the syntax of the source text and if this defines a correct Modula-2 compilation unit it produces a *compiled form* of the source text.
— Once compiled forms of all the source texts are available, they must be successfully processed by a linker. The linker checks the existence of and compatibility of each compiled form and if no errors are detected it produces an executable program.
— The compilation of an implementation (implementation module) requires that the compiled form of the corresponding interface (definition module) be accessible to the compiler.
— The compilation of any compilation unit requires that the compiler have access to the compiled forms of the interfaces (definition modules) of all modules it imports.

These last two features define a partial ordering of the required compilations.

Some Modula-2 processors generate and attach unique sequence codes to each definition module that is compiled. This enables the linker to ensure that, when a module is imported, all compiled forms of the associated implementation had access to exactly the same compiled form of the interface. If this is not done, there is a serious risk of introducing an insecurity if different forms of an interface are used. For example, in the MacModula™ implementation (Modula Corporation, 1985), the recompilation of a definition module forces the recompilation of the associated implementation module and all modules that import it, before linking is allowed.

Some of the consequences of the arrangement should be pointed out explicitly. Consider first the case of a library module which has been written for general use. Such a module will have been written and tested long before its use. The supplier of the module must make available to the user the compiled form of both the interface and the implementation, but need not make the source text of either available.

In practice, there is little point in suppressing publication of the source text of the interface, since it merely defines the facilities provided by the module of which a user of a module must be told anyway. However, if the module implements a deeper level of service, and the user should not import it directly, then even this source text can be kept private. It is normal, however, for the source text of the interface to be published in printed form but not supplied in electronic form. The reason is simple: anyone who recompiles the interface, and thus destroys the supplied compiled form, thereby destroys the utility of the supplied implementation which cannot thereafter be used.

However, the source text of the implementation may be kept private with three consequent advantages:
— Since a module user is unable to determine how the module is implemented, he or she is less likely to write code that depends on a particular implementation. The code is therefore more likely to be correct, as it will be solely based on the interface definitions.
— If the implementation turns out to be flawed, the errors may be corrected and a new release of the module delivered with no change in the documentation.
— Keeping the implementation private partially protects the copyright in the module in two possible ways:
  — it cannot be ported to another machine since it cannot be recompiled, and
  — the compiled form may contain a hidden serial (licence) number.

In some library module contexts, it may be appropriate for the module designer to prepare several implementations for the same (source text) interface. For example, a module to implement a symbol table might be provided with a hash table implementation, a binary tree implementation, and a binary tree implementation with rebalancing. The scheme outlined above would require that the library module be provided to the user as source text and compiled form of the interface, and three compiled forms of the implementations. Only one of the implementations should be resident in an environment while a program is being processed, though it does not seem that many processors would be able to check this.

If the module structure does not wholly involve the use of a library, the order of compilations also becomes important. This is the case for program development involving several modules when the components of the program may be being developed simultaneously and perhaps by different persons. To illustrate this, consider the program of Figure 1. An outline of the relevant import parts are shown in the following program outline, and Figure 2

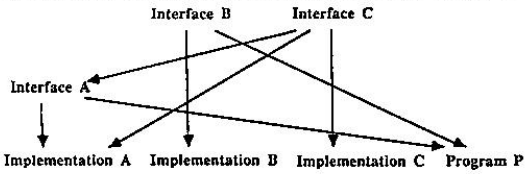

Figure 2. Conventional partial ordering of compilation of source texts.

```
MODULE P;
FROM A IMPORT ...;
FROM B IMPORT ...;
...
END P.

DEFINITION MODULE A;
FROM C IMPORT ...
...
END A.

IMPLEMENTATION MODULE A;
FROM C IMPORT ...
...
END A.

DEFINITION MODULE B;
...
END B.

IMPLEMENTATION MODULE B;
...
END B.

DEFINITION MODULE C;
...
END C.

IMPLEMENTATION MODULE C;
...
END C.
```

shows the partial ordering of compilations required by the particular form of Modula-2 processor construction described earlier.

It will be clear that the interface of C must be delivered in compiled (checked) form before the interface of A can be compiled. However, once all the interfaces are delivered, the three implementations and the program module may be written and compiled in any order. Indeed, the program module may be finalized before any of the implementations of the modules it requires.

Revisions of the program are also affected. If the compiled forms can be preserved after an executable program has been created, then a change in one component forces all the components that depend on it to be recompiled. Thus alteration of the definition module of C requires recompilation of all the source texts except the interface and the implementation of B, but alternation of the implementation module of C requires no compilations other than of itself. This last case is important since it is implementations and the program module (which can be considered to be an implementation without an interface) which are likely to contain errors and to be revised. In all cases a relinking is required.

