Diss. ETH No. 7286

Separate Compilation in Modula-2 and the Structure of the Modula-2 Compiler on the Personal Computer Lilith

A dissertation submitted to the
SWISS FEDERAL INSTITUTE OF TECHNOLOGY ZURICH

for the degree of
Doctor of Technical Sciences

presented by
LEO BERNHARD GEISSMANN
Dipl. Math. ETH
born April 10, 1953
citizen of Hägglingen (Aargau)

accepted to the recommendation of
Prof. Dr. N. Wirth, examiner
Prof. Dr. C.A. Zehnder, co-examiner

Zürich 1983

Preface

In the fall of 1977, Prof. N. Wirth started the Lilith project, an integrated software and hardware effort with the main goal to develop a modern work-bench for the software engineer. The main components of the project have been the design and implementation of the programming language Modula-2 which includes features for the programming of large software systems, the design and construction of the personal computer Lilith with modern devices (e.g. high-resolution screen, pointing device "mouse") and with an efficient architecture to support the needs of a high-level programming language, and the implementation of a comfortable programming environment (e.g. operating system, editor, compiler, debugger) which supports and facilitates program development.

This thesis grew out of the authors participation in the Lilith project. It concentrates on the aspect of separate compilation which is a very important tool for the design and development of large software systems. Separate compilation has been successfully implemented and completely embedded into the Lilith computer system. This was enabled by a corresponding concept in Modula-2 and by the support of the operating system and the Lilith architecture.

I am indebted to Prof. N. Wirth for conceiving and coordinating the Lilith project, for giving me the opportunity to participate in the design of Modula-2 and the development of a Modula-2 compiler, and for supervising this thesis.

I would like to thank Prof. C.A. Zehnder for his helpful advices on this thesis.

My thanks go to all my colleagues who participated in the Lilith project and contributed with their work to the success of the whole system. In particular, many thanks are due to A. Gorrengourt, Ch. Jacobi, and Sv.E. Knudsen for the valuable discussions and their participation in the development of the compiler. I am also grateful to F. Ostler for reading my thesis and improving my English.

Above all, I am indebted to my wife Hanni for her encouragement and patience.

Contents

Abstract

Separate compilation allows the separation of a program into smaller units which are compiled separately. For high-level programming languages it is mandatory that a compiler also provides *full type- and parameter-checking* on references among separate units. A possibility to implement these checks is that the compiler saves *interface descriptions* of the compiled units and uses them upon subsequent compilations. For a programmer it must be possible to specify the interface of a separate unit. For this purpose, a *concept for separate compilation* should be integrated in a programming language.

The purpose of this thesis is to show how the programming language *Modula-2* provides a *powerful* concept for separate compilation which is the base for a *simple, comfortable*, and *successful* implementation. Units for separate compilation are *modules*; in order to provide a useful tool, the specification of the interface (definition module) is split from its actual implementation (implementation module). This contributes to the stability of the interface, and this is very important when a program is developed by a group of programmers.

The Modula-2 compiler which has been developed as a main component of the Lilith personal computer project generates a *symbol file* upon compilation of a definition module. The symbol file contains a complete description of the definition module, and it is considered by the compiler to be the only and unique interface description during the compilation of neighbor modules.

Apart from a good concept for separate compilation in a programming language, the convenience of an implementation of separate compilation mainly depends on the support provided by the operating system and the architecture of the target machine. The architecture of the personal computer *Lilith* supports the separation of programs into separate modules. Code and global data of a separate module are loaded into so-called frames and addressed relatively within these frames. This allows the compiler to generate almost definitive code.

This thesis further describes the structure of the implemented Modula-2 compiler and the functions of the various compiler parts. The compiler is written in Modula-2 itself and separate compilation was successfully used for structuring and development of the compiler.

Kurzfassung

Separate Compilation ermöglicht das Aufteilen eines Programms in kleinere Einheiten, die vom Compiler einzeln übersetzt werden. Für höhere Programmiersprachen ist es dabei notwendig, dass der Compiler auch bei Referenzen zwischen separaten Einheiten eine *vollständige Typen- und Parameterprüfung* durchführt. Eine Möglichkeit für die Implementierung dieser Überprüfungen ist, dass der Compiler *Schnittstellenbeschreibungen* der compilierten Einheiten aufbewahrt und diese bei nachfolgenden Compilationen verwendet. Damit ein Programmierer die Schnittstelle einer separaten Einheit eindeutig festlegen kann, muss die Programmiersprache ein *Konzept für die separate Compilation* enthalten.

In dieser Dissertation wird gezeigt, wie in der Programmiersprache *Modula-2* ein *leistungsfähiges* Konzept für die separate Compilation definiert wurde, das die Grundlage für eine *einfache*, *bequeme* und *erfolgreiche* Implementierung ist. Einheiten für die separate Compilation sind *Module*, wobei die Beschreibung der Schnittstelle (Definitions-Modul) eines separaten Moduls von der eigentlichen Codierung (Implementations-Modul) abgetrennt und für sich compiliert wird. Dies erhöht die Stabilität der Schnittstelle, was sehr wichtig ist, wenn ein Programm von einer Gruppe von Programmierern gemeinsam entwickelt wird.

Der Modula-2-Compiler, der im Rahmen des Lilith-Arbeitsplatzrechner-Projekts entwickelt wurde, erzeugt bei der Compilation eines Definitions-Moduls ein *Symbol-File*, das eine vollständige Beschreibung der im Definitions-Modul definierten Objekte enthält. Das Symbol-File wird vom Compiler als einzige und eindeutige Beschreibung der Schnittstelle während der Compilation angrenzender Module betrachtet.

Neben einem guten Konzept für die separate Compilation in der Programmiersprache hängt die Bequemlichkeit ihrer Implementierung vor allem von der Unterstützung durch das Betriebssystem und von der Architektur des Zielrechners ab. Die Architektur des Arbeitsplatzrechners *Lilith* unterstützt die Aufteilung eines Programms in separate Module. Code und globale Daten eines separaten Moduls werden in sogenannte Frames geladen und innerhalb dieser Frames relativ adressiert. Dies erlaubt dem Compiler, beinahe definitiven Code zu erzeugen.

Die Dissertation beschreibt im weitern die Struktur des implementierten Modula-2-Compilers und die Aufgaben der verschiedenen Compilerteile. Der Compiler ist selbst in Modula-2 geschrieben, und die separate Compilation wurde erfolgreich für die Strukturierung und Entwicklung des Compilers angewendet.

1 Introduction

1.1 Separate Compilation

Compilation of a program is considered by most programmers as a waste of time and they would prefer it to be an instant operation. It is admitted that a compiler has to provide a great deal of support, to detect and report the programming errors, and to generate optimal code, but, nevertheless, all this should happen within no time. Compilation time, however, increases with the length of a program and therefore it is very inconvenient to compile large programs. It is even worse to recompile a large program which has been modified at a few places only.

The remedy appears to be *separate compilation*. The compiler allows a program to be split into smaller units which are compiled separately. After compilation of all separate units, the generated code parts are combined and the program is ready for execution. Separate compilation saves compilation time, if only a small unit, instead of the whole program, must be recompiled after a local modification. After combination with the new code part, the program is ready again for execution. The advantage of separate compilation becomes more obvious, if a large program is developed by a group of programmers. It allows each group member to develop his contribution more or less independently of the rest of the group.

Another positive aspect of separate compilation concerns library routines, e.g. mathematical functions or input-output routines. Instead of copying the source text of library routines into the program text, it makes more sense to compile them separately and to combine their code with the other code of the program before its execution.

As soon as a program is split into several separate units, the units will reference objects which are specified in other units. This means that there must be some information available about these objects, e.g. the name of a procedure and the types of its parameters. The information about all objects of a separate unit which might be referenced from other units is called the *interface* of the unit. For separate compilation it is important that the interface of a unit remains unchanged when the unit is modified and recompiled. In this case the other units are not affected. If, however, the interface of a unit is changed, all units referring to it must be recompiled as well.

We learn from this that behind an interface the programmer enjoys a great deal of freedom for the implementation of a separate unit, and that separate units become more independent of each other the smaller their interfaces are. The art of good software engineering is to design program structures such that the interfaces of separate units remain small and stable.

Separate compilation is not a new invention. Nearly all *assembly languages* support it and also *Fortran* uses this principle. But according to the simplicity of these languages neither assembler programs nor the Fortran compilers check references to other units. They are assumed to be correct. An interface description only exists on a piece of paper or in the mind of the programmers, but it is not known to the translator. Full responsibility rests on the programmer who must be sure that the

interfaces are correctly used, and who also has to detect whether an interface has been changed.

With the next language generation, the Algol- and Pascal-like *high-level programming languages*, more security was introduced in programming. This security is gained by a consistent *type concept* which requires that each variable must be declared to be of an explicit data type and that only values of this type may be assigned to a variable. Operands in expressions must be of compatible types, and in procedure calls the parameters must be substituted consistent with the type rules. All these conditions are tested by the compiler. It is one of the advantages of high-level languages to give the programmer a certain amount of security in these important details and to allow him to concern himself more with the algorithm to be implemented.

The improvement of security should not be abandoned, when a program is split into separately compiled units. Full type and parameter checking should still be guaranteed by the compiler for references among separate units. To fulfill this demand, it is necessary that the compiler exactly knows the interfaces of the separate units. A possibility to gain the needed information could be that the compiler reads and processes for each compilation of a separate unit the *source texts* of the involved interfaces. For reasons of economy and also of security, however, it is more appropriate that the compiler saves a *symbolic interface description* after compilation of a separate unit. This description will be needed by the compiler when it compiles other units which refer to the corresponding separate unit. Saving a symbolic description also enables the compiler to easily detect whether an interface was changed and recompilation of other units is therefore necessary.

Definitions

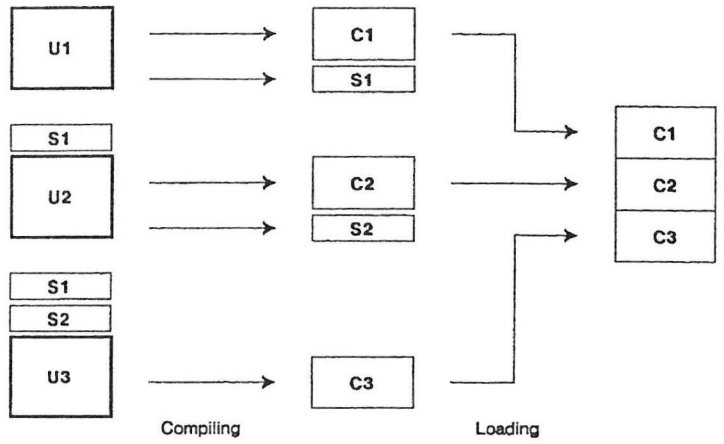
For further use in this text, the term *separate compilation* is reserved for implementations which provide full type and parameter checking across the boundaries of separate units at compilation time (see Figure 1.1).

If the responsibility for the correct use of the interfaces is left to the programmer, or if references are checked upon combining the separate code parts, the term *independent compilation* will be used (see Figure 1.2).

Assemblers and Fortran compilers therefore provide independent rather than separate compilation.

A consistent type concept, as provided by Algol and Pascal, is a necessary pre-condition for the implementation of separate compilation. But without additional language features, it is impossible to provide a reliable and convenient implementation. Separate compilation is not a feature which might be provided by a compiler only. It is rather necessary that separate compilation is *integrated as a concept* into a programming language. It should be possible to explicitly specify separate units and especially their interfaces in terms of a programming language.

What happens if such a concept is missing in a programming language might, for example, be demonstrated with the programming language *Pascal* [Wir71]. Pascal



U Separate Unit
C Object Code
S Symbolic Interface Description

Figure 1.1 Separate Compilation

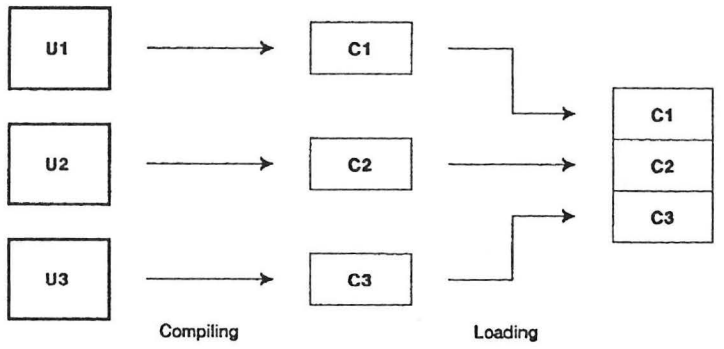


Figure 1.2 Independent Compilation

programs were intended to be compiled as a whole, but compiler implementors soon detected that at least they should support access to library routines. The door was opened to a wide range of proposals and implementation efforts. Many different concepts were implemented, but in most cases it was independent compilation without explicit compiler tests. Pascal 6000 [Jen75], for example, allows the declaration of procedure headings to represent procedures which are externally encoded. No tests at all are provided to check the correctness of the interface. Other concepts are proposed in [Cel80] and [Eul82]. They allow the specification of further kinds of external objects. Their correctness is, however, checked after compilation upon combining the separate code parts. A concept which respects the block-structure of Pascal and provides separate compilation on all procedure levels is presented in [LeB79]. This is done by dumping and restoring the whole environment (symbol table) of a separately compiled segment. A quite simple separate compilation concept has been implemented in UCSD Pascal [Sof81]. Programs may be subdivided into so-called *units*, and these may be compiled separately. Units are typically used to supply library routines to a program.

A new generation of programming languages, which may be considered as successors of Pascal, introduced the *module concept* as an additional tool for structured programming. The module concept both supports data abstraction and explicit visibility control and is therefore very useful for structuring large programs and, obviously, for separate compilation purposes. It allows a programmer to explicitly control the references which cross the boundaries of a module, and he may therefore reduce the visibility of objects and protect data and data structures from unauthorized access. This possibility to specify and control an interface is precisely what is needed for separate compilation. A module is an ideal program part to be compiled separately. It is therefore obvious that most programming languages which provide modules also include a concept for separate compilation.

1.2 Modula-2 and its Implementation

One of these new programming languages with a module and separate compilation concept is *Modula-2* [Wir82] which has been designed and implemented as successor of *Pascal* and *Modula* [Wir77] at the Institut für Informatik, ETH Zürich. Modula-2 is a modern programming tool for software engineers and supports both programming of systems and high-level applications. A philosophy for the language design was to include such concepts which are simple, but powerful and appropriate for a large number of applications. Overloading of the language by rarely used special concepts was intentionally avoided.

The separate compilation concept of Modula-2 has been designed according to this philosophy. It provides separate compilation of modules which are considered to be defined in a *universal* (global) environment. Local modules and procedures, however, are excluded from being compiled separately. It seems not to be a severe restriction, because compiling in a universal environment is assumed to be appropriate for most applications.

The purpose of this thesis is to show that Modula-2 provides a *clear* and *powerful* concept for separate compilation, and that this concept is an excellent base for a

simple and *comfortable* implementation. This work bases on the implementation of Modula-2 on the personal computer *Lilith* (see section 1.3). Modula-2 is the sole programming language on this machine and the developed programming environment completely integrates separate compilation.

The module construct, as provided by Modula-2, needs a slight improvement to become a powerful tool for separate compilation. It is most important that the interface of a separate module is not affected when a local change is performed. The best method to guarantee the stability of the interface is to split it completely from the actual implementation and to compile it separately. Modules and separate modules, as defined in Modula-2, are described in Chapter 2. In order to show that the chosen concept is a good compromise between simplicity and capacity, separate compilation concepts of other programming languages are described in Chapter 3 for comparison.

The advantage of simple and clear language concepts is that their implementation in a compiler is easily and efficiently possible without causing unexpected overhead. A *comfortable*, *successful*, and *efficient* implementation of separate compilation has been realized on the personal computer Lilith. Important factors for a good implementation are, in addition to the language concept, the symbol file which contains the interface description of a separate module, the support given by the operating system, and also the architecture of the target machine. This is described in Chapter 4.

The further chapters of this text describe the organization, structure, and function of the Modula-2 multi-pass compiler which has been implemented on the personal computer Lilith. It is written in Modula-2 and the separate compilation facility is used for its internal structuring. Especially mentioned are the components of the compiler which are important for the implementation of separate compilation. Chapter 5 reports the history of the compiler development and describes the overall organization of the compiler. Chapter 6 describes the global compiler data structures and the organization of the symbol table. Chapter 7 describes the scope handling structure which controls the visibility of the declared objects. Chapter 8 describes the functions of the compiler parts and shows their decomposition into separate modules.

The conclusions which are given in Chapter 9 concern the use of Modula-2 and the separate compilation facility, separate modules, symbol files, and the implemented Modula-2 compiler. Further, a possible improvement of the module concept is proposed.

1.3 The Lilith Project

The development of the programming language Modula-2 was embedded into a large project which was started at the Institut für Informatik in 1977. The general aim of the whole project was the design and the development of a *personal work-station computer system for software engineers*. This includes a comfortable programming environment as well as an efficient computer whose architecture supports the needs of a high-level programming language. The final product of the co-operative

software and hardware effort is the *Lilith computer system* [Wir81].

The main components of the Lilith project

Modula-2

Design of the programming language Modula-2 and development of a multi-pass Modula-2 compiler for the personal computer Lilith.

M-code

Design of the M-code machine architecture which enables efficient code generation by a Modula-2 compiler, as well as efficient interpretation by a computer.

Lilith

Design and development of the Lilith hardware which efficiently executes the microcode in which the M-code interpreter is encoded.

Medos-2

Development of a single-user operating system which supports the linking, loading, and execution of programs, file access, and management of memory resources.

Programs and Modules

Development of a large set of programs and modules which utilize the capabilities of Lilith and provide a modern and efficient environment for the user of the machine (e.g. text editor, graphic editor, debugger, window handler, input-output routines).

2 Separate Compilation Concept in Modula-2

The programming language *Modula-2* [Wir82] is a modern programming tool for software engineers. It supports both systems programming and programming of high-level applications. The language has been designed in parallel with the development of the personal work-station computer *Lilith* [Wir81], and one of the main goals of the language design was to produce a tool powerful enough that all software running on Lilith, including the operating system, can be written in Modula-2.

Generally, Modula-2 may be considered as a successor of Pascal. It mainly differs from Pascal by the lack of files, which can be expressed in terms of more primitive features of the language, by an improved syntax, and by the new concepts of modules and separate compilation. In addition and mainly for systems programming, Modula-2 provides some low-level features and a simple coroutine concept.

Because modules are ideal units for the purposes of separate compilation, the first section describes the module concept. The second section gives a description of the concept for separate modules.

2.1 The Module Concept

The module concept extends the set of tools for structured programming. It is necessary for the design and implementation of large programs and enables the construction of modular components which combine program parts which logically belong together.

The syntactic construction of modules is quite similar to that of procedures. A module consists of a parameterless heading, the import and export specification, the declaration of local objects (constants, types, variables, procedures, and modules), and a statement part. Semantically, however, there are major differences.

A first difference concerns the validity or visibility of objects. Each module and each procedure opens a validity range for objects, a so-called *scope*. A general language rule requires that the names of all objects visible in a scope must be unique. To support this some visibility rules are defined.

Visibility rules for procedures:

- Objects defined within a procedure are invisible outside.
- Objects visible outside a procedure are also visible inside, unless an object with the same name is defined within the procedure.

Example 2.1

```
VAR aa, bb, cc : CARDINAL;
PROCEDURE pp;
  VAR bb, dd : REAL;
    (* bb, dd          visible objects declared inside the procedure *)
```

```
    (* aa, cc, pp       visible objects declared outside the procedure *)
BEGIN
END pp;
    (* aa, bb, cc, pp   visible objects declared outside the procedure *)
```

The visibility rules for modules are both more restrictive and more general. With the exception of standard defined objects (e.g. CARDINAL), the visibility of objects across module boundaries is explicitly controlled by import and export specifications.

Visibility rules for modules:

- An object visible outside a module is visible inside only if it is imported.
- An object defined within a module is visible outside only if it is exported.
- Standard defined objects are implicitly visible inside a module.

Example 2.2

```
VAR aa, bb : CARDINAL;
MODULE mm;
  IMPORT bb;
  EXPORT pp;
  VAR cc : CARDINAL;
  PROCEDURE pp;
    VAR bb, dd : REAL;
  BEGIN
  END pp;
    (* cc, pp       visible objects declared inside the module *)
    (* bb           visible object declared outside the module *)
BEGIN
END mm;
    (* pp           visible object declared inside the module *)
    (* aa, bb, mm   visible objects declared outside the module *)
```

Apart from their visibility, there is also a difference in the *life-time of variables*.

Life-time rules:

- The life-time of variables declared within a procedure is bound to the execution of the procedure.
- The life-time of variables declared within a module is the same as that of variables declared outside.

In the examples above the variables *bb* and *dd* of procedure *pp* are established when *pp* is called and they disappear when the procedure terminates. The variable *cc* has the same life-time as the variable *aa*, even when it is declared local to module *mm* and invisible outside the module.

Another difference concerns the statement part. As the life-time of variables declared local to a module is not bound to the execution of the module's statement part, this statement part is reserved for special purposes. Modula-2 defines that it is used for initialization. It cannot be called explicitly like a procedure; instead it is implicitly activated when the environment of the module is established, i.e. when the enclosing procedure is activated. In most cases the statement part has to initialize the module's variables.

Example 2.2 is derived from example 2.1 by inserting the scope of a module around procedure *pp* and variable *cc*. This reduces the visibility of *cc* to the new scope, whereas *pp* is still visible in the same environment as it was before. Further, variable *aa* is no longer visible in procedure *pp*.

The idea behind the module concept is to combine objects which logically belong together (e.g. a procedure package) and to separate them from other objects of the entire program. The explicit control of visibility reduces the number of visible objects, and it also allows to *hide* objects which should not be affected by other program parts. In particular, it is desirable to protect variables from unauthorized access. Hiding them is an effective way to achieve this goal. All this is a contribution to the improvement of security in programming.

Example 2.3

Handling of a queue is a typical example, where the use of a module structure is appropriate. The main ingredients of a queue handler are procedures which allow to *enter* and to *remove* information. In a specific implementation these procedures operate on a circular buffer which represents the queue. The module allows to protect the buffer from direct access. For clients of the module, it is only accessible via the procedures.

```
MODULE Queue;

  EXPORT InfoEntry, Enter, Remove;

  CONST buffleng = 128;

  TYPE InfoEntry = RECORD (* some fields *) END;
       BufferIndex = [0 .. buffleng - 1];

  VAR buffer : ARRAY BufferIndex OF InfoEntry;
      enter, remove : BufferIndex;

  PROCEDURE Enter(info : InfoEntry; VAR ok: BOOLEAN);
  BEGIN (* Enter *)
    ok := (enter + 1) MOD buffleng <> remove;
    IF ok THEN
      buffer[enter] := info;
      enter := (enter + 1) MOD buffleng;
    END;
  END Enter;
```

```
  PROCEDURE Remove(VAR info: InfoEntry; VAR ok: BOOLEAN);
  BEGIN (* Remove *)
    ok := enter <> remove;
    IF ok THEN
      info := buffer[remove];
      remove := (remove + 1) MOD buffleng;
    END;
  END Remove;

BEGIN (* Queue *)
  enter := 0;
  remove := 0;
END Queue;
```

For large programs there still remains a visibility problem. A name conflict occurs, if several modules (possibly written by different programmers) export objects with the same name. To prevent ambiguities, *qualified export* may be specified in the export list. In this case the exported objects are only visible outside the module, if their name is preceded by the module's name, i.e. qualified access. Direct access to such objects is possible in other modules, if they are explicitly imported (*from import*). Qualified access in other modules is necessary in any case, if only the module name is imported.

Example 2.4

```
MODULE mm;
  EXPORT QUALIFIED pp;
END mm;
    ------- mm.pp -------    qualified access
MODULE nn;
  FROM mm IMPORT pp;
    ------- pp ----------    direct access
END nn;

MODULE oo;
  IMPORT mm;
    ------- mm.pp -------    qualified access
END oo;
```

2.2 Separate Modules

For separate compilation purposes, the module turns out to be an ideal unit. Visibility rules for modules give the programmer an explicit control over the references across the module boundaries, and the export list is well suited for the specification of the interface of a separate compilation unit. It was therefore obvious to choose modules as separate compilation units in Modula-2 [Gei79, Wir82].

Definitions

A *separate module* is a unit which constitutes a separately compiled part of a program. In the further parts of this paper separate modules will simply be called *modules*.

For modules nested within a separate module the term *local module* will be used.

The partitioning of a program into separate modules is supported on the global level only. The environment of each separate module is considered as the "universe" in which the separately compiled modules are embedded. Separate compilation of local modules and procedures is intentionally not provided. The chosen solution seems to be a good compromise which enables almost all possible applications, but reduces the complexity of the concept and allows a simple implementation. Compilers have only to save the interface description of a separate module, but not also a description of the module's environment.

The interface of a separate module may be considered as a contract of the module with its importers (clients). If the interface is changed, all importers of the module must be recompiled. For achieving a great degree of independence in the development of separate program parts, it is important that the interface of a separate module remains stable (unchanged) upon its recompilation.

To guarantee the stability of an interface upon recompilation, a compiler could provide extensive tests which compare an already existing symbolic interface description with the actual state of the interface. This is, however, not satisfactory at all, because an unintended change of an interface could have far-reaching consequences. Modula-2 improves the security for the programmer by separating the interface definition from the actual implementation. It splits the description of a separate module which exports objects to other program parts into two complementary parts, a *definition module* and an *implementation module*.

The definition module contains all declarations which are needed for a complete specification of the interface. For procedures, only the heading with the procedure's name and parameter list are relevant for the interface. Local declarations and the statement parts of the procedures must therefore not occur in the definition module. Considering that a definition module exports to an unknown environment, qualified export must be specified in definition modules.

Example 2.5

If the module *Queue* of example 2.3 is compiled separately, the following definition module would be specified.

```
DEFINITION MODULE Queue;

  EXPORT QUALIFIED InfoEntry, Enter, Remove;

  TYPE InfoEntry = RECORD (* some fields *) END;

  PROCEDURE Enter(info : InfoEntry; VAR ok: BOOLEAN);
```

```
  PROCEDURE Remove(VAR info: InfoEntry; VAR ok: BOOLEAN);

END Queue.
```

The implementation module is considered as the complement of the definition module. It is the actual encoding of the separate module. It contains local objects and statements which need not be known to the clients of the module. Generally, the objects declared in the definition module are implicitly defined within the implementation module. Only procedures must be declared again, this time as normal procedures with heading, local object declarations, and a statement part. It is obvious that their headings must be the same as the headings of the procedures in the definition module.

Implementation modules do not have an export list, because exported objects of the separate module are declared in the definition module. But they have their own import list to import the needed external objects. This again contributes to the separation of the actual implementation from the interface.

Example 2.6

Implementation module of the separate module *Queue*.

```
IMPLEMENTATION MODULE Queue;

  CONST buffleng = 128;

  TYPE BufferIndex = [0 .. buffleng - 1];

  VAR buffer : ARRAY BufferIndex OF InfoEntry;
      enter, remove : BufferIndex;

  PROCEDURE Enter(info : InfoEntry; VAR ok: BOOLEAN);
  BEGIN (* Enter *)
    ok := (enter + 1) MOD buffleng <> remove;
    IF ok THEN
      buffer[enter] := info;
      enter := (enter + 1) MOD buffleng;
    END;
  END Enter;

  PROCEDURE Remove(VAR info: InfoEntry; VAR ok: BOOLEAN);
  BEGIN (* Remove *)
    ok := enter <> remove;
    IF ok THEN
      info := buffer[remove];
      remove := (remove + 1) MOD buffleng;
    END;
  END Remove;
```



```
BEGIN (* Queue *)
  enter := 0;
  remove := 0;
END Queue.
```

The separation of the interface description from the actual implementation is highly advantageous. The compiler can generate the symbolic description of the interface upon compilation of the definition module. This will be the valid reference for all importing separate modules, and in particular for the compilation of the implementation module. It guarantees that changes of the implementation module do not affect the interface. It is even possible to write several implementation modules for the same interface, which might be appropriate to different situations. The only condition which must be respected by the implementor is that the semantics of his implementation corresponds to that which is intended. Although the separate compilation concept of Modula-2 supports full checking of types and parameters, it does not provide features to completely control the semantics of an implementation. For example, it would be still possible to supply the interface defined in definition module *Queue* with an implementation module of inexpected behavior, i.e. a stack instead of an ordinary queue might be implemented.

Example 2.7

A different implementation of the separate module *Queue* stores the information in a linear list instead of in a circular buffer. The intended semantics of queue handling are still observed.

```
IMPLEMENTATION MODULE Queue;

  FROM Storage IMPORT ALLOCATE, DEALLOCATE;

  TYPE ElemPtr = POINTER TO Element;
       Element = RECORD
                   info : InfoEntry;
                   link : ElemPtr;
                 END;

  VAR enter : ElemPtr;
      remove : ElemPtr;

  PROCEDURE Enter(info : InfoEntry; VAR ok: BOOLEAN);
  BEGIN (* Enter *)
    ok := TRUE;
    entert.info := info;
    NEW(entert.link);
    enter := entert.link;
  END Enter;
```

```
  PROCEDURE Remove(VAR info: InfoEntry; VAR ok: BOOLEAN);
    VAR help : ElemPtr;
  BEGIN (* Remove *)
    ok := remove <> enter;
    IF ok THEN
      info := removeт.info;
      help := remove;
      remove := helpт.link;
      DISPOSE(help);
    END;
  END Remove;

BEGIN (* Queue *)
  NEW(enter);
  remove := enter;
END Queue.
```

A reason for splitting a separate module into definition and implementation modules is to separate the interface information, which must be available to the clients of the module, from implementation details, which usually are considered private to the module. This separation is simply achieved for procedures. For calling a procedure, a client only needs to know the procedure's name and parameter list. Therefore, local declarations and the statement part of an exported procedure are declared in the implementation module.

