

SEPARATE COMPILATION AND NEW LANGUAGE FEATURES

by



Gordon V. Cormack

A thesis
presented to the University of Manitoba
in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy
in
Computer Science

Winnipeg, Manitoba, 1981

(c) Gordon V. Cormack, 1981

SEPARATE COMPILATION AND NEW LANGUAGE FEATURES

BY

GORDON V. CORMACK

A thesis submitted to the Faculty of Graduate Studies of
the University of Manitoba in partial fulfillment of the requirements
of the degree of

DOCTOR OF PHILOSOPHY

© 1981

Permission has been granted to the LIBRARY OF THE UNIVERSITY OF MANITOBA to lend or sell copies of this thesis, to the NATIONAL LIBRARY OF CANADA to microfilm this thesis and to lend or sell copies of the film, and UNIVERSITY MICROFILMS to publish an abstract of this thesis.

The author reserves other publication rights, and neither the thesis nor extensive extracts from it may be printed or otherwise reproduced without the author's written permission.

ABSTRACT

A facility to effect separate compilation within block-structured languages is presented and its utility discussed. This facility is expanded to support ongoing maintenance to, multiple versions of, and local modifications to large software systems. The following new abstraction mechanisms are described as they appear in the language L. A facility for user-defined types which is compatible with block structure and separate compilation is presented. More powerful parameter mechanisms is described, namely parameters of type TYPE, FIELD SELECTOR, and DURATION, as well as implicit parameters which allow aspects of the type of a formal parameter to be unspecified. A facility for parameterization of types and pointers as well as procedures is presented. The SCOPE of a variable is divided into VISIBILITY and DURATION. Storage of permanent duration is described which allows the language system to maintain control over disk and tape storage. The division between VISIBILITY and DURATION is applied to constants. The duration and modifiability of storage is included as part of the type of a pointer, preventing dangling references and allowing pointers to refer to constants. Overloading and user-defined automatic conversions are incorporated into the separate compilation system. This involves the solution to multi-dimensional matrix equations where "minimum" and "addition" are used in place of the conventional "summation" and "multiplication" in performing matrix multiplication.

ACKNOWLEDGEMENTS

I would like to thank my advisor, Dr. P. R. King, whose course entitled "New Directions in Programming Languages" sparked my original interest in this field. Thanks are due also to my examiners, Dr. Marc Joliat, Dr. Howard Ferch, and Dr. Robert Thomas, whose criticisms have greatly improved this thesis. I would also like to thank Eugene Reimer, Richard Tilley, Kip Rugger, and Bill Hoskins, each of whom spent many hours discussing various aspects of this thesis with me. I would also like to thank the Natural Sciences and Engineering Research Council of Canada for their support in the form of postgraduate scholarships.

CONTENTS

ABSTRACT	ii
ACKNOWLEDGEMENTS	iii

Chapter

page

I. INTRODUCTION	1
DESIGN CRITERIA FOR THE SEPARATE COMPILATION SYSTEM	2
DESIGN CRITERIA FOR L	3
ABSTRACTION MECHANISMS AND THEIR PARAMETERS	4
WHAT IS AN ABSTRACTION?	4
WHAT IS A PARAMETER?	4
ABSTRACTIONS IN L	4
PROCEDURES AND THEIR PARAMETERS	4
TYPES	5
TYPE GENERATORS	5
STORAGE GENERATORS AND POINTERS	5
PROCEDURE OVERLOADING	6
AUTOMATIC CONVERSIONS	6
ABSTRACTIONS NOT IN L	7
THESIS ORGANIZATION	7
II. A SEPARATE COMPILATION SYSTEM	9
INTRODUCTION	9
THE UNIT OF COMPILATION	10
ORDER OF COMPILATION	12
LIBRARIES AND DISTRIBUTED PROGRAMS	14
IMPLEMENTATION OF THE SEPARATE COMPILATION SYSTEM	16
THE OBJECT DATA BASE	17
THE LOADING ALGORITHM	18
STORAGE ALLOCATION CONSTRAINTS ON THE COMPILER	20
RAMIFICATIONS OF PROGRAM MODIFICATION	20
COMPATIBILITY INFORMATION IN OBJECT MODULES	22
COMPATIBILITY MAINTENANCE VIA THE LOADER	24
AN EXAMPLE OF THE USE OF THE PROGRAM EDITOR	25
OTHER FEATURES OF THE PROGRAM EDITOR	27
OBSOLETE VERSIONS	27
MULTI-CONFIGURATION COMPILATIONS	28
MULTI-CONFIGURATION DECLARATION CHANGES	29
SELECTABLE FEATURES	32
CONCLUSIONS	32

III. A VIEW OF PARAMETERS AND SCOPE	35
INTRODUCTION	35
PROCEDURES AND MACROS	35
MACROS	36
REMOTE PROCEDURES	36
SCOPED LANGUAGES	36
PARAMETERS	38
MORE POWERFUL PARAMETERS	39
PARAMETERS OF TYPE PROCEDURE	39
PARAMETERS OF ARBITRARY PRECISION	41
PARAMETERS OF TYPE TYPE	43
DEFAULT PARAMETERS	48
DEFINITION SITE AND CALL SITE RESOLUTION	49
FIELD SELECTOR PARAMETERS	51
POINTER PARAMETERS	54
TYPE DECLARATIONS AND THEIR PARAMETERS	55
EQUIVALENT TYPES	55
NEW TYPES	56
INITIAL VALUES ON TYPE DECLARATIONS	58
TYPE GENERATORS	59
MODULES AND ENCAPSULATION	62
DURATION OF VARIABLES	66
DURATION OF ARRAY BOUNDS	69
DURATION OF EXPRESSIONS	69
DURATION OF TYPES	73
CONSTANTS AND "READ ONLY VARIABLES"	74
INITIAL VALUES AND INVOCATION EXITS	75
POINTERS AND SCOPE	76
CONVERSION OF POINTER VALUES	78
DURATION OF PARAMETERS	80
COMPILE-TIME INVOCATION OF PROCEDURES	83
CONCLUSIONS	83
IV. OVERLOADING AND AUTOMATIC CONVERSION	85
INTRODUCTION	85
OVERLOADING IN FORTRAN	86
OVERLOADING IN PL/I	86
OVERLOADING IN ALGOL 68	87
OVERLOADING IN CLU	88
OVERLOADING IN ADA	88
THE LANGUAGE L	89
OVERLOADED PROCEDURES IN L	89
USER-DEFINED AUTOMATIC CONVERSIONS	91
USER-DEFINED AUTOMATIC CONVERSIONS IN L	93
IMPLEMENTATION OF OVERLOADING IN CONVENTIONAL LANGUAGES	98
AN ALGORITHM FOR THE SELECTION OF OVERLOADED FUNCTIONS IN	
ADA	98
AN ALGORITHM TO COUNT THE NUMBER OF INTERPRETATIONS OF AN	
EXPRESSION	100
OPTIMIZATIONS TO THE ALGORITHM	101
OPTIMIZATIONS WHICH REQUIRE NO EXTRA STORAGE	101
AN OPTIMIZATION WHICH USES EXTRA STORAGE	102

YIELDING THE SELECTED FUNCTIONS	103
CONCLUSIONS	105
IMPLEMENTATION OF USER-DEFINED CONVERSIONS	106
BEST FIT SELECTION WHERE CONVERSIONS ARE PERMITTED	107
OPTIMIZATIONS	108
YIELDING THE SELECTED FUNCTIONS	109
BEST FIT AND SEPARATE COMPILATION	112
WEAKENING THE LOGICAL EXPRESSION	115
FINDING THE MINIMUM COMPATIBILITY LIST	116
A GRAPHICAL REPRESENTATION OF THE OPERATOR SELECTION	
PROBLEM	117
A REPRESENTATION FOR ONE-PARAMETER FUNCTIONS	117
SOLVING THE MATRIX EQUATION $A \cdot L + X = B$ FOR X	122
CALCULATING ALL THE U_i	123
MINIMIZING THE COMPATIBILITY LIST USING MATRICES	124
FURTHER REDUCTIONS TO THE COMPATIBILITY LIST	128
A DUAL APPROACH TO REDUCTION	129
FUNCTIONS WITH MORE THAN ONE PARAMETER	130
MATRIX MULTIPLICATION IN N DIMENSIONS	131
A NOTATION FOR MULTI-DIMENSIONAL MATRIX EQUATIONS	133
REPRESENTING THE SELECTION PROCESS AS A MATRIX	
EXPRESSION	137
CONCLUSIONS	140
V. CONCLUSIONS	142
IMPLEMENTATION AND USE OF L	142
BLOCK STRUCTURE	143
OPTIMIZATION	145
REFERENCES	148

Chapter I

INTRODUCTION

This thesis presents two components of a complete programming system: a separate compilation system and a new programming language, L. The syntax and control structures of L are not fully and precisely defined in this thesis; they are described only to the degree necessary to illustrate the novel features of L which can be classified as abstraction mechanisms. The separate compilation system is presented first as a general system which could support a wide variety of languages. Next, the features of L are presented. All of the features of L have been designed with compilation in mind, and are intended to fit into the SCS.

Traditionally, the fields of language design, compiler design, and program support environments have been addressed independently. This independence has led frequently to programming languages which have been hard to implement, and general purpose operating systems not well suited to providing the facilities required to run programs written in some languages. For example, block-structured languages have existed for twenty years, yet very few systems have provisions for compiling nested programs other than as one monolithic unit. Similarly, very few linkage editors and loaders provide support for type checking.

The design aims of the SCS and L are now discussed.

1.1 DESIGN CRITERIA FOR THE SEPARATE COMPILATION SYSTEM

The major aim in the design of the SCS is to support the language L. Since L is a block-structured language, support for the separate compilation of block-structured languages is the major requirement of the SCS. In order to prevent massive recompilation effort for each program modification, the SCS must allow segments of the program to be recompiled. It is not desirable to force the programmer to break his program into independent units (thus defeating block structure at the macroscopic level) in order to compile them independently, as such a requirement would be inflicting the system's view of the world on the language design.

It is also undesirable that recompilation of one unit necessitate the recompilation of others. However, it is essential that the programmer be notified if this situation occurs; it should be impossible to run an inconsistent set of program units.

Since L is a statically type safe language it is essential that the SCS maintain this type security at all times.

Additional facilities which must be within the scope of the programming environment, and therefore must be supported by the SCS, are libraries of common abstractions, multiple versions of a program, and distribution of a program from one site to another.

1.2 DESIGN CRITERIA FOR L

L was designed according to three basic criteria: general applicability, compilability, and usability. Each is now discussed.

L is intended to be general purpose programming language; L is intended to be useful in a wide variety of applications, such as systems programming, mathematical programming, simulation, data base, and business applications. This generality is not achieved by attempting to anticipate all areas of application and to include built-in facilities for each. Rather, the tools are provided for each user to develop his own environment within which he can write his programs. Abstraction tools, block structure, and separate compilation permit such development.