It is also possible to tolerate a cycle of importations (for example A imports B, B imports C, C imports A) provided that at least one of the importations in the cycle occurs only in an implementation and not an interface. This means that there is no cyclic dependency between the interfaces and an allowed order of compilation can be found. Then if all the compiled forms of the interfaces are available all the implementations can be compiled in any order. Caution with such instances is required as the order of elaboration of the initialization sections of the associated implementations is not defined and is likely to be implementation-dependent.

3. BENEFITS OF INTERMODULE INTERACTIONS

The conventional Modula-2 processor construction described in Section 2 minimizes the recompilation of implementations. However, there are at least two distinct disadvantages to this construction.

3.1 Opaque export

The first is a language feature. Modula-2 allows a module designer to declare a type identifier in the interface but not give the corresponding type definition. This is called an opaque export of the type and it offers a number of important facilities. Most important of these is the ability to protect the data in an instance of the type from alteration by any means other than the facilities provided by the module, while allowing the user of the module to declare and preserve such instances in their activation records. Thus it becomes possible to opaquely export a type *tree* together with basic tree operations such as *create, insert, delete, search* and *traverse,* and thereby to use the module as a mechanism to extend the power of the language.

In a similar usage it is possible to allow the user to hold data pertaining to a resource (for example a file or an i/o device) while denying access to the details of the resouce other than by calls to module procedures. An illustration of the technique can be seen in the following fragment.

```
DEFINITION MODULE FileSystem;
...
TYPE
    FileType; (* Opaque export *)

    PROCEDURE OpenFile
        (DirectoryName : ARRAY OF CHAR;
        VAR f : FileType);

    PROCEDURE Read
        (VAR f : FileType;
        VAR NextChar : CHAR);
    ...
END FileSystem.
```

However, the Modula-2 Report (Wirth, 1985) states 'Opaque export is restricted to pointers and to subranges of standard types'. Many processors go further and restrict opaque export to pointer types. Why does this restriction exist?

It is not hard to find a plausible reason. At the point of compilation of an interface the text of the implementation is not known; therefore the compiler has a problem even at this stage with the representation of a type opaquely exported. Some representation has to be assumed if it is to be inserted into the compiled form of the interface

(remember that this is all that the compiler looks at in order to compile an importing module).

In most computer systems all pointer values have the same size and representation, regardless of what their bound type might be, and hence the simplest compilation technique is to map all opaquely exported types into the pointer type representation, thereby forcing the module designer to declare the elaboration of the type in the implementation as a pointer type. Such a technique is not unduly limiting as the bound type of a pointer type is not constrained.

The suggested freedom to also allow subranges of the standard types (meaning the types BOOLEAN, CHAR, CARDINAL and INTEGER) is predicated on the assumption that values of these types have the same storage requirements as a pointer type. Wherever this assumption is violated then opaque export is correspondingly restricted. For example, in many computer systems it is not reasonable to use the same representation for characters (type CHAR) as for whole numbers (type CARDINAL).

A user of a module with an opaquely exported type may use the type:
— in structured type declarations,
— as the type of a formal parameter,
— as the result type of a function procedure, and
— as the type of a variable.

No selection or dereferencing of a value of an opaquely exported type is allowed, and no operations are defined on such a value other than those provided by the exported procedures. The only allowed forms of an expression are a simple variable or a selection of a variable of a structured type that yields a variable of the opaquely exported type. Correspondingly the declared variables or selected variables may be used as actual parameters corresponding to variable parameters, as actual parameters corresponding to value parameters, and in assignments.

There is a problem with this resolution. The user must be *fully aware* that the opaque type is a pointer type and must not be allowed to think that the type is 'really' a record or some other kind of type, because the assignment statement

```
variable := expression
```

copies a pointer value, not the hidden value. If the hidden value is later altered by calls to the module's procedures, then all variables of the opaquely exported type which point to it adopt the new changed value. This is not the expected semantics for assignment with non-pointer types.

3.2 Removing the restriction

Suppose for argument that a program is viewed solely as the totality of all its source texts. Since an opaquely exported type is defined in the corresponding implementation, there is no problem in giving a meaning to opaque export of any type, nor is there difficulty in deciding (as humans) how each specific case should be implemented. Consequently the restriction of opaque export to pointer types with the consequent problems is a consequence of separate compilation, or at least of the form of separate compilation embodied in the usual Modula-2 processor construction. Section 4 examines how this restriction can be removed.