More problems arise with type declarations. The declaration rules of Modula-2 require that all types used in declarations must be declared as well. This implies that, apart from an exported type, a definition module possibly contains additional type declarations which are components of the exported type. For example, consider that a pointer type is exported by a definition module. In this case also the referenced type structure (e.g. a record type) must be specified in the definition module. This might be cumbersome for two reasons. First, the referenced type structure could be implementation-dependent and therefore it could be inconvenient to declare it already in the definition module. Second, it could be important to protect objects of the referenced type from direct access and modification by clients. As in Modula-2 the structures of exported types are completely *transparent* outside the module, it is impossible to protect a referenced type structure.

In order to improve this unpleasant situation, Modula-2 allows to specify types with *opaque export*. A type definition in a definition module may consist of a type identifier only, i.e. without specification of a type structure. This means that outside the module only the type's name is known but not its properties. Client modules may declare objects of such types, but they are not allowed to directly operate on them. All possible operations on objects of types with opaque export must be provided by the exporting module, i.e. with a collection of procedures. In particular, a procedure must be provided which initializes the corresponding objects.

According to exported procedures, types with opaque export must be completely declared in the corresponding implementation module. In order to avoid complex

code generation, it was decided that opaque export of types is restricted to pointer and integer types.

Example 2.8

A typical example for the use of opaque type export is a simple file handling module which hides the actual representation of a file. File variables may be declared in a user program, but it is required that only procedures of the file module operate on these variables.

```
DEFINITION MODULE SimpleFiles;

  EXPORT QUALIFIED File, Open, Close, Read, .... ;

  TYPE File;          (* type with opaque export *)

  PROCEDURE Open(VAR f: File);
  PROCEDURE Close(VAR f: File);

  PROCEDURE Read(VAR f: File; VAR ch: CHAR);

  ....

END SimpleFiles.


IMPLEMENTATION MODULE SimpleFiles;

  FROM Storage IMPORT ALLOCATE, DEALLOCATE;

  TYPE File = POINTER TO FileDesc; (* complete declaration *)
       FileDesc = RECORD (* some fields *) END;

  PROCEDURE Open(VAR f: File);
  BEGIN (* Open *)
    NEW(f);
    WITH fт DO (* initialize file *) END;
  END Open;

  PROCEDURE Close(VAR f: File);
  BEGIN (* Close *)
    WITH fт DO (* terminate file *) END;
    DISPOSE(f);
  END Close;
```

```
  PROCEDURE Read(VAR f: File; VAR ch: CHAR);
  BEGIN (* Read *)
    WITH f↑ DO (* assign 'ch' a value from buffer *) END;
  END Read;

  ....

END SimpleFiles.
```


3 Separate Compilation in Other Programming Languages

This chapter gives a short overview of the separate compilation concepts that are provided by some other representants of the new programming language generation. As examples the languages *UCSD Pascal* [Sof81], *Mesa* [Mit79], and *Ada* [Ada80] are chosen. They all base on the type and block-structure concept of Pascal and include a module concept. UCSD Pascal is a slight extension of Pascal with a very simple separate compilation concept. Its main disadvantage is that the interface specification is coupled with the implementation part. This makes the use of separate compilation very inconvenient. Mesa and Ada provide more complex separate compilation concepts than Modula-2. In Mesa it is possible to have definition parts which are partially exported by several implementation parts, and Ada provides separate compilation of local modules and procedures. The disadvantage of these features, which are useful in some special applications, is that their implementation is expensive and that also the normal user pays for the complexity.

For illustration, the queue handling module which has been used as example in the previous chapter will be encoded in all the discussed languages.

The approximate date of development of the programming languages is listed in the following table:

1970			Pascal	
1972				
1974	Mesa			
1976		Modula	UCSD Pascal	
1978		Modula-2		
1980				**Ada**

3.1 UCSD Pascal

The lack of a separate compilation concept in the programming language Pascal [Wir71], as mentioned above, opened the door for a wide spectrum of language extensions for this purpose. Only a few of the proposed and implemented extensions are genuine separate compilation facilities which check the correct use of the interfaces upon compilation. There are many concepts which defer the actual checking to linking time when the code parts are combined.

UCSD Pascal [Sof81] is a variant of Pascal that has a quite simple separate compilation concept which is basically similar to that of Modula-2. UCSD Pascal is intended to be *"one Pascal for all Microcomputers"*. It is the programming language of a programming system called *UCSD p-SYSTEM*. This system, developed at the University of California at San Diego and based on the *Pascal-P compiler* developed at ETH, provides the programmer with a machine independent programming environment which also facilitates portability. The UCSD p-SYSTEM is widespread in the world of microcomputers. The main differences between Pascal and UCSD Pascal concern string handling, file handling, concurrency, and separate compilation.

Separately compiled entities in UCSD Pascal are either normal *programs* or so-called *units*. A unit is a module-like construction which contains a group of routines and data structures. It consists of an *interface part* and an *implementation part*. The interface part contains all object declarations which are visible for client units and programs. The implementation part contains the actual implementations of the procedures and functions which are declared in the interface and possibly further declarations of local objects. The implementation part may further contain a statement part which is used for initialization and termination purposes.

Example 3.1

```
UNIT Queue;

  INTERFACE

    TYPE InfoEntry = RECORD (* some fields *) END;

    PROCEDURE EnterInfo(info : InfoEntry; VAR ok: BOOLEAN);

    PROCEDURE RemoveInfo(VAR info: InfoEntry; VAR ok: BOOLEAN);

  IMPLEMENTATION

    CONST buffleng = 128;
          buffmax = 127;  (* buffleng - 1 *)

    TYPE BufferIndex = 0 .. buffmax;

    VAR buffer : ARRAY [ BufferIndex ] OF InfoEntry;
        enter, remove : BufferIndex;

    PROCEDURE EnterInfo;  (* no parameter list in redeclaration *)
    BEGIN (* EnterInfo *)
      ok := (enter + 1) MOD buffleng <> remove;
      IF ok THEN
        BEGIN
          buffer[enter] := info;
          enter := (enter + 1) MOD buffleng;
        END;
    END; (* EnterInfo *)
```



```
    PROCEDURE RemoveInfo;
    BEGIN (* RemoveInfo *)
      ok := enter <> remove;
      IF ok THEN
        BEGIN
          info := buffer[remove];
          remove := (remove + 1) MOD buffleng;
        END;
    END; (* RemoveInfo *)

BEGIN (* Queue *)
  (* initialization code *)
  enter := 0;
  remove := 0;
END. (* Queue *)
```

A program or unit may *use* another unit by listing its name in the *uses clause*. In programs the uses clause immediately follows the program heading, in units it may appear at the beginning of either the interface or the implementation part. If a unit name appears in the uses clause, all its objects declared in its interface part are visible within the importing client. UCSD Pascal does not provide a feature to indicate whether or not an imported object is directly accessible, nor is it possible to select individual entities while omitting others. This may be very inconvenient, because the user has to know all names defined in an imported interface. If more than one unit is imported, it is also necessary that the names of the objects provided by the different units are distinct.

The initialization code of a unit is implicitly executed before the importing client is started, and the termination code is executed immediately after termination of the importing client.

Example 3.2

```
PROGRAM QueueUser;

  USES Queue;              (* interface objects of unit Queue *)
                           (* are visible within the program  *)

  VAR info : InfoEntry;    (* type from unit Queue *)
      ok : BOOLEAN;

BEGIN (* QueueUser *)
  (* implicit initialization of unit Queue *)
  ...
  EnterInfo(info, ok);     (* procedure from unit Queue *)
  ...
  RemoveInfo(info, ok);    (* procedure from unit Queue *)
  ...
END. (* QueueUser *)
```

Apparently, the separate compilation concept in UCSD Pascal has been influenced by the simplicity of its implementation, probably a too simple one. The compiler on the UCSD p-System does not save a symbolic interface description of a unit. Instead, it copies the whole text of the interface part on the object code file. When a unit is used this file is read again and the interface declarations are inserted as text into the declaration part of the using client. Objects of other units which are referenced by an interface part are not described on the object code file. It is therefore necessary to import (use) also those units needed for the interface description of an imported unit, and it is necessary that the units are imported in a sequence which respects the declaration rule which reqires that an object must be declared before its use.

Another disadvantage of the separate compilation concept in UCSD Pascal is that interface and implementation part belong together textually. Upon each recompilation of a unit, the interface description on the object code file is renewed. This prohibits easy checking of the rule that all clients of a unit must be recompiled when its interface has been changed. The compiler does not check this at all, and there is also no check provided by the linker. Therefore, it may easily occur that code units are linked whose interface references are incorrect. Full responsibility is again with the programmer. The only way to guarantee that all interfaces are correctly used would be to recompile all units and programs which import a recompiled unit, and this is very inconvenient. It is a pity to loose the advantages of separate compilation upon linking because of a merely too simple concept.

3.2 Mesa

The *Mesa* programming language has been designed and implemented at the Xerox Palo Alto Research Center. In the language manual [Mit79] it is characterized by the following abstract:

"The Mesa language is one component of a programming system intended for developing and maintaining a wide range of systems and applications programs. Mesa supports the development of systems composed of separate modules with controlled sharing of information among them. The language includes facilities for user-defined data types, strong compile-time checking of both types and interfaces, procedure and coroutine control mechanisms, and control structures for dealing with concurrency and exceptional conditions."

The design of Modula-2 was, in addition to Pascal and Modula, also inspired by Mesa. Especially the separation of the interface definition from the actual implementation was a hint in the right direction for the design of the separate compilation concept in Modula-2.

Separate compilation units in Mesa are called *modules*. There are two kinds of modules: *definitions modules* which define the interface of a separate part, and *program modules* which contain actual data and executable code and which possibly export an interface.

Example 3.3

```
Queue : DEFINITIONS =
  BEGIN
    -- interface elements
    Enter : PROCEDURE [info : InfoEntry] RETURNS [ok : BOOLEAN];
    Remove : PROCEDURE RETURNS [info : InfoEntry, ok : BOOLEAN];
    -- non-interface elements
    InfoEntry : TYPE = RECORD [ -- some fields -- ];
  END.


DIRECTORY Queue : FROM "queue";       -- connection to a symbolic
                                      -- interface description
BufferQueue : PROGRAM EXPORTS Queue =
  BEGIN
    OPEN Queue;                       -- open visibility of public
                                      -- names of module Queue
    buffleng : CARDINAL = 128;        -- constant
    BufferIndex : TYPE = [0 .. buffleng-1];  -- type

    enter : CARDINAL ← 0;             -- variable with initialization
    remove : CARDINAL ← 0;            -- variable with initialization
    buffer : ARRAY BufferIndex OF InfoEntry;  -- variable

    Enter : PUBLIC PROCEDURE [info : InfoEntry]
      RETURNS [ok : BOOLEAN] =
      BEGIN
        ok ← (enter + 1) MOD buffleng # remove;
        IF ok THEN
          BEGIN
            buffer[enter] ← info;
            enter ← (enter + 1) MOD buffleng;
          END;
      END;                                -- procedure Enter

    Remove : PUBLIC PROCEDURE
      RETURNS [info : InfoEntry, ok : BOOLEAN] =
      BEGIN
        ok ← enter # remove;
        IF ok THEN
          BEGIN
           info ← buffer[remove];
           remove ← (remove + 1) MOD buffleng;
          END;
      END;                                -- procedure Remove

  END.
```

A program module is a separately compiled program part consisting of data definitions and of code statements. Access to its objects from outside the module boundaries is controlled by *access attributes* which are assigned to each name defined in the module. An object is either *private* or *public*. Private objects are known within the defining module only. Public objects may also be accessed from other separate modules, if they import the module. Objects are assumed to be private, unless they are explicitly specified to be public.

Import of separate modules is achieved by specifying module names in the *import list* in the heading of a module. The public objects of an imported module are, however, not directly visible inside. Instead, their names must be preceded by the module's name (qualification). By stating an *open clause* at the beginning of the block, the public objects become directly visible (consider *qualified export* and *from import* in Modula-2).

In principle, program modules would satisfy all needs for separate compilation. But, of course, recompilation of a separate program module would be very inconvenient for its clients, because a specific compilation order must be respected and therefore importing modules must be recompiled too. In order to avoid unnecessary recompilation, Mesa provides the definitions modules which represent the *interface extraction* of a program part.

Usually, a definitions module is a collection of definitions of publicly accessible objects, defined in a program module. These are variables, procedures, and signals. They are called *interface elements*. Procedures must be specified with their parameter list only. Variables may be declared as *read-only* variables. This prohibits assignment to the variables from outside. Apart from the interface elements which represent actually existing objects of a program module, a definitions module also may contain so-called *non-interface elements*. These are compile-time constants, i.e. constant and type definitions, which are the definitions module's *own* objects. They must not be redeclared in the program module. According to the purpose of a definitions module the access attribute is set to public by default, and private objects must be specified explicitly.

In contrast to Modula-2, definitions modules in Mesa are not implicitly coupled with program modules. The names of the definitions module and the corresponding implementation are different. A program module must explicitly mention a definitions module in its *export list*. This means that it exports an interface, i.e. the interface elements of the indicated definitions module. For this purpose public variables, procedures, and signals must match in type and name with interface elements.

As described so far, the separate compilation concept of Mesa is not essentially different from that of Modula-2. Mesa, however, provides some further features. It allows that *a program module implements only a part of an exported interface*. Therefore, a definitions module may be implemented by several program modules. On the other hand it is also possible that *more than one interface is exported by a program module*. A module may also import another module more than once, i.e. *several instances* of a module may exist upon execution of the program. This is a powerful feature. Consider for example that a program requires two different

queues. In this case the queue module would be imported twice and identified with two different names. Each instance would have its own set of data. It is even possible to supply two different implementations of the interface.

Example 3.4

```
DIRECTORY Queue : FROM "queue";       -- connection to a symbolic
                                      -- interface description

MultiQueueUser : PROGRAM
  IMPORTS firstQueue : Queue,         -- import of two instances
          secondQueue : Queue =       -- of the same interface
  BEGIN
    ok : BOOLEAN;

    info : Queue.InfoEntry;           -- access non-interface elements

    ok ← firstQueue.Enter[info];      -- access to interface elements
    [info, ok] ← firstQueue.Remove;   -- of the first queue

    ok ← secondQueue.Enter[info];     -- access to interface elements
    [info, ok] ← secondQueue.Remove;  -- of the second queue

  END.
```

A module instance is not implicitly initialized when a program is started. The initialization is delayed until any of its exported procedures is called for the first time. At that moment, first the variable initializations and, afterwards, the statements of the module body are executed. Another possibility is to explicitly initialize a module by a *start statement*. This gives full control on the initialization phase to the programmer.

All these and further features cover a wide range of needs for complex program applications. They look quite nice, but they also have their price and complicate the normal use. Before execution of a program, it is necessary to combine the separate modules and to set up the connections according to the export and the import lists. This is done by a so-called *binder*. According to the complexity of the possible combinations, a special *configuration language* has been designed. This language is called *C/Mesa*. It has a Mesa-like syntax, but its own semantics. It allows to describe program configurations, i.e to specify which program modules export which interfaces, which instance of a module should be substituted to a client, an so on. A C/Mesa program is interpreted (processed) by the binder and the resulting output is a program code which is ready for execution.

In the simplest case, a configuration description lists the names of all program modules which must be bound together. An instance of each named module will be part of the program configuration, and if a module imports any interfaces, they will be supplied by those which are exported from other modules of the configuration. A program module may be assigned to be the *control component* of the program. This means that it must be initialized first when the program is started. It may be considered as the main program of the configuration.

Example 3.5

```
QueueProgram : CONFIGURATION
  CONTROL SingleQueueUser =          -- main program
  BEGIN
    BufferQueue;                     -- exports Queue
    SingleQueueUser;                 -- imports Queue
  END.
```

The simple appearance of the above configuration description is achieved by the use of default conventions. For more complex applications this is no longer appropriate. To prevent mis-interpretations, the generation of a module's instances and their assignment to importing clients must explicitly be stated.

Example 3.6

```
QueueProgram : CONFIGURATION
  CONTROL MultiQueueUser =           -- main program
  BEGIN
    buffer : Queue ← BufferQueue[];  -- first instance of Queue
    list   : Queue ← ListQueue[];    -- second instance of Queue
    MultiQueueUser[buffer, list];
  END.
```

Compared with Modula-2, the separate compilation concept of Mesa is more complex. The language supports features which might be useful for some special applications, but are rarely used by most programmers. The price for all these capabilities is not low and must be paid by the implementation as well as by the users.

One important difference between Modula-2 and Mesa concerns the interface specification. In Modula-2 the interface of a separate module is specified in the definition module and this is the only valid reference for all importing clients, and also for the corresponding implementation module. Definition module and implementation module are considered as complementary parts of a separate module. In Mesa, however, an interface is conceptionally provided (exported) by a program module and definitions modules are considered as supplementary constructions which summarize an interface specification for clients and therefore reduce the dependency on local modifications of a program module. As a consequence, the language allows that an interface represented by a definitions module may partially be exported by several program modules, and that a program module exports objects which are specified in different definitions modules. A programmer does not get complete information from definitions modules only. For binding a program, he explicitly needs information about the actual implementations of the involved program modules. Possible ambiguities must be resolved by explicit assignments in a configuration description.

3.3 Ada

In 1975 a competition for the design of a new programming language was started by the United States Department of Defense. The main goal was to establish a *"single high order computer programming language appropriate for Department of Defense embedded computer systems"*. The language should support the programming of *large scale and real time systems*, and programs written in this language should also be easily portable. As result of a four year period of alternating design, evaluation, and requirement refinement phases emerged the programming language *Ada* [Ada80] which was designed by CII Honeywell Bull under the leadership of J.D. Ichbiah.

The programming language Ada primarily includes facilities of general purpose high-level languages with procedure and module structures, with parallel processing facilities, and with separate compilation. Apart from these features the language also includes facilities for specialized applications. Separate compilation in Ada is based on the concepts introduced and implemented for the programming language *LIS* [Ich77].

Program units in Ada are *subprograms* (procedures and functions), *tasks* (for parallel processing), and *packages* (modules). They all may be compiled separately. A package in Ada is the equivalent to the module in Modula-2. It is generally split into a *package specification* and a *package body* which must have the same name. The interface of a package must be specified in the package specification which may contain declarations of constants, types, variables, and program units. All objects declared in a package specification are *visible*, i.e. they may be referenced from program units declared outside the package. They are made directly visible in a program unit, if the package name is listed in a *use clause*. A package body complements the package specification. It is only necessary, if the package specification contains declarations of program units. The package body must in this case contain the bodies of these program units. It may contain further declarations and program units which are invisible outside the package. The statement part of a package body serves for initialization.

Example 3.7

```
PACKAGE Queue IS

  TYPE InfoEntry IS
    RECORD
      -- some fields
    END RECORD;

  PROCEDURE EnterInfo(info: IN InfoEntry; ok: OUT Boolean);

  PROCEDURE RemoveInfo(info: OUT InfoEntry; ok: OUT Boolean);

END Queue;
```

```
PACKAGE BODY Queue IS

  buffleng : CONSTANT := 128;          -- constant

  TYPE BufferIndex IS RANGE 0 .. buffleng-1; -- type

  enter : BufferIndex := 0;            -- variable with initialization
  remove : BufferIndex := 0;           -- variable with initialization
  buffer : ARRAY (BufferIndex) OF InfoEntry; -- variable

  PROCEDURE EnterInfo(info: IN InfoEntry; ok: OUT Boolean) IS
  BEGIN
    ok := (enter + 1) MOD buffleng /= remove;
    IF ok THEN
      buffer(enter) := info;
      enter := (enter + 1) MOD buffleng;
    END IF;
  END EnterInfo;

  PROCEDURE RemoveInfo(info: OUT InfoEntry; ok: OUT Boolean) IS
  BEGIN
    ok := enter /= remove;
    IF ok THEN
      info := buffer(remove);
      remove := (remove + 1) MOD buffleng;
    END IF;
  END RemoveInfo;

END Queue;
```

In package specifications, it is possible to hide the structure of a type to external users of the package by declaring a *private type*. In this case the package specification must contain a *private part* where the private type is completely declared. Private types are the equivalent to Modula-2's *opaque exported types*. The two concepts mainly differ in the place where the hidden structure of the types must be declared. In Modula-2 this place has been chosen according to the priciple that implementation details should consequently be separated from the interface specification. The price paid for this decision, however, is a restriction of opaque export to pointer and integer types. In Ada a restriction for private types is avoided. As a consequence, implementation details must be specified in the package specification, i.e. the user knows about the actual type structure. A disadvantage of this concept is that any change of a private type implies a modification of the package specification, and this may be inconvenient for the users of the package.

Example 3.8

```
PACKAGE SimpleFile IS

  TYPE File IS PRIVATE;

  PROCEDURE Open(f: IN OUT File);
  PROCEDURE Close(f: IN OUT File);

  PROCEDURE Read(f: IN File; ch: OUT Character);

  .....

PRIVATE  -- complete declaration of private type 'File' necessary

  TYPE FileDesc IS RECORD .... END;
       File IS ACCESS FileDesc;

END SimpleFile;
```

Ada programs may be compiled separately. Each program is a collection of one or more *compilation units* which are said to belong to a *program library*. (The language, however, does not specify the organization of a program library.) Compilation units are package specifications, package bodies, subprogram declarations, and subprogram bodies. This allows partitioning of a program on the global level (as with Modula-2). If a package or a subprogram of the program library is referenced by a compilation unit, this must be indicated in a *with clause* at the beginning of the compilation unit.

Example 3.9

```
WITH Queue;                          -- imports Queue
PROCEDURE QueueUser IS               -- main program
  USE Queue;                         -- for direct visibility

  info : InfoEntry;
  ok : Boolean;

BEGIN
  --
  EnterInfo(info, ok);
  --
  RemoveInfo(info, ok);
  --
END QueueUser;
```

Ada also provides separate compilation of nested program units, so-called *subunits*. A subunit is a body of a subprogram, task, or package whose interface is specified in the outermost declaration part of another compilation unit and whose separate

compilation of the body is indicated by a *body stub*. The original environment of the body stub must be saved by the compiler, and it must be reconfigured when the subunit is compiled. A subunit may refer to all objects visible in its environment. As a consequence, after recompilation of a compilation unit all its subunits must be recompiled as well.

Example 3.10

Package *Queue* implemented as a subunit

```
PROCEDURE QueueUser IS

  PACKAGE Queue IS
    -- interface specifications
  END Queue;

  PACKAGE BODY Queue IS SEPARATE;     -- body stub

  -- declarations

BEGIN
  --
END QueueUser;


SEPARATE(Queue)                       -- subunit
PACKAGE BODY Queue IS
  -- implementation of Queue
END Queue;
```

Compared with Modula-2, the separate compilation concept of Ada mainly differs in the support of subunits. This is probably a nice feature to support the development of block-structured programs, i.e. it is most useful for programming in Pascal-style without using the facilities for modular programming. This is, however, not the appropriate style for programming large scale systems. For modularized programs, it is not crucial to have the possibility to compile objects within a local environment separately. On the contrary, separate compilation on the global level is appropriate and adequate for a large number of applications. In view of the additional complexity in the compiler, caused by saving and restoring the environment of a subunit, it seems that separate compilation of subunits is not worth to be supported.

4 Implementation of Separate Compilation on Lilith

A necessary pre-condition for a successful implementation of separate compilation is a powerful concept of the programming language. In addition, the quality of an implementation is mainly determined by the support of the compiler, of the operating system, and also of the code architecture of the target machine.

A convenient and heavily used implementation is running on the personal computer Lilith. The compiler performs complete checking among separate program parts and provides version control on the module interfaces. This is achieved by encoding the interface information on *symbol files*. The Lilith architecture supports separate compilation by appropriate addressing and code features. This allows the compiler to generate efficient and almost definitive code. Finally, the operating system well supports linking, loading, and execution of programs such that program development and execution is very comfortable for the programmer.

4.1 General Implementation Concept

Separate compilation with full type checking among separate program parts requires that interface information about the separate parts must be available to the compiler. Supplying the interface information in the form of source text would insufficiently regard the security aspects of separate compilation, because it is very important that all compiled program parts refer to identical information about a separate part. A better method for reliable version control is to supply the interface information in the form of a *symbolic interface description* which has been generated by the compiler itself and represents an extraction of the compiler's internal symbol table.

The implementation of this concept is facilitated by the separate compilation concept of Modula-2 which splits the interface definition of a separate module from its actual implementation. The symbolic interface description can be generated upon compilation of the definition module. The Modula-2 compiler on Lilith writes this information on a *symbol file* and considers this file as the only valid representation of the definition module for subsequent compilations. The symbol file is needed for the compilations of the corresponding implementation module and all compilation units (i.e. definition and implementation modules) which import the separate module.

This implies the simple rule that before a compilation unit can be compiled, all symbol files of the imported modules must exist, i.e. the corresponding definition modules must have been compiled previously. For a *correct sequence of compilations*, it is in particular very important to respect the dependencies of other modules when definition modules are recompiled. As soon as all symbol files of a program have been generated, it does not matter in which sequence the implementation modules are compiled. This makes programming flexible and allows fast modifications and recompilations of program parts.

The recompilation of a definition module implies that a new version of the corresponding symbol file is generated, because a change of the module's interface must be assumed. This means that all compilation units which import from this module must be compiled again. This might be very inconvenient when a basic

definition module of a large system is modified and a recompilation of the whole software is forced. Another philosophy, however, like updating an existing symbol file, if the new definition module is a true extension of the old one, would be too dangerous and end in chaos sooner or later.

It is obvious that upon compilation, and also when the the code parts are combined to a program, it must be checked that the correct sequence of compilations is respected and that all references to a separate module are based on the same interface information, i.e. the same symbol file version. For this purpose the compiler generates a time stamp, called *module key*, upon compilation of a definition module and includes it on the symbol file. Together with the name of the module, the module key allows exact identification of a certain version of a separate module.

Upon compilation of an implementation module the compiler writes the generated code of the separate module on an *object file*. It also copies the module key on the object file. This allows identification when the code is combined for execution. A simple and quick check on matching module keys ensures the correct use of the interfaces. No complex checks are necessary now; these were previously performed upon compilation.

4.2 Organization of the Symbol Files

For the generation of the symbol files two questions are important: What kind of information should be contained on the symbol file and what kind of format is appropriate to describe this information? This section describes the considerations which guided the definition of a symbol file format for the Modula-2 compiler. A main goal was a simple solution which does not enlarge the compiler unnecessarily.

4.2.1 Contents

As mentioned above, the symbol files must contain the module key of the described separate module and also the names and the module keys of the separate modules which are imported by the definition module. This information is needed to check the compatibility of the different interface descriptions.

The symbol file further must describe *all* objects which are declared in the definition module. These object descriptions usually represent a superset of the interface information needed for the compilation of importing compilation units. For the compilation of the corresponding implementation module, however, a full description of the definition module is needed. It is therefore necessary to indicate on the symbol file which of the described objects belong to the interface and may be referenced by the importing clients.

The objects described on the symbol file must be *well defined*. This means that attributes assigned to the objects upon compilation of the definition module must be indicated on the symbol file. Their values are considered as definitive and must be respected in all compilations to which the symbol file is supplied. Assigned attributes are, for example, the *size of data types*, the *offsets of record fields*, and the *address offsets of the variables*. These values are needed for code generation in

importing modules. For code generation, the address offsets of the exported procedures would also be interesting. But these are not to determine when a definition module is compiled (the procedure bodies are specified in the implementation module). Instead, *procedure numbers* are assigned to the procedures and written on the symbol file. For code generation in importing modules, these procedure numbers may be used as provisional references to the procedure code, which must be updated when the different code parts are combined to a program.

Another important fact for the contents of the symbol file is that types in Modula-2 are only compatible by declaration and not by structure. Two different type declarations with the same type structure are considered as incompatible. Therefore it is to guarantee that a type description on the symbol file keeps its identity, i.e. its name and the module environment where it is declared. An imported type cannot be replaced by an implicitly declared local type with the same structure. Apart from the objects declared in the definition module, a symbol file therefore also describes types which are imported from other modules.

It is very important for convenient use of separate compilation that the information on the symbol file is *complete*. A programmer would be disappointed, if he had to supply symbol files for modules which are not directly imported by a compilation unit. This is confirmed by experience gained with the use of the UCSD p-system. In this system, the compiler requires that information about separate units which are used by other imported units is explicitly supplied. This is really disturbing for the programmer.

4.2.2 Format

The symbol file mainly consists of information which is stored by the compiler in its symbol table. A first idea for the generation of the symbol file could be to write an exact copy of the symbol table on the file. This is a fast and efficient generation method, but it does not consider the reading of the symbol files. Upon other compilations it should be possible to restore the saved information, or at least a part of it, into the actual symbol table. The saved symbol table entries, however, cannot be restored at the same place as in the original symbol table. The references between the symbol table entries must be reassigned and this makes the integration process heavy and complex.