L is a compilable language; all symbols and types are resolvable at compile time. Compilation allows efficient object code to be generated, early error detection, and permits operator selection (as discussed in chapter 4). It is also intended that L should be easily compilable; that is, each feature of L should have a straightforward translation into a low-level object code. In addition to initial compilability, L has been designed to allow separate compilation: recompilation of subprograms without necessitating recompilation of entire programs.

L is intended to be usable. This criterion indicates that programming in L should not be excessively cumbersome. Programmers familiar with traditional programming languages such as PL/I or Pascal should find that programming in L is less work, not more. An additional requirement imposed by this criterion is that all of the abstraction

mechanisms be usable by all levels of programmers, not just by system designers. For example, the programmer should not have to submit a request to an administrator in order to use these facilities for his own program or subsystem.

1.3 ABSTRACTION MECHANISMS AND THEIR PARAMETERS

1.3.1 WHAT IS AN ABSTRACTION?

"To abstract" is defined by the Oxford English Dictionary as

"to separate qualities common to all individuals of a group from the peculiarities of each individual."

1.3.2 WHAT IS A PARAMETER?

The Oxford English Dictionary defines a parameter as

"a quantity which is constant in a particular case considered, but which varies in different cases."

1.3.3 ABSTRACTIONS IN L

1.3.3.1 PROCEDURES AND THEIR PARAMETERS

A procedure provides a solution to a class of problems. The parameter mechanism permits the caller to specify which instance of the problem is to be solved. For example, we may write a procedure which solves a quadratic equation. Supplying the coefficients as parameters indicates which particular quadratic is to be solved. Thus, the ability to create procedures is an abstraction mechanism. More powerful parameterization facilities in L allow procedures to be written which solve a larger class of problems and are therefore better abstractions.

1.3.3.2 TYPES

A type in L describes a class of variables and values. The type defined here should not be confused with an abstract data type [LIS75], which is, in fact, a type plus a set of operations on values of that type. This distinction is also made by Steensgard-Madsen [STE81]. Since a type describes a set of variables it is an abstraction, and a facility for the user to define his own types is an abstraction mechanism.

1.3.3.3 TYPE GENERATORS

A type generator is an abstraction which defines a class of types. In conventional languages, the type generators ARRAY and STRUCTURE are usually found. An instance of a type generator is a type, and is created when parameters are supplied to a type generator. For example, ARRAY is not a type. However [10] ARRAY OF REAL is. 10 and REAL are parameters to the type generator ARRAY. Clearly, a facility for the user to create his own type generators is an abstraction mechanism.

1.3.3.4 STORAGE GENERATORS AND POINTERS

A variable is one instance of a given type. A storage generator allows many instances of storage of a given type to be created at execution time. A pointer variable contains all of the information which is unique to a particular instance of storage (its address). Thus a storage generator is a method of creating instances of an abstraction and a pointer is the mechanism identifying each instance. If more than just the address of storage varies from one instance to the next, this

information as well can be stored (along with the address) as a component of the pointer. For example, if the storage generated were of variable length, a pointer to such storage would consist of its address and length.

1.3.3.5 PROCEDURE OVERLOADING

Overloading allows many procedures to be defined with the same name. Thus the name is an abstraction representing many procedures which (presumably) implement a similar function. The particular instance of a procedure whose name is called is selected according to the number and type of its parameters. The most familiar example of overloading is the overloading of built-in operators in languages such as FORTRAN. The operator "+", for instance, performs addition, even though a wide variety of object code may be generated, depending on the types of the operands.

1.3.3.6 AUTOMATIC CONVERSIONS

Automatic conversions not only increase the convenience of using a programming language; they allow more powerful procedural abstractions to be created. For example, we may write a procedure, GAMMA, which evaluates the gamma function of a LONG REAL argument and returns a LONG REAL result. If automatic conversion exists between INTEGER, REAL, and LONG REAL, this GAMMA procedure will function with any combination of these types as argument and result. This abstraction is the equivalent of writing nine overloaded functions. Furthermore, such conversions will increase the abstraction of all procedures which operate on LONG REALs.

1.3.4 ABSTRACTIONS NOT IN L

A very powerful abstraction mechanism which is not found in L is a macro facility for textual substitution. Such a facility is outside the scope of a compilable language, as a macro language is itself a (different) programming language whose execution time output is source code for another language. Macros are often used where existing facilities are inadequate; the abstraction mechanisms in L are intended to be powerful enough to solve most problems for which macros are currently used.

1.4 THESIS ORGANIZATION

Chapter 2 of this thesis presents the separate compilation system which can be used for L or, indeed, for any block-structured language. The separate compilation system allows the programmer to recompile any procedure within the program without necessarily having to recompile that procedure's subprocedures. This system provides type checking across compilation boundaries, library support, and provision for on-going maintenance to and multiple versions of a program. In addition, the concepts of distributed programs and local versions of programs are presented.

Chapter 3 discusses scoping rules and four abstraction mechanisms: procedures, user-defined types, type generators, and storage generation. New parameters which enhance the procedural abstraction are presented, namely, parameters of type TYPE, parameters of type FIELD SELECTOR, and implicit parameters which are automatically supplied at each call. In addition, a mechanism is described whereby defaults may be supplied for omitted parameters; these defaults may be looked up in the symbol table either at the call site or at the site of the procedure's definition.

Two forms of user-defined type are presented in chapter 3: equivalent types and new types. The MODULE mechanism of L is shown to provide the information hiding necessary to implement an abstract data type as defined in [LIS75]. Type generators are provided by allowing the same parameterization to be applied to user-defined types as to procedures. The implicit parameter mechanism is used to interrogate these parameters. Lastly, storage allocation and pointers are discussed as they appear in L. The duration of storage, as well as whether or not that storage may be modified, may be controlled explicitly by the programmer. The duration and modifiability of storage to which a pointer refers are included in that pointer's type. A parameter facility for pointers is defined whereby a pointer may refer to storage whose duration or type is not fully specified at compile time.

Chapter 4 discusses the facilities of procedure overloading and automatic conversions. An algorithm for the selection of overloaded procedures based on the type of their parameters and result is presented. A facility for automatic conversion is introduced and found to interact with the selection process. For this reason, a best fit solution is presented for the selection process. The addition of user-defined automatic conversions has a large impact on the separate compilation system. Therefore, algorithms are presented which minimize this impact.

Chapter 5 contains some concluding remarks and makes suggestions as to further research arising from this thesis.

Chapter II

A SEPARATE COMPILATION SYSTEM

2.1 INTRODUCTION

This chapter presents a separate compilation system for L. However, most of the features of this system are applicable to other compilable languages such as Pascal [WIR71], Algol 68 [WIJ76], Ada [ICH79a ICH79b ICH80], Protel [FOX79], and Mesa [MIT79].

A separate compilation system is a system which allows the programmer to recompile a section of an existing program without having to recompile the entire program. Such a system is necessary for several reasons.

Firstly, a program may become so large that it is impossible to compile it in its entirety; it is necessary to break apart such a program into many compilation units, each of which is manageable by the compiler.

Secondly, a large program will inevitably require maintenance and modification. A considerable saving in compilation expense can be had by merely recompiling the affected compilation units, rather than the whole program.

Thirdly, it is not reasonable to expect the user to recompile each routine which he uses in his program. Such a requirement would cause

the recompilation of a large fraction of the operating system for most user programs.

Fourthly, the user must be able to create his own library or environment. Such a facility is necessary for the user to suit the language, L, to his specific purpose. It is through the use of such user libraries, coupled with the abstraction mechanisms presented, that L is intended to obviate the need for many of the special-purpose languages that exist and are being developed today.

Finally, a separate compilation system can be used to maintain several versions of a particular program without having to duplicate the large amount of code which may be common to many specific versions.

2.2 THE UNIT OF COMPILATION

The separate compilation system described here fulfils the above requirements by allowing the programmer to recompile any procedure within a program at any time. The procedure was chosen as the unit of compilation for the following reasons.

No extra compilation information need be specified by the programmer when he is writing his program. Blocks and procedures are natural units; current programming methodology dictates that programs be subdivided into fairly small procedures which could be compiled easily on a minicomputer. Blocks and procedures may be further combined into modules or packages, however such a combination is not forced by the compilation system. In fact, these modules are often too large to be compiled in their entirety. For example, Ada's MATH_LIB and TEXT_IO

packages are both enormous. There is, of course, no reason why all the components of a module could not be compiled together; there is no such requirement.

Other languages, such as FORTRAN and PL/I, use procedures as the unit of compilation. These languages require that these procedures be independent; that they share no common declarations. It is seldom convenient to break up a program into independent procedures; often this process is deferred until the program becomes so large that it can no longer be compiled.

Ada allows the separate compilation of procedures which are not independent. However, the header of the separately compiled procedure must be specified within its containing procedure, and the body marked as separate. The header, therefore, may not be separately compiled. Furthermore, the programmer has to predict which procedures may require recompilation and do extra work for these procedures. In the author's experience, it is impossible to predict which procedures will require maintenance or modification.

Another reason for the use of small compilation units is that when a modification is made to a program, the programmer making the modification must take control of the entire compilation unit. The use of small compilation units reduces the probability that two modifications will intersect.

In fact, any scope delimiter can be used as the unit of compilation; the procedure was chosen in L because it is the smallest unit. Under some circumstances, an entire scope may be too large a unit. We may

wish to recompile only one or two declarations from a particular scope. This could be done using the facilities discussed here. However, only compilation time would be saved; the effect on the system would be the same as if the entire scope had been recompiled.

2.3 ORDER OF COMPILATION

All separate compilation systems, other than those which recompile only independent units, require that these units be compiled according to some partial ordering. Furthermore, this same ordering is imposed on recompilations. In L, an order is required for the initial compilation of a program. However, after the initial compilation, units (procedures) may be recompiled in any order. Recompile of one procedure will necessitate recompilation of another only if that second procedure is invalidated by the recompilation of the first.

Since L is a block-structured language, the order of initial compilation can be determined easily. Since any procedure can reference the declarations of its ancestors, each of its ancestors must first be compiled before the procedure itself can be. Thus, the program may be compiled in any top-down order. Languages such as Ada, Mesa, and Protel explicitly name the units whose declarations are referenced. Such units must be compiled in an order such that any included unit is compiled before the unit which includes it.

When a unit is recompiled, any unit which can reference its declarations may be affected. In other compilation systems, these units must immediately be recompiled, along with any unit which may depend on

them. In the L compilation system, the new version is compared to the object code for the old, and it is determined whether or not dependent units could possibly be affected. If they could not, the change is called a compatible modification, and no further processing occurs. If the change is not compatible, the incompatible changes are recorded with the object code and are checked by the linker or linkage editor. If the change does not affect a unit, no further action is taken. If the unit is affected, the linker makes the appropriate fixups to the object code if possible. If a fixup is not possible, the linker rejects the unit as being incompatible with the new version of the program.

There are two reasons for such an approach. First, a large amount of recompilation time can be saved. Second, the system may be used where the dependent units are not necessarily available when the recompilation is done.