3.3 Optimization problems

A second constraint imposed by the usual construction of a Modula-2 processor is the difficulty of carrying out any optimizations across the boundary between a separate module and its users. For example, the compilation of an importing module cannot take into account any information which might be derived from the compilation of the implementation. Similarly, the compilation of the implementation cannot take into account the usage of an importing module. Neither is a problem, of course, for local modules which are compiled in the context of their enclosing module.

A particular optimization example may highlight the issue. Many module writers have adopted the practice of exporting a variable identifier whose value holds some important state information. Even some standard modules (such as InOut) adopt this practice. The practice is regrettable since the module itself cannot rely on the user leaving the value of the variable unaltered, and the user cannot protect his own module from an inadvertent modification by his or her own code. The insecurity introduced demands that the module itself make *no* use of the value of this exported variable, and the user would be well-advised to encapsulate access to its value in a function procedure declaration. Why would anyone choose to create self-imposed problems of this kind when a secure option is available — providing the state as the result value of a parameterless function procedure call?

However, the only plausible explanation of their use is a misplaced concern for efficiency of access, assuming a variable access is a direct access to a register and thus faster or more compact than a call of a parameterless procedure, and presuming that it may be a significant efficiency determinant. However, the variable is in a different module and probably in a different addressing environment, so if the access involves the alteration of address or display registers (a context switch) then the speed of a variable access may be nearly the same as that of a parameterless procedure call. One conclusion that can be drawn is that if a language permits the export of variables from modules, someone will want to use the facility.

However, if the compiler had access to the implementation while it was compiling an importing module it could optimize a call of a parameterless function by substituting the code body in-line, thereby removing even any slight advantage. Consider an exported procedure *Done()* and the following declaration in the implementation as an example.

```
PROCEDURE Done( );
BEGIN
    (*  Return the value of the private variable
        InternalDoneState as the result. *)
    RETURN InternalDoneState;
END Done;
```

When the compiler finds a call in the importing module, such as

```
WHILE Done( ) AND (ch <> ",") DO
```

then it should be able to substitute code equivalent to

```
WHILE InternalDoneState AND (ch <> ",") DO
```

4. ALLOWING OPTIMIZATION

The key question to be answered is whether there exist alternative constructions of processors, which implement the essentials of separate compilation, and yet permit some of these problems to be circumvented. They exist, and in this section the possibilities will be explored. The essentials of satisfactory solutions are clear: either code generation must be delayed to some processing stage following compilation, or the information must be made available to the compiler at the time it generates code.

One solution has been suggested by K.J. Gough (private communication, 1986) that does not involve either of these, but it is relatively unsatisfactory. Suppose that the code generated by a straightforward compiler is such that it is always larger than optimized code. Then it is possible for this code to be flagged so that the linker may substitute for the tentative code something more efficient. Such a system may permit an in-line varible access to be substituted for a procedure call, to use the earlier example. The difference between a replacement strategy and a delayed code generation scheme is not large, and resides only in the containment of optimization decisions to a highly local fragment of code.

4.1 Delayed code generation

The simplest suggestion is attributed to G. Goos (private communication, 1982) in relation to Ada. Suppose that the role of the compiler is defined to be the analysis of the syntax and static semantics of each compilation unit (including the reporting of errors) and the production of an intermediate analysed tree form of each unit, with no code-generation. Suppose further that two new processing steps are substituted for the linking stage. Firstly a program which will be called the *resolver* is given access to all the compiled trees and modifies them so as to take into account all available information. The resolver also has to handle the compatibility problems of a linker. Secondly a program which will be called the *code generator* takes all the trees and generates specific code for the machine from them. This solution will readily handle both the opaque export problem and the optimization problem.

It is also possible to combine the resolver and the code-generator into a single pseudo-linker so as to hide the individual steps. Delayed code-generation reduces the task which the compiler has to undertake and shifts extra tasks to the final phase (resolver and code-generator). Separate compilation is retained at the expense of reducing the significance of compilation. However the scheme is attractive, since all its components are well-understood, and it has the potential to be able to apply complex transforms in pursuit of optimizations.

It is worth exploring a few permutations on this construction scheme. Suppose the task of the compiler were further reduced to the analysis of the syntax and static semantics of each compilation unit (including the reporting of errors), with correct source texts simply being marked in some way to indicate their checked status. Alternatively the source texts may be copied with some identifying mark to show them as checked (compiled) units. Nothing changes in the scheme except that the resolver needs to reparse the source texts (with the assurance that they are correct and hence no checking nor error-recovery is needed) before tackling the tree processing. The order of reparsing will be determined by the dependencies, and hence either the compiler must somehow inform the resolver of these, or the resolver can establish a sub-task for each reparsing which interlocks and waits when it reaches a point where it needs unavailable information. Provision for resolving or detecting importation cycle deadlock will be needed in this case.