A better idea is to select the needed information in the original symbol table and to write it on the symbol file in a format which facilitates its restoring. If we consider that a compiler has procedures which parse the source text and generate symbol table entries of the declared objects, it is obvious to choose a format of the symbol file which is similar to the language syntax and which may be parsed by the same compiler routines. According to the fact that symbol files are more often read than generated, this method facilitates the integration of the symbol files at the expense of their generation. A further advantage is that the symbol files do not depend on internal compiler structures and therefore are not affected by changes of the symbol table representation.

These considerations lead to the definition of a symbol file format for the Modula-2

compiler (see Appendix 2) similar to the Modula-2 syntax for definition modules. It is a sequence of so-called *symbol modules* with module key, import and export list, and a sequence of declarations. The keywords are encoded in symbols and the object declarations are expanded with size and address information. The last symbol module on a symbol file represents the complete definition module. The preceding symbol modules represent the *subset interfaces* of the imported modules containing the descriptions of the types which are imported by the definition module. Within each symbol module it is important that the declaration rules of Modula-2 are respected. The declaration of an object may only reference objects which have been declared previously (except for pointers). This allows straight forward parsing by the compiler. How the Modula-2 compiler handles the symbol files is described in Chapter 8.

4.3 The Lilith Architecture for Separate Modules

The Modula-2 compiler on Lilith generates code for a so-called *M-code machine*. The M-code, which is encoded in microcode on Lilith, was designed to support the language structures of Modula-2. It enables the Modula-2 compiler to generate short, efficient, and almost definitive code. This section describes some aspects of the M-code and the Lilith architecture which are important for the separate compilation. For a complete and detailed description of the M-code and the Lilith architecture refer to [Wir81] and [Jac83].

The separate compilation concept of Modula-2 is well supported by the Lilith architecture. Each separate module may be loaded as a unit consisting of a *data frame* for its global data and a *code frame* for the code of its procedures. A global table, the *data frame table*, holds the addresses of the data frames of the loaded separate modules. A reference to the code frame of a separate module is stored in the first word of its data frame. All separate units are accessible via the data frame table, and the index of their entry in this table is used for their identification in the code. It is called the *module number*. M-code instructions which allow access to the data frame and the call of procedures from other separate modules contain as reference this module number (load external, store external, call external).

Very important for the code generation are the available addressing modes in the target code. It is desirable that a compiler generates *almost definitive code* which does not require a lot of additional updates when the code of the separate modules is combined to a program. This may be achieved by addressing code and data relatively.

The Lilith architecture enables relative addressing with a set of address registers. The currently executed instruction is addressed by the *PC* register (program counter) which is defined relatively to the beginning of the code frame of the corresponding separate module. The address of this code frame is held by the *F* register (code frame address). The address of the data frame of the same module is assigned to the *G* register (global data frame address). Code and global data addresses within the own module can therefore be assigned definitively by the compiler. If a procedure of another separate module is called, new values are implicitly assigned to the F and G registers. The Lilith architecture for addressing

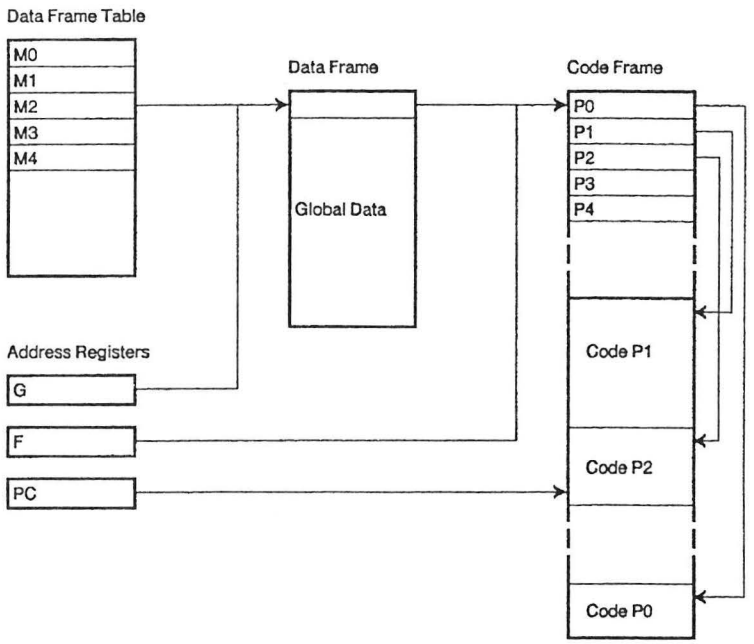


Figure 4.1 Lilith Architecture: Addressing of a Separate Module

Figure 4.2 M-Code Instructions: External References

separate modules is illustrated in Figure 4.1.

M-code instructions for calling procedures do not contain an address offset. Instead, a *procedure number* must be specified. This is an index to the *procedure entry table* which must be included at the beginning of the code frame and contains the address offsets of all procedures of the module.

The Lilith architecture is well suited not only for local references. Also the code for external references may be generated almost definitively. It is possible to insert the information contained on the symbol file directly. These are the procedure numbers for procedure calls and the address offsets for access to variables. Only the module numbers cannot be assigned definitively. For program execution the actual indices to the data frame table are needed which are only known when the modules are loaded. Instead of the actual module numbers of the loaded code, the generated code contains provisional module numbers which must be updated by the linking-loader. Examples of M-code instructions for external references are given in Figure 4.2.

4.4 Program Linking, Loading, and Execution

Modula-2 programs on Lilith run under the *Medos-2* operating system [Knu83]. Medos-2 itself is written in Modula-2 and consists of several separate modules. Some of these modules represent the public interface of Medos-2. From Modula-2 programs it is therefore easy to call functions of the operating system and, of further advantage, the compiler can check whether the interface is correctly used. The most important modules of the Medos-2 interface are the module *FileSystem*, the module *Terminal*, and the module *Program*. The former two modules support input/output, the latter supports memory layout and program linking, loading, and execution.

Generally, a program consists of a *main module* (i.e. the main program) and of all separate modules that are (directly or indirectly) imported by the main module. The fact that the code generated by the compiler is almost definitive allows to collect and combine the needed modules when the program is loaded. No explicit previous linking is necessary. This is respected by the compiler which writes the code onto the *object file* in a format as it is expected by the linking-loader of Medos-2 (see Appendix 3).

Apart from the code information, the object file also contains an *interface table* and *fix-up lists*. Both of them are interpreted by the linking-loader. The interface table consists of pairs of module names and module keys. The first entry identifies the module represented by the code itself, the other entries indicate which separate modules are imported by this module. The fix-up lists refer to places in the code where the provisional module numbers must be updated.

Upon loading of a module, the linking-loader first determines its actual module number (index to the data frame table) and the actual module numbers of the imported modules, and then it enters the addresses of the data frame and the code frame. After loading the code, it executes the fix-up commands. The provisional module numbers in the code correspond to the module references in the interface

table. They are replaced by the actually assigned module numbers. Afterwards, the linking-loader loads the imported modules (as far as they are not already loaded) and checks the module keys to approve that the modules are compatible.

After all needed modules are loaded, the execution of the program is started by a call to the main module. According to the implicit initialization of local modules, a separate module first calls the initialization parts of the imported modules before its body is executed. A special code sequence guarantees that each module is initialized only once.

Medos-2 applies a stack principle to the program execution. It allows a program to *call* another program, which is immediately loaded and started like a procedure. Medos-2 may be considered as the first entry in the program stack. If upon calling a new program some of the imported modules have already been loaded with another program on the stack, the linking-loader takes their module numbers for the updates in the code. Generally, a module cannot be loaded twice.

During its execution, a program may require certain resources which are controlled by the operating system (e.g. memory space in the heap for its pointer-referenced data). Normally, this resources are automatically reclaimed by the operating system after termination of the program. In some cases, however, it makes sense that resources are shared by several programs (e.g information stored in the heap). For this purpose, Medos-2 provides a *sharing facility*. Upon calling a program, it must be indicated whether or not the called program should *share* its resources with the caller.

The sharing facility is, for example, used for the execution of the Modula-2 multi-pass compiler (see Chapter 5). Several compiler parts are called sequentially from a common compiler base, while the information stored in the heap survives and the connected files remain opened.

5 Overview of the Modula-2 Compiler

The Modula-2 multi-pass compiler on the personal computer Lilith has been developed at the Institut in the time from 1977 to 1980. It is written in Modula-2 itself and uses heavily the separate compilation facility. This chapter first gives a short review on the most important development steps, and then it describes the overall organization of the Modula-2 compiler. The internal structure of the compiler and the functions of the compiler parts are described in the subsequent chapters.

5.1 History of the Compiler Development

The design of the programming language Modula-2 was started in 1977. The language design was accompanied by an implementation effort on a PDP-11/34 computer.

The starting-point for the development of the Modula-2 compiler was the seven-pass compiler for the programming language *Modula* [Wir77] (from here on called *Modula-1*), which was designed and implemented for the PDP-11 by Le Van Kiet between 1975 and 1977. This Modula-1 compiler is described in [Le78].

Step by step the Modula-1 compiler was changed to a Modula-2 compiler written in Modula-2 itself. The first step was the implementation of pointer types and record types with variants. A restructuring of the compiler followed by using pointers and variant records for the representation of the symbol table. In the next steps further extensions and changes from Modula-1 to Modula-2 were implemented and applied in the compiler (e.g. for-statement, value parameters instead of constant parameters). At the same time the number of compiler passes was reduced to five. By the end of 1978, a first version of the Modula-2 compiler for the PDP-11 was completed. It accepted the language as described in the first Modula-2 report [Wir78].

The next development goal was the implementation of separate compilation, which again needed several bootstrapping steps. In a first phase, the compiler was extended to accept definition modules, to handle the symbol files, and to generate relative code. It was also necessary to develop a program linker and a new basic executive system on which the separately compiled programs are executed. Afterwards, the compiler passes were split into separate modules. This allowed to test the appropriateness and usefulness of the new facility. An advantage for the internal security of the compiler was that now all compiler parts could refer to a unique definition of the commonly used data structures. This work was finished by the end of 1979. This compiler version accepted the language described by the second Modula-2 report [Wir80].

In the meantime a cross-compiler project for the new personal computer Lilith [Wir81] was started as well. A goal of this project was to keep the first three passes of both compilers as similar as possible. The code generation part was rewritten from scratch and compressed into one pass.

With the cross-compiler it was possible to develop the software for the operating system of Lilith using the PDP-11. In the summer of 1980, the Lilith hardware and the

operating system *Medos-2* were developed to a state which allowed to transport and run the Modula-2 compiler on the new computer. After this first implementation on Lilith, the compiler was consolidated and improved. Apart from error corrections, also inappropriate *"inherited"* structures and names were changed. This sometimes turned out to be time consuming when the basic structure of the compiler was touched.

In this paper the current state of the Modula-2 compiler on Lilith is reported. Most of the description is also valid for the compiler on the PDP-11. The actual language reference is contained in [Wir82]. A user's guide and a list of the compiler's error messages is contained in the *Lilith Handbook* [Gei82].

5.2 Compiler Parts

The Modula-2 multi-pass compiler is divided into several parts. For its execution, the feature of shared program calls (see section 4.4) is used. A base part, which remains loaded during the whole compilation time and in which the global compiler data are declared, controls the compilation process and calls subsequently the other compiler parts, i.e. the initialization part, the various passes, and the listing generator. The names and the duties of the various compiler parts are:

Modula	Base part, common data, control of the compilation process.
Initialization	Initialization of output files.
Pass1	Syntax analysis.
Pass2	Declaration analysis.
Pass3	Body analysis.
Pass4	M-code generation.
Symfile	Symbol file generation.
Lister	Listing file generation.

The decomposition of the compiler parts into separate modules and the structures and functions of these modules are described in Chapter 8.

Information transfer between the compiler passes is based on *inter-pass files* and on the *symbol table*. On the inter-pass files a symbolic skeleton of the compiled unit is transmitted. The symbol table is stored in the heap of the memory and resides there during the whole compilation process. It describes the declared objects and is a large network of mainly two kinds of entries: *name entries* for the representation of objects and *structure entries* for the representation of data structures. The handling of these global compiler data is described in Chapter 6 and Chapter 7.

The following table shows some figures about the size of the Modula-2 compiler. For each compiler part it lists the number of *separate modules*, the number of *source text lines* (definition and implementation modules), and the memory space needed for *code* and *global data*. More details about the size of the compiler modules are given in Appendix 6.

Modula	5 modules	1177 lines	1.97 kword
Initialization	4 modules	1008 lines	1.88 kword
Pass1	8 modules	3299 lines	6.46 kword
Pass2	5 modules	3479 lines	5.91 kword
Pass3	4 modules	2887 lines	4.92 kword
Pass4	7 modules	3832 lines	7.80 kword
SymFile	3 modules	837 lines	1.08 kword
Lister	2 modules	618 lines	2.17 kword
Modula-2 Compiler	38 modules	17137 lines	32.23 kword

Upon execution, the base part Modula and one of the other compiler parts are loaded. The space needed in the heap of the memory (e.g. for the symbol table) depends on the size and complexity of the compiled unit. For compiling a large compilation unit (2000 lines), about 10 kword are needed in the heap.

5.3 Compiler Execution

Source text units accepted by the compiler for compilation (i.e. *compilation units*) are, according to the Modula-2 syntax definition, definition modules, implementation modules, and separate modules without export. Apart from the input file with the source text, the compiler requires the symbol files of the imported modules.

The kind of the compiled unit and the success of the compilation process determine which compiler parts are called and which output files are generated by the compiler. Generally, Initialization is the first and Lister, which generates the program listing of the compiled unit, is the last called part. In between, (some of) the compiler passes are called subsequently. If not all needed symbol files are available, the compilation is immediately stopped in Pass1, and no output file is generated.

For definition modules the compilation process stops after Pass2 and, if no errors had been detected so far, the compiler part Symfile is activated to generate the *symbol file*.

For other compilation units, Pass2 generates a *reference file* which describes the structure of the compiled unit and is used by the post-mortem debugger. Pass4 is called, if no error has been detected by the first three passes. It translates the compiled unit into M-code and writes the generated code on the *object file*.

6 Global Compiler Data

Data types and variables which are commonly used by the compiler parts are declared and exported by a separate module (MCBase) of the compiler base part Modula. Information about the compiled unit is stored in the *identifier table*, on the *inter-pass files*, and in the *symbol table*.

6.1 Identifier Table

The multi-pass organization of the compiler requires that the objects declared in the compilation unit be recognized in the different passes. Instead of passing identifiers to the subsequent passes, they are stored by Pass1 in an identifier table and replaced by a number, called *spelling index*. Replacing the identifiers by numbers is also reasonable for searching an object in the symbol table, because it is easier and faster to compare numbers instead of strings. Further, the compiler is not forced to restrict the number of significant characters in the identifiers for implementation purposes.

Identifiers are no longer internally used after Pass1, but they are needed again for identification on some output files (the symbol file and the reference file). This means that the identifier table must survive Pass1. For historical reasons (limited memory space on the PDP-11), this table is not declared in the base part. Instead, it is dumped on a compiler work file, called ASCII, and the spelling indices refer to the position of the identifiers in this file. The file is used by Pass2 and the symbol file generation part Symfile.

6.2 Inter-pass Files

The scanner of Pass1 transforms the source text of the compilation unit into a *sequence of symbols* which represents the syntactical structure of the module. Each compiler pass reads, processes, and modifies this information and proceeds the remaining skeleton to the subsequent pass. For passing the symbol sequence to the next compiler pass, it is written on an *inter-pass file*. For this purpose the compiler alternately uses the work files IL1 and IL2; IL1 between Pass1 and Pass2 and between Pass3 and Pass4 or Lister, IL2 between Pass2 and Pass3 or Lister and between Pass4 and Lister.

The symbols are encoded according to the global compiler type *Symbol* which reflects the syntactical units of Modula-2:

```
Symbol =
  (eop,
   andsy, divsy, times, slash, modsy, notsy, plus, minus, orsy,
   eql, neq, grt, geq, lss, leq, insy,
   lparent, rparent, lbrack, rbrack, lconbr, rconbr,
   comma, semicolon, period, colon, range,
   constsy, typesy, varsy,
   arraysy, recordsy, variant, setsy, pointersy, tosy, arrow, hidden,
   importsy, exportsy, fromsy, qualifiedsy,
```

```
codesy, beginsy,
casesy, ofsy, ifsy, thensy, elsifsy, elsesy, loopsy, exitsy,
repeatsy, untilsy, whilesy, dosy, withsy,
forsy, bysy, returnsy, becomes, endsy,
call, endblock,
definitionsy, implementationsy, proceduresy, modulesy, symbolsy,
ident, intcon, cardcon, intcarcon, realcon, charcon, stringcon,
option, errorsy, eol,
namesy,
field, anycon)
```

In addition to this code, each symbol contains a position number which refers to the position of the represented item within a source text line. The position numbers are needed by the Lister to indicate detected errors at the corresponding position in the program listing. Some symbols contain further information. Symbols for constant numbers contain the number's value and the symbol representing an identifier contains the corresponding spelling index.

Interspersed in the symbol sequence representing the syntactical structure of the compiled unit, additional symbols represent the line structure of the source text, the compiler options which may be specified in the source text, and the error messages which are generated by the compiler. The complete syntax of the symbol sequences on the inter-pass files is described in Appendix 4.

6.3 Symbol Table Entries

The *symbol table* of the Modula-2 compiler is a collection of symbolic information about the objects declared in a compilation unit. It is built up by Pass2 and stored in the heap of the memory as a large network of primarily two different types of entries. These are the *name entry* which describes named objects and the *structure entry* which describes type structures. References and links between these entries are established by pointers (*Idptr* is a pointer to a name entry; *Stptr* is a pointer to a structure entry):

```
Idptr = POINTER TO Identrec;     (* name entry *)
Stptr = POINTER TO Structrec;    (* structure entry *)
```

Note: In the Modula-1 compiler the symbol table consisted, due to lack of pointer types, of several array-tables allocated in a common data space. The more flexible representation in the heap was introduced as soon as pointers and variant records were implemented by the new compiler. The redesign of the symbol tables, now as a network of pointer-referenced entries was also influenced by the symbol table representation in the Pascal-6000 compiler [Amm75]. In fact, the symbol table is similar to that of the Pascal compiler.

6.3.1 The Name Entry

Several *classes* of named objects are distinguished by the compiler (e.g. constant-identifier, type-identifier). The enumeration type *Idclass* specifies the classes:

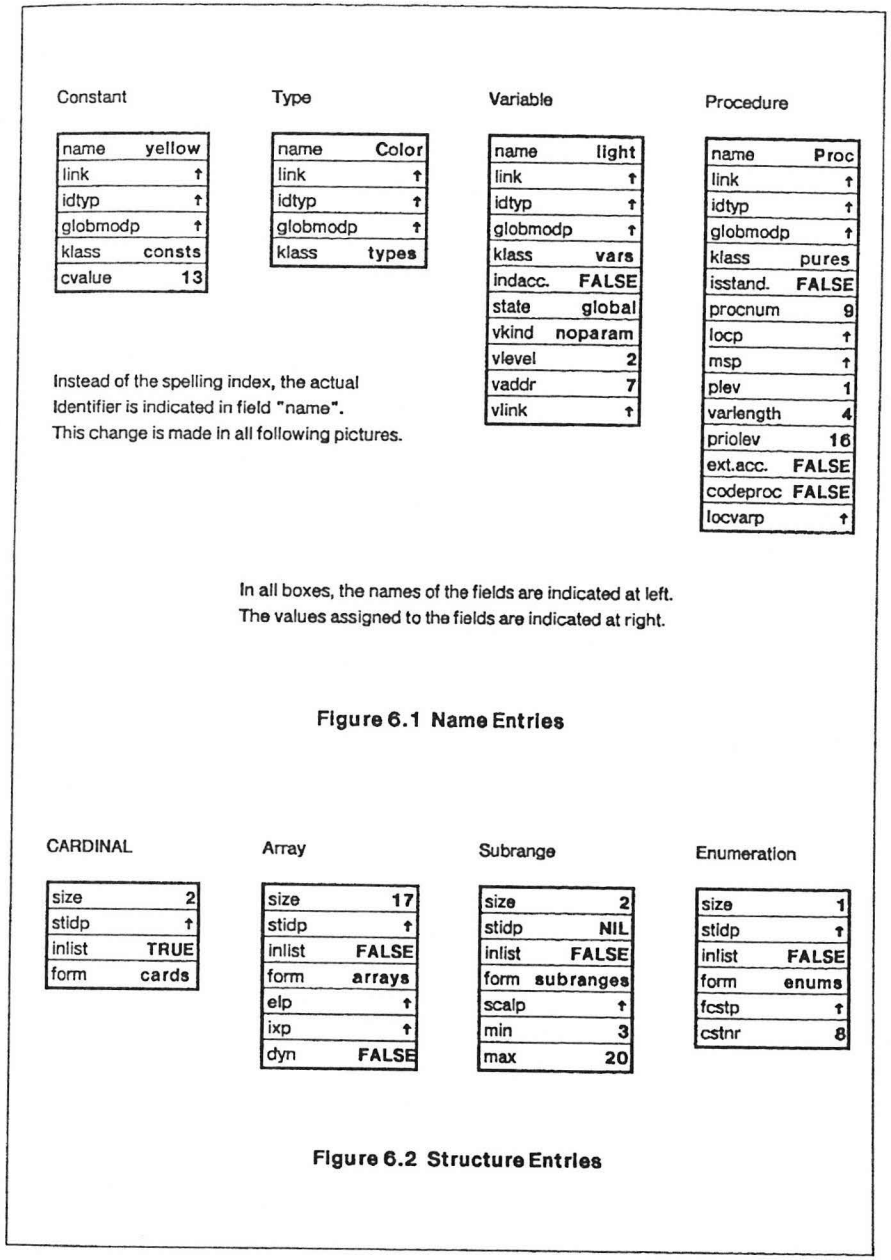


Figure 6.1 Name Entries

Figure 6.2 Structure Entries

```
Idclass =
  (consts, types, vars, fields, pures, funcs, mods, unknown, indrct)
```

A name entry is a record which consists of a part which is common to all object descriptions and a variant part which contains class-specific information (e.g. the value of a constant). It is defined by the type *Identrec*:

```
Identrec =
  RECORD
    name: Spellix;
    link: Idptr;
    CASE BOOLEAN OF
      FALSE: nxtidp: Idptr;
    | TRUE : idtyp : Stptr;
    END (* CASE BOOLEAN *);
    globmodp: Idptr; (* pointer to global module *)
    CASE klass: Idclass OF
      types: (* no further fields *)
    | consts, unknown: (* unknown may convert to consts *)
        cvalue: Constval;
    | vars:
        indaccess : BOOLEAN; (* indirect access to value *)
        state : Kindvar;
        vkind : Varkind;
        vlevel : Levrange;
        vaddr : CARDINAL; (* offset *)
        vlink : Idptr;    (* variables or parameters *)
    | fields:
        fldaddr: CARDINAL;
    | pures, funcs, mods:
        CASE isstandard: BOOLEAN OF
          TRUE:
            CASE Idclass OF
              pures: pname: Stpures;
            | funcs: fname: Stfuncs;
            END (* CASE Idclass *)
        | FALSE:
            procnum : CARDINAL;
            locp : Idptr;
            extp : Listptr;
            plev : Levrange;
            varlength : CARDINAL;
            priolev : CARDINAL;
            externalaccess : BOOLEAN;
            CASE Idclass OF
              pures, funcs:
                CASE codeproc : BOOLEAN OF
                  FALSE:
```

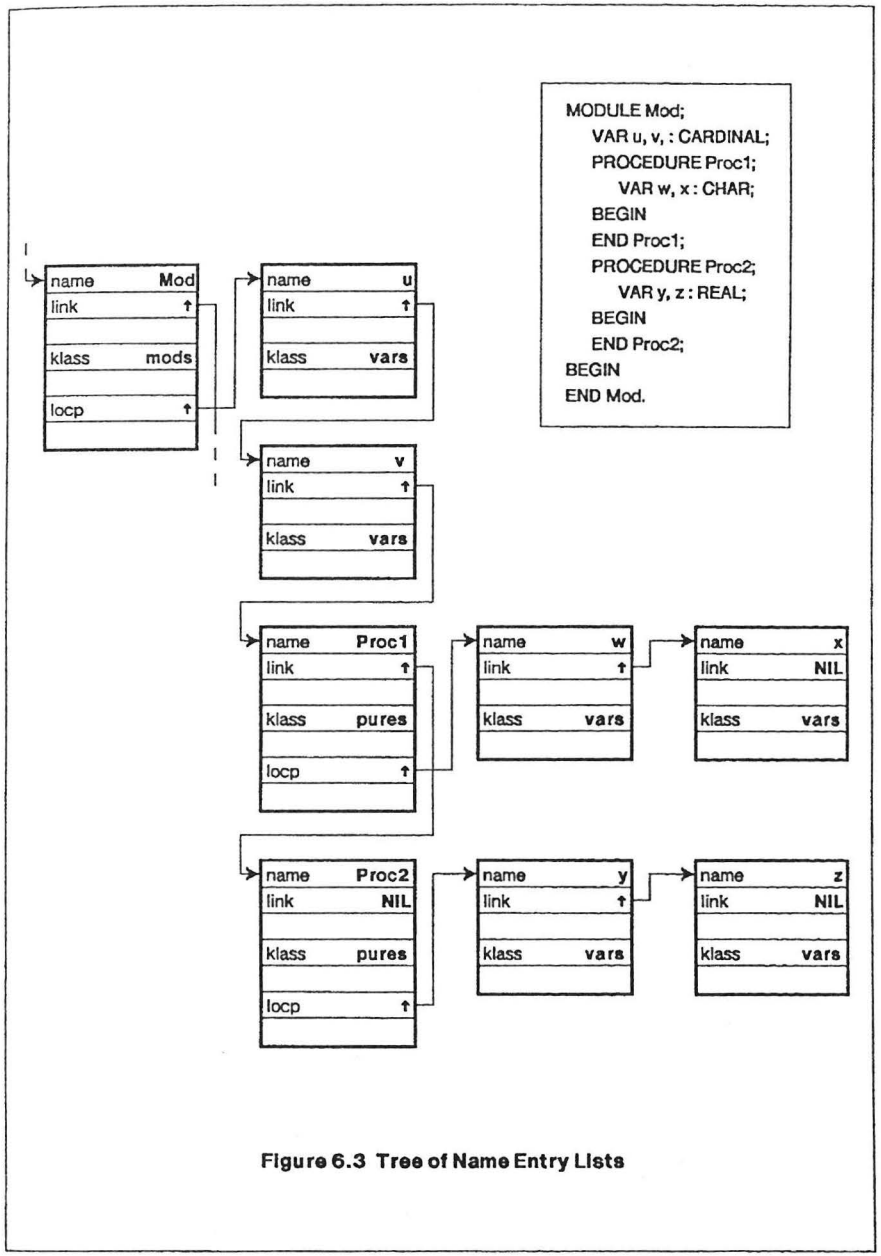



Figure 6.3 Tree of Name Entry Lists

```
                locvarp : Idptr; (* variables, no parameters *)
              | TRUE:
                codelength : CARDINAL;
                codeentry : CARDINAL;
              END; (* CASE BOOLEAN *)
          | mods:
              impp: Listptr;
              expp: Idptr;
              qualexp: BOOLEAN;
              CASE globalmodule : BOOLEAN OF
                FALSE: (* no further field *)
              | TRUE:
                globvarp : Idptr; (* global variables *)
                modnum : CARDINAL;
                modulekey : Keyarr;
                identifier : Modnamarr;
              END; (* CASE BOOLEAN *)
          END; (* CASE Idclass *)
      END; (* CASE BOOLEAN *);
    | indrct: (* no further fields *)
  END; (* CASE Idclass *)
END (* RECORD*)
```

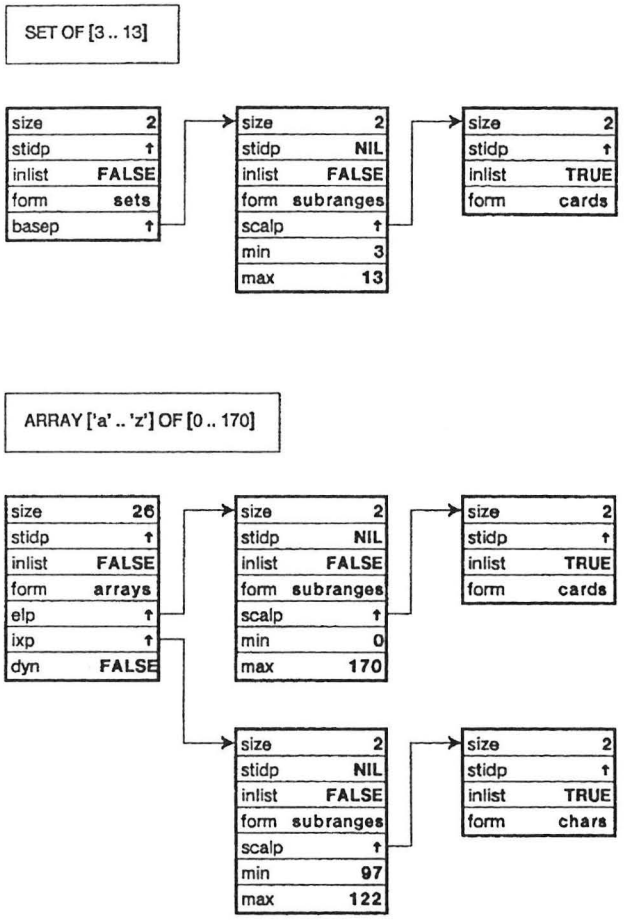
Examples of name entries are given in Figure 6.1.