The amount of recompilation time which can be saved is not insignificant. Many common units will be referenced by almost every program in the system. Lauer and Satterthwaite state that such a universal recompilation for the Pilot project [LAU79] took one-and-one-half to two days of enforced inactivity among the members of the project.

Universal compilation, as mentioned above, is possible only if all components of the program are available to be recompiled. Consider, however, the standard prelude available to all programs. If such a separate compilation system were used to create this prelude, then all programs written in the language would be found to be dependent on this

prelude. After any modification to this prelude, such as adding a new procedure, it would be necessary to find and recompile every program in existence.

Several similar issues, including upward compatibility, are discussed in [JOL81].

2.4 LIBRARIES AND DISTRIBUTED PROGRAMS

A library can be implemented in the following way:

1. BEGIN
2. PROC SIN(X: REAL) RETURNS Y: REAL = BEGIN ... END
3. PROC COS(X: REAL) RETURNS Y: REAL = BEGIN ... END
4. BEGIN
5. END
6. END

A compilation of a users' program using this library can be viewed as a replacement of the block beginning on line 4. The program can be compiled whether or not the bodies of SIN and COS have been written. If they have not, a null body may be supplied at the outset, later to be replaced by the actual implementations. The user's program need not be recompiled when these replacements are done. If three programmers are developing the three programs, much of their work can proceed in parallel. Furthermore, if the library is a system library, maintenance may be done without forcing existing programs to be recompiled. The system therefore facilitates top-down as well as bottom-up program development. The Algol 68C system [BOU74] provides a subset of such a facility; only top-down compilation of nested environments is provided.

Of course, the programmer cannot actually change a public library. Instead, his modifications are effected in his own object data base

(ODB). When he links his program, both the library's and the programmers ODB are supplied. Any routines which the programmer has modified will reside in the programmer's ODB and will be selected in preference to the versions in the library's ODB. The linker verifies that the same ODB's are supplied at link time as those which were present at compile time.

There can be an unlimited number of user's ODBs, each of which is under the sole control of a user. It is not possible to have any pointers or references from the library ODB to the user ODB. On the other hand, references from the user ODB to the library ODB are numerous. If a program resides on more than one ODB, it is called a distributed program.

In a large program, there will probably be more ODB's than just those of the library and the user. There may be a user library, as well as system and sub-system libraries. In addition, there may be various special-purpose libraries which the programmer wishes to use, such as a linear algebra package or a plotting package. Each of these libraries would have its own ODB. When the user's program is compiled, all of the ODBs are supplied to the compiler to form a distributed object data base (DODB).

Distributed programs also arise from a user's modifications to a supplier's software. In this case, the user makes whatever modifications are necessary to make the software conform to his requirements. These modifications are reflected in his ODB. Now, if the distributor applies maintenance to the software, the user can still

make use of this new release so long as none of the procedures he modified are also modified by the maintenance. Reducing the size of the unit of compilation reduces the possibility of such conflicting modifications. IBM has recently developed a System Modification Program [IBM80] to aid the user in applying such modifications.

2.5 IMPLEMENTATION OF THE SEPARATE COMPILATION SYSTEM

The heart of the separate compilation system is the ODB. The ODB contains the object code for each procedure in the program. In addition, the ODB contains all of the symbol table information necessary to compile (or recompile) the procedure or one of its sons. The dependency structure of the procedures is maintained in the ODB. In addition, control information is stored which indicates when recompilation is necessary.

Two programs which access the ODB are part of the separate compilation system: the program editor and the linker/loader. The program editor compiles or recompiles procedures (supplied to it by a text editor, which is not discussed in this thesis) and updates the ODB appropriately. The linker brings together all the procedures of the program, checks their consistency, does fixups if possible, and produces executable machine code. The loader performs all of the functions of the linker and then executes the machine code. The loader is discussed in this chapter; the linker merely performs a subset of its function.

2.5.1 THE OBJECT DATA BASE

The object data base must be capable of primary-key and secondary-key retrieval, where there may be multiple occurrences of the same secondary key. It must be possible to concatenate these data bases, so that the ones closest to the front are searched first. Thus, a user may have his own library which takes precedence over that of the system.

Each procedure is stored as an entity in this data base. The secondary key of the entity is the unqualified name of the procedure. The primary key is a unique key generated when the entity is defined for the first time. Each such object module contains the following

information:

1. the name of the object module containing this procedure's father. The father is that procedure which immediately contains this one.
2. a timestamp. (TS)
3. the time stamp of the father under which this procedure was compiled. (FTS)
4. the symbol table entries corresponding to the local declarations of this module.
5. a list of procedures called by this module, and the addresses within the object code of each of the calls. A procedure passed by this procedure as an actual parameter will be in this list; formal parameters will not.
6. the size of local storage for the procedure
7. a list of the addresses in the object code of all variable references. Alternatively, object code could be generated so that variable references could be identified by context.
8. the object code for the procedure.
9. backwards compatibility records to be described in section 2.5.5. (CR)

2.5.2 THE LOADING ALGORITHM

The algorithm by which a program is loaded and subsequently executed from the object data base is now presented. This algorithm ignores

compatibility criteria which will be discussed later.

1. Read the object module corresponding to the mainline program.
2. Bring in all object modules which this procedure references and which not already present.
3. Repeat 1,2,3 (recursively) for every object module brought in by 2.
4. Resolve all interprocedural references by filling in the appropriate calls with the address of the module called. If all the necessary modules cannot be loaded into memory at one time, we can place the address of a dynamic loader into the call address fields. When this dynamic loader is called it can load modules into memory using an appropriate paging algorithm.
5. Call the mainline procedure.

This algorithm traverses the call graph of the program, loading all procedures on that graph. Note that, in step 3, only the procedures called by procedures which were not previously loaded need themselves be loaded; if the calling procedure were already loaded, then steps 1,2, and 3 would have already been executed for the called procedures when the calling procedure was loaded. This same check prevents infinite recursion if the call graph has a loop in it caused by recursive procedure calls. Only routines which are actually called are loaded by this algorithm.

2.5.3 STORAGE ALLOCATION CONSTRAINTS ON THE COMPILER

In order to accommodate the program editor, certain constraints on the object code generated by the compiler are necessary. A declaration of a variable of any type must cause a fixed amount of storage to be allocated on the stack. This is a very common approach to compilation. Local variables of unknown or variable length are handled by storing a pointer on the stack and using a level of indirection whenever referencing them. Thus, all local variables are addressed as a displacement from the beginning of the local storage area for the procedure. A given procedure can reference variables in the n scopes which contain it (n is small). The set of n addresses of the local areas corresponding to these scopes is called the display. All variable references within the object code consist of a pair of integers b, d where b is the index into the display of the base address, and d is the displacement from this address at which the variable to be referenced will be found.

2.5.4 RAMIFICATIONS OF PROGRAM MODIFICATION

When a program is modified using the program editor, the editor must locate the corresponding object module in the object data base. The programmer may have specified an incompletely qualified name. Using secondary key access all modules with the same unqualified name can be found. By following the father pointers of these modules, the editor can eliminate all of the incorrect choices.

Instead of building the symbol table from the program, the editor reads the symbol table segments from the object modules for the procedure and all of its ancestors.

When the declarations are processed, each symbol is looked up in the existing symbol table. If the symbol exists and the types are the same, no action is taken as no change has been made. If the types are different, the declaration is recognised as a modification to the old one. If the symbol is not found, the declaration is recognised as an insertion. After all of the new declarations have been processed, any symbol table entries which have not been referenced are recognised as deletions.

When a declaration is added to a procedure, it is added to the symbol table and its storage is allocated at the end of the local area. The length of the local area is updated appropriately. If the type of a variable is changed so as to change its length, or a declaration is deleted, the locations of local variables must be reassigned.

If a type declaration is changed, all variables of that type are affected as if their type was changed directly. Furthermore, any subprocedure which references that type will be affected.

If a procedure is inserted, a unique name for the procedure is created, and so is an object module in the object data base. If a procedure is deleted, its object module is also deleted.

Replacing executable code causes no problem; the new code is compiled using the symbol table from the appropriate object modules, and the object code portion of the appropriate object module is replaced.

The symbol-table entry for a procedure header belongs to the symbol table for the father of that procedure. Thus, any change to the header

is treated by the editor as a change to the father as well as to the procedure.

2.5.5 COMPATIBILITY INFORMATION IN OBJECT MODULES

If we have done merely insertions which do not obscure more global names, and executable code modifications, we have no fear of affecting lower-level routines. This will be the situation for most modifications. Such modifications are called compatible changes. All other changes are called incompatible because they may affect dependent procedures. If we have a compatible modification and the "father time stamp" (FTS) field of the module being changed is equal to the "time stamp" (TS) field of the father, then the editor can merely update the data fields in the object module. If the FTS does not correspond to the father's TS, or an incompatible change was made, we must create a new-generation module.

Creation of a new-generation module involves writing the module with a new TS and FTS. Furthermore, a compatibility record (CR) is appended to the module. The CR contains the old TS and the old FTS.

If an incompatible change was made, subprograms to the module changed may be affected. If there were deletions or modifications to procedures, the CR contains a list of the unique names which were generated for these procedures. The loader ensures that no module accessing these procedures will be run. If there were deletions or changes to variable declarations, the CR contains a list of b,d addresses which are no longer valid. If a reallocation of storage was

done for any reason, the mapping from the old b,d addresses to new b,d addresses is included in the CR. If a type definition was modified so as to change the length of the internal representation, then the name of that type would be included in the CR, so that the loader could re-map local storage of subprograms as necessary.

There is a special case of insertion and deletion. If an insertion obscures a more global item, and the type of the inserted declaration is different from that of the more global, the b,d (or the unique name in the case of a procedure) of the obscured symbol is added to the invalid list in the CR. If the two types match, a mapping from the global b,d to the new b,d is added to the CR. The situation of deleting a declaration to uncover a more global one is analagous; if the more global declaration does not have the same type, the name or b,d is added to the invalid list in the CR; if the types are the same, a mapping from the local b,d to the global b,d is provided. This special case is useful for the addition and deletion of local replacements which override library or system procedures and variables. For example, the programmer may wish to intercept all calls to the system routine, PRINT, in order to format his output differently. This end could be met by adding a more local version of PRINT with exactly the same parameter types.

In summary, the CR contains the following information:

1. the TS of the old object module
2. the FTS of the old module
3. a list of b,d addresses which may no longer be used
4. a list of unique procedure names which may no longer be called
5. a list types which have been changed
6. a list of old b,d to new b,d mappings.
7. a list of mappings of old unique names to new unique names of procedures.

2.5.6 COMPATIBILITY MAINTENANCE VIA THE LOADER

We can ensure compatibility of modules by enhancing the loading algorithm. For every module which is loaded, we check the FTS field against the TS field of the father. If all of the time stamp fields match, the program can be executed without further concern. For every module, M, whose father is no longer compatible, there must be a compatibility record, associated with the father, whose TS is equal to M's FTS. We then check the list of procedures which are no longer valid against the list of procedures called. If there is any intersection, we terminate the loading and inform the user that his program must be recompiled because of an obsolete reference to the procedure in the invalid list. Similarly, we check the list of b,d addresses in the invalid list with the list of b,d addresses used in the procedure being loaded. If there is no intersection, the module can be modified to be compatible with its father by applying the b,d mappings. The

modification can then be saved back into the object data base to create a new-generation module which is compatible. Or, if write access to the object data base is not available or we wish to maintain the older version of the module, we can apply the modifications to the in-memory copy only. In this instance, the compatibility modifications must be re-applied every time the module is loaded.