A further step along the same path eliminates the marking of source texts. The resolver now has a front-end which compiles the source texts with checking and error-recovery before it tackles tree-processing. The compiler is reduced to an optional (but highly desirable) program for providing independent checking of compilation units, in the interests of efficiency. It plays no role in the task of code generation.

Since the executable program is always derived from a consideration of all the compilation units, the dependencies that remain are the essential ones. This is exactly the same as in the usual Modula-2 processor structure.

4.2 Changing the processor structure

A second departure from the conventional processor construction involves changing the conceptual structure of modules and their compiled forms. Such a new processor structure is based on that described earlier in Section 2, but the fourth and last condition is altered to:

— The compilation of any compilation unit requires access to the compiled forms of all modules it imports.

One of the significant features of this change is the compiler has access to both the interface and implementation of every imported module, consequently it is capable of making optimization decisions while compiling an importing module which involve use of information regarding the implementation of imported modules. In addition the partial ordering of compilations is altered, as shown in Figure 3 for the earlier example.

This new ordering requires implementations to be compiled before any module that imports the features of its interface. Several detailed constructions are possible.

— The compiler might be able to accept just a definition module (interface) for checking only, or an interface plus its implementation. Only in the latter case may it generate a compiled form of the module.
— The compiler might generate a compiled form as a result of compiling the interface. However, when compiling the implementation, the data in the compiled form of the interface are incorporated into a new, merged, compiled form of the total module.

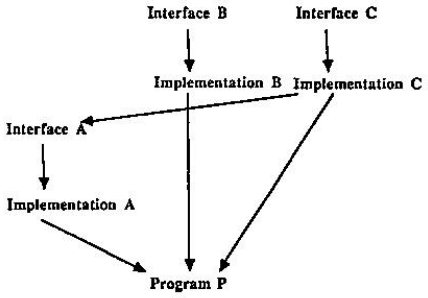

Figure 3. Revised partial ordering of compilation of source texts.

— The compiler might, as with the conventional construction, generate compiled forms of both parts of a module. The compilation of importing modules will involve searching for both compiled components.

The consequences of this change need careful examination, for it is generally assumed by Modula-2 programmers that the conventional construction is the only one feasible or practical. The first such consequence is that all importation cycles are disallowed. Recall that the conventional construction allowed them, providing at least one of the imports was confined to an implementation. With this new processor construction, even this restriction does not allow an order of compilation to be determined. Modules involving importation cycles cannot be compiled. Fortunately such cycles rarely arise in real programs.

Secondly the processor behaves differently under conditions of change. Under the conventional model, an alteration of an implementation module involves only its own recompilation and relinking. In this new structure, altering an implementation involves nearly the same consequences as alteration of the interface; all the dependent compilation units, except the interface itself, need compilation.

5. CONCLUSIONS

Is the processor construction part of the definition of a programming language such as Modula-2? Is tampering with the conventional structure permissible? Are there Modula-2 features from which a particular processor construction must be inferred and no others are possible?

In this paper it is argued that the processor construction is not fixed and that advantages can be gained from such departures. Of course, there are frequently consequent disadvantages which accompany the advantages, but it is important to see the situation as involving choice.

A second conclusion is that it may not be desirable to separate the interface and the implementation as done in the language Modula-2, leaving a linker and operating system to provide the connection links. It would appear to be better to retain all parts of a module as a compilation unit, with appropriate textual separation of the interface for public distribution (but not recompilation).

ACKNOWLEDGEMENTS

The work described in this paper was carried out with the assistance of Australian Research Grant F85160571 and a University Research Grant from the University of Tasmania. The comments of members of the staff of the Department of Information Science and members of the Silicon Research Group in the University of Tasmania are also gratefully acknowledged.

REFERENCES

MODULA CORPORATION LTD. (1985): *MacModula™ Reference Manual*, Modula Corporation, Provo, Utah.

SALE, A.H.J. (1986): *Modula-2: Discipline & Design*, Addison-Wesley, Wokingham, England.

WIRTH, N. (1985): *Programming in Modula-2*, 3rd Ed, Springer-Verlag, Berlin.

BIOGRAPHICAL NOTE

Professor Sale has been Professor of Information Science at the University of Tasmania since 1974, and is currently Chairman of the University's Professorial Board. His active contribution to the standardisation of Pascal led to the development of the Pascal Certification Process. More recently he has published a book on Modular-2, which is fast replacing Pascal and has a second in draft form.