The fields of the common part are *name* which is the spelling index of the represented object, *link* which points to the next name entry in a linked list, *idtyp* which refers to the type structure of the object, *globmodp* which refers to the name entry of the separate module in which the object is declared, and *klass* which indicates the class of the object. Instead of *idtyp*, the field *nxtidp* is alternatively used in some special cases (see section 7.4). In order to save memory space this two fields are overlaid in the name entry.

The reference to the entry of the own separate module (field *globmodp*) is needed for the generation of the symbol file and for code generation. In both cases the compiler must know to which separate module an object belongs.

The name entries are organized as a *tree of linear lists*, representing the nesting structure of the declarations in the compiled unit. Name entries for objects declared in the same block are linked in a linear list (field *link*), which is *ordered* according to the spelling index. Each linear list may be considered as the list of the *local* objects of a procedure or module. Its heading is the field *locp* of the corresponding procedure or module entry (for exported objects see section 7.4). Figure 6.3 shows the organization of the name entries for a small module.

Originally, the local objects were organized as a binary tree. The tree was replaced by the linear list when the compiler was extended for separate compilation. Two reasons forced this change. First, the limited memory space of the PDP-11 no longer allowed reserving two words for linking a tree. Second, the spelling index became an index to the identifier table, and no longer to the hash table of the scanner in



Remark: There is just one structure entry "cards" in the symbol table.

Figure 6.4 Representation of Type Structures

Pass1. (This change was caused by the need to regain the character representation of the identifier from the spelling index for the generation of the symbol file and the reference file.) The identifiers are sequentially entered into the identifier table, according to their first occurence in the compiled text. This means that generally an identifier declared later gets a higher spelling index, and therefore the binary tree would degenerate to a nearly linear list as well. Compilation with a linear list was marginally slower (about 3% for a large program) than compilation with the binary tree organization. So it was decided that it was not worth loosing one word in each name entry. A side effect of this change was that the searching procedures became simpler.

6.3.2 The Structure Entry

Several *forms* of type structures are described by the structure entries. The different forms are specified by the enumeration type *Structform*:

```
Structform =
  (enums, bools, chars, ints, cards, words, subranges, reals,
   pointers, sets, proctypes, arrays, records, hides, opens)
```

A structure entry is a record which also consists of a common part and of a variant part with form-specific information (e.g. base type and bound values of a subrange). It is defined by the type *Structrec*:

```
Structrec =
  RECORD
    size: CARDINAL; (* word size *)
    stidp: Idptr; (* identifier defining this structure *)
    inlist: BOOLEAN; (* structure entered into a list *)
    CASE form: Structform OF
      bools, chars, ints, cards, words, reals: (* no field *)
    | enums:
        fcstp: Idptr;
        cstnr: CARDINAL;
    | subranges:
        scalp: Stptr;
        min: CARDINAL;
        max: CARDINAL;
    | pointers:
        elemp: Stptr;
    | sets:
        basep: Stptr;
    | arrays:
        elp: Stptr;
        ixp: Stptr;
        dyn: BOOLEAN;
    | records:
        CASE rpart: Recpart OF
```

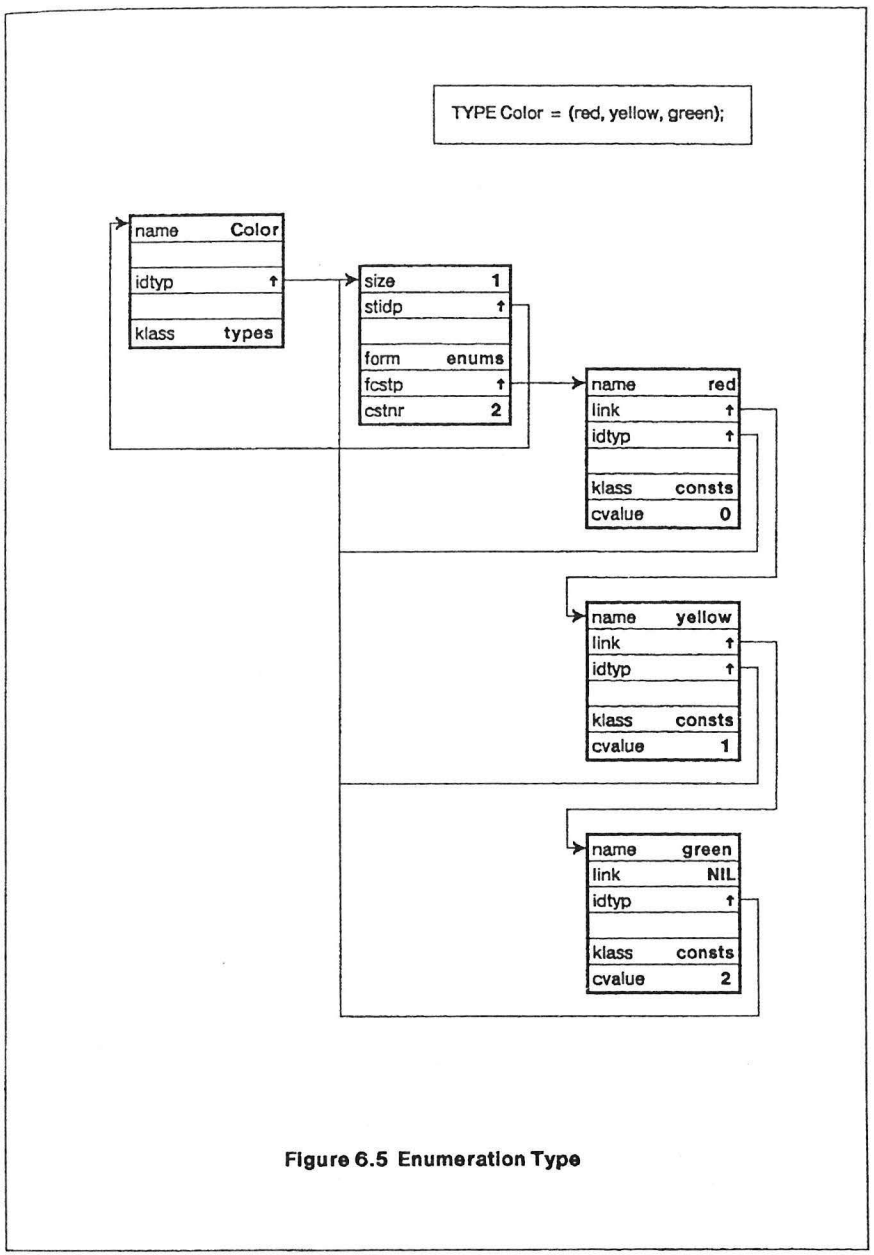



Figure 6.5 Enumeration Type

```
        fixedpart:
          fieldp: Idptr;
          tagp: Stptr;
      | tagfield:
          fstvarp: Stptr;
          elsevarp: Stptr;
          tagtyp: Stptr;
      | variantpart:
          nxtvarp: Stptr;
          subtagp: Stptr;
          varval: CARDINAL;
      END (* CASE Recpart *)
  | proctypes:
      fstparam : Idptr; (* pointer to parameter list *)
      CASE rkind: Idclass OF
        funcs : funcp : Stptr; (* pointer to function type *)
      | pures : (* no further fields *)
      END; (* CASE Idclass *)
  | hides, opens: (* conversion from hides to opens *)
      openstruc : Stptr; (* used for opens *)
  END (* CASE Structform *)
END (* RECORD *)
```

Examples of structure entries are given in Figure 6.2.

The fields of the common part are *size* which indicates the memory space needed for allocation of a variable of this type, *stidp* which refers to the name entry of the object which declared the structure (e.g. the type identifier), *inlist* which is a flag for the compiler, and *form* which indicates the specific form of the structure.

A structure entry is considered as a *unique* description of a type structure, and this implies that the name entries of objects with the same data structure always refer to the same structure entry. If a type is explicitly declared with a type identifier, a back-reference from the structure entry to the type name entry is established (field *stidp*). This pointer and the field *inlist* are used by the compiler for the generation of the symbol file and the reference file.

The complexity of the structural representation of a type corresponds to the complexity of its declaration. Almost all structure entry refer to further type descriptions. Figure 6.4 shows examples for the representation of subranges, sets, and arrays. For an array, the index type is always entered as a subrange.

In the structure entry of an enumeration type, field *fcstp* is the heading of the name entry list of the corresponding enumeration constants. These values are entered into a separate name entry list and not into the local list of the corresponding procedure or module. The separation is necessary because of the import and export rules of Modula-2. If an enumeration type is imported (exported), all enumeration constants are implicitly imported (exported) as well (see Chapter 7). Figure 6.5 shows the representation of an enumeration type.

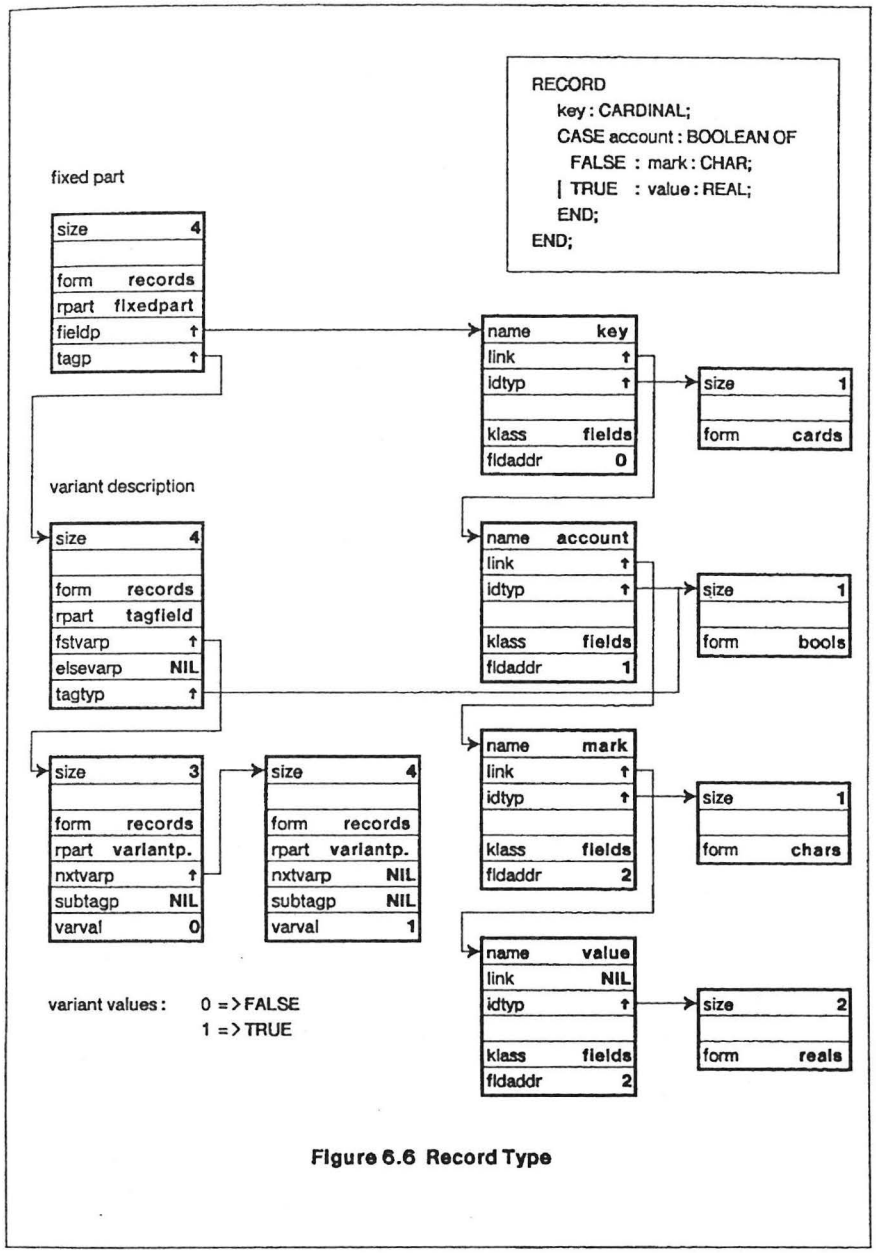


Figure 6.6 Record Type

A record type is, in general, a very complex structure. Each record field may have its own type, and it is possible to overlay fields within variant sections. A structure entry of a record type primarily consists of a so-called *fixed part* with field *fieldp* as a pointer to the name list of all fields declared in the record, and with field *tagp* as a pointer to the description of the variants. According to the language definition which does not support access control on variant sections, the compiler does not check whether or not fields declared in variant sections are accessed legally, i.e. consistent with the tag field value. Therefore, the description of the variant structure in the symbol table serves as size information only. It is separated from the representation of the field list and is only generated for a variant section at the end of the record declaration. Figure 6.6 shows the representation of a record type.

Modula-2 allows the declaration of procedure types. Therefore, a structure entry for procedures is needed. The structural information consists of the parameter list, assigned to field *fstparam*, and, for function procedures, the result type. The parameter list links the parameters (variable entries) via field *vlink*. Each (constant) procedure is considered as specifying its own type structure (the handling of compatibility with procedure variables is explained in 8.6.2). Figure 6.7 shows the representation of a procedure.

6.4 Pointers to the Symbol Table

In module MCBase, several pointer variables are declared which hold the *roots of the symbol table*. They provide direct access to important name entries:

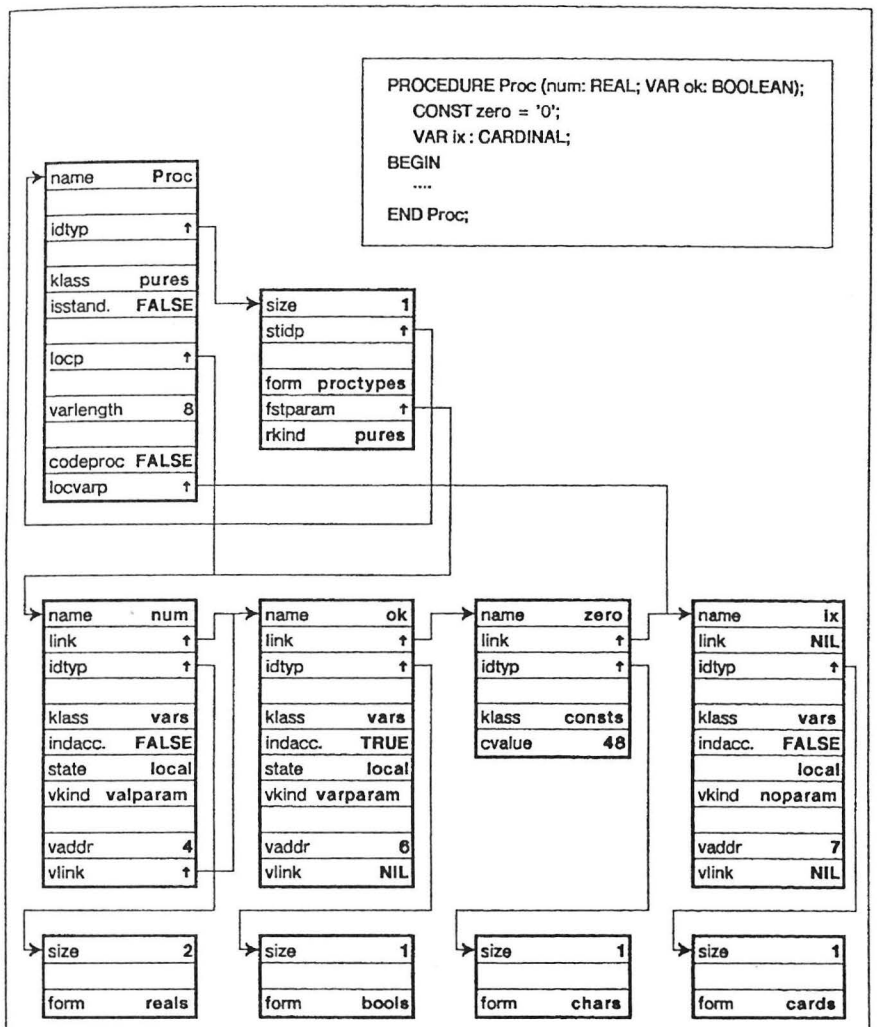
```
root     : Idptr;
sysmodp  : Idptr;
mainmodp : Idptr;
```

The variable *root* represents the root of the whole symbol table. The first entry marks the so-called *standard module*. Its export list contains the objects which are declared *standard* in Modula-2. The local list of the standard module links the separate modules which are involved in the current compilation. These are the module SYSTEM, the compiled unit, and all imported separate modules.

The pointer *sysmodp* provides a direct reference to the name entry of the module *SYSTEM*. The pointer *mainmodp* does the same for the compilation unit. Both pointers are used by the compiler very frequently.

Furthermore, the compiler must explicitly know the pointers to the structure entries of the *standard types*. For example, they are used in Pass1 for the assignment of a type to constants, and in Pass3 for the implementation of the type compatibility tests. These pointers are declared in module MCBase as follows:

```
boolptr   : Stptr;
charptr   : Stptr;
intptr    : Stptr;
cardptr   : Stptr;
intcarptr : Stptr;
```

Remark: The procedure mark entry needs four words on the stack. Therefore, the first parameter is alloccated with offset 4.

Figure 6.7 Procedure Declaration

```
realptr   : Stptr;
procptr   : Stptr;
bitsetptr : Stptr;
```

Figure 6.8 gives an illustration of the overall organization of the symbol table.

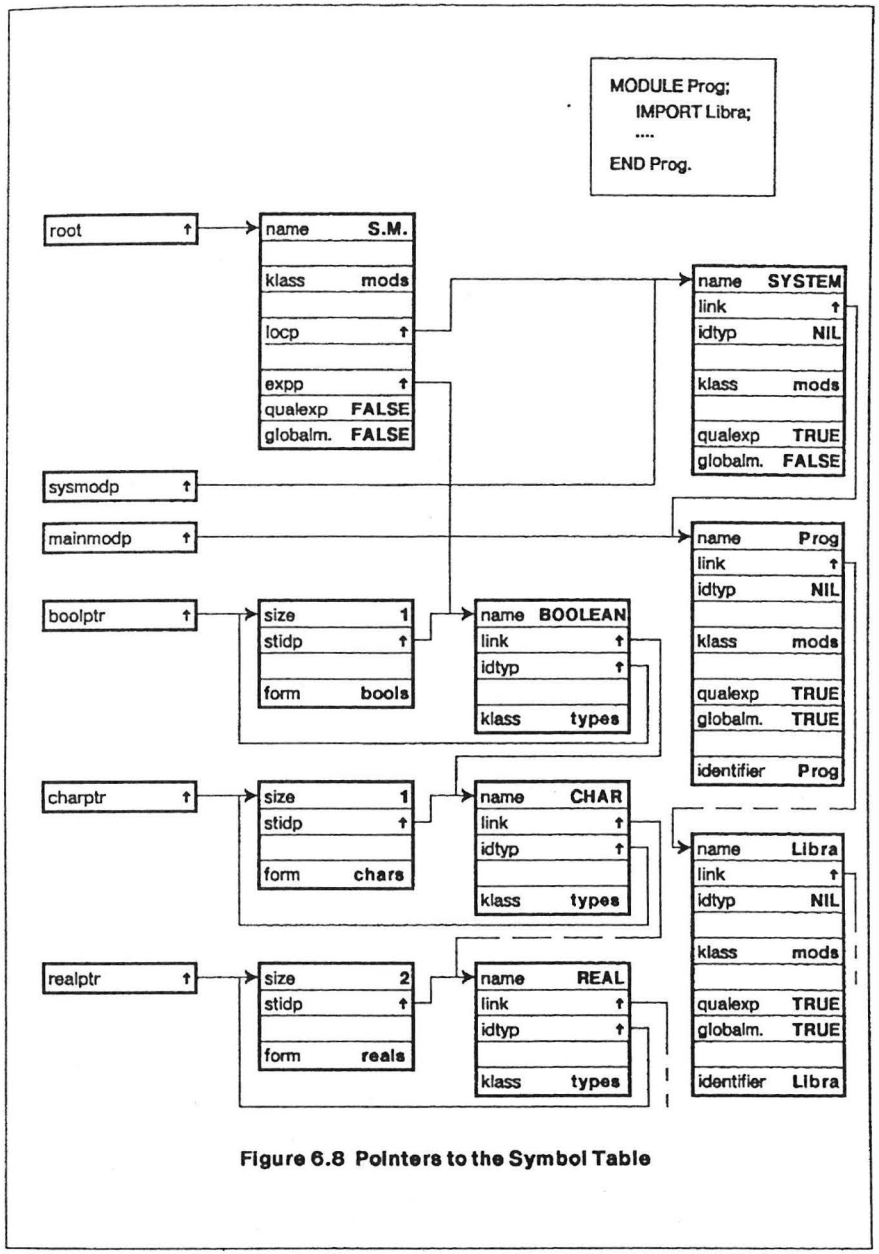


Figure 6.8 Pointers to the Symbol Table

7 Scope Handling

7.1 Scope Rules

The *scope rules* of Modula-2 specify in which parts of a program a declared object is known and accessible. Procedures and modules are considered to constitute a so-called *scope*. So a compilation unit defines a system of scopes which are nested in the same manner as procedures and modules. In addition, a new scope is also established by record declarations (for the declared fields) and by with-statements.

For a better understanding of the structures built up in the compiler for scope handling, the scope rules are listed here:

1) Generally, an object is known in the scope where it is declared. The identifier of the object must be *unique* within the set of objects in this scope.

2) If an object is used in a declaration, then it must be declared itself prior to its use. This rule is relaxed for a type referenced in a pointer declaration; it may be declared later in the same scope. This rule does not apply to objects referenced in statements.

3) For a scope established by a procedure or a with-statement, an object known outside the scope is also known inside, unless a new object with the same identifier is specified within this scope.

4) For a scope established by a module, an object defined outside is also known inside if it is imported. An object specified inside the scope is also known outside if it is exported unqualified. The objects defined as standard objects of Modula-2 are always implicitly known within the scope of a module (*pervasive* objects).

5) Qualified exported objects are accessible outside the module only if prefixed by the module's name. If the objects are explicitly imported by another module (import with FROM), they are directly known by their name within this module.

6) The fields of a record are accessible only in field designators or directly in with statements which refer to a variable of this record type.

7.2 Scope Display

Pass2 and Pass3 check that the scope rules are respected by the compiled unit. Both passes use an identical control structure to handle the scopes, the so-called *scope display*. This is organized as a stack, representing the current nesting of the scopes. On each level of the display several lists are appended, which provide access to the objects specified in the corresponding scope. In Pass2 the display is also used to enter the new declared objects into the name entry lists of the top scope.

Name searching procedures use the scope display. To find an object with a given identifier (spelling index), the searching procedures must scan through all lists from the top scope down to the next scope of a module until an object with the specified name is found.

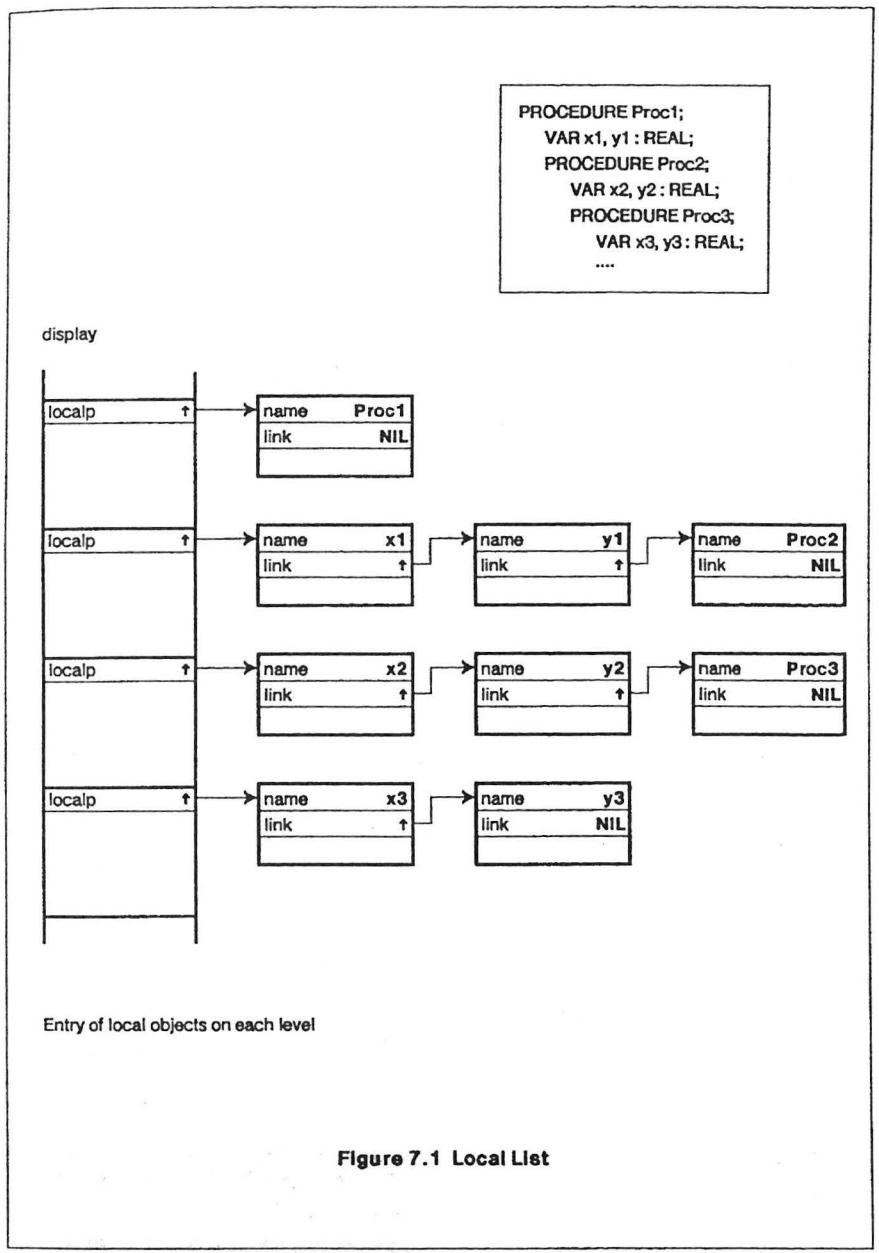


Figure 7.1 Local List

In Pass2, the display is declared as an array of records:

```
display : ARRAY Scoperange OF
            RECORD
              localp : Idptr;
              extendp : Listptr;
              exportp : Idptr;
              modulep : Idptr;
              forwardp : Forwardptr;
            END;
```

In each display entry, field *localp* holds the locally declared objects and field *extendp* the objects with extended validity range (e.g. exported by an inner scope). The fields *exportp* and *modulep* are only used for the scopes of modules and refer to the export list and the module itself (e.g. to provide access to the import list of the module). The field *forwardp* holds the list of pointer types whose referenced type are declared later.

The display grows and shrinks according to the depth of nesting. At the end of a procedure or module, Pass2 assigns the lists of the top scope to the corresponding fields of the procedure or module name entry in the symbol table (*localp* to *locp*, *extendp* to *extp*, and *exportp* to *expp*). Saved in this way, the display can easily be reconfigured in Pass3.

A detailed description of the different lists is given in the next sections. It refers to the organization and handling of these lists in Pass2.

7.3 Local List

The *local list* links name entries (via field *link*) and is appended to the display entry field *localp*. For procedures, it contains the objects declared local to the procedure. For modules, it includes only those objects which are not exported. The local list is ordered according to the spelling index of the objects (see 6.3.1). Therefore, a new object is inserted at the corresponding position in the list.

Examples of local lists are given in Figure 7.1.

7.4 Export List

The *export list* links name entries (via field *link*) and is appended to the display field *exportp*. It contains the exported objects of a module. This list is used for modules only.

When a module is compiled, Pass2 reads, before a new scope is established, the names of the exported objects and generates a provisional list of name entries. In the case of unqualified export, it checks at the same time that the exported names are *unique* in the environment of the module, i.e. not already known in the current scope.

In this provisional version of the export list, all name entries have the field *klass* set to

the value *unknown* (only the name, but not the object is known) and a back-reference to the preceding entry is assigned to field *nxtidp*. This field is overlaid on the field *idtyp*, which is not used for name entries of unknown class. The list is assigned to field *expp* of the module entry. For an example see Figure 7.2.

After the provisional export list has been generated, a new scope of the module is established and the provisional export list is assigned to the display entry field *exportp*. If the declaration of an exported object is processed, the new symbol table entry is not linked into the local list of the current scope. Instead, the provisional unknown entry in the export list is replaced by the new definitive entry. Field *nxtidp* is used to update the field *link* of the preceding list element. The exchange is illustrated by Figure 7.3.

At the end of processing the module, Pass2 scans the actual export list and checks whether or not all name entries are definitive; i.e. all exported objects have been declared.