If a compatibility record is applied to a module being loaded, then all newer compatibility records associated with the father must also be applied. Furthermore, if the FTS in the compatibility record does not correspond to the father, the necessary compatibility records from the father's module must first be applied. Once this recursive process is complete, the module can be executed.

2.6 AN EXAMPLE OF THE USE OF THE PROGRAM EDITOR

Suppose we have a hierarchy of procedures with C a subprogram of B, and B a subprogram of A. The object data base after the initial compilation looks like this.

name	FP	TS	FTS
A	null	1	-
B	A	1	1
C	B	1	1

At time 3, we make a compatible change to B. B is updated but none of the above information is changed.

At time 4, we recompile A making incompatible changes. We build a compatibility record and arrive at the following structure:

| --- original module --- | --- CR --- |

name	TS	FTS	TS	FTS
A	4	-	1	-
B	1	1		
C	1	1		

At time 5, we make a compatible modification to B. We detect that B's FTS is not equal to A's TS. We therefore apply A's CR to B. A compatibility record is added to B and B's TS and FTS are brought up to date.

| --- original module --- | --- CR --- |

name	TS	FTS	TS	FTS
A	4	-	1	-
B	5	4	1	1
C	1	1		

At time 6, we make an incompatible modification to A. We write another compatibility record and arrive at the following:

| --- original module --- | --- CR --- | --- CR --- |

name	TS	FTS	TS	FTS	TS	FTS
A	6	-	1	-	4	-
B	5	4	1	1		
C	1	1				

At time 7, we attempt to run the program. We check the modules for compatibility in top-down order. B assumes version 4 of A. We must therefore apply A's CR for TS=4. This brings B into line with A, version 6. We now have:

--- original module --- --- CR --- --- CR ---						
name	TS	FTS	TS	FTS	TS	FTS
A	6	-	1	-	4	-
B	7	6	1	1	5	4
C	1	1				

C assumes version 1 of B. We must eventually apply B's CRs to bring C up to B level 7. B's CR for bringing TS=1 up to TS=5 requires A level 4. Since B level 1 assumed A level 1, we must apply A's CR for TS=1 in order to bring C into line with A level 4. Then we apply B's CR for TS=1 to bring C into line with B level 5. Similarly, in order to apply B's CR for TS=5 we must first bring C up to A level 6 by applying A's CR for TS=4. Finally we have:

--- original module --- --- CR --- --- CR ---						
name	TS	FTS	TS	FTS	TS	FTS
A	6	-	1	-	4	-
B	7	6	1	1	5	4
C	7	7	1	1		

2.7 OTHER FEATURES OF THE PROGRAM EDITOR

2.7.1 OBSOLETE VERSIONS

The program editor described above supports a forward moving system. That is, all versions are ordered chronologically and only the most recent version is valid. It is possible, however, that the system is installed at many locations, and every location does not have the same version of the system.

Thus, when a program is being compiled on one system, the donor system, to be used on another, the receptor system, it is quite possible that the receptor system is obsolete, and therefore the donor system must be back-levelled in order to do the necessary compilation.

A back-levelled compilation may be requested by supplying an old time stamp to the program editor. If the editor needs to bring the system forward to this time, it does so by applying the appropriate CRs as described previously.

If, however, the system needs to be regressed to an earlier version, the inverses of the compatibility records must be applied. The algorithm to apply the inverse CRs in the correct order is much the same as the original algorithm.

The inverse CRs may be generated easily; the inverse of an insertion is a deletion, the inverse of a deletion is an insertion, and the mappings between b,d pairs are trivially inverted.

Thus, with the addition of inverse CRs to the object data base, any old configuration of the program may be reproduced.

2.7.2 MULTI-CONFIGURATION COMPILATIONS

When maintenance to a program becomes necessary, it is often necessary to make the same change to all older versions of the program as well as to the current version. In this section the possibility of making the same change to all or many versions of the program at the same time is discussed.

The most frequent change will be the recompilation of executable code in some procedure. It would be desirable for the same object code to be valid for distribution to all receptors, regardless of the level of each receptor's system. The programmer can request such a multi-level compilation by specifying a list of TSs along with the appropriate command to the program editor. Alternatively, the programmer may specify a range of times or may specify that all possible versions are to be supported.

In order to implement such a request, the program editor compiles the program in the context of his existing system. Then the appropriate inverse CRs are applied to bring the object code through the various versions. If there is an intersection between an invalid list and the corresponding reference list, the editor must stop and indicate that previous versions are not compatible.

If a b,d pair needs to be translated, then more than one version of the object code is produced, each version handling a subset of the desired configurations. If there is no intersection, the object code applies to all versions and can be marked with all the relevant time stamps indicating that the object code may be inserted into an object file at any of those levels.

2.7.3 MULTI-CONFIGURATION DECLARATION CHANGES

If a change to be effected on multiple versions of a program involves replacement of declaration parts, extra processing is required of the program editor. As mentioned earlier, the program editor compares the new declarations to the ones which exist in the object data base and

recognises additions, deletions, and changes to declarations. These elementary operations are effected and a new symbol table is produced.

This new symbol table will be appropriate for older versions of the program only if no declaration changes were made to the procedure in question since the older versions. It is also necessary that none of the declarations in the symbol table are affected by more global CRs. If these two conditions are met, the same symbol table can be used for the older versions as well as the current one.

Rather than treating a declaration change as the replacement of all of the declarations in a procedure, each elementary operation recognised by the editor can be considered individually. The CRs may be extended by adding the necessary information to convert a symbol table from one version to the next. In the case of additions, the CR would contain the new symbol table entry. In the case of a deletion, the CR would indicate such. In the case of a change, the CR would contain the new symbol table entry. From this information, all the abbreviated information previously in the CR can be deduced. Inverse CRs would contain similar information.

These CRs need not be applied in strict chronological order. For example, a system may be created at time 1. At time 2 some declarations are added. At time 3 some other declarations are added. A user running level 1 may want the later changes applied but not want to bring his entire system to level 2. The CR taking level 2 to level 3 may be applied to the user's system. He does not end up with level 3. However, he does benefit from this latest modification.

Clearly, identifying versions by TS alone is no longer adequate. Instead, versions must be identified by their genealogy. For example a level 3 system is called a 1-2-3 configuration whereas the user's system discussed is a 1-3 configuration. The CRs may be applied in any order provided that an attempt is never made to insert an existing declaration or to delete or change a non-existent declaration.

It is possible that the modifications at time 2 and 3 both affect the executable code for a procedure, P. In this case the code for P for version 1-3 and for version 1-2-3 would not be the same. Thus, a different version of P would have to be maintained by the programmer. There are two ways in which the versions of P may differ. If one version references declarations which are not available in another version, the program editor will enforce the distinction between versions. If there is merely a logical difference between the procedures, a general purpose version which works for all configurations can be written.

In any case, these conflicts occur only if the same declaration or block of code is involved in more than one modification. The probability of such interference is greatly reduced by the fact that these units are generally small. In a system with larger compilation units, the probability of interference is much higher as any two modifications which affect the same compilation unit will conflict.

2.7.4 SELECTABLE FEATURES

A selectable feature (SF) is an enhancement or modification to a program which may be selected or not depending on the requirements of an individual user. Such modifications are neither chronological nor hierarchical; they may be applied in any order or combination in order to generate a special-purpose version of the program. It is impossible to maintain each possible version separately; 10 SFs generate 1024 possible versions while 20 SFs generate 1,048,576. The program editor handles SF's in much the same way as the chronologically ordered modifications discussed in section 2.5. Each SF is given a name instead of a number. This name is found in the CRs instead of the TS. The appropriate combination of CRs is applied in order to generate a particular system. The program editor can determine whether or not SFs conflict and also minimize the number that do conflict. If a new SF conflicts with existing configurations, the programmer must decide whether to maintain many different versions of the procedures in contention, to write a general purpose procedure, to effect his SF in a different manner, or to disallow certain combinations of SF's.

2.8 CONCLUSIONS

This chapter has presented a program modification system based on block structured languages. The unit of compilation has been chosen to be the procedure. A program editor has been presented which will allow separate compilation of any procedure without extra programming or foresight as to which procedures would require recompilation. An object data base was presented which stores the compiled version of the program

and enough control information so that program fragments can be recompiled, and so that procedures can be recompiled without necessitating recompilation of subordinate procedures. Subordinate procedures which are invalidated are detected when they are loaded. Thus, the need for a massive compilation effort when a core declaration or procedure is changed is obviated. Extensions to the object data base and program editor which allow many configurations of a program to be maintained concurrently were presented.

The separate compilation system would be of little use if its use was just as expensive as recompilation. Expense involves many criteria including CPU time, computer memory, and disk storage. In addition, CPU time and memory usage are divided into that used at compile time and that used at run time.

At run time, normally the only expense of the SCS is that a time stamp must be checked every time a procedure is loaded. Unless a change has been made and the program has not yet been thoroughly tested, the time stamps will always match. If it is essential that no additional overhead be incurred, the program can be pre-linked before execution.

Since a large number of changes are compatible, many recompilations of subordinate programs are prevented. When incompatible changes are made, however, the time stamp checking will cause compatibility analysis to be performed on all subordinate programs. Only if this analysis fails does recompilation need to be done. The compatibility analysis is cheaper than recompilation; it consists merely of merging the father's and son's compatibility lists, looking for matches. This process

requires no symbol table, very little memory, and the time required is linear with the size of the CRs. In addition, it could be implemented on the target machine, which may not have a compiler. On the other hand, recompilation requires that the source for the program be read in, that all global symbol tables be read in (and searched), and that the syntactic and semantic analysis be done. Thus recompilation uses more storage and is more complex than performing the compatibility analysis. Since a single change may cause many procedures to be checked, the saving is very significant.

The SCS uses some disk storage. The symbol tables stored in the ODB would be required by any separate compilation system. The disk overhead of the compatibility analysis is the extra storage required to cross reference each non-local b,d address, and the storage required to store the CRs. A cross-reference entry need be no larger than the b,d address itself. Addresses themselves make up a fraction of the object code. Of these addresses only a fraction will be global. It is the author's estimate that the cross-reference information will not increase the size of the object code by more than 10%. This estimate is based on one third of the object code being addresses and one third of these addresses being global. The storage required for the CRs depends entirely on the amount of maintenance applied to the system. Because of the CRs, the source for the program may be stored off line, freeing a considerable amount of storage.