7.5 Import List

The *import list* is a chain of pointers to name entries, representing the imported objects of a module. It is generated when the import clauses of the module are processed and contains:

- The objects *explicitly specified* in the import clauses.
- The *nested local modules* which are exported unqualified by imported local modules, to provide access to the list of their unqualified exported objects.
- The *enumeration types* which are exported unqualified by imported local modules, to provide access to the list of the corresponding enumeration constants.
- The *"standard module"* exporting the standard objects, which are implicitly known within all modules and therefore are called *pervasives*.

Modules and enumeration types nested within an imported module are entered separately. This allows the implementation of a simpler algorithm for searching an object in the import list, i.e. only linear lists must be scanned and not also nested structures.

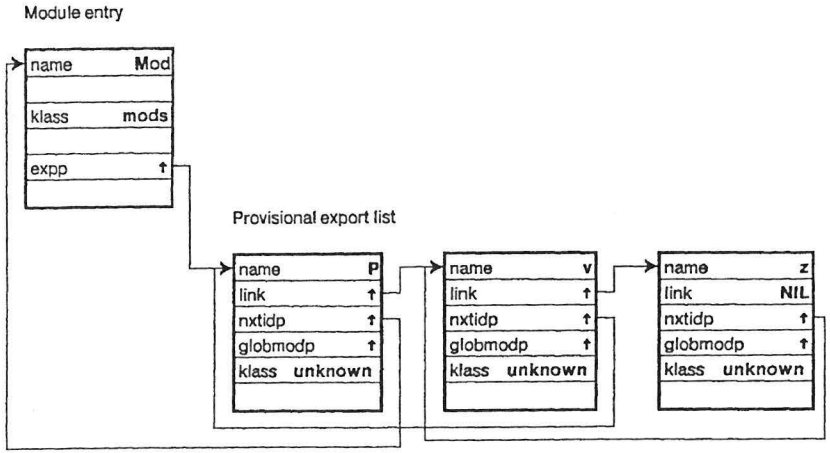
The names of the imported objects must not be ambiguous. This is checked when a new object is entered into the import list. The import rules of Modula-2 say that with a module its unqualified exported objects are also imported, and that with an enumeration type its enumeration constants are imported too. The unambiguity of all these names is checked as well.

Pass2 enters only those objects into the import list which previously have been declared and therefore may be used for declarations (scope rule 2). The remaining object names are directly passed to Pass3, which updates the import list (scope rule 1) and checks that all imported objects have been declared.

The generated import list is assigned to the field *impp* of the module entry. When the scope of the module is entered into the scope display, then the import list is not

Provisional version of the export list
after processing the export specification
by Pass2.

```
MODULE Mod;
    EXPORT p, v, z;
    TYPE t = (t1, t2, t3);
    VAR v : t;
    ....
END Mod;
```

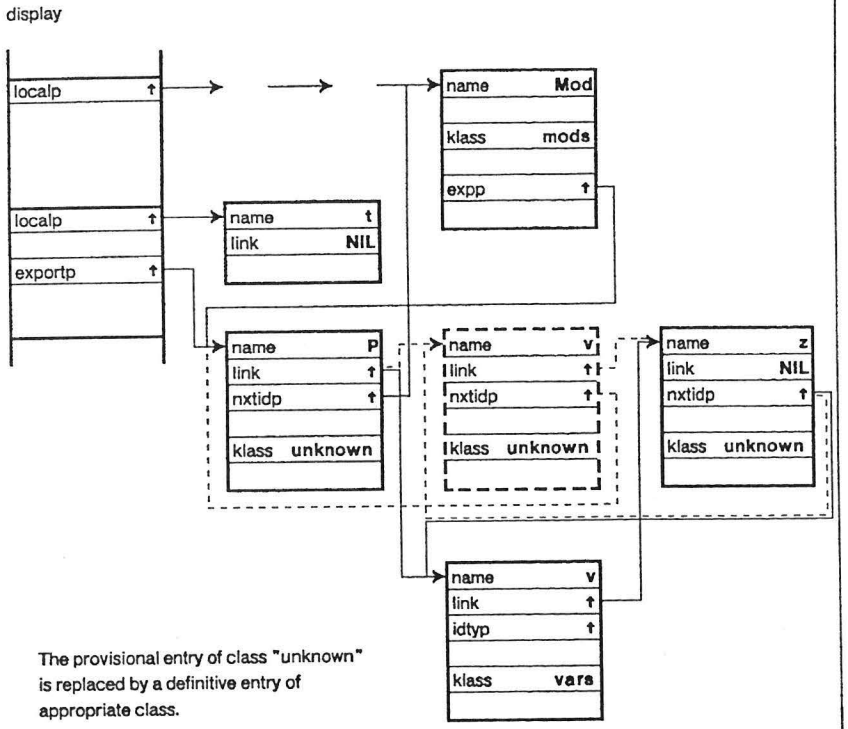


Field "nxtidp" is overlaid on field "idtyp"

Figure 7.2 Export List

Situation in display and export list
after processing the first declaration
of an exported object.

```
MODULE Mod;
    EXPORT p, v, z;
    TYPE t = (t1, t2, t3);
    VAR v : t;
    ....
END Mod;
```



The provisional entry of class "unknown"
is replaced by a definitive entry of
appropriate class.

Figure 7.3 Export List

directly assigned to a display entry field. Instead, the pointer to the module entry is assigned to the field *modulep*, and searching of an object in the import list is started via this detour. The module reference in the scope display entry is also needed for further purposes.

For an example see Figure 7.4.

7.6 Extend List

The *extend list* is a chain of pointers to name entry lists. It represents those objects in a scope, which are declared or specified here, but not accessible via local list, export list, or import list. These objects are considered to have an *extended validity range* (scope).

Two kinds of name entry lists are referenced by the extend list: Enumeration constant lists of enumeration types and export lists of modules which are declared within the scope. At the insertion of a new name list, it is always checked that the new names do not conflict with the names already known in the extend list.

The import/export rules of Modula-2 say that with an enumeration type, the corresponding enumeration constants are also imported/exported. For this and for other implementation reasons, these constants are not directly inserted into the local list. Instead, a separate name entry list is generated and appended to the structure entry of the enumeration type. The extend list makes the enumeration constant lists accessible within the scope.

Scope rule 4 says that objects exported unqualified from a module are also known in the scope around the module. After compilation of a module, its entry in the scope display is released. But access to its export list is still necessary. For this purpose the export list is entered into the extend list of the next lower scope display entry. Here again, it is necessary to enter separately the enumeration constant lists and the export lists which are implicitly contained in an export list.

For an example see Figure 7.5.

7.7 Forward List

The *forward list* contains the type identifiers referenced in pointer declarations, which are not already known in the current scope. This list is used in Pass2 only.

Scope rule 2 allows a pointer-referenced type to be declared after the pointer declaration in the same scope. To respect the scope rules in this case, Pass2 first searches the type identifier in the current scope. If it is not known yet, the pointer structure entry is left incomplete. Instead, the spelling index and the pointer to the structure entry are inserted into the forward list. At the end of processing the corresponding procedure or module, the forward list is scanned. All missing types are searched (now through all possible scope levels) and the corresponding pointer structure entries are completed.

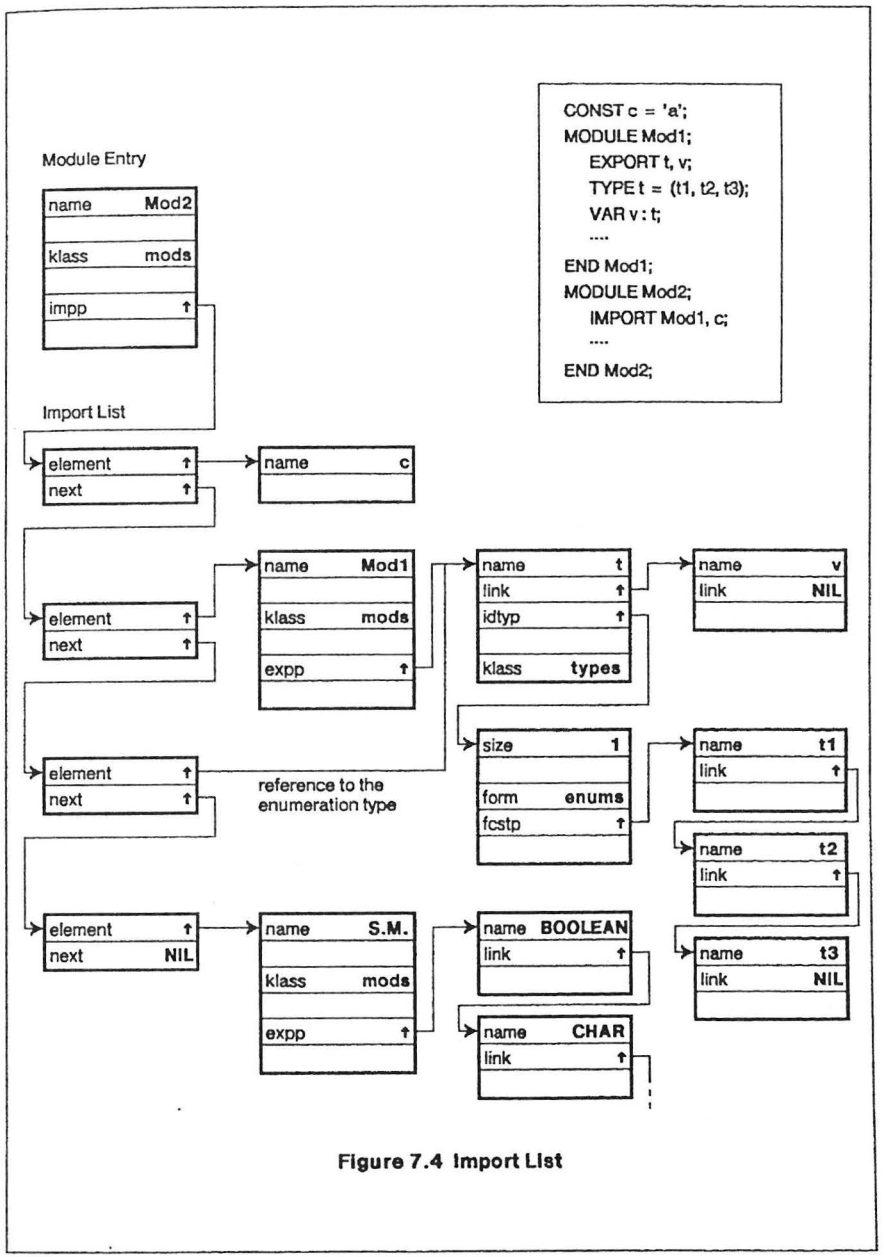


Figure 7.4 Import List

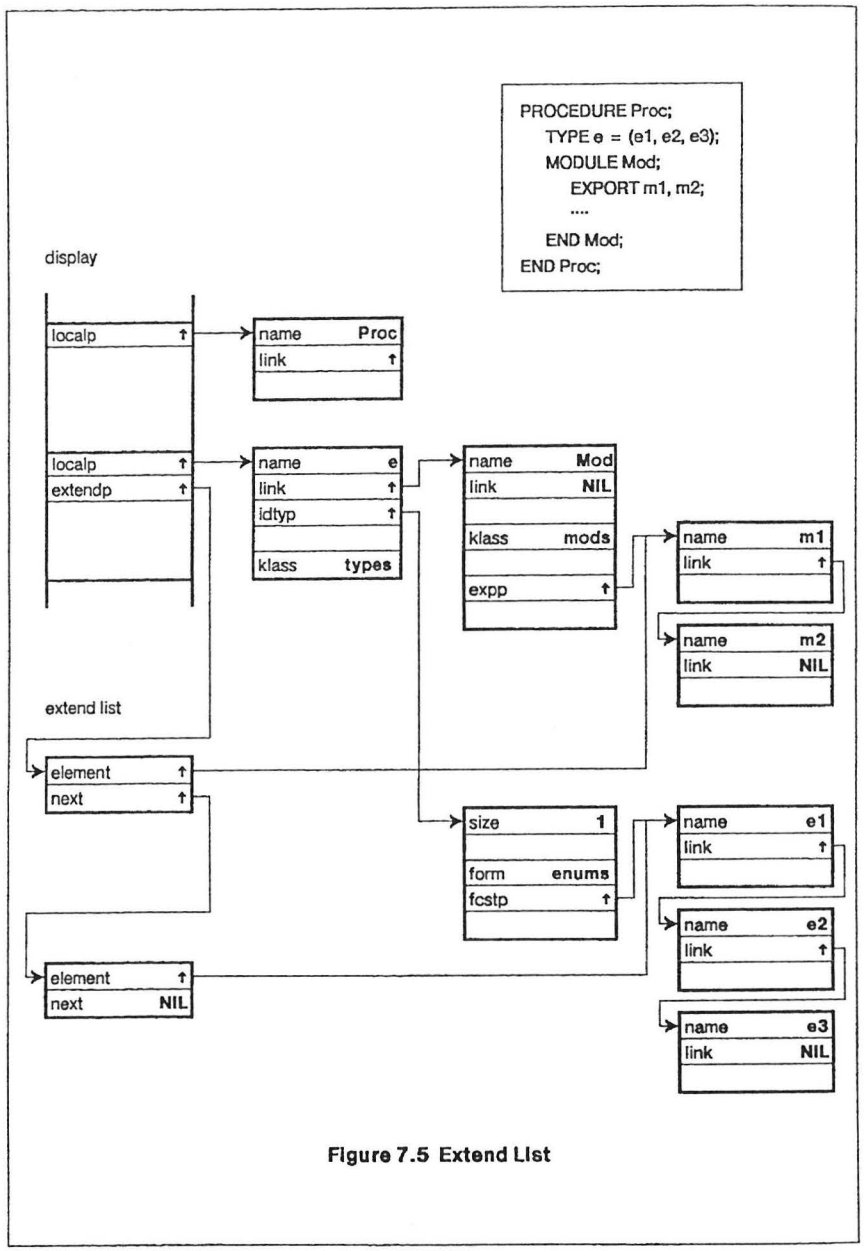


Figure 7.5 Extend List

8 Structure and Function of the Various Compiler Parts

This chapter describes the implementation of the various compiler parts. It shows their decomposition into separate modules and the concepts and functions implemented by these modules.

8.1 Interface to the Environment

One goal of the implementation of the Modula-2 compiler has been to keep the dependence on its environment as small as possible. This means that the compiler modules generally should not import objects from separate modules which do not belong to the compiler. Especially references to the operating system (e.g. input/output procedures) should not be spread over the whole compiler, because each operating system has its own interface and behavior, and an adaptation to another operating system could cause many modifications of the compiler source.

Therefore, two modules which constitute an *Independent Interface* between operating system and compiler have been developed. These are the module *CompFile* for the handling of compiler files, and the module *WriteStrings* for writing on the terminal. A third module, *Conversions*, has been developed for the conversion of integer values to character strings.

Only one module which might be considered as part of the operating system is referenced all over the various compiler parts. It is the module *Storage* which administrates the heap of the memory. It exports the procedures *ALLOCATE* and *DEALLOCATE* which are needed for substitution of the standard procedures *NEW* and *DISPOSE*.

Generally, only these modules outside the compiler are imported by the compiler modules. Exceptions are three special compiler modules (MCPublic, MCInit, and MCLookup) which also import other modules from outside the compiler. The advantage of this interface restriction became evident when the compiler was transported from the PDP-11 to Lilith. The transfer was easily accomplished within a few days. Only the interface modules and the special compiler modules were to be adapted to the new environment.

The compiler base cannot be completely independent of the environment because of the calling mechanism that is prescribed by the loader. Another dependency is caused by the file names which may differ among various systems.

8.2 Compiler Base Modula

The compiler part Modula is the base part of the compiler. Its main purpose is to control the execution of the other compiler parts and to establish a global base for type and data definitions which are valid for all compiler parts. The base part consists of the separate modules:

MCBase	Main module and private part, with data descriptions and variables used by all compiler parts.
MCPublic	Public part, with execution control.

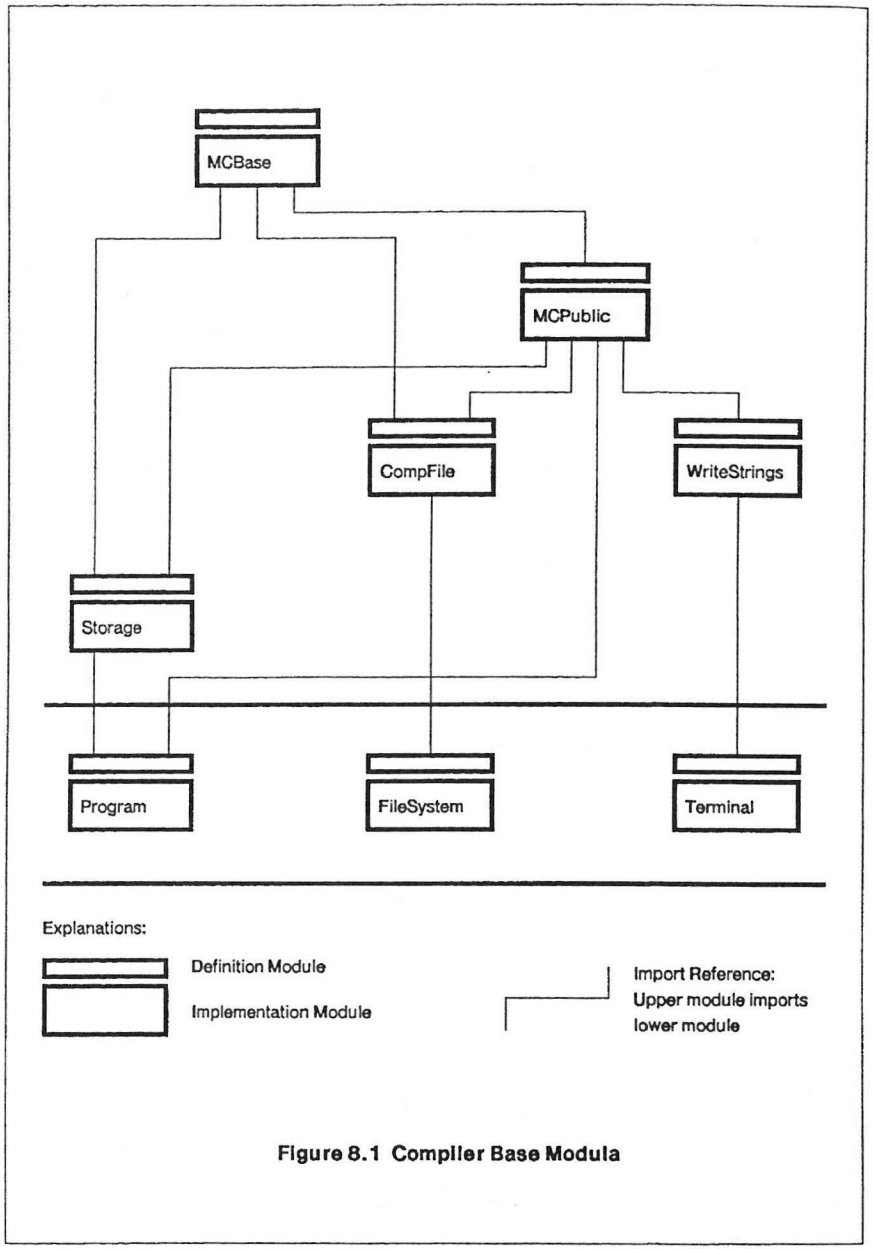


Figure 8.1 Compiler Base Modula

The import-references are illustrated in Figure 8.1.

The reason for splitting the compiler base into two separate modules was to separate those parts which are independent of the underlying operating system from those which handle the system-dependent activities. The module MCBase is mainly used for defining the global constants, types, and variables which describe the symbol table of the compiler. Its implementation module is empty. Module MCPublic has to call the other compiler parts according to the kind of the compiled unit and the current state of the compilation process (e.g. call of the Lister after Pass3 if an error has been detected). For this purpose, module MCPublic exports a *state* variable to which the called compiler parts may assign values. The type of the variable is a set of possible status values such as the kind of the compiled unit, whether or not errors have been detected in a pass, the validity of program options, and so on.

8.3 Initialization

The initialization part is a pre-pass. It handles the dialog with the user in order to open the source text file and to accept the program options (parameters for the compilation). It further creates the output files, determines a new module key (time stamp), and initializes the execution state. This part consists of the modules:

MCInit	Main module, with control of the interactive dialog.
MCLookup	Lookup on input files.

The import-references are illustrated in Figure 8.2.

The separation of the initialization part from the compiler base has been forced by the need to save space in the memory of the computer (crucial on the PDP-11 with 28 kword memory space). It is also reasonable that the code for the compiler initialization should not remain in memory during the whole compilation process.

8.4 Pass1

Pass1 provides the lexical and syntactic analysis of the source text. It is split into the separate modules:

MCP1Main	Main module, with the syntax parser.
MCP1IO	Input/output handler and source text scanner.
MCP1Ident	Module to enter the standard objects into the symbol table.
MCP1Reals	Module for the compilation of constant real numbers.
MCLookup	Lookup on input files.
MCSymDefs	Symbol definitions for symbol files.

The import-references are illustrated in Figure 8.3.

8.4.1 Main Functions

The first activity of Pass1 is the initialization of the identifier table and the symbol

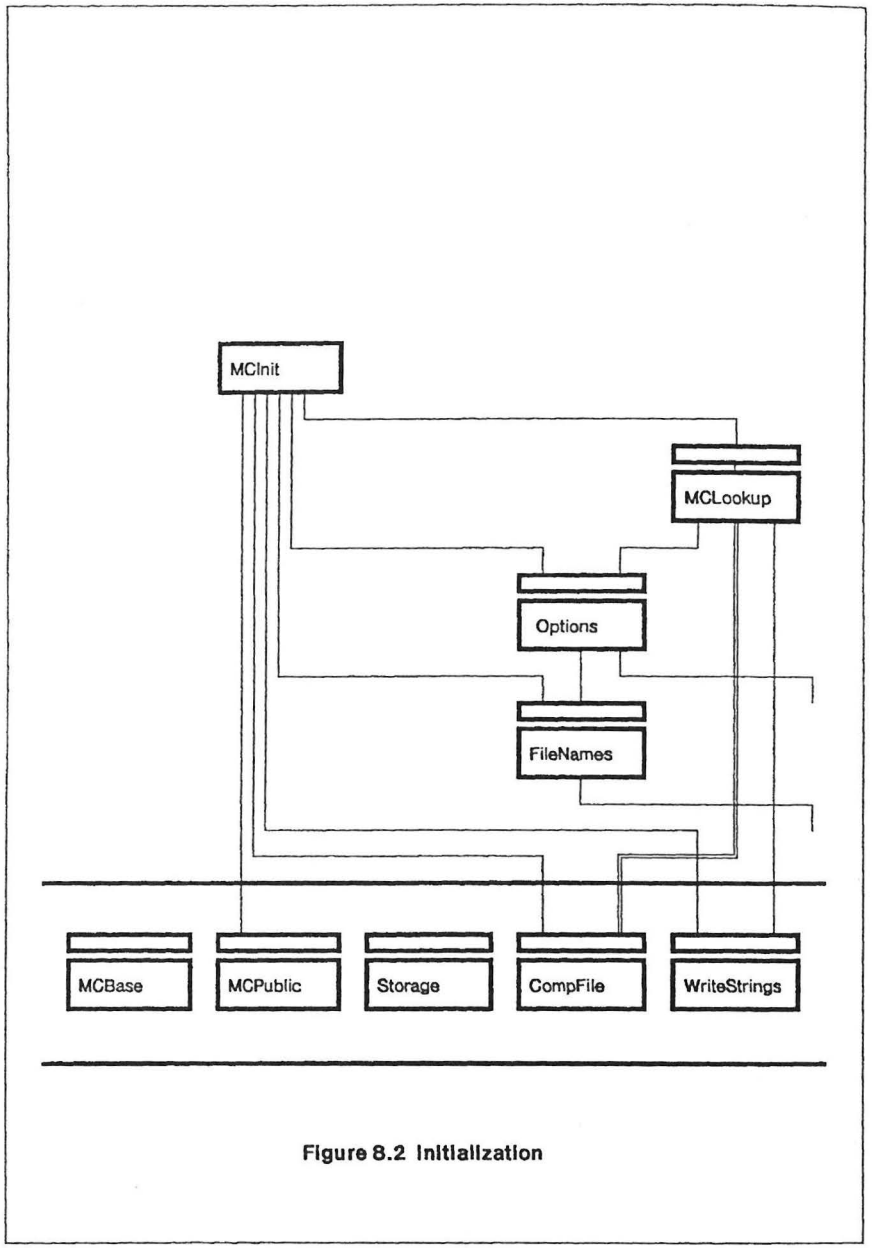


Figure 8.2 Initialization

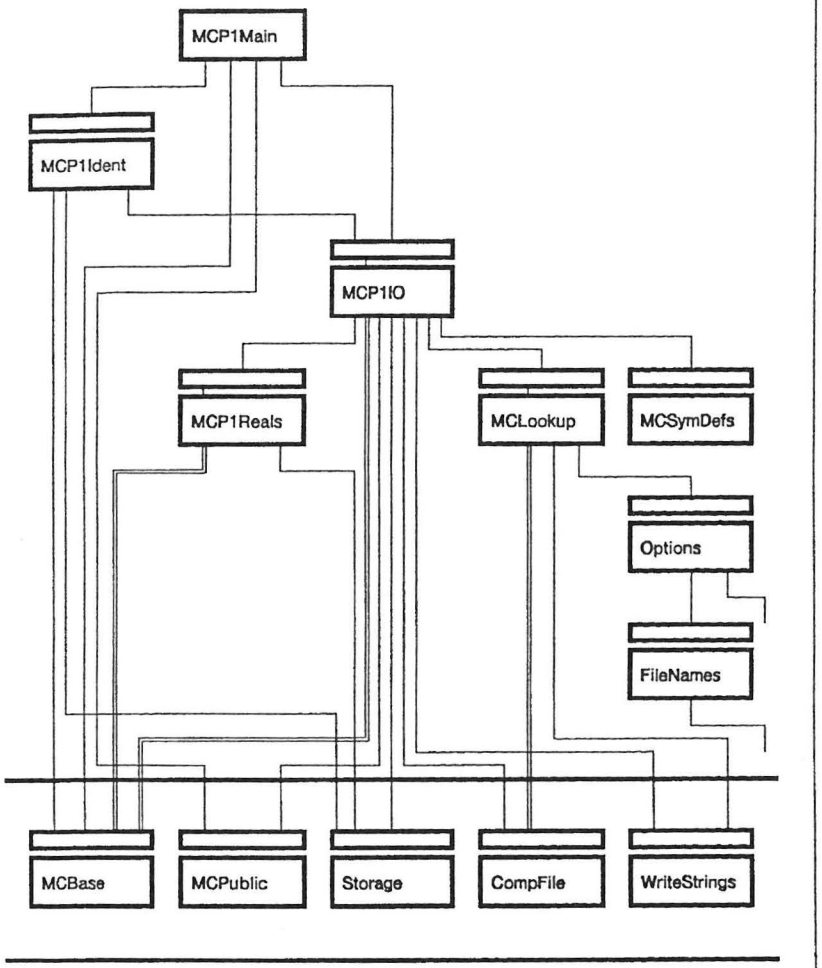


Figure 8.3 Pass1

table. Procedures of module MCP1Ident enter the key-words of the Modula-2 syntax into the identifier table (key-words are considered as special identifiers) and enter the *standard declared objects* of Modula-2 and the objects of module *SYSTEM* into the symbol table.

Afterwards, Pass1 starts to process the compilation unit. The *scanner* in module MCP1IO reads the source text and transforms it into a symbolic representation based on the global compiler type Symbol (see section 6.2). The scanner also evaluates the constant numbers (see below) and enters the identifiers into the identifier table and determines their spelling index.

The *parser* in module MCP1Main checks the generated symbol sequence for correctness of syntax and, in case of mistakes, generates error messages and synchronizes the symbol sequence with the requested syntax. The symbol sequence proceeded to the subsequent passes must merely respect the syntax defined for the inter-pass languages (see Appendix 4). Its correctness is assumed by these passes, in order to not waste time for syntax checking. Redundant symbols (e.g. semicolons) are eliminated and the remaining symbols are written on the inter-pass file IL1. In some special cases the sequence of the symbols is changed in order to allow the subsequent passes unambiguous parsing, independent of any contextual information. An ambiguity occurs, for example, for the statements *assignment* and *procedure-call* which both start with a *designator*. Pass1 reverses the symbol sequence of an assignment by putting the assignment symbol in front of the designator. For procedure calls, it inserts a call symbol in front of a procedure designator.

8.4.2 Reading Symbol Files

When an implementation module is compiled, and when separate modules are imported by a compilation unit, Pass1 searches for the corresponding symbol files. The lookup on these files is handled in module MCLookup. The name of a symbol file is derived from the module's name according to a default strategy. It is also possible to explicitly specify the file name in an interactive dialog.

The symbols on the symbol files which are encoded according to a type declared in module MCSymDefs are transformed by Pass1 into compiler internal symbols. It also replaces the identifiers by new assigned spelling indices. Afterwards, it writes the information at the beginning of the inter-pass file. The inter-pass file therefore consists of a sequence of *symbol modules* which are supplied on the symbol files (see section 4.2), followed by the symbolic representation of the actually compiled unit. See also Figure 8.4.

8.4.3 Constant Values

Numbers, characters, and character strings are recognized by the scanner, which checks them for correct syntax and computes their values.

Integer numbers are simulatneously computed for all possible number bases (octal, decimal, hexadecimal) as long as the "digits" are in the allowed range and the computed number would not exceed the value range.

```
IMPLEMENTATION MODULE Scopes;
   IMPORT BaseTypes, Storage;
   ...
   ...
END Scopes.
```

Required Symbol Files: Read by Pass1

Scopes

Symbol Module **BaseTypes** (subset)	Symbol Module **Scopes**

BaseTypes

Symbol Module **BaseTypes**

Storage

Symbol Module **Storage**

Module Sequence: On inter-pass file from Pass1 to Pass2

Sym. Module **BaseTypes**(ss)	Sym. Module **Scopes**	Sym. Module **BaseTypes**	Sym. Module **Storage**	Impl. Module **Scopes**

Figure 8.4 Reading Symbol Files

Real numbers are computed by procedures exported by the separate module MCP1Reals to which the scanner passes the characters. The separation of this module is very useful. It concentrates all conversion operations in one place, and the internal representation of the real numbers can easily be adapted to the required hardware format. In the view of portability it is essential that this module does not use constants of type REAL for its own implementation.

Apart from evaluating a constant specified in the source text, the scanner also must assign a data type to it. This is easily possible for characters, strings, and real numbers. A problem arises in the case of integer numbers, because of the overlapping value ranges of the types INTEGER and CARDINAL. The compiler distinguishes three intervals:

ints : minint .. -1
intcards : 0 .. maxint
cards : maxint + 1 .. maxcard

A constant integer number within the intcard range is compatible with objects of either type INTEGER or CARDINAL.

The compiler's internal representation of constant values respects the fact that constants must be written on the inter-pass file. This is simple for values represented in one memory word (e.g. integers), but the dynamic length of string values is very inconvenient. The solution is that string values are stored in the heap and only a pointer to this entry is passed on the inter-pass file. The same solution has been chosen for real numbers in view of a possible implementation of extended-precision real arithmetic. The following data structure is used by the compiler:

```
Constval =
  RECORD
    CASE Structform OF
      arrays : svalue : Stringpointer
    | reals : rvalue : Realpointer
    ELSE value : CARDINAL
    END
  END
```

this must be dealt with for POINTERS that are larger than 16-bit CARDINALS.

This VARIANT is the SAME SIZE FOR 16-bit pointers & is NOT the same for 32-bit pointers.

8.5 Pass2

Pass2 processes and analyzes all declaration parts of the compiled unit. It is split into the separate modules:

MCP2Main	Main module, with the parser of the declarations.
MCP2IO	Input/output handler and scanner of the inter-pass file.
MCP2Ident	Identifier and scope handler.
MCP2Reference	Reference file generator.
MCOperations	Module for the evaluation of constant expressions.

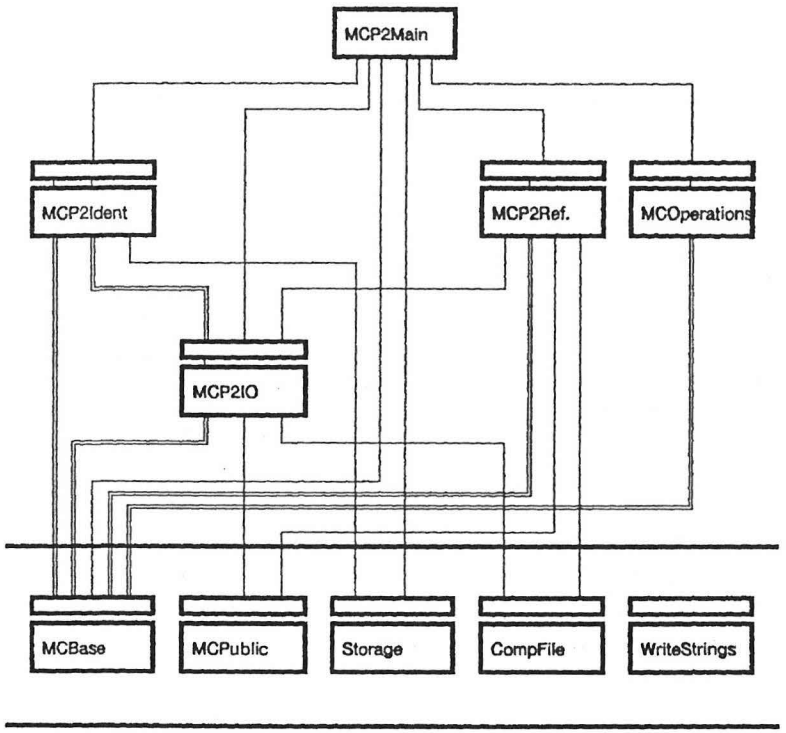


Figure 8.5 Pass2

The import-references are illustrated in Figure 8.5.

8.5.1 Main Functions

The inter-pass file IL1 is read by the scanner (in MCP2IO), which passes the symbols to module MCP2Main. This main module is a parser for the declaration parts and it transforms the declarations into a structured representation in the symbol table. The statement parts are skipped and copied on the second inter-pass file IL2. The correct use of the scope rules is verified by procedures exported from the separate module MCP2Ident.

A duty of Pass2 is the organization of the initialization of local modules. According to the language definition of Modula-2, the module bodies (statement parts) must be executed as soon as the procedure to which the module is local becomes active. This implies an implicit call of the module body. Pass2 inserts this call at the beginning of the surrounding statement part.

The following list gives an overview of the various activities of Pass2. More details about some of these activities are given afterwards:

- Entering the declared objects into the symbol table.
- Testing of scope rules (see Chapter 7).
- Support of initialization of the module bodies.
- Evaluation of offset addresses for variables and record fields (see 8.5.2).
- Assignment of procedure numbers.
- Generation of the reference file (see 8.5.3).
- Evaluation of constant expressions (see 8.6.3).
- Check for complete declaration of exported objects and forward referenced types.
- Check for redeclaration of procedures and opaque exported types in implementation modules.

8.5.2 Size Evaluation and Address Assignment

An important duty of Pass2 is the evaluation of data sizes and the assignment of offset addresses to variables and record fields.

On Lilith, the smallest accessible data unit is one memory word (16 bit). Therefore, the elements of most types are represented in *one memory word*. These types are:

- Standard types: INTEGER, CARDINAL, BOOLEAN, CHAR
- Enumerations
- Subranges
- Sets
- Pointers
- Procedure types
- System types: WORD, ADDRESS, PROCESS

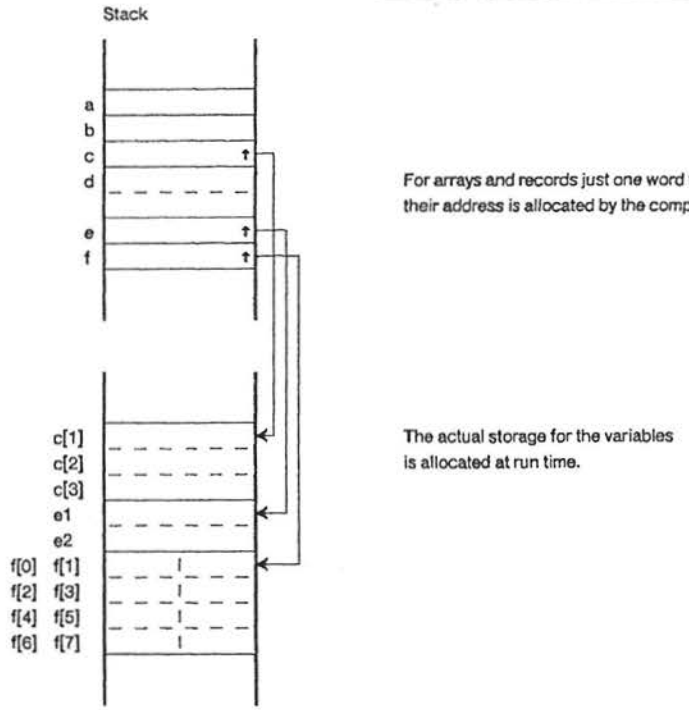
For elements of type REAL, *two memory words* are needed.

For record types, the needed space is the sum of the space needed for the fields. The fields in a record are allocated consecutively according to the order of

watch the alignment when we go between files & access these

not this goes to the target addressing.

```
VAR  a : CARDINAL;
     b : BOOLEAN;
     c : ARRAY [1 .. 3] OF CARDINAL;
     d : REAL;
     e : RECORD
           e1 : BOOLEAN;
           e2 : (red, green, blue);
         END;
     f : ARRAY [0 .. 7] OF CHAR;
```



For arrays and records just one word for their address is allocated by the compiler.

The actual storage for the variables is allocated at run time.

Figure 8.6 Data Allocation

declaration and get an offset relative to the beginning of the record. Fields declared in variant parts are overlaid.

For array types (except for character arrays) the needed space is the product of the number of elements and the element size. In character arrays (strings) two characters are packed into one word.

For addressing variables, the M-code instructions provide addressing relative to a base address. For global variables this is the address of the data frame of the separate module, for local variables it is the address of the procedure mark stack entry of the corresponding procedure. A characteristic of M-code is that the instructions for loading and storing information allow an offset in the range from 0 to 255 (by one byte). The consequence of this fact is that large elements should not lie within the offset range and therefore reduce the number of variables possible. Therefore, arrays and records are generally addressed indirectly. Their actual address is held by a word allocated in the offset range. The space actually needed is allocated after all one (or two) word entries have been determined. These addresses are computed at run time. See Figure 8.6.

Parameters are treated as local variables, whereby, for variable parameters, a single word is needed for the address entry. For dynamic arrays a two-word entry is generated with the second word holding the upper index bound.

- Size evaluation and offset assignment cause Pass2 to be *machine dependent*. One reason for doing it at this point is the separate compilation facility. It is desirable that the addresses of the variables exported from a separate module be assigned during compilation of the definition module. Then all modules importing a separate variable can work with fixed values and no offset insertions are necessary at execution time. Another reason is that the compiler should not have to scan all declarations twice to complete their representation in the symbol table.

8.5.3 Reference File Generation

If the execution of a program fails, it is useful to analyze the program's actual state. This is done by a *post-mortem debugger*. For the debugger, the compiler provides some information about addresses of variables, procedure entries, and names and structures of the different objects. The needed information is compiled in Pass2. For each declaration of a module, procedure, variable, or type, a procedure of the separate module MCP2Reference is called. This procedure extracts the appropriate piece of information from the symbol table and writes it on the reference file.

Reference files are generated upon compilation of an implementation module or a module without export and describe exactly this module, i.e. all objects declared in the module. This rule is broken for type descriptions only. Types declared in other separate modules are contained in the reference file, if they are used in the declaration of an object. This is reasonable because for data structures which are accessible in a module, it should be possible to inspect their values during a debugging session.

The separation of the module MCP2Reference from the main module has been very useful. The compiler is largely independent of the debugger. Since the introduction

of the reference file, the debugger has been developed to provide more detailed information about the program's actual state. This caused at least twice an improvement of the reference file format. It has been possible to make all the necessary changes in module MCP2Reference, without any adaptation of the interface or replacement of procedure calls in other compiler modules. The syntax of the information on the reference file is listed in Appendix 5.

8.5.4 Integration of Symbol Modules

The symbol modules which represent the imported separate modules are processed by Pass2 and transformed into symbol table entries. Full type information about the imported objects is therefore available in the symbol table, and the compiler applies the same type and parameter tests to them as to objects declared in the compiled unit itself. The advantage of choosing a format on the symbol files similar to the inter-pass language is that no special procedures are needed to process the symbol modules. Slight differences, such as already assigned variable offset addresses or procedure numbers, cause a few extensions to the parsing procedures for the compiled unit in module MCP2Main.

As several symbol files are merged by Pass1, it is possible that parts of the same separate module are represented by more than one symbol module on the inter-pass file. Matching module keys guarantee that the symbol modules are derived from the same compilation of the corresponding definition module. Each of these symbol modules possibly specifies another subset of the interface. Therefore some objects could be declared more than once. To tackle this situation, Pass2 tries to complement the existing representation of the separate module, and to skip the objects already entered into the symbol table.

8.5.5 Matching Definition and Implementation Module

A definition module and its corresponding implementation module represent complementary parts of the same separate module. Objects declared in the definition module are implicitly available in the implementation module. For *procedures* and *opaque exported types*, the implementation module even has to provide a new, complete declaration. Upon compilation of the implementation module, Pass2 checks whether or not the two modules are matching.

The symbol module representing the definition module precedes the implementation module on the inter-pass file between Pass1 and Pass2. This symbol module is therefore already entered into the symbol table, when the implementation module is processed. Before processing the latter, Pass2 scans the export list and the local list of the current representation of the module in the symbol table and determines the already assigned variable addresses and procedure numbers. The entries of the procedures and the opaque exported types, which must be redeclared in the implementation module, are removed from the export or local list and entered into a separate list, the so-called *old list*.

This old list is scanned, when a procedure or a type is declared in the implementation module and a redeclaration of such an object is *possible* (see

below). If an object is found in the old list, it is checked for equivalence with the new declared object. Afterwards, the old entry is removed from the old list and the symbol table. An empty old list at the end of the compilation indicates successful redeclaration of *all* procedures and opaque types of the definition module.

Redeclaration of an object of the definition module is *possible*, if the new declared object is *globally known*, i.e. known at the global level of the implementation module. This means that the object either must be declared directly at the global level, or declared within a local, possibly nested, module and exported unqualified to the global level. Procedure *GlobalKnown* in module MCP2Ident provides this test. Starting from the current scope level down to the global scope level, it checks all export lists for the object's name. According to the export rules, that an object may be exported implicitly with a module name, it also checks for the corresponding module names. Redeclaration is possible, if one of these names is known at the global level.

Example 8.1

In the following module the name *Nested* is known at the global level. For a complete test, the names *Nested* and *Mod2* must be checked.

```
IMPLEMENTATION MODULE Nest;

  MODULE Mod1;
    EXPORT Mod2;

    MODULE Mod2;
      EXPORT Nested;

      PROCEDURE Nested(level: CARDINAL);
      BEGIN .....
      END Nested;

    END Mod2;

  END Mod1;

END Nest.
```

8.6 Pass3

Pass3 processes and analyzes the statement parts of the compiled unit. It is split into the separate modules:

MCP3Main	Main module, with the parser of the statements.
MCP3IO	Input/output handler and scanner of the inter-pass file.
MCP3Ident	Identifier and scope handler.
MCOperations	Module for the evaluation of constant expressions.

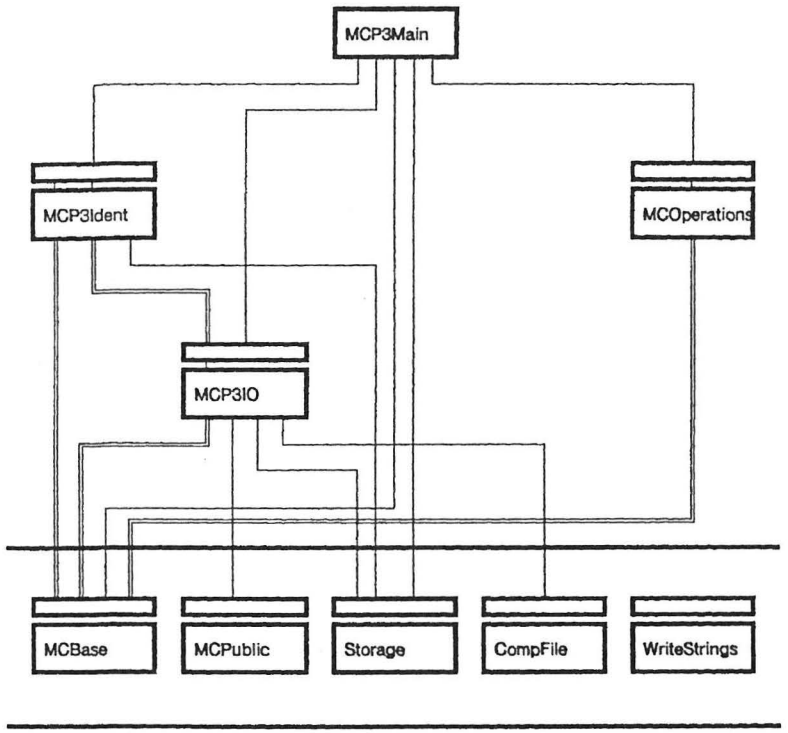


Figure 8.7 Pass3

The import-references are illustrated in Figure 8.7.

8.6.1 Main Functions

The information on the inter-pass file IL2 is read by the scanner in module MCP3IO. The main module MCP3Main parses the statement parts. It checks on the correct use of the statements and the type compatibility of objects in expressions, assignments, and procedure calls. For all identifiers (spelling indices), the compiler tries to find an entry of an appropriate object in the symbol table, and replaces the index by the pointer to this entry.

The following list gives an overview of the various activities of Pass3. More details about some of these activities are given afterwards:

- Test of type compatibility (see 8.6.2).
- Evaluation of constants in expressions (see 8.6.3).
- Allocation of data space needed in WITH statements.
- Substitution of the standard procedures NEW and DISPOSE by user defined procedures.
- Linking of string constants.
- Completion of import list.

8.6.2 Type Compatibility Tests

Modula-2 defines type compatibility rigorously. Generally, objects are (type) *compatible*, if they refer to the *same* declaration. For subranges the base types must be compatible. Objects with merely an *equivalent* data structure are considered incompatible.

Example 8.2

In the following declaration the variables *a* and *b* are incompatible:

```
VAR  a : ARRAY [1 .. 10] OF CARDINAL;
VAR  b : ARRAY [1 .. 10] OF CARDINAL;
```

This rule allows a simple and fast implementation of type compatibility testing in Pass3. Objects are recognized to be compatible if they (or their base types) refer to the same structure entry. The first, general test therefore compares the pointers to the structure entries.

Unfortunately, compatibility rules for constants and procedures are more complex and in some cases it is even necessary to check on *structural equivalence*.

Compatibility rules for constants and procedures:

- *"intcard" numbers* are compatible with objects of either type INTEGER or CARDINAL.
- *NIL* is compatible with all pointer types.

- *String constants* are compatible with character arrays which are at least as long as the string constant.
- *Procedures* are compatible with procedure variables with the same structure, i.e. the same parameter sequence, including the parameter kind (value or variable parameter), and, for function procedures, with the same result type.

The above rules determine the basic set of compatibility tests which are always performed. In the compiler this set is known as *expression compatibility*. Relaxations of the compatibility rules are allowed in some places.

Relaxations of the compatibility rules:

- In *assignments*, objects of the types CARDINAL and INTEGER are said to be compatible. The assigned value must lie within the overlapping range of both types: 0..maxint; for constants this is checked by Pass4, for variables a run-time check is generated.
- In *procedure calls*, a parameter of type WORD may be substituted by any object matching the size of one word.
- In *procedure calls*, a parameter of type ARRAY OF WORD may be substituted by an object of any type.
- *Objects of type ADDRESS* are compatible with "cardinals" and with pointers.

Special compatibility tests according to these relaxations are performed by the compiler in the appropriate situations.

8.6.3 Evaluation of Constants in Expressions

With the introduction of constant-expressions in Modula-2, it was necessary to implement expression evaluation at compilation time. Constant expressions are processed by Pass2 (declarations) and by Pass3 (set constructors and case labels). In Pass3 also normal expressions are evaluated, if possible.

Direct evaluation could be dangerous for the compiler because of possible arithmetic errors (overflow). To prevent this situation the compiler must be able to *control the execution of all (arithmetic) operations*. This is provided by the separate module MCOperations.

The implementation part of module MCOperations will depend on the computer's facilities. On Lilith, where overflow ends with a trap, addition and subtraction are checked against overflow before execution of the operation. For multiplications, a preceding overflow test is impossible, or at least too expensive, and therefore this operation is simulated by an additive algorithm.

Beside evaluation of constant-expressions, it is also interesting to perform an evaluation of normal expressions as far as it is possible. It is an inexpensive feature which shortens the generated code, and, sometimes, also helps to determine the actual type of the expression. It even saves code in Pass3, because it is possible to parse expressions and constant-expressions with the same procedures.

The Modula-2 language definition specifies that an expression is evaluated from left to right on each precedence level. This rule must be respected for evaluations at compilation time as well. It follows that constants in expressions may be evaluated by the compiler as long as they are not "preceded" by a variable. Evaluation after a variable on the same level would disobey the rule.

Example 8.3

In this example the stepwise pre-evaluation of an expression is shown:

```
      4 * (16 - 13) + 2 * 6 * (3 - a) * 3 * (7 DIV 2)

-->   4 * 3         + 2 * 6 * (3 - a) * 3 * (7 DIV 2)
-->   12            + 2 * 6 * (3 - a) * 3 * (7 DIV 2)
-->   12            + 12    * (3 - a) * 3 * (7 DIV 2)
-->   12            + 12    * (3 - a) * 3 * 3
```

The information written on the inter-pass file IL1 depends on whether or not an evaluation is possible. If it is not possible, the operands and the operator must be passed in the original sequence to Pass4. In the other case (both operands are constants) the result of the operation should be passed only, i.e. operands and operator are replaced by the result.

While parsing an expression, the decision whether or not an operation can be evaluated is only possible, after the second operand is known. This means that all components of the expression must be saved for a certain time. For this purpose Pass3 provides a backtracking feature. Normal output is generated, but at the beginning of an expression, simple expression, term, or factor a *save position* referring to the current output item is retained. After a successful evaluation of an operation, the output is reset to the corresponding save position and the new result replaces the previous output.

One could imagine different implementations of this saving mechanism. A first approach could be to stack the whole output information until an expression is parsed completely and afterwards to write it on the inter-pass file. Another possibility is to write the output directly on the inter-pass file and, in case of replacement, to reposition the file.

In the compiler the second possibility has been implemented. One reason for this decision was the small memory space on the PDP-11 computer. Another reason is that repositioning on the file is necessary for about 2 to 4 percent of all save positions only. Therefore, file positioning causes negligible overhead.

Additional difficulties arise due to the replacement of the constants. The interspersed symbols like line numbers, errors, and compiler options must not be lost. The save mechanism has to remember the interspersed information in the replaced part and to rewrite it on the inter-pass file.

8.7 Pass4

Pass4 provides M-code generation for the Lilith computer. It is split into the separate modules:

MCP4Main	Main module with the parser of the statements and expressions.
MCP4Global	Input and lister output handler and scanner of the inter-pass file.
MCP4CodeSys	Code output handler and generator of simple code patterns.
MCP4AttributSys	Attribute handling and code generation for load and store operations.
MCP4CallSys	Code generator for procedure calls.
MCP4ExprSys	Code generator for variables and expressions.
MCMnemonics	Definition of the instruction mnemonics.

The import-references are illustrated in Figure 8.8.

Pass4 reads the inter-pass file IL1 and translates the compiled unit into M-code. The generated code is written on the object file in a format that is accepted by the loader. Information needed by the Lister is written on the inter-pass file IL2.

The description of Pass4 and the M-code generation is the topic of a separate paper [Jac83].

8.8 Symfile

The compiler part Symfile is called after Pass2, when a definition module is compiled. It selects in the symbol table all objects (name entries; see section 6.3) which are needed for a symbolic description of the definition module and writes the symbol file. It consists of the modules:

MCSymFile	Main module, which generates the symbol file.
MCSymDefs	Symbol definitions for symbol files.

The import-references are illustrated in Figure 8.9.

Module MCSymFile first generates a *name list* for each separate module described in the symbol table. These lists contain pointers to the selected name entries and are ordered according to the sequence, in which the objects of the module should be described on the symbol file.

For the generation of the name lists, the export list (*expp*) and the local list (*locp*) of the symbol table entry of the compiled definition module are scanned. For all name entries in these lists, a recursive procedure is called which selects all needed name entries and appends references on them to the name lists. This procedure works according to the rule that an object referenced in a declaration must be declared first and therefore its reference must be appended first to the name lists. Before a reference to a selected object is appended to the name lists, this procedure inspects the corresponding structure entry and selects all types used in this structure. The

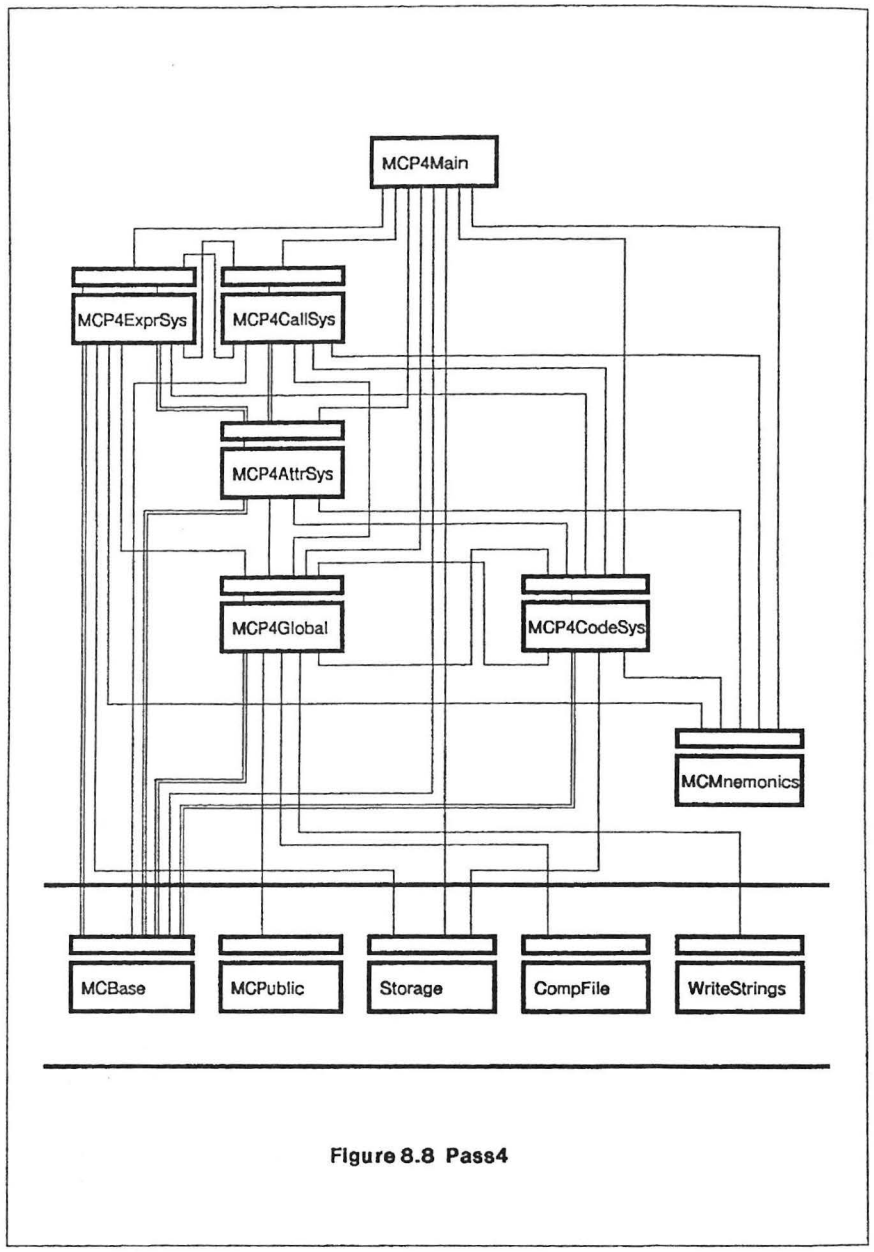


Figure 8.8 Pass4

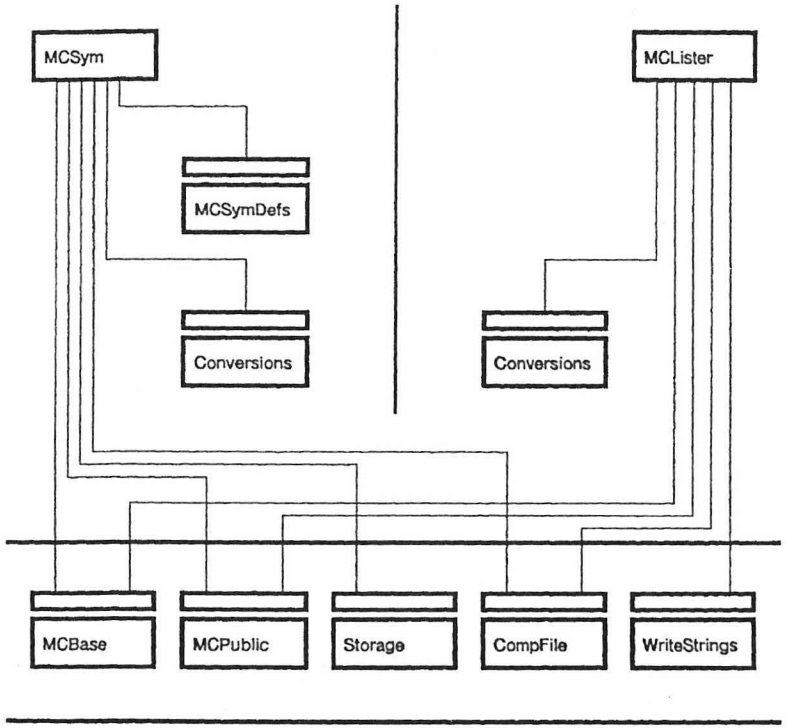


Figure 8.9 Symfile and Lister

separate module to which a selected object belongs is indicated by the name entry field *globmodp*.