Chapter III

A VIEW OF PARAMETERS AND SCOPE

3.1 INTRODUCTION

Many different concepts of parameter passing and scoping rules have been developed. Three, in particular, have been widely accepted: the Algol system of static scoping [NAU62], the system of textual substitution as with MACRO parameters [IBM75 IBM76], and the system of dynamic scoping as in APL [IVE62 IBM78] and LISP [WEI67]. Some are more powerful and some are more clear in some instances. Some are more "compilable" than others. In this chapter we attempt to unify the three parameter mechanisms to provide the most power while still maintaining the security necessary for compilability. New language features such as user-defined types are discussed (in the context of this parameter mechanism).

3.2 PROCEDURES AND MACROS

In all programming, certain processes are performed many times. If the code necessary to perform a process is replicated as necessary by the programmer, two problems arise. The first problem is that manually repeating the code is tedious and error prone. The second is that this duplicated code causes program size to swell. Two independent mechanisms have been developed to counter the two problems: "macro" or in-line procedures, and remote procedures.

3.3 MACROS

Macros provide a mechanism to avoid the tedium of repeated code; one merely needs to copy the pertinent code wherever it is required by calling the macro. Once the macro has been debugged there is no possibility of a clerical error in copying. However the problem of program size is not alleviated.

3.4 REMOTE PROCEDURES

A programmer can also compile the repeated statements, and, in place of each instance of the statements, he can place a branch to this remote procedure. He must tell the procedure where to return control after the statements are executed. This information which is transmitted to the procedure is in fact a parameter.

Normally this return address is placed in global storage at a fixed location such as a particular register and the procedure finds the information there by convention.

The problem of program size is reduced but the programmer has had to write code to save the return address and branch to the routine. However, in a high level language, this parameter is invisible.

3.5 SCOPED LANGUAGES

In an unscoped language, where there can only be one instance of a variable with a given name, it is impossible to detect whether the compiler has generated remote or in-line code i.e. macros and procedures are functionally identical in this case.

Macros and procedures can be distinct within a scoped language, where a given identifier may have different meanings in different parts of a program.

Suppose we define a procedure, P , which references the non-local variable X . At a particular call site a different X is visible from that which was visible where P was defined. The X visible at the call site and at the definition site will be known as X_c and X_p respectively. Which is referenced within this call of P ?

In an Algol-like language X_p is referenced. This choice is justified using many criteria. The reference can be resolved when P is compiled, whereas the meaning of X_c is not known until the call is compiled. Furthermore, X_c is not necessarily the same for each call, meaning that a great deal of compilation would have to be done for each call, resulting in many different versions of P . Furthermore, the reader of the procedure does not know which X is referenced without examining every call site. Because Algol-like languages avoid this decision making at run time, they are said to be statically scoped.

In APL, which is interpreted rather than compiled, X_c is used. This approach is possible because each symbol is resolved at execution time; when P executes it looks up the current version of X , which is X_c for the particular call being executed. Such a language is said to be dynamically scoped.

Macros usually reference X_c ; the source code is merely inserted at the call site and X is resolved later by the compiler. Thus

parameterless macros and parameterless procedures may be distinguished in a scoped language.

In-line procedures, on the other hand, usually obey the same scoping rules as remote procedures. Thus Algol-like procedures, even in-line procedures, lack "adaptability" to their calling environment. Call-site versus definition-site resolution is discussed further in section 3.8.1.

3.6 PARAMETERS

Many times the code which is repeated within a program is not identical in all cases but is similar. The most common instance of this situation is one in which the same operation is applied to different data. For this reason procedures are allowed to have parameters, which are references which are not resolved until the call site. Implicit in this mechanism is the ability for the parameter's name to be different from the actual variable which it references.

Parameters can easily be incorporated in a statically scoped language for three major reasons:

1. the number of parameters is relatively small and so any extra time spent referencing these variables is insignificant,
2. the names of the parameters are not relevant outside the procedure; the call merely lists variables which are matched in a one to one correspondence with the parameters,

3. the type of each parameter is specified by the procedure and must be matched by the caller; this typing makes it possible to compile the procedure, as all necessary declaration information is present at the procedure definition.

Macro facilities also incorporate parameters. These parameters permit the internal name of a global variable to be different from the actual name. Furthermore, many additional facilities can be incorporated if parameters are treated merely as textual substitutions. Names of procedures, types, or even sequences of code can be passed as parameters. The price of such generality is that no compilation or error checking can be done at macro-definition time.

3.7 MORE POWERFUL PARAMETERS

In some languages, more powerful parameters are found which introduce to statically scoped procedures some of the power of macros. Three such features will be discussed in this section, as they appear in existing languages and as they appear in L.

3.7.1 PARAMETERS OF TYPE PROCEDURE

PROCEDURE parameters are used within a routine which implements an abstract process which may have variations from one call to the next. For example, Simpson's rule is a well-defined process to integrate a function. A macro facility with parameters can produce a new variant for each function, F , to be integrated. However, in order to compile a procedure to effect Simpson's rule we require that F be passed as a procedure parameter.

Languages such as FORTRAN and Pascal [WIR71] allow procedure parameters to be declared by stating only the type of the result. This type of declaration does not adequately describe the parameters to the compiler. The compiler is unable to verify the type and number of parameters to this procedure parameter, or to do any conversions which are necessary. In order to perform this matching, a substantial fraction of the compilation effort must be done at each call site.

Algol 68 and PL/I (as an option) regard the types in the parameter list to be part of the type of the procedure. The routine with a procedure parameter can be compiled in the same way as a procedure with variable parameters; only the address of the procedure is resolved at the call site.

Due to the design criterion of compilability, the latter approach is taken in L; a formal procedure parameter must be fully typed by specifying the types of all of its parameters and the type of its result. The corresponding actual procedure must match in these types. In order to match, the types must either be exactly the same as those specified on the formal parameter, or must be convertible to that type. If any conversions are required, a dummy procedure is passed which has parameters of the exact type required by the formal parameter, performs the necessary conversions and then calls the procedure supplied as the actual parameter.

3.7.2 PARAMETERS OF ARBITRARY PRECISION

In Pascal and other languages, array and subrange bounds must be specified on formal parameters. This requirement is consistent with the fact that such precisions are part of the type of the parameter. Nevertheless the requirement is restrictive; we cannot write a procedure to sum the elements of an integer vector of indeterminate size.

FORTRAN provides a facility called "execution time dimensioning"; if an integer is passed as a parameter to a procedure, another parameter may use that integer as a bound. The procedure can be compiled by making the object code sufficiently general to operate on arrays of any length.

PL/I provides an important simplification to FORTRAN's execution time dimensioning. Instead of requiring that the bound be passed as an additional parameter, PL/I derives the array bound from the array which is passed. If X has 100 elements, we can call SUM(X), which is more convenient and more clear than FORTRAN'S SUM(X,100). Alternatively, if the length of the array is significant, a constant bound may be specified in the formal parameter definition. Algol 68 provides a facility similar to that of PL/I except that a constant bound may not be specified for an array parameter.

The following procedure headers illustrate the three modes of dimensioning arrays.

```
PROC SUM( X: [100] INT) RETURNS R: INT
```

The above header defines SUM, a procedure with an integer array parameter, X, and returning an integer, R. The length of the array is fixed at 100.



```
PROC SUM(X: [N] INT, N: INT) RETURNS R: INT
```

This header represents FORTRAN style execution time dimensioning. X is an array of INT whose bound is the INT parameter, N.

```
PROC SUM(X: [*] INT) RETURNS R: INT
```

In this header, the asterisk indicates that the bound of X is unspecified; an array of any size may be passed as an actual parameter. In order to determine what the bound of the actual parameter was the writer of SUM must use a magic built-in function such as HBOUND, UPB, LENGTH, etc.

Such functions are magic because they could not be written as procedures in the language. In fact, they must be built into the compiler. Ada's attribute syntax ('FIRST and 'LAST) more clearly indicates that such a built-in language feature is being used. But the built-in approach is not general; there must be a built-in attribute for every variable part of every type. These built-in attributes increase the complexity of the language definition, as well as introducing a vast number of keywords to the language. Most importantly, such attributes are not available for user-defined types, as described in section 3.10.

In L, parameters of arbitrary precision may be passed using the implicit parameter mechanism:

```
PROC SUM(X: [N:INT] INT ) RETURNS R: INT .
```

This syntax makes N an implicit parameter. An implicit parameter is a parameter which appears as a parameter within the called procedure, but is passed automatically at the call site and is therefore invisible to the caller. N may be used within SUM to query the length of X but the

caller has no knowledge of N. Such implicit parameters achieve the same generality as PL/I or Algol 68, but obviate the need for the built-in magic functions, allowing any parameter of a user-defined type to be automatically passed.

The syntax of a formal implicit parameter is the same as that of an ordinary formal parameter: a name followed by a colon and then a type. Any such parameter which appears as part of the type of another parameter is an implicit parameter and no corresponding actual parameter is supplied by the caller.

3.7.3 PARAMETERS OF TYPE TYPE

It is desirable to further generalize SUM so that it will operate not only on an array of integers, but also on arrays of real, complex, or any numeric type. In order to accomplish this generalization, the type of the elements of the array must be a parameter.

Consider the following procedure headers.

```
PROC SUM(X: [100] T, T: TYPE) RETURNS R: T
```

We have used an array of fixed length for simplicity. X is an array whose elements are of type T. SUM returns R, a variable of type T. T itself is an extra parameter to SUM. Note that, as is the case with FORTRAN's execution time dimensioning, T is an unnecessary parameter. The following header makes T an implicit parameter, as it would appear in L.

```
PROC SUM(X: [100] T:TYPE) RETURNS R: T
```

Allowing X to be of arbitrary size gives us:

```
PROC SUM(X: [N:INT] T:TYPE) RETURNS R: T
```

The code of SUM is now presented and the problems of compiling it are considered.

```

1. PROC SUM(X: [N:INT] T: TYPE) RETURNS R: T
2.     R := 0
3.     FOR I FROM 1 TO N DO
4.         R := R + X[I]
5.     OD
6. END

```

First consider statement 2. This statement requires the assignment of the integer literal 0 to the variable R of type T. Such a conversion may not exist, and certainly cannot be found at compile time. A possible solution is to require the appropriate zero to be passed as a parameter. The header for SUM would then become:

```
PROC SUM(X:[N:INT] T:TYPE, ZERO:T) RETURNS R:T .
```

Statement 2 would be replaced by:

```
R := ZERO .
```

Now consider statement 4. This statement contains three operations: subscripting, assignment, and "+". The subscript calculation requires the length of an object of type T, which is not known at compile time. However, code can be generated to make use of this length when it is contained in the information passed as the actual parameter to T. Similarly, the assignment operation can move a number of bytes which is specified by the actual parameter to T.

Compiling the "+" operation is more difficult. Since the operator is overloaded (as described in chapter 4), we do not have enough type information to select the appropriate "+". Indeed, if a user-defined T is used, neither the type T nor the appropriate "+" may exist within

scopes known to SUM. This problem has been approached differently in EL1 [WEG74], CLU [LIS79], and Ada.

In EL1, the programmer would have to specify a list of all possible types which T could take on. Object code would be generated for all possibilities and the selection done at run time. Such a constraint greatly reduces the generality of the procedure SUM.