When a structure entry is inspected, it is marked in the field *inlist*. This prevents that a structure is inspected more than once. The field *stidp* refers to the name entry of the structure's name. This reference is also appended to the name lists. If the value NIL is assigned to *stidp*, no explicit name for this structure exists, i.e. it has been directly declared in a variable declaration or within another type declaration.

In declarations of variables and record fields, Modula-2 allows the specification of a list of objects to which the same type is assigned. A reconfiguration of such an object list from the information stored in the symbol table would be difficult, because name entry lists are ordered according to the spelling indices and not according to the declaration sequence. Instead, each variable and record field is declared separately on the symbol file. This implies problems with type declarations without explicit type name, because of specifying the same type structure in more than one declaration would be a contradiction to the type compatibility rules (equivalent type structures are incompatible). Therefore, the symbol file generator assigns a *dummy name* to such type structures. For other type structures which are elements of larger type structures (e.g. index range of an array), no explicit type name is required.

Example 8.4

Consider the following definition module:

```
DEFINITION MODULE Example;
  EXPORT QUALIFIED r1, a1;
  TYPE
    Rec = RECORD f1: REAL; f2, f3: [1..100] END;
  VAR
    r1, r2 : Rec;
    a1, a2 : ARRAY [0..33] OF CHAR;
END Example.
```

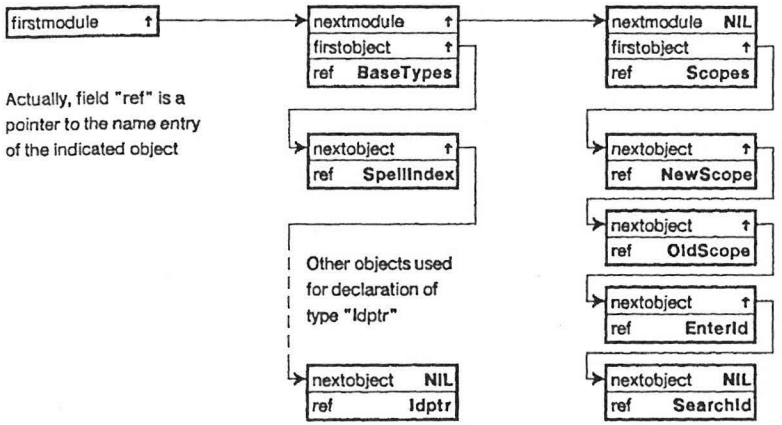
Dummy names are assigned to the subrange [1..100] and to the array ARRAY [0..33] OF CHAR. The generated name list references the objects in the following sequence:

dummy-1 type: [1..100]
Rec
r1
dummy-2 type: ARRAY [0..33] OF CHAR
a1
r2
a2

After the selection is completed, the name lists are scanned, and the selected objects are transformed into a sequence of symbols which are written on the symbol file. The symbols are encoded according to a type declared in module MCSymDefs. For all objects the actual identifiers are inserted instead of the spelling indices. For variables the address offsets, for procedures the procedure numbers, and for type

```
DEFINITION MODULE Scopes;
   FROM BaseTypes IMPORT SpellIndex, Idptr;
   EXPORT QUALIFIED
      NewScope, OldScope, EnterId, SearchId;
   PROCEDURE NewScope;
   PROCEDURE OldScope;
   PROCEDURE EnterId(id: Idptr);
   PROCEDURE SearchId(spix: SpellIndex; VAR id: Idptr);
END Scopes.
```

Name Lists: Selected objects to be described on the symbol file



Symbol File: A sequence of symbol modules

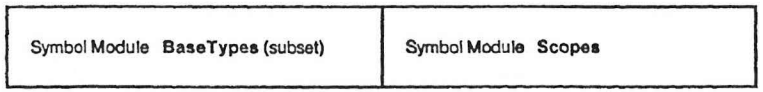


Figure 8.10 Symbol File Generation

specifications the size information are included. See also Figure 8.10.

Information concerning the symbol file is given in Chapter 4. Its syntax is described in Appendix 2.

8.9 Lister

The Lister is designed to generate a listing file of the compiled unit after either Pass2 or Pass3, when syntactical errors have been detected, or after Pass4, when the compilation has been successful. This part consists of a main module only:

MCLister Main module, generation of a program listing.

The import-references are illustrated in Figure 8.9.

After Pass2 or Pass3, the Lister reads the interspersed line numbers and error information from the inter-pass file. Pass4 generates a file in a special format containing only line numbers, code addresses and error numbers; the listing file is generated by merging the source text with the information on this file.

The sequence of line numbers on the inter-pass files is not ordered (e.g. because of the change in the symbol sequence in Pass1; see 8.4.1). Therefore, it is impossible that the Lister could directly generate the listing file. It has to reorder the line number information before the file can be written.

In the current implementation the Lister maintains a circular buffer, into which the incoming messages are inserted in correct order. After the buffer is filled, also an output procedure is started which sequentially reads the buffer and writes the listing file. This implementation is very tricky and difficult to understand. It has been chosen because the error messages may occur in any number and the length of the compiled unit is unknown.

Nevertheless, a simpler and probably faster implementation would be possible, if a maximum number of lines in the source file of a compilation unit is assumed (3000 lines seem to be an unreached limit). This is acceptable for a compiler with a separate compilation facility. With this restriction, the Lister first could read the whole line number information from the inter-pass file and, after this information has been ordered internally, write the listing file.

9 Conclusions

The first section of this chapter shows that Modula-2 is successfully used at ETH Zürich and all over the world. Large program systems have been developed which extensively use the separate compilation feature of Modula-2.

In the further sections, some aspects of the separate compilation concept of Modula-2 and its implementation on Lilith are analyzed. This is done for three subjects. The first is the *separate module* which is analyzed for its usefulness as a programming tool. A classification of typical separate modules is given. Second, the importance of *symbol files* is discussed. It turns out that symbol files in an open system environment do not only serve for information transfer of the compiler, but they also guarantee the stability and uniqueness of a system interface. Finally, the actual compiler implementation is discussed.

9.1 Use of Modula-2

Modula-2 is about to become a successful and widely used programming language. The simple, but powerful language concepts allowed a comfortable, reliable, and efficient implementation of Modula-2. About 200 copies of the Modula-2 compiler developed for the PDP-11 have been distributed all over the world, and the Modula-2 compiler for Lilith has been mailed to more than 100 destinations. In addition, several Modula-2 compilers for other computers (e.g. VAX, Motorola 68000, Intel 8086) have been derived from these compilers; at the Computer Center of ETH Zürich, the Modula-2 compiler for the PDP-11 has been translated into Pascal and developed to a package of cross-compilers, called *SMILER-2*, running on the CDC 6000 / Cyber installation and generating code for the PDP-11, Motorola 6809, and Motorola 68000.

Modula-2 is also heavily used on the 30 Lilith computers of the Institut für Informatik ETH Zürich. Modula-2, Lilith, and the comfortable programming environment, are *the tools* which are used by most members of the Institut. Several large system and application programs have been implemented on Lilith. They all extensively use the separate compilation facility. Separate compilation turned out to be *highly useful* and *mandatory* for the development of large programs and software systems. For some of these programs, the number of *separate modules* and the memory space needed for *code* and *global data* of these modules is listed in the following descriptions.

LIDAS

The program system *LIDAS* [Reb83] is a *data base management system* which interactively helps the user to design and use a data base. A data base is defined according to an extended relational data model, i.e. by entity sets (relations) and relationships between them. It is represented by a collection of *data modules* encoded in *Modula/R* [Koc82]. Modula/R is an extension of Modula-2 with concepts of the relational data base model.

One component of LIDAS is *Gambit* which interactively supports the design of a data base. It consists of several programs which are subsequently called by a common

base part and which transform the user defined relations into a Modula/R data base representation. The base part includes the common data of Gambit, the relational data system of LIDAS, and a package of screen software. The following table shows the size of the base part and the data definition program which is called first.

Gambit Data Definition	8 modules	11.90 kword
Gambit Base Part	5 modules	2.38 kword
Relational Data System	6 modules	18.48 kword
Screen, Windowhandler	6 modules	8.34 kword
Library Modules	4 modules	1.47 kword
LIDAS-Gambit	29 modules	42.59 kword

XS-1

The program system *XS-1* [Ber82] is an *interactive operating system* which supports the execution of application programs. According to dialog structures supplied by the application program, it controls the user's dialog with the program. It is a philosophy of the system to provide the user with a consistent user interface.

The main components of XS-1 are the dialog control modules, the tree-file system which maintains a tree structure on files, and a package for displaying information on the screen.

Dialog Control	8 modules	10.06 kword
Tree-File System	22 modules	11.81 kword
Screen, Library Modules	13 modules	7.69 kword
XS-1	43 modules	29.57 kword

Caliban

The program *Caliban* [Fre83] is an *information retrieval system*. It helps the user to retrieve information from a set of data which is stored on a file. For retrieving the stored information, the system uses a large number of indices which are organized in a tree structure. In one of its application, Caliban is supplied with a catalog of a library. In this catalog, information about 800 documents is stored. The documents are indexed by a tree containing 13000 nodes.

The main components of Caliban are the user interface, a package to display the tree structure, modules which supply the retrieval functions, and the tree-file system which administrates the index tree.

Caliban User Interface	25 modules	14.83 kword
Tree Display	14 modules	5.46 kword
Retrieval Functions	6 modules	3.78 kword
Tree-File System	22 modules	12.24 kword
Screen, Windowhandler	6 modules	4.24 kword
Library Modules	6 modules	1.41 kword
Caliban	79 modules	42.00 kword

Medos-2

All the programs mentioned above base on the operating system *Medos-2*. It is the resident part of the programming environment on Lilith. Medos-2 is also programmed in Modula-2 and mainly supports file access, terminal input/output, and program execution.

File Handling	3 modules	8.30 kword
Terminal Input/Output	4 modules	2.20 kword
Program Execution	2 modules	2.32 kword
Further Modules	5 modules	1.35 kword
Medos-2	14 modules	14.18 kword

9.2 Separate Modules

The concept of separate modules is *highly useful* and *mandatory* for the development of large programs and software systems. The splitting of separate modules into definition and implementation modules enables flexible and, as long as the definition modules remain unchanged, independent development of the separate program parts. Moreover, it makes it possible for the system to *detect* a change in a definition module, because the change manifests itself as recompilation. Three characteristic classes of separate modules may be distinguished: *Program components*, *library modules*, and *system interfaces*.

Program Components

For reasons of convenient development of a program, it is sometimes useful to split a program into a collection of separate modules. This decomposition helps to reduce the size of the compilation units and therefore to save compilation time. Program components are typically defined for a specific application in a program.

Library Modules

Library modules may be considered as general program components whose interface is not specifically tailored to a single program. Usually, they provide activities or services which may be required by several programs. Such services are, for example, mathematical functions, conversion and formatting routines, and procedures for dialog control. Library modules typically export a collection of procedures or functions and do not maintain data of their own.

System Interfaces

System interfaces are provided by modules which control the access to certain resources, e.g. memory management, disk organization, screen layout. Such modules, and in particular their local data, typically survive the execution of a program and are responsible for consistency of the controlled resources. It is an important fact that a resource cannot be simultaneously controlled by several modules (e.g. if more than one module would control the files on a disk, inconsistencies would occur within no time). This implies that a module with a

system interface must be specified quite generally and respect the needs of most programs. This is especially true for the interfaces of the operating system.

The quality of a separate module is highly dependent on the quality and stability of its definition module. Frequent changes and recompilations of definition modules are very inconvenient for the importing clients and should be avoided. As the number of dependencies increases with the generality of a module, it is important that library modules and especially system interfaces be well designed before they become public.

The definition of a good interface is not trivial at all and requires a great deal of experience in modularization and knowledge about the anticipated use. A good guideline might be to have a small and simple interface which hides a (possibly complex) internal implementation. This is especially proposed for the design of program components. Many programmers are tempted to express the modular structure of a program by a corresponding number of separate modules. That number thereby tends to become excessively large. There are also programmers who derive the number of separate modules from the number of source code lines. The result of such "modularization" is that there will exist modules which are imported by just one other module and whose interface is usually complicated. All these programmers forget that Modula-2 also supports *local modules* which are more appropriate in some cases. Local modules help to keep the interfaces and dependencies of separate modules small.

A reason for not using local modules might be that the syntax for local modules is slightly different from the syntax for separate modules, i.e. in local modules the specification of the exported objects is not separated from the implementation. This makes some changes of the source text necessary when a local module is converted to a separate module and vice versa. Program development could be facilitated if no changes would be necessary for conversion. Certain components could be separated temporarily, until they are in a stable stage which allows to convert them again into a local module. This could help to keep the number of separate modules in larges programs small.

Proposal

Separate modules and local modules should be identical in syntax, i.e. the concept of splitting a module into definition and implementation module should be generally introduced for modules in Modula-2. This makes a slight modification of the Modula-2 syntax is necessary, i.e. the redefinition of the syntactic elements *ModuleDeclaration* and *CompilationUnit* (see also Appendix 1):

```
ModuleDeclaration  = DEFINITION MODULE ident ";" { import } [ export ]
                       { definition } END ident |
                     [ IMPLEMENTATION ] MODULE ident [ priority ] ";"
                       { import } block ident .
CompilationUnit    = ModuleDeclaration "." .
```

This generalization of definition and implementation modules, which is also provided by Ada, would further be valuable to improve the readability of a program text. The

description of the exported objects would be concentrated in an explicit place and no longer be dispersed over a whole module body. The partitioning also helps to avoid forward references in the statement parts. It would be possible to apply the declaration rule, that objects used in declarations must be declared previously, on statement parts as well. And this would probably make single-pass compilation possible.

9.3 Symbol Files

The importance of the symbol files has grown during the development of the Lilith system. When the project was started, the symbol files were considered as a medium for transferring compiler internal information between several compilations. The first versions therefore were directly based on the type *Symbol* used for the inter-pass communication. Therefore, with each modification of this type a new version of encoding of symbol files was generated which was incompatible with its ancestors. This forced the recompilation of all programs only for reasons of compiler modifications.

For making the symbol files more independent of the internal compiler organization, a separate enumeration type, *SymFileSymbols* (see Appendix 2), was defined which allowed a more general encoding of symbol files. This new organization has been satisfactory until now, but further considerations show that the information on the symbol file should be represented in a still more generalized format.

On Lilith, the operating system Medos-2 is completely written in Modula-2 and the interface to the operating system is directly specified by a collection of definition modules. The advantage of such an interface specification is that it may be considered by the user as a collection of library modules which provide special services. The compiler can perform *complete parameter checking on all references to the operating system*. This improves security in the use of the functions of the operating system. As a consequence, the symbol files which represent the system interface must be stable, as long as the interface is stable. This makes the symbol file organization dependent on the operating system, far from belonging to the compiler only.

Generality of the symbol file is also necessary if more than one programming language is supported on a computer system and if combination of separate parts written in different languages is desired.

Proposal

The format of the information described on the symbol file should be highly independent of the internal data representation of a compiler. It should be globally defined by the operating system and respected by all compilers generating code for this system. Perhaps, a similar encoding as used for object files (see Appendix 3) would be appropriate. In addition, a common strategy for allocating data and a common calling sequence for activating procedures should be defined.

9.4 The Modula-2 Compiler on Lilith

The actual structure and organization of the Modula-2 compiler on Lilith has been influenced to some degree by its history, and this has not always been to its advantage. Instead of writing the compiler from scratch, the development was started with the existing Modula-1 compiler on the PDP-11. The advantage of this choice was a rapid implementation of a first Modula-2 version. Afterwards, many hours were spent to convert the given compiler structures into structures which take advantage of Modula-2's enriched facilities. Relics of its ancestor still survive in the current compiler version. This is not all negative. Especially, the given modularization was well chosen and facilitated the further development of the compiler. Perhaps, in retrospect, a better decision would have been to first change the Modula-1 compiler to accept Modula-2, and afterwards to write a new compiler from scratch. A more independent general design of the compiler structures would have been possible.

A second important fact which influenced the compiler organization was the available hardware. The small memory space on the PDP-11 (28 kword) was a limitation which, in order to save memory space, forced many reorganizations (e.g. identifier table on a file, linear links in the symbol table). It also influenced the multi-pass organization of the compiler. The final partition into four passes was an optimal choice for the PDP-11. As one goal of the compiler project was to first have a cross compiler on the PDP-11 to develop the basic software for Lilith, and afterwards to transport this compiler to Lilith, the multi-pass character was also given for the final compiler.

The memory space of Lilith (128 kword) would certainly allow the implementation of a single-pass compiler. This is, however, impossible without a slight modification of the Modula-2 language in order to tackle forward references. Such a modification might be, for example, the generalized module concept as it is proposed above.

The separate compilation feature has been very useful for the development of the compiler. On the one hand it was possible to reduce the size of the compilation units, on the other hand it helped to guarantee that all compiler parts refer to the same global data structure definitions. This improved the security and reliability of the compiler. The first three passes are similar in their modular structure and mainly consist of three separate modules. The main module contains the parser which processes and checks the symbol sequence of the compiled unit. The input-output module handles all direct operations on files and makes the other modules independent of the underlying file system. The third module controls the scopes and the links in the symbol table. Beside this general subdivision some supplementary separate modules (MCP1Reals, MCP2Reference, MCOperations) were designed. These provide special services. The main reason for splitting them from the other modules was that their implementation was considered as instable and heavily dependent on the environment of the compiler. For example, in order to provide the debugger with the needed information, the implementation of module MCP2Reference has been changed several times without affecting the definition module and the other modules of Pass2.

This paragraph closes with a comment on hiding data in modules. It is widely

accepted that a good concept in modular programming is to use a module structure to hide data and data structures and to allow external access and operations on these data by a collection of exported procedures only. This protects the data against unauthorized and inconsistent modifications and makes the clients independent of the actual data organization. This concept is generally respected in the Modula-2 compiler, with one important exception: The symbol table structure is globally defined and directly accessible from all compiler modules. It is a very fundamental structure in the compiler and its data are frequently inspected and modified by all compiler parts. Access via procedures would heavily burden the compilation process. The disadvantage of the chosen solution is that the interface becomes large and contains a detailed structure description which always fills the symbol table when an importing module is compiled. Compromises between security and efficiency are sometimes necessary and inevitable in large programs. In library and system modules, however, open structures should be avoided and security should be the highest principle.

Appendix 1 Syntax of Modula-2

First, the Modula-2 syntax is listed as it is defined in the Modula-2 report [Wir82]. A Lilith specific extension is described later.

```
ident                = letter { letter | digit } .
number               = integer | real .
integer              = digit { digit } | octalDigit { octalDigit } ( "B" | "C" ) |
                       digit { hexDigit } "H" .
real                 = digit { digit } "." { digit } [ ScaleFactor ] .
ScaleFactor          = "E" [ "+" | "-" ] digit { digit } .
hexDigit             = digit | "A" | "B" | "C" | "D" | "E" | "F" .
digit                = octalDigit | "8" | "9" .
octalDigit           = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" .
string               = "'" { character } "'" | '"' { character } '"' .
qualident            = ident { "." ident } .
ConstantDeclaration  = ident "=" ConstExpression .
ConstExpression      = SimpleConstExpr [ relation SimpleConstExpr ] .
relation             = "=" | "#" | "<>" | "<" | "<=" | ">" | ">=" | IN .
SimpleConstExpr      = [ "+" | "-" ] ConstTerm { AddOperator ConstTerm } .
AddOperator          = "+" | "-" | OR .
ConstTerm            = ConstFactor { MulOperator ConstFactor } .
MulOperator          = "*" | "/" | DIV | MOD | AND | "&" .
ConstFactor          = qualident | number | string | set |
                       "(" ConstExpression ")" | NOT ConstFactor .
set                  = [ qualident ] "{" [ element { "," element } ] "}" .
element              = ConstExpression [ ".." ConstExpression ] .
TypeDeclaration      = ident "=" type .
type                 = SimpleType | ArrayType | RecordType | SetType |
                       PointerType | ProcedureType .
SimpleType           = qualident | enumeration | SubrangeType .
enumeration          = "(" IdentList ")" .
IdentList            = ident { "," ident } .
SubrangeType         = "[" ConstExpression ".." ConstExpression "]" .
ArrayType            = ARRAY SimpleType { "," SimpleType } OF type .
RecordType           = RECORD FieldListSequence END .
FieldListSequence    = FieldList { ";" FieldList } .
FieldList            = [ IdentList ":" type |
                         CASE [ ident ":" ] qualident OF variant { "|" variant }
                         [ ELSE FieldListSequence ] END ] .
variant              = CaseLabelList ":" FieldListSequence .
CaseLabelList        = CaseLabels { "," CaseLabels } .
CaseLabels           = ConstExpression [ ".." ConstExpression ] .
SetType              = SET OF SimpleType .
PointerType          = POINTER TO type .
ProcedureType        = PROCEDURE [ FormalTypeList ] .
FormalTypeList       = "(" [ [ VAR ] FormalType { "," [ VAR ] FormalType } ] ")"
                       [ ":" qualident ] .
```



```
VariableDeclaration  = IdentList ":" type .
designator           = qualident { "." ident | "[" ExpList "]" | "↑" } .
ExpList              = expression { "," expression } .
expression           = SimpleExpression [ relation SimpleExpression ] .
SimpleExpression     = [ "+" | "-" ] term { AddOperator term } .
term                 = factor { MulOperator factor } .
factor               = number | string | set | designator [ ActualParameters ] |
                       "(" expression ")" | NOT factor .
ActualParameters     = "(" [ ExpList ] ")" .
statement            = [ assignment | ProcedureCall |
                         IfStatement | CaseStatement | WhileStatement |
                         RepeatStatement | LoopStatement | ForStatement |
                         WithStatement | EXIT | RETURN [ expression ] ] .
assignment           = designator ":=" expression .
ProcedureCall        = designator [ ActualParameters ] .
StatementSequence    = statement { ";" statement } .
IfStatement          = IF expression THEN StatementSequence
                         { ELSIF expression THEN StatementSequence }
                         [ ELSE StatementSequence ] END .
CaseStatement        = CASE expression OF case { "|" case }
                         [ ELSE StatementSequence ] END .
case                 = CaseLabelList ":" StatementSequence .
WhileStatement       = WHILE expression DO StatementSequence END .
RepeatStatement      = REPEAT StatementSequence UNTIL expression .
ForStatement         = FOR ident ":=" expression TO expression
                         [ BY ConstExpression ] DO StatementSequence END .
LoopStatement        = LOOP StatementSequence END .
WithStatement        = WITH designator DO StatementSequence END .
ProcedureDeclaration = ProcedureHeading ";" block ident .
ProcedureHeading     = PROCEDURE ident [ FormalParameters ] .
block                = { declaration } [ BEGIN StatementSequence ] END .
declaration          = CONST { ConstantDeclaration ";" } |
                       TYPE { TypeDeclaration ";" } |
                       VAR { VariableDeclaration ";" } |
                       ProcedureDeclaration ";" | ModuleDeclaration ";" .
FormalParameters     = "(" [ FPSection { ";" FPSection } ] ")" [ ":" qualident ] .
FPSection            = [ VAR ] IdentList ":" FormalType .
FormalType           = [ ARRAY OF ] qualident .
ModuleDeclaration    = MODULE ident [ priority ] ";" { import } [ export ]
                         block ident .
priority             = "[" ConstExpression "]" .
export               = EXPORT [ QUALIFIED ] IdentList ";" .
import               = [ FROM ident ] IMPORT IdentList ";" .
DefinitionModule     = DEFINITION MODULE ident ";" { import } [ export ]
                         { definition } END ident "." .
definition           = CONST { ConstantDeclaration ";" } |
                       TYPE { ident [ "=" type ] ";" } |
                       VAR { VariableDeclaration ";" } |
```

```
                ProcedureHeading ";" .
ProgramModule   = MODULE ident [ priority ] ";" { import }
                  block ident "." .
CompilationUnit = DefinitionModule |
                  [ IMPLEMENTATION ] ProgramModule .
```

The only extension of Modula-2 for Lilith is the addition of so-called *code procedures*. A code procedure is a declaration in which the procedure body has been replaced by a (sequence of) code number(s), representing M-code instructions (see [Wir81]). Code procedures are a facility to make available routines that are micro-coded at the level of Modula-2. This facility is reflected by the following extension to the syntax of the *procedure declaration*:

```
ProcedureDeclaration= ProcedureHeading ";" ( block | codeblock ) ident .
codeblock           = CODE CodeSequence END .
CodeSequence        = code { ";" code } .
code                = [ ConstExpression ] .
```


Appendix 2 Syntax of the Symbol File

The terminal symbols of the symbol file syntax correspond to the constants in type *SymFileSymbols*, declared in module MCSymDefs:

```
SymFileSymbols =
  (endfileSS,
   unitSS, endunitSS, importSS, exportSS,
   constSS, normalconstSS, realconstSS, stringconstSS,
   typSS, arraytypSS, recordtypSS, settypSS, pointertypSS,
   hiddentypSS,
   varSS, procSS, funcSS,
   identSS, periodSS, colonSS, rangeSS,
   lparentSS, rparentSS, lbracketSS, rbracketSS,
   caseSS, ofSS, elseSS, endSS)
```

The information on the symbol file is a sequence of bytes, packed into words as follows:

word = firstByte + secondByte * 400B

Symbols and characters are represented in one byte by their ordinal value. Numbers are split into two bytes with *byte1 = number DIV 400B* and *byte2 = number MOD 400B*.

```
Unit            = Header { SymbolModule } ENDFILESS .
Header          = SymFile ModuleKey DefModName .
SymFile         = Value .       / symbol file syntax version /
Value           = NORMALCONSTSS Number .
ModuleKey       = Value Value Value .   / compilation time stamp /
DefModName      = Ident .   / name of compiled definition module /
Ident           = IDENTSS Character { Character } '0C' .
SymbolModule    = UNITSS ModuleKey Ident [ IMPORTSS { Ident } ]
                    [ EXPORTSS { Ident } ] { Definition } ENDUNITSS .
Definition      = CONSTSS { ConstDeclaration } |
                  TYPESS { TypeDeclaration } |
                  PROCSS ProcHeading |
                  VARSS { VarDeclaration } .
ConstDeclaration = Ident Constant .
Constant        = Value QualIdent | RealConst | StringConst .
QualIdent       = Ident [ PERIODSS Ident ] .
RealConst       = REALCONSTSS RHigh RLow .
RHigh           = Number .      /upper part of real number/
RLow            = Number .      /lower part of real number/
StringConst     = STRINGCONSTSS Character { Character } '0C' .
TypeDeclaration = Ident Type .
Type            = SimpleType | HIDDENTYPSS | ArrayType | RecordType |
                  SetType | PointerType | ProcType .
SimpleType      = QualIdent | Enumeration | Subrange .
```

```
Enumeration     = LPARENTSS { Ident Value } RPARENTSS .
Subrange        = LBRACKETSS Constant RANGESS Constant RBRACKETSS .
ArrayType       = ARRAYTYPSS SimpleType OFSS Type .
RecordType      = RECORDTYPSS { Fields } [ Variants ] ENDSS Size .
Fields          = Ident Offset COLONSS Type .
Variants        = CASESS COLONSS QualIdent
                    { OFSS { VarVal } COLONSS [ Variants ] Size }
                    [ ELSESS [ Variants ] Size ] ENDSS .
Size            = Value .
Offset          = Value .
VarVal          = Value .
SetType         = SETTYPSS SimpleType .
PointerType     = POINTERTYPSS Type .
ProcType        = PROCSS LPARENTSS { [ VARSS ] [ ARRAYTYPSS ]
                    QualIdent } RPARENTSS [ COLONSS QualIdent ] .
ProcHeading     = Ident ProcNum ProcType .
ProcNum         = Value .
VarDeclaration  = Ident Address COLONSS Type .
Address         = RelAddr | AbsAddr .
RelAddr         = Value .
AbsAddr         = LBRACKETSS Value RBRACKETSS .
```


Appendix 3 Syntax of the Object File

The format of the object file is chosen according to a linker-loader input format of Medos-2. Its syntax is generally defined as follows:

```
Loadfile        = { Block } .
Block           = BlockType BlockSize { BlockWord } .
BlockType       = "200B" | "201B" | .... | "377B" .
BlockSize       = Number . /number of BlockWords/
BlockWord       = Number .
```

The load file is a sequence of words, with BlockType and Number each represented in one word.

The object file generated by the compiler obeys a syntactic structure, called *Module*.

```
Module          = [ VersionBlock ] HeaderBlock [ ImportBlock ]
                    { ModuleCode | DataBlock } .
VersionBlock    = VERSION BlockSize VersionNumber .
BlockSize       = Number .
VersionNumber   = Number .
HeaderBlock     = MODULE BlockSize ModuleName DataSize
                    [ CodeSize Flags ] .
ModuleName      = ModuleIdent ModuleKey .
ModuleIdent     = Letter { Letter | Digit } { "0C" } . /exactly 16 characters/
ModuleKey       = Number Number Number .
DataSize        = Number . /in words/
CodeSize        = Number . /in words/
Flags           = Number .
ImportBlock     = IMPORT BlockSize { ModuleName } .
ModuleCode      = CodeBlock [ FixupBlock ] .
CodeBlock       = CODETEXT BlockSize WordOffset { CodeWord } .
WordOffset      = Number . /in words from the beginning the module/
CodeWord        = Number .
FixupBlock      = FIXUP BlockSize { ByteOffset } .
ByteOffset      = Number . /in bytes from the beginning of the module/
DataBlock       = DATATEXT BlockSize WordOffset { DataWord } .
DataWord        = Number .
```

Following frame types are assigned:

```
VERSION         = "200B" .
MODULE          = "201B" .
IMPORT          = "202B" .
CODETEXT        = "203B" .
DATATEXT        = "204B" .
FIXUP           = "205B" .
```

A loaded module consists of two nonoverlapping frames: A *data frame* and a *code*

frame. Data blocks are loaded into the data frame, code blocks into the code frame. Fix-up blocks contain binding information for the linker-loader. The offsets of a fix-up block refer to the loaded code in the code frame where provisional module numbers must be replaced by actual module numbers which are determined and assigned by the linker-loader. The provisional modules are assigned by the compiler and correspond to the modules listed in the import block. A provisional module number 0 indicates a reference to the own module.

Appendix 4 Syntax of the Inter-pass Files

In the following syntax descriptions the identifiers written in capitals are terminal symbols and correspond to the constant values of type *Symbol*, declared in module MCBase:

```
Symbol =
  (eop,
   andsy, divsy, times, slash, modsy, notsy, plus, minus, orsy,
   eql, neq, grt, geq, lss, leq, insy,
   lparent, rparent, lbrack, rbrack, lconbr, rconbr,
   comma, semicolon, period, colon, range,
   constsy, typesy, varsy,
   arraysy, recordsy, variant, setsy, pointersy, tosy, arrow, hidden,
   importsy, exportsy, fromsy, qualifiedsy,
   codesy, beginsy,
   casesy, ofsy, ifsy, thensy, elsifsy, elsesy, loopsy, exitsy,
   repeatsy, untilsy, whilesy, dosy, withsy,
   forsy, bysy, returnsy, becomes, endsy,
   call, endblock,
   definitionsy, implementationsy, proceduresy, modulesy, symbolsy,
   ident, intcon, cardcon, intcarcon, realcon, charcon, stringcon,
   option, errorsy, eol,
   namesy,
   field, anycon)
```

Information on the inter-pass files is a sequence of words. Numbers and pointers are represented in one word. Symbols and position numbers are packed together into one word as follows:

ORD(symbol) * 400B + pos

Line marks, error messages, and compiler options are interspersed on the inter-pass files at any position. They apply to following syntax definition:

```
Interspersed   = LineMark | Error | Option .
LineMark       = EOL LineNumber .
LineNumber     = Number .
Error          = ERRORSY ErrorNumber .
ErrorNumber    = Number .
Option         = OPTION LetterVal Switch .
LetterVal      = Number.
Switch         = PLUS | MINUS .
```

4.1 Inter-pass File between Pass1 and Pass2

Information on this file consists of a sequence of ***symbol modules*** followed by the symbolic representation of the compiled unit. Non-teminal symbols beginning with *Sym* indicate supplementary information which is contained in symbol modules only.

```
Unit          = { SYMBOLSY ModuleKey Module }
                  ( DEFINTIONSY | IMPLEMENTATIONSY | MODULESY )
                  Module EOP .
ModuleKey     = CARDCON Value CARDCON Value CARDCON Value .
Value         = Number .
Module        = Ident [ Priority ] { Import } [ Export ] Block .
Ident         = IDENT SpellIndex .
SpellIndex    = Number .
Priority      = LBRACK Constant RBRACK .
Constant      = ConstExpr .
ConstExpr     = SimpleConstExpr [ RelOp SimpleConstExpr ] .
SimpleConstExpr = [ PLUS | MINUS ] ConstTerm { AddOp ConstTerm } .
ConstTerm     = ConstFact { MulOp ConstFact } .
ConstFact     = INTCON Value |
                CARDCON Value [ SymType ] |
                INTCARCON Value |
                CHARCON Value |
                REALCON EntryAddr |     /real value in heap/
                STRINGCON EntryAddr Length |
                QualIdent [ SetConstr ] |
                SetConstr |
                LPARENT ConstExpr RPARENT |
                NOTSY ConstFact .
SymType       = QualIdent .  /typename of constant in symbol file/
QualIdent     = Ident { PERIOD Ident } .
EntryAddr     = Number .
Length        = Number .
SetConstr     = LCONBR [ ElementList ] RCONBR .
ElementList   = Element { COMMA Element } .
Element       = Constant [ RANGE Constant ] .
MulOp         = TIMES | SLASH | DIVSY | MODSY | ANDSY .
AddOp         = PLUS | MINUS | ORSY .
RelOp         = EQL | NEQ | GRT | GEQ | LSS | LEQ | INSY .
Import        = IMPORTSY IdentList | FROMSY Ident IdentList .
IdentList     = Ident { Ident } .
Export        = EXPORTSY IdentList | QUALIFIEDSY IdentList .
Block         = { Definition } [ BEGINSY StatSequence ] ENDBLOCK .
Definition    = CONSTSY { ConstDecl } |
                TYPESY { TypeDecl } |
                VARSY { VarDecl } |
                PROCEDURESY Procedure |
                MODULESY Module .
ConstDecl     = Ident Constant .
TypeDecl      = Ident Type .
Type          = SimpleType |
                HIDDEN |
                ARRAYSY SimpleType OFSY Type |
                RECORDSY { FieldList } ENDSY [ SymSize ] |
```



```
                SETSY SimpleType |
                POINTERSY Type |
                PROCEDURESY [ FormalTypeList ] .
SimpleType    = QualIdent | Subrange | Enumeration .
Subrange      = LBRACK Constant RANGE Constant RBRACK .
Enumeration   = LPARENT { Ident [ SymValue ] } RPARENT .
SymValue      = CARDCON Value .
FieldList     = { Ident [ SymOffset ] } COLON Type |
                CASESY [ Ident ] COLON QualIdent
                  { OFSY ( ElementList | { SymVarVal } ) COLON
                    { FieldList } [ SymSize ] }
                  [ ELSESY { FieldList } [ SymSize ] ] ENDSY .
SymOffset     = CARDCON Value .
SymVarVal     = CARDCON Value .
SymSize       = CARDCON Value .
FormalTypeList = LPARENT { [ VARSY ] FormalType } RPARENT
                  [ COLON QualIdent ] .
FormalType    = [ ARRAYSY ] QualIdent .
VarDecl       = { Ident [ Address ] } COLON Type .
Address       = AbsAddress | SymAddress .
AbsAddress    = LBRACK Constant RBRACK .
SymAddress    = CARDCON Value .
Procedure     = Ident ( SymProcNum PROCEDURESY FormalTypeList |
                  [ FormalParList ] [ Block | CodeBlock ] ) .
SymProcNum    = CARDCON Value .
FormalParList = LPARENT { [ VARSY ] { Ident } COLON FormalType }
                  RPARENT [ COLON QualIdent ] .
CodeBlock     = CODESY [ Constant { COMMA Constant } ] ENDBLOCK .
StatSequence  = { Statement } .
Statement     = BECOMES Designator COMMA Expression |
                CALL Designator ParamList |
                IFSY Expression StatSequence
                  { ELSIFSY Expression StatSequence }
                  [ ELSESY StatSequence ] ENDSY |
                WITHSY Designator StatSequence ENDSY |
                CASESY Expression
                  { OFSY ElementList COLON StatSequence }
                  [ ELSESY StatSequence ] ENDSY |
                LOOPSY StatSequence ENDSY |
                WHILESY Expression StatSequence ENDSY |
                REPEATSY StatSequence UNTILSY Expression |
                FORSY Ident COMMA Expression TOSY Expression
                  [ BYSY Constant ] StatSequence ENDSY |
                RETURNSY [ LPARENT Expression RPARENT ] |
                EXITSY .
Designator    = QualIdent
                  { LBRACK Expression RBRACK | PERIOD Ident | ARROW } .
Expression    = SimpleExpr [ RelOp SimpleExpr ] .
```

```
SimpleExpr      = [ PLUS | MINUS ] Term { AddOp Term } .
Term            = Factor { MulOp Factor } .
Factor          = INTCON Value |
                  CARDCON Value |
                  INTCARCON Value |
                  CHARCON Value |
                  REALCON EntryAddr |     /real value in heap/
                  STRINGCON EntryAddr Length |
                  SetConstr |
                  Designator [ ParamList | SetConstr ] |
                  NOTSY Factor |
                  LPARENT Expression RPARENT .
ParamList       = LPARENT [ Expression { COMMA Expression } ] RPARENT .
```

4.2 Inter-pass File between Pass2 and Pass3

Information on this file contains the statement parts of the procedures and modules of the compiled unit. The procedure and module headers reference the corresponding name entries in the symbol table.

```
Unit            = Module EOP .
Module          = MODULESY Nptr { Import } Block .
Nptr            = Pointer .
Import          = FROMSY Ident { Ident } | IMPORTSY { Ident } .
Ident           = IDENT SpellIndex .
SpellIndex      = Number .
Block           = { Module | Procedure } [ BEGINSY StatSequence ]
                    ENDBLOCK .
Procedure       = PROCEDURESY Nptr [ Block | CodeBlock ] .
CodeBlock       = CODESY [ Constant { COMMA Constant } ] ENDBLOCK .
Constant        = ConstExpr .
ConstExpr       = SimpleConstExpr [ RelOp SimpleConstExpr ] .
SimpleConstExpr = [ PLUS | MINUS ] ConstTerm { AddOp ConstTerm } .
ConstTerm       = ConstFact { MulOp ConstFact } .
ConstFact       = INTCON Value |
                  CARDCON Value |
                  INTCARCON Value |
                  CHARCON Value |
                  REALCON EntryAddr |     /real value in heap/
                  STRINGCON EntryAddr Length |
                  QualIdent [ SetConstr ] |
                  SetConstr |
                  LPARENT ConstExpr RPARENT |
                  NOTSY ConstFact .
Value           = Number .
EntryAddr       = Number .
Length          = Number .
```



```
QualIdent       = { Ident PERIOD } Ident .
SetConstr       = LCONBR [ ElementList ] RCONBR .
ElementList     = Element { COMMA Element } .
Element         = Constant [ RANGE Constant ] .
MulOp           = TIMES | SLASH | DIVSY | MODSY | ANDSY .
AddOp           = PLUS | MINUS | ORSY .
RelOp           = EQL | NEQ | GRT | GEQ | LSS | LEQ | INSY .
StatSequence    = { Statement } .
Statement       = BECOMES Designator COMMA Expression |
                  CALL ( Name | Designator ) ParamList |
                  IFSY Expression StatSequence
                    { ELSIFSY Expression StatSequence }
                    [ ELSESY StatSequence ] ENDSY |
                  WITHSY Designator StatSequence ENDSY |
                  CASESY Expression
                    { OFSY ElementList COLON StatSequence }
                    [ ELSESY StatSequence ] ENDSY |
                  LOOPSY StatSequence ENDSY |
                  WHILESY Expression StatSequence ENDSY |
                  REPEATSY StatSequence UNTILSY Expression |
                  FORSY Ident COMMA Expression TOSY Expression
                    [ BYSY Constant ] StatSequence ENDSY |
                  RETURNSY [ LPARENT Expression RPARENT ] |
                  EXITSY .
Designator      = QualIdent
                    { LBRACK Expression RBRACK | PERIOD Ident | ARROW } .
Expression      = SimpleExpr [ RelOp SimpleExpr ] .
SimpleExpr      = [ PLUS | MINUS ] Term { AddOp Term } .
Term            = Factor { MulOp Factor } .
Factor          = INTCON Value |
                  CARDCON Value |
                  INTCARCON Value |
                  CHARCON Value |
                  REALCON EntryAddr |     /real value in heap/
                  STRINGCON EntryAddr Length |
                  Designator [ ParamList | SetConstr ] |
                  SetConstr |
                  NOTSY Factor |
                  LPARENT Expression RPARENT .
ParamList       = LPARENT [ Expression { COMMA Expression } ] RPARENT .
Name            = NAMESY Nptr .
```

4.3 Inter-pass File between Pass3 and Pass4

Information on this file consists of a sequence of procedures which may contain nested procedures. The module structure of the compiled unit is no longer visible. The code sequences of the code procedures are stored in the heap and not

contained on the file.

```
Unit          = { Procedure } ENDBLOCK EOP .
Procedure     = PROCEDURESY Nptr [ Block | CodeBlock ] .
Nptr          = Pointer .
Block         = { Procedure } [ BEGINSY StatSequence ] ENDBLOCK .
StatSequence  = { Statement } .
Statement     = BECOMES Variable COMMA Expression |
                CALL Variable ParamList |
                IFSY Expression StatSequence
                 { ELSIFSY Expression StatSequence }
                 [ ELSESY StatSequence ] ENDSY |
                FORSY Name COMMA Expression TOSY Expression
                 [ BYSY Constant ] StatSequence ENDSY |
                CASESY Expression
                 { OFSY { Element } COLON StatSequence }
                 [ ELSESY StatSequence ] ENDSY |
                WHILESY Expression StatSequence ENDSY |
                REPEATSY StatSequence UNTILSY Expression |
                LOOPSY StatSequence ENDSY |
                RETURNSY [ LPARENT Expression RPARENT ] |
                EXITSY |
                WITHSY Variable StatSequence ENDSY .
Variable      = [ FIELD FieldLevel PERIOD ] Name
                 { LBRACK Expression RBRACK | PERIOD Name | ARROW } .
FieldLevel    = Number .
Name          = NAMESY Nptr .
Expression    = SimpleExpr [ RelOp SimpleExpr ] .
SimpleExpr    = [ MINUS ] Term { AddOp Term } .
Term          = Factor { MulOp Factor } .
Factor        = Constant |
                Variable [ ParamList ] |
                LPARENT Expression RPARENT |
                NOTSY Factor .
Constant      = ANYCON TypeStruct Value .
MulOp         = TIMES | SLASH | DIVSY | MODSY | ANDSY .
AddOp         = PLUS | MINUS | ORSY .
RelOp         = EQL | NEQ | GRT | GEQ | LSS | LEQ | INSY .
TypeStruct    = Pointer .
Value         = Number .
ParamList     = LPARENT [ Expression { COMMA Expression } ] RPARENT .
Element       = Constant .
CodeBlock     = CODESY ENDBLOCK .
```

4.4 Inter-pass File between Pass4 and Lister

Information on the file is a sequence of double word blocks, which either are line

marks or error messages. The first word in a block is a positive number for a line mark, and a negative number for an error message.

```
Unit          = { InfoBlock } .
InfoBlock     = LineMark | Error .
LineMark      = LineNum CodeAddr .
LineNum       = Number .      /positive value/
CodeAddr      = Number .
Error         = ErrNum ErrPos .
ErrNum        = Number .      /negative value/
ErrPos        = Number .
```

Appendix 5 Syntax of the Reference File

The terminal symbols of the reference file syntax correspond to the constants in type *RefSymbol*, declared in module MCP2Reference:

```
RefSymbol =
  (reffileRS, endRS,
   moduleRS, procRS, typeRS, varRS,
   typerefRS, undefRS,
   integerRS, cardinalRS, charRS, booleanRS,
   realRS, bitsetRS, proctypRS,
   wordRS, addressRS, processRS,
   subrRS, enumRS, setRS, pointerRS,
   arrayRS, arrdynRS, recordRS,
   hiddenRS, openRS,
   constRS, fieldRS,
   absRS, indRS, relRS)
```

The information on the reference file is a sequence of bytes, packed into words as follows:

word = firstByte + secondByte * 400B

Symbols and characters are represented in one byte by their ordinal value. Numbers are split into two bytes with *byte1 = number DIV 400B* and *byte2 = number MOD 400B*.

```
Unit        = RefFile Module .
RefFile     = REFFILERS Number . / ref file syntax version /
Module      = MODULERS Head Block ENDRS .
Head        = LineNum ObjectNum Name .
LineNum     = Number .          / refers to source text /
ObjectNum   = Number .          / refers to procedure table /
Name        = { Character } "0C" .
Block       = { Module | Procedure | Type | Variable } .
Procedure   = PROCRS Head Block ENDRS .
Type        = TYPERS TypeNum Name TypeDesc .
TypeNum     = Number .
TypeDesc    = SUBRRS TypeSize Min Max TypeRef |
              ENUMRS TypeSize { Constant } ENDRS |
              SETRS TypeSize TypeRef |
              POINTERRS TypeSize TypeRef |
              ARRAYRS TypeSize TypeRef TypeRef |   / index, element /
              ARRDYNRS TypeSize TypeRef |
              RECORDRS TypeSize { Field } ENDRS |
              HIDDENRS TypeSize |
              OPENRS TypeSize TypeRef .
TypeSize    = Number .          / number of words /
Min         = Number .
```



```
Max         = Number .
TypeRef     = TYPEREFRS TypeNum |
              UNDEFRS |
              INTEGERRS |
              CARDINALRS |
              CHARRS |
              BOOLEANRS |
              REALRS |
              BITSETRS |
              PROCTYPRS |
              WORDRS |
              ADDRESSRS |
              PROCESSRS .
Constant    = CONSTRS Number Name .
Field       = FIELDRS Offset Name TypeRef .
Offset      = Number .
Variable    = VARRS LineNum AddrMode Address Name TypeRef .
AddrMode    = ABSRS | INDRS | RELRS .
Address     = Number .
```

Appendix 6 Compiler Statistics

The following tables show some statistics about the size of the Modula-2 compiler. For each separate module the number of *source text lines* (definition and implementation modules) and the memory space needed for *code* and *global data* is listed.

MCBase		267 lines	0.06 kword
MCPublic		234 lines	0.51 kword
CompFile		401 lines	1.01 kword
Storage		233 lines	0.24 kword
WriteStrings		42 lines	0.13 kword
Modula Base Part	5 modules	1177 lines	1.97 kword
MCInit		276 lines	0.68 kword
MCLookup		218 lines	0.32 kword
Options		139 lines	0.23 kword
FileNames		375 lines	0.65 kword
Initialization	4 modules	1008 lines	1.88 kword
MCP1Main		936 lines	2.42 kword
MCP1IO		940 lines	1.59 kword
MCP1Ident		389 lines	0.87 kword
MCP1Reals		231 lines	0.34 kword
MCLookup		218 lines	0.32 kword
MCSymDefs		71 lines	0.02 kword
Options		139 lines	0.23 kword
FileNames		375 lines	0.65 kword
Pass1	8 modules	3299 lines	6.46 kword
MCP2Main		1793 lines	3.59 kword
MCP2IO		390 lines	0.46 kword
MCP2Ident		635 lines	0.88 kword
MCP2Reference		343 lines	0.47 kword
MCOperations		318 lines	0.49 kword
Pass2	5 modules	479 lines	5.91 kword

MCP3Main		1692 lines	3.39 kword
MCP3IO		382 lines	0.44 kword
MCP3Ident		495 lines	0.58 kword
MCOperations		318 lines	0.49 kword
Pass3	4 modules	2887 lines	4.92 kword
MCP4Main		556 lines	1.04 kword
MCP4Global		294 lines	0.41 kword
MCP4CodeSys		1037 lines	3.04 kword
MCP4AttributSys		655 lines	1.07 kword
MCP4ExprSys		660 lines	1.19 kword
MCP4CallSys		529 lines	1.01 kword
MCMnemonics		101 lines	0.02 kword
Pass4	7 modules	3832 lines	7.80 kword
MCSymFile		645 lines	0.90 kword
MCSymDefs		71 lines	0.02 kword
Conversions		121 lines	0.15 kword
SymFile	3 modules	837 lines	1.08 kword
MCLister		497 lines	2.02 kword
Conversions		121 lines	0.15 kword
Lister	2 modules	618 lines	2.17 kword
Modula	5 modules	1177 lines	1.97 kword
Initialization	4 modules	1008 lines	1.88 kword
Pass1	8 modules	3299 lines	6.46 kword
Pass2	5 modules	3479 lines	5.91 kword
Pass3	4 modules	2887 lines	4.92 kword
Pass4	7 modules	3832 lines	7.80 kword
SymFile	3 modules	837 lines	1.08 kword
Lister	2 modules	618 lines	2.17 kword
Modula-2 Compiler	38 modules	17137 lines	32.23 kword

Glossary

The glossary explains the specific meaning of some terms used in this text.

Client

A separate module which imports objects specified in the interface of an other separate module.

Compilation

Transformation of a program text (source text) written in a high-level programming language into a more primitive encoding, e.g. a machine language.

Compilation Unit

Source text unit accepted by a compiler for compilation. In Modula-2 this is a definition module, an implementation module, or a separate module without export.

Debugger

A program which helps to analyze the state of execution of a program. The post-mortem debugger on Lilith operates on a memory dump which is saved onto a file at the moment of a program crash. Structural information about the objects used in a program program is read from reference files.

Definition Module

Interface part of a separate module in Modula-2. It contains all declarations which are needed for a complete specification of the interface of a module.

Export

Specification of objects defined within a module which are visible and may be used outside the module.

Implementation Module

Implementation part of a separate module in Modula-2. It complements the corresponding definition module, i.e. it is the actual encoding the specified interface. It contains local objects and statements which need not be known to the clients of the module.

Import

Specification of objects defined outside a module which may be referenced inside the module.

Independent Compilation

Compilation of a program split into several separate program parts without checking of the references among the separate parts.

Interface

A collection of information about objects of a separately compiled unit. It is a description of those objects which may be referenced by other separate units.

Inter-pass File

Work file of the Modula-2 compiler on Lilith which contains the symbol sequence passed between subsequent compiler passes.

Linker-loader

A component of the operating system Medos-2 on Lilith which loads the code of the separate modules of a program, checks the module keys, and establishes references among the separate modules.

Local Module

A nested unit within a separate module which is used for internal modularization.

Main Module

A separate module which is called for program execution. After loading of the code of this module and all directly or indirectly imported modules, a program is started by execution of the code sequence of the main module.

Module

1) A tool provided by high-level programming languages for structuring large programs and for data abstractions. A module combines objects which logically belong together and allows to control the use of objects across the module boundaries explicitly.

2) A syntactical unit provided by the programming language Modula-2. Visibility control of objects is achieved by specification in import and export lists.

3) (As generally used in the text, meaning) a separate module.

Module Key

A time stamp which is assigned to a separate module by the Modula-2 compiler on Lilith. It allows correct identification of a compiled version of the separate module.

Name Entry

A representation of a named object in the symbol table of the Modula-2 compiler on Lilith.

Object File

File containing the code of a separate module. It is generated by the Modula-2 compiler on Lilith upon compilation of an implementation module or a module without export. It is written in a format as it is accepted by the linking-loader of the operating system Medos-2.

Pass

A compiler part which performs one or more compilation steps on the whole program part to be compiled. A pass processes a representation of a compilation unit and analyzes and transforms the processed information. The output of a pass is possibly processed by a subsequent pass. The Modula-2 compiler on Lilith is organized as multi-pass compiler.

Program

A main module and a collection of separate modules which are directly of indirectly imported by this main module.

Reference File

File containing structural information about the objects declared in a separate module. It is generated by the Modula-2 compiler and used by the post-mortem debugger on Lilith.

Scope

A validity and visibility range for objects, usually constituted by the boundaries of modules and procedures. The identifiers of the objects must be unique within a scope.

Separate Compilation

Compilation of a program split into several separate program parts with full type- and parameter-checking among the separate parts upon compilation.

Separate Module

A unit which constitutes a separately compiled part of a Modula-2 program. A separate module which exports some of its objects is split into a definition and an implementation module. (A separate module is often simply called module in the text.)

Spelling Index

Internal representation of an identifier in the Modula-2 compiler on Lilith. It is a reference to the identifier table where the actual character representation of the identifier is stored.

Structure Entry

A representation of a type structure in the symbol table of the Modula-2 compiler on Lilith.

Symbol File

Symbolic interface description of a separate module. It is generated by the Modula-2 compiler on Lilith upon compilation of a definition module and considered as the only valid representation of the definition module for subsequent compilations of the corresponding implementation module and client modules.

Symbol Table

A compiler internal representation of the objects declared in a compilation unit. The symbol table of the Modula-2 compiler on Lilith is organized as a large network of name and structure entries which are stored in the heap of the memory.

References

[Ada80] The Programming Language Ada, Reference Manual, Proposed Standard Document, United States Department of Defense, 1980. Lecture Notes in Computer Science, Volume 106, Springer, Berlin, 1981.

[Amm75] U. Ammann, Die Entwicklung eines Pascal-Compilers nach der Methode des Strukturierten Programmierens, ETH Diss. Nr. 5456, Zürich, 1975.

[Ber82] G. Beretta, E. Biagioni, H. Burkhart, P. Fink, J. Nievergelt, J. Stelovsky, H. Sugaya, A. Ventura, J. Weydert, XS-1: An integrated interactive system and its kernel, Proc. 6th International Conference on Software Engineering, Tokyo, 1982.

[Cel80] A. Celentano, P. Della Vigna, C. Ghezzi, D. Mandrioli, Separate Compilation and Partial Specification in Pascal. IEEE Transactions on Software Engineering, Volume 6, Number 4, 1980, p. 320-328.

[Eul82] M. Eulenstein, An Extension to Pascal for Modular Programming and a Proposal of a Conceptionally Machine Independent Linker. H. Langmaack, B. Schlender, J.W. Schmidt, Eds., Implementierung Pascal-artiger Programmiersprachen, Teubner, Stuttgart, 1982, p. 29-45.

[Fre83] H.P. Frei, J.F. Jauslin, Graphical Presentation of Information and Services: A User Oriented Interface. C.J. van Rijsbergen, Ed., Information Technology: Research and Development, Volume 2, Number 1, Butterworths, Oxford, 1983, p. 23-42.

[Gei79] L. Geissmann, Modulkonzept und Separate Compilation in Modula-2. W. Remmele, H. Schecher, Eds., Microprogramming, Teubner, Stuttgart, 1979, p. 98-114.

[Gei82] L. Geissmann, J. Hoppe, Ch. Jacobi, Sv.E. Knudsen, W. Winiger, N. Wirth, Lilith Handbook, Institut für Informatik, ETH, Zürich, 1982.

[Ich77] J.D. Ichbiah, G. Ferran, Separate Definition and Compilation in LIS and its Implementation. J.H. Williams, D.A. Fisher, Eds., Design and Implementation of Programming Languages. Lecture Notes in Computer Science, Volume 54, Springer, Berlin, 1977, p. 288-297.

[Jac83] Ch. Jacobi, Code Generation and the Lilith Architecture, ETH Diss. Nr. 7195, Zürich, 1983.

[Jen75] K. Jensen, N. Wirth, PASCAL User Manual and Report, Second Edition, Springer, Berlin, 1975.

[Koc82] J. Koch, M. Mall, P. Putfarken, M. Reimer, J.W. Schmidt, C.A. Zehnder, Modula/R Report, Lilith Version, Institut für Informatik, ETH, Zürich, 1982.

[Knu83] Sv.E. Knudsen, A paper in preparation, concerning the Medos-2 operating system on Lilith, Institut für Informatik, ETH, Zürich, 1983.

[Le78] V.K. Le, The Module: A Tool For Structured Programming, ETH Diss. Nr. 6153, Zürich, 1978.

[LeB79] R. LeBlanc, Ch. Fischer, On Implementing Separate Compilation in Block-Structured Languages, Sigplan Notices, Volume 14, Number 8, 1979, p. 139-143.

[Mit79] J.G. Mitchel, W. Maybury, R. Sweet, Mesa Language Manual, Version 5. Xerox PARC, CSL-79-3, Palo Alto, 1979.

[Reb83] J. Rebsamen, M. Reimer, P. Ursprung, C.A. Zehnder, A. Diener, LIDAS - The Database System for the Personal Computer Lilith, Proc. INRIA Workshop on Relational DBMS Design / Implementation / Use on Micro-Computers, Toulouse, 1983. (in print)

[Sof81] SofTech Microsystems, UCSD Pascal, Users Manual, Version IV.0, SofTech Microsystems, Inc, San Diego, 1981.

[Wir71] N. Wirth, The programming language PASCAL, Acta Informatica 1, 1971, p. 35-63.

[Wir77] N. Wirth, Modula: a Language for Modular Multiprogramming, Software - Practice and Experience, Volume 7, 1977, p. 3-35.

[Wir78] N. Wirth, Modula-2, Institut für Informatik, ETH, Report Nr. 27, Zürich, December 1978.

[Wir80] N. Wirth, Modula-2, Institut für Informatik, ETH, Report Nr. 36, Zürich, March 1980.

[Wir81] N. Wirth, The personal computer Lilith. A.I. Wassermann Ed., Software Development Environments, IEEE Computer Society Press, 1981.

[Wir82] N. Wirth, Programming in Modula-2, Springer, Berlin, 1982.

Curriculum Vitae

I was born on 10th April 1953 in Baden. From 1960 to 1965 I attended the primary schools in Spreitenbach and Brugg, and from 1965 to 1969 the Bezirksschule in Brugg. In 1969 I entered to the Kantonsschule Baden where I obtained 1972 the Matura Typ C.

In the fall of 1972 I began my studies in mathematics at the Swiss Federal Institute of Technology (ETH) in Zürich and finished them in the spring 1977 with the diploma "Dipl. Math. ETH". As a diploma-thesis I implemented a Pascal course for an interactive education system (XS-0).

From spring 1977 to autumn 1977 I was teacher for mathematics at the Kantonsschule Baden.

Since October 1977 I have worked as an assistant at the Institut für Informatik of ETH Zürich in the research group of Prof. N. Wirth at the Lilith project.