In CLU, operations and types are combined in CLUSTERS. Rather than passing a type as a parameter, a CLUSTER containing the type and operation "+" would have to be passed to SUM. The formal parameter T would indicate that any cluster which is an actual parameter must contain the operation "+". This overall approach of using clusters would preclude many of the features of L and therefore this type passing mechanism is not used.

In Ada, type parameters are permitted only for macro-like GENERIC units. Such parameters are supplied at the instantiation site, rather than the call site. An instantiation is an extra statement in which parameters to a generic unit are supplied, yielding an instance of the generic unit which is given a new name and is called in the normal way. Any operations or functions which refers to a generic type parameter in one of its parameters or in its result must itself be a generic parameter and be supplied at the instantiation site.

The definition of SUM in Ada is:

```

GENERIC
  TYPE T IS PRIVATE
  WITH FUNCTION '+'(A:T, B:T) RETURN T
  ZERO: T
  SIZE: INTEGER;
FUNCTION SUM(X: (1..SIZE) OF T) RETURN T IS
  R: T;
BEGIN
  R := ZERO;
  FOR I IN 1..SIZE LOOP
    R := R + X(I);
  END LOOP
END SUM

```

The necessary instantiations:

```

FUNCTION INTSUM99 IS NEW SUM(INTEGER,"+",0,99);
FUNCTION MPSUM50 IS NEW SUM(MULTI_PRECISE,MPADD,MPINT(0),50);

```

Typical declarations and calls:

```

MPARRAY: (1..50) OF MULTI_PRECISE;
INTARRAY: (1..99) OF INTEGER;
I: INT;
MP: MULTI_PRECISE;

```

Typical calls:

```

I := INTSUM99(INTARRAY);
MP := MPSUM50(MPARRAY);

```

In L, any procedure may have implicit or explicit TYPE parameters. Any procedure which references a TYPE parameter in one of its parameters or in its result must have that procedure as a parameter. This rule is the same as that for ADA's generic parameters. However, in L, all parameters are supplied at the call site; there is no concept of instantiation.

A definition of SUM written in L is:

```

PROC SUM(X:[N:INT] T:TYPE, ZERO:T,
        "+" : PROC(T,T) RETURNS T) RETURNS T
  R := ZERO
  FOR I FROM 1 TO N DO
    R := R + X[I]
  END SUM

```

```

VAR I: INT
VAR MP: MULTI_PRECISE
VAR INTARRAY: [1..50] INT
VAR MPARRAY: [1..99] MULTI_PRECISE

```

```

I := SUM(INTARRAY,0,"+")
MP := SUM(MPARRAY,MPINT(0),MPADD)

```

The L version avoids entirely the complexity of instantiation. This is not the same as automatic instantiation as described in [ICH79a BER78]. Instead, we define the procedure, SUM, which sums an array, and then call it to sum various arrays.

The majority of calls to SUM will, in fact, use 0 and "+" as the second and third parameters. In fact, the specification of SUM says nothing of these parameters; SUM should merely sum an array. However, they cannot be implicit parameters. Instead, they can be supplied using a general default parameter mechanism which allows symbols to be resolved at the call site instead of the definition site.

Using the default mechanism described in the next section, we can call SUM in the simplest possible way:

```
I := SUM(INTARRAY) .
```

We will have defined a compilable procedure which sums an array, which is the abstraction we wished to implement.

3.8 DEFAULT PARAMETERS

In L, as in Ada, it is possible to specify default values for parameters. In L, defaults may be specified for parameters of type PROCEDURE or TYPE as well. parameter is omitted, this default value will be used. For example, we may write a procedure, LOG, which calculates the logarithm of a number. Unfortunately, the term "logarithm" is ambiguous; it could mean natural logarithm, common logarithm, or logarithm to some arbitrary base. We define here a procedure which accepts two parameters: the number whose logarithm is to be taken, and the base to which the logarithm is to be taken. However, if the base is not specified, the value e (2.718 ...) is used resulting in the natural logarithm being calculated.

```
PROC LOG(X:REAL, B:REAL DEFAULT(2.718)) RETURNS R:REAL
```

```
END LOG
```

Given the above definition,

```
A := LOG(X)
```

and

```
A := LOG(X,2.718)
```

would have exactly the same effect. However, if the the common logarithm were desired, the following call could be used.

```
A := LOG(X,10)
```

3.8.1 DEFINITION SITE AND CALL SITE RESOLUTION

A default value may also be a symbol or expression. In either case, the value is evaluated when the call is executed. However, any symbols appearing in the DEFAULT clause are resolved at the definition site. For example, we may have:

```
CONST E: REAL VALUE(2.718)
PROC LOG(X:REAL, B:REAL DEFAULT(E)) RETURNS R: REAL
.
.
.
END LOG

BEGIN
  VAR E: REAL INIT(123.456)
  A := LOG(C)
END
```

The statement

```
A := LOG(C)
```

is equivalent to

```
A := LOG(C,2.718)
```

not

```
A := LOG(C,123.456) .
```

Even though a different E is visible at the call site, the E which was visible when LOG was defined is used.

In this case, the proper E was referenced. Consider, however, a slightly different definition of LOG.

```

CONST LOGBASE: REAL VALUE(2.718)
PROC LOG(X:REAL, B: REAL DEFAULT(%LOGBASE)) RETURNS R:REAL

```

```

END LOG

```

```

BEGIN
  CONST LOGBASE: REAL VALUE(10)
  A := LOG(C)
END

```

In this case, the statement

```
A := LOG(C)
```

is equivalent to

```
A := LOG(C,LOGBASE)
```

or

```
A := LOG(C,10)
```

not

```
A := LOG(C,2.718) .
```

The difference is that the % preceding LOGBASE in the definition indicates that the resolution of this symbol is to be deferred until the call site. Only a single symbol may be so deferred, not an expression, as the symbol itself is part of the specification of the procedure and therefore must reside in the symbol table. Since the parameter is fully typed, this deferral does not impair compilation of LOG. This deferral allows the programmer to use dynamic resolution for any symbol provided that he identifies and types that variable via a parameter declaration. Thus, much of the power of macros and dynamic scoping can be had when desired.

LOGBASE is an example of an important facility; the ability of a programmer to set up his own environment. By one declaration, the programmer specifies that either common or natural logarithms are used within a particular scope. Other languages, such as APL and Algol-W, have used global variables to alter the environment within which a program executes. There are two problems with this approach: it is the programmer's responsibility to reset these variables so as not to interfere with other programs which assume the normal environment, and the normal environment must be restored before any library routines are called. The approach of deferred resolution ensures that the environment is affected only for code which is statically contained within the scope of the declaration.

3.8.2 FIELD SELECTOR PARAMETERS

Further generalization of procedures may be effected by the introduction of parameters which select fields from structures. A structure is a type which is composed of components or fields, each of which has its own type. Each field is identified by name. Such names are called FIELD SELECTORS.

The following declaration declares DATE to be a structure with fields whose selectors are YEAR, MONTH, and DAY. The types of the fields are INT, CHAR(3), and INT respectively.

```
VARIABLE DATE: STRUCT(YEAR: INT, MONTH: CHAR(3), DAY: INT)
```

We can easily write a procedure which accepts DATE as a parameter:

```
PROC NEXTDAY(DATE:STRUCT(YEAR:INT, MONTH:CHAR(3), DAY:INT))
```

Furthermore, the types of the fields of D may be implied parameters:

```
PROC NEXTDAY( DATE:STRUCT( YEAR:Y:TYPE, MONTH:M:TYPE, DAY:D:TYPE))
```

Of course, any operators which operate on YEAR, MONTH, or DAY must be additional parameters.

It is also feasible to treat YEAR, MONTH, and DAY as implicit parameters. The field selectors in the parameter could be matched positionally with the actual field selectors. A more general approach is to treat any structures whose fields have the same type as being the same type. We have not pursued this approach; in L field names are not interchangeable. The equivalence rules for structures are as in Algol 68.

Within the context of L, it is possible to use the following header:

```
PROC NEWDAY(D: T: TYPE)
```

The field names have not been specified. In fact, D may be of any type. However, this specification is not very useful. Consider, for example, trying to compile the following statement within NEWDAY:

```
D.DAY := D.DAY + 1 (not valid within NEWDAY as defined above).
```

This statement adds 1 to the DAY field of D. Two problems arise when compiling this statement: the type of D.DAY is not known, and the position within D of the DAY field is not known.

Thus DAY must be a parameter; the information passed is the offset of DAY within a variable of type T. DAY must also be typed in the same way that procedure and variable parameters must be typed. In order to specify the parameter, DAY, we write:

```
PROC NEWDAY(D: T: TYPE, T.DAY: INT)
```

The declaration of the parameter DAY states that it is a field selector for structure T, where the type of the field is INT.

The ability to pass field names as parameters enables an important generalization of procedures. Of particular significance is the fact that a general set of routines to manipulate data structures can be written. For example, a sort routine can be written to which is passed the name of the field on which it is to sort an array, file, or linked list. A general linked list routine may also be written where the name of the link field is a parameter. This is useful because the same records may be linked into several different lists; the appropriate list is selected by passing the name of the link field as a parameter. LINK or NEXT could be chosen as the default name. As an example, a routine to insert into an ordered linked list is given.

```

PROC INS(VAR TOP,NEW: PTR(REC: TYPE), REC.LINK PTR(REC),
        REC.DATA: T: TYPE,
        "<": PROC(T,T) RETURNS BOOL DEFAULT(%"<"),
        NULL: PTR(REC) DEFAULT(%NULL))
  VARIABLE P,PREV: PTR(REC)
  FOR PREV,P FROM NULL,TOP REPEAT P,P.LINK
    WHILE P~=NULL AND P.DATA < NEW.DATA DO OD
  NEW.LINK := P
  IF PREV = NULL THEN TOP := NEW ELSE PREV.LINK := NEW
END

```

Pointers are parameterized by the type to which they point and automatic dereferencing is provided. A more complete discussion of pointers follows.

3.9 POINTER PARAMETERS

In L, the type of a pointer includes the type of storage to which the pointer refers. This approach has been used in Algol-W, Algol 68, and Pascal, but not in PL/I. The inclusion of this information enables all pointer references to be statically type-checked. However, it is often necessary to omit some of this typing information and have it maintained dynamically. In L, this facility is provided by means of a parameterized pointer.

A typical non-parameterized pointer declaration is

```
VAR P: PTR( [100] INT )
```

indicating that P is a pointer which refers to an array of integers of length 100. The storage for P will contain merely the address in storage being referenced. On the other hand, a pointer Q may be defined which can refer to an integer array of any length:

```
VAR Q: PTR( [N:INT] INT ) .
```

N is an implicit parameter to Q which, at any given time, will indicate the length of storage to which Q refers. The storage for Q will contain both an address and N. Some examples of valid and invalid uses of P and Q are presented.

```
VAR A: [100] INT
```

```
VAR B: [50] INT
```

```
P := A  -- valid, Algol 68 dereferencing rules are used
          therefore P gets address of A
```

```
Q := A  -- valid, address of A and 100 copied to Q
```

```
Q := P  -- valid, P and 100 copied to Q
```

```
Q := B  -- valid, address of A and 50 copied to Q
```

```
P := B  -- invalid, length does not match
```

P := Q -- invalid, length will not necessarily match

Thus, any information which may vary from one instance of storage to the next is stored within the pointer by way of an implicit parameter. This facility makes the maintenance of hidden dope vectors and descriptors unnecessary. Any variable length information must be stored with one level of indirection. However, this level of indirection may be made invisible using the user-defined type facility described in the next section.

3.10 TYPE DECLARATIONS AND THEIR PARAMETERS

L has two distinct forms of type declaration. One form creates a name which is merely a shorthand for another type; equivalence rules such as those used in Algol 68 apply. The other form creates an entirely distinct type; no operations, procedures, or literals are inherited from an existing type. This dichotomy avoids the compromises found in Pascal's name equivalence and Ada's derived types.

3.10.1 EQUIVALENT TYPES

The first form of type declaration is very similar to the MODE declaration of Algol 68. For example, the declaration of Q

```
VAR Q: [10,10] REAL
```

is identical to the declarations

```
TYPE M: [10,10] REAL
```

```
VAR Q: M .
```

If several variables of type [10,10] REAL are used, such a type declaration saves much tedious, error-prone repetition. In addition, if

the type of all of the variables needed to be changed to [20,20] REAL, such a change could be made by altering only the type declaration.

3.10.2 NEW TYPES

It has become apparent since the design of Algol 68 that a more powerful form of user-defined type is desirable; one which is not interchangeable with the original type (the original type is the type on the right side of the type declaration). For example, we may wish to define the new type "COUNTER". Variables of type COUNTER are intended to be incremented, decremented, initialized, and compared to zero. Thus it is natural that a COUNTER should be represented by an INTEGER.

The declaration of such a new, distinct type differs from the first form of declaration in that the keyword "NEW" is inserted before the original type:

```
TYPE COUNTER: NEW INT
```

None of the operations, literals, functions, or primitives such as field selection and subscripting are inherited from the original type. Values of the new type may not be assigned to variables of the original; values of the original type may not be assigned to variables of the new. Operator selection (discussed in chapter 4) regards the two types as being distinct.

It is necessary for the programmer to define the functions which operate on COUNTERS. The headers for these four functions are:

```
PROC UP(VAR C: COUNTER)
```

```
PROC DOWN(VAR C: COUNTER)
```

```
PROC ZERO RETURNS C: COUNTER
```

PROC ISZERO(C: COUNTER) RETURNS B: BOOL

UP and DOWN are functions which increment and decrement a counter, respectively. DOWN does nothing if the counter is already zero. ZERO returns a zero counter for initialization purposes. ISZERO tests a counter to determine whether or not its value is zero.

The following procedure is an incorrect version of UP:

```
PROC UP(VAR C: COUNTER)
  C := C + 1
END UP
```

This procedure is incorrect because the statement "C := C + 1" cannot be compiled as the operation "+" does not exist for the type COUNTER. It is necessary within the procedures which define the operations on the type COUNTER to be able to treat counters as if they were integers. For this reason, the language primitive 'REP is defined thus: any variable, X, of new type, T, whose original type is O shares its storage with a variable named X'REP whose type is O. If all of PL/I's types are considered to be represented as bit strings, then L's 'REP is identical to PL/I's UNSPEC.

The four procedures may be written using 'REP:

```
PROC UP(VAR C: COUNTER)
  C'REP := C'REP + 1
END UP
```

```
PROC DOWN(VAR C: COUNTER)
  IF NOT ISZERO(C) THEN C'REP := C'REP - 1
END DOWN
```

```
PROC ZERO RETURNS C: COUNTER
  C'REP := 0
END ZERO
```

```
PROC ISZERO(C: COUNTER) RETURNS B: BOOL
  IF C'REP = 0 THEN B := TRUE
  ELSE B := FALSE
END ISZERO
```

Note that DOWN refers to both C and C'REP; it is often necessary to access the representation of a variable in one instance and not in another.

This division is not possible in CLU, where accessing the representation of a variable is tied to the parameter mechanism. The type of a formal parameter in CLU may be specified in the normal way, in which case the representation of the parameter may not be referenced. On the other hand, the type of a formal parameter may be specified as CVT, in which case all references to that parameter will access its representation.

It is possible to restrict the scope within which 'REP may be used for a particular type, thus allowing information hiding. This facility is provided by the MODULE, as discussed in section 3.12.

3.10.3 INITIAL VALUES ON TYPE DECLARATIONS

It is essential that the programmer be able to supply statements which will initialize new storage of a particular type. For example, if we define a variable of type COUNTER, we want it to be initialized to zero. If a queue is represented as a linked list, the head and tail pointers must be set to NULL. Such a facility is provided by the INIT clause on a TYPE declaration:

```
TYPE COUNTER: NEW INT INIT(ZERO) .
```

The expression in the INIT clause is evaluated whenever storage of type COUNTER is created. Any symbols in the INIT clause are resolved at the site of the type declaration; thus the type definer can be sure that his

initialization routine will be called, even if the user has a more local procedure with the same name. If the symbol in the INIT clause is to be resolved at the site of storage allocation, the deferred default parameter mechanism discussed in the next section must be used.

Suppose, however, that we did not wish to provide the function ZERO for counters; the init clause could not be written. For this reason, an inverse to the 'REP facility is provided in L. The name of this inverse is 'PER (REP backwards). 'PER will treat a value of any type O which is the original type of new type T as if it were of type T. The INIT clause may be written using 'PER:

```
TYPE COUNTER: NEW INT INIT(0'PER) .
```

Note that 'PER is overloaded; the type yielded by 'PER must be determined from context. This determination from context is discussed in chapter 4. It should be noted that PL/I lacks an inverse to UNSPEC, frequently causing inconvenience to the programmer.

3.11 TYPE GENERATORS

A parameterized type declaration is a type generator; a type is created only when actual parameters are supplied to the type generator. Built-in type generators are ARRAY and STRUCT. A programmer may create his own type generators by adding a formal parameter list to a type declaration. For example the declaration

```
TYPE MATRIX(N:INT): [N,N] INT
```

creates the type generator MATRIX which has one parameter, N, indicating the size of a square matrix. Whenever a variable is declared, a type (not a type generator) must be specified and therefore all parameters to MATRIX (or any type generator) must be supplied.

```
VAR X,Y,Z: MATRIX(100)
```

Formal parameters to procedures may be fully typed, or a type generator may be used as the type of the parameter. If a type generator is used, the parameters supplied to the type generator to form the type of the corresponding actual parameter may be determined using the implicit parameter mechanism. An example of each using the built-in type generator, ARRAY, has already been given. An example is now given using MATRIX.

The type of the parameter is fully specified in this case:

```
PROC INVERSE(M: MATRIX(100)) RETURNS I: MATRIX(100) .
```

But the more general INVERSE is written thus:

```
PROC INVERSE(M: MATRIX(S:INT)) RETURNS I: MATRIX(S) .
```

Note that only the size of M is derived using the implicit parameter mechanism. The size of I is then determined. The implicit parameter facility is thus more general than as originally presented.

Parameters to type generators may be of type TYPE. For example, a linked list is composed of elements, each of which contains a pointer and some data. A type generator to create such elements is

```
TYPE LISTELEM(T:TYPE): STRUCT(LINK: PTR(LISTELEM(T)), DATA: T) .
```

In order to set up head and tail pointers for a linked list of integers we need merely declare:

```
VAR HEAD,TAIL: PTR(LISTELEM(INT)) .
```

Type generators may also have implicit parameters. For example, a type generator which is similar to PL/I's LIKE can be defined.

```
TYPE LIKE(X: T: TYPE): T
```

```
VAR A: LIKE(B)
```

This pair of declarations creates a variable A with the same type as another variable B.

As may parameters to procedures, parameters to type generators may have defaults. For example, we may wish to make the default size of a MATRIX 3 x 3. This can be accomplished thus:

```
TYPE MATRIX(N: INT DEFAULT(3)): [N,N] REAL .
```

Normally, symbols in default expressions are resolved at the type definition site, e.g.

```
CONST THREE: INT VALUE(3)
TYPE MATRIX(N: INT DEFAULT(THREE)): [N,N] REAL
BEGIN
  VAR THREE: INT INIT(165)
  VAR X: MATRIX
END
```

In the above example, the size of X is 3 x 3.

On the other hand, resolution of symbols may be deferred to the storage allocation site by using a %.

```
CONST NUMBER_OF_DIMENSIONS: INT VALUE(3)
TYPE MATRIX(N: INT DEFAULT(%NUMBER_OF_DIMENSIONS)): [N,N] REAL
BEGIN
  CONST NUMBER_OF_DIMENSIONS: INT VALUE(4)
  VAR X: MATRIX
END
```

In this case, the default size of a MATRIX is the currently visible NUMBER_OF_DIMENSIONS, 4.

3.12 MODULES AND ENCAPSULATION

All of the primitives necessary to create distinct user-defined types have already been presented. However, in order to provide information hiding, the MODULE facility is included in L. A module is merely a set of declarations whose visibility is restricted; names defined within a module are not normally known outside. However, the duration of variables and the semantic meaning of the declarations is the same as if they were not enclosed in the module.

A name defined within a module may be made visible outside by prefixing its declaration with the keyword VISIBLE. If a variable is made visible, it may be used in the normal way, since its duration is no different within the module than outside. If a procedure is made visible, its formal parameter list and result are made visible in the normal way, but not its body. If an equivalent type is made visible, both the type, its parameters, and the original type are made available. If a new type is made visible, the corresponding 'REP and 'PER suffixes are not made available. If these suffixes are to be made available, the keyword NEW must be prefixed by VISIBLE. Normally, variables of a new type may be assigned to each other; the copy function of the representation is used (if there is one). However, this default copy facility may be overridden by supplying a user-defined function, ':='. If such a definition is present, that function will be used instead. If the definition is VISIBLE, the copy may be used outside the module; if the definition is not VISIBLE, no copying is permitted outside the module. If the standard copy is not overridden, it will be used inside and outside of the module.

The module may be used to create an abstract data type, whose internal representation is not known outside the module. As an example, the abstract type COUNTER is presented.

```

MODULE
  VISIBLE TYPE COUNTER: NEW INT INIT(0'PER)
  VISIBLE PROC UP(C:COUNTER)
    C'REP := C'REP + 1
  END UP
  VISIBLE PROC DOWN(C:COUNTER)
    IF NOT ISZERO(C) THEN C'REP := C'REP - 1
  END DOWN
  VISIBLE PROC ISZERO(C:COUNTER) RETURNS R:BOOL
    IF C'REP = 0 THEN R := TRUE
    ELSE R := FALSE
  END ISZERO
MEND

VAR X:COUNTER
CALL UP(X)
CALL DOWN(X)

X := 17      -- INVALID
X'REP := 17  -- INVALID
X := 17'PER  -- INVALID

```

Within the module, the representation of a counter is referenced using 'REP and 'PER. Outside the module, such references are not allowed. Had the declaration

```
VISIBLE TYPE COUNTER: VISIBLE NEW INT INIT(0'PER)
```

been used, such references would have been legal. Of course, it is always illegal to say `X := 17` as no conversion between integer literals and counters is provided. Note that, with the exception of the statements which are made invalid by the module, the semantic meaning of this program would be the same if all `MODULE`, `MEND`, and `VISIBLE` keywords were removed.

Although modules are not themselves parameterized, they may be used to realise parameterized abstract type generators. As an example, the

type generator QUEUE is presented. A queue is implemented as a linked list and therefore has no maximum size. However this implementation is invisible to the user of a queue.

```

MODULE
  VISIBLE TYPE QUEUE(T:TYPE): NEW PTR(LISTELEM(T)) INIT(NULL'PER)
  TYPE LISTELEM(Z:TYPE): STRUCT(LINK: PTR(LISTELEM(Z)), DATA: T)
  VISIBLE PROC PUSH(Q:QUEUE(T:TYPE),ELEM:T)
    Q'REP := GLOBAL LISTELEM(T) := (Q'REP,ELEM)'PER
  END PUSH
  VISIBLE PROC POP(Q:QUEUE(T:TYPE))
    Q'REP := Q'REP.LINK
  END POP
  VISIBLE PROC TOP(Q:QUEUE(T:TYPE)) RETURNS E:T
    E := Q'REP.DATA
  END TOP
  VISIBLE PROC EMPTY(Q:QUEUE(T:TYPE)) RETURNS R:BOOL
    IF Q'REP = NULL THEN R := TRUE
    ELSE R := FALSE
  END EMPTY
MEND

```

The declaration of the type LISTELEM is not visible as it is not relevant to the user of the QUEUE type generator. Similarly, making QUEUE a NEW type ensures that the user cannot depend on the fact that the QUEUE is implemented as a linked list.

The fact that parameters are associated with procedures and types, not modules, enables us to provide facilities which are not possible with parameterized modules. For example, within the module which defines QUEUE, we may define a procedure which copies a queue whose elements are of one type to a queue whose elements are of another, provided that the elements themselves can be copied.

```

VISIBLE PROC COPY(Q:QUEUE(QT:TYPE),
                  COPY: PROC(QT) RETURNS QT) RETURNS R:QUEUE(RT:TYPE)
VAR QP: PTR(LISTELEM(QT))
VAR RP: PTR(LISTELEM(RT))
IF Q'REP = NULL THEN
  R'REP := NULL
ELSE
  QP := Q'REP
  RP:=R'REP:=GLOBAL LISTELEM(RT):=(NULL, COPY(QP.DATA))
  WHILE QP.LINK ~= NULL DO
    RP.LINK := GLOBAL LISTELEM(RT) :=
      (NULL,COPY(QP.LINK.DATA))
    RP := RP.LINK
    QP := QP.LINK
  END COPY

```

Such a procedure cannot be written in CLU or Ada.

In CLU, COPY would have to belong to either the instance of the cluster defining QUEUE(QT) or to the instance defining QUEUE(RT). If it belonged to the former, it would not be able to reference the representation of the latter; if it belonged to the latter, it would not be able to reference the representation of the former. It is essential to reference both in order to implement COPY.

In Ada, QUEUE would have to be defined in a generic package. As in CLU, a single instance of this package could not reference both kinds of QUEUES. A procedure outside of the package defining QUEUE would not be able to reference either representation.

Another application of a module is to define two or more related types. An example of such related types are the types VECTOR and MATRIX which might be found in a linear algebra package:

```

MODULE
VISIBLE TYPE VECTOR(N:INT): NEW [N] REAL
VISIBLE TYPE MATRIX(N:INT,M:INT): NEW [N,M] REAL
VISIBLE PROC '*'(A,B: VECTOR(N:INT)) RETURNS R:REAL
.
.
.
END '*'
VISIBLE PROC '*' (A: MATRIX(M,N:INT), B: VECTOR(N))
      RETURNS VECTOR(N)
.
.
.
END '*'
VISIBLE PROC '*' (A: VECTOR(M:INT), B: MATRIX(M,N:INT))
      RETURNS VECTOR(M)
.
.
.
END '*'
VISIBLE PROC '*' (A: MATRIX(L:INT,M:INT), B: MATRIX(M,N:INT))
      RETURNS R: MATRIX(L,N)
.
.
.
END "*"
MEND

```

3.13 DURATION OF VARIABLES

The storage for an ordinary local variable exists while the procedure in which it is declared is active. It is frequently desirable that the duration of a variable be longer than the duration of the procedure to which it belongs (its visibility).

The classic application of such a variable is in a pseudo-random number generator. It is necessary for the generator to remember a seed value from one call to the next.

Algol 68 and Pascal support no such concept. Algol 60 and PL/I provide a crude facility in OWN and STATIC variables respectively. The

storage for these variables persists for the duration of the main program. The pseudo-random number generator may thus be implemented using a static variable for the seed value. Unfortunately, an OWN variable may not be used, as initialization is impossible.

This distinction between static and automatic storage allocation is too coarse; the duration of a static variable may be too long or too short. A static variable would have a longer duration than was necessary for the seed of the random number generator. What is really required is a variable whose storage has the same duration as the local storage of the procedure containing the random number generator.

This excessive duration is harmful because the initial value of the static variable must be a constant whereas the initial value may need to be calculated at run time. For example, it may be read in or generated from the clock. Initialization of variables with longer duration than visibility will be discussed in more detail later. A second problem is that storage is wasted; when a stack discipline is used, storage is minimized by making all durations as short as possible. A third problem is that many asynchronous procedures may wish to use the random number generator, yet have separate seed values.

On the other hand, the duration of a static variable may not be sufficient. It is quite reasonable for a program to remember data from one invocation to the next. Such information is usually stored in a general purpose data set or external file, and its use is not consistent with other data accesses in the programming language. None of the language's type security is available for data sets. If storage for a

variable were given permanent duration, that variable could serve all the purposes of a data set yet benefit from the security and ease of access provided by the programming language.

In order to provide as much flexibility as possible, the duration of storage for a variable may be specified within the declaration. The duration may be one of the following: local, containing, intermediate, global, or permanent.

Variables of local duration are conventional local variables where the duration is the same as the visibility. Any variable whose duration is not specified by the programmer is of local duration.

Variables of containing duration have the duration of the scope which immediately contains the procedure in which the variable is declared.

Global duration is that of the main procedure.

Permanent duration persists between program invocations.

All other durations correspond to procedures whose scopes are between containing and global. Such durations are known as intermediate durations, and must be referred to by the name of the procedure whose local variables are of that duration.

It is also desirable to control the scope of dynamically allocated storage. Most languages allow storage to be generated only at the global level. Algol 68 allows generators to specify local storage also. In L, the same duration control may be placed on dynamically allocated storage as on the storage for variables. In order to be compatible with

existing languages, the default duration could be global. However, in the author's opinion, the containing duration would be more useful. Of course, data sets of any organization may be created using dynamically allocated permanent storage.

3.14 DURATION OF ARRAY BOUNDS

When an array is created, its bounds must be evaluated. Thus any variable which is used as an array bound must exist when this creation is done. For an array variable the duration of any variables used as bounds must be greater than or equal to the duration of the array. For constant arrays, the values in the bounds must themselves be constant, i.e. of infinite duration. For dynamically created arrays any visible variable can be evaluated at allocation time and is therefore valid.

The same duration rules also apply to values used to initialize variables. In fact, all parameters to the type of the object being allocated must obey these rules.

3.15 DURATION OF EXPRESSIONS

In general, arbitrary expressions may be specified as array bounds, initial values, or as type parameters. It is necessary to define the duration of an expression such that the duration of a variable must be less than or equal to the duration of any expression used as a parameter to its type, or as an initial value.

The duration of a variable and of a constant have already been defined. Thus, the duration of an expression consisting of a single

constant is infinity whereas the duration of an expression consisting of a single variable could be the duration of that variable.

If the expression consists of more than one term, none of the terms may have a duration which is less than that of the variable being declared. Thus, the duration of an expression does not exceed the duration of any of its terms. An expression with more than one term involves operators or procedures as well as variables and constants. These functions have a duration which must equal or exceed that of the variable being declared.

The duration of a procedure variable is the duration of the variable itself. However, the duration of a procedure constant is not necessarily infinite. If the procedure makes references to variables whose duration is non-local or calls procedures which make such references, the duration of the procedure constant must be considered to be reduced accordingly. References to variables with local duration are not relevant as such variables are created and discarded every time the procedure is called. Thus, the duration of a procedure constant is the duration of the variable or procedure of least non-local duration which is referenced by the procedure. This definition is similar to the definition of the scope of a routine text in Algol 68.

Since procedure constants may have other than infinite duration, it is necessary to state an additional rule which was previously impossible to violate: the duration of a value must be greater than or equal to the duration of the variable to which that value is assigned. Since the duration of most values is infinite, this rule is usually easily met.

However, care must be taken to observe this rule when procedure variables are assigned. Further discussion of this rule is necessary when discussing pointer variables in section 3.19.

Since the duration of a procedure value affects assignment compatibility, it is reasonable to consider this duration to be part of the procedure's type. It is also reasonable that all type information be specified in the header of the procedure. Thus, a programmer would have to specify INFINITE, PERMANENT, CONTAINING, etc. as part of the procedure header. Any code compiled within the context of this declaration could be checked to recognise violations to the duration rules.

A restricted form of this concept can be found in preliminary Ada: subprograms can be declared as either procedures or functions. Procedures are assumed to have containing duration i.e. are allowed to reference global variables, while functions have infinite duration; i.e., are allowed to reference only constants, local variables, and other functions.

On the other hand, it may be argued that the duration of a procedure can be determined from the text of that procedure and therefore its specification on the header is redundant. Freeing the programmer from having to specify too much redundant information is desirable. The costs of doing so are the following. The compilation process becomes more complicated. The program editor has more difficulty. The programmer may lose sight of the significance of the duration of a procedure.

Deriving the duration of a procedure from the text causes more work for the compiler. Not only must the text of the procedure be examined, so must the text of all contained procedures. The duration of all non-contained procedures which are called must also be known. This derivation can, however, be done using a fairly simple extra pass for duration checking. First, a list is made of which of the possible durations each routine accesses. From these lists, the duration of all procedures can be calculated.

The separate compilation problem may also be addressed. The problem is that recompilation of executable code is no longer guaranteed to be a compatible change. However, such situations can be recognised. If the text is modified so as to reference storage of a duration which was not previously referenced, then the procedure's duration or the duration of one of its fathers may be affected.

More specifically, if there exists a procedure, P, whose executable code is being modified or whose descendant's executable code is being modified, and that modification causes a reference to an item whose duration is less than the duration of P but greater than or equal to the visibility of P, then P's duration is reduced by the modification.

If one of these durations is affected, the change must be regarded as a change to the declarations containing the header of the procedure whose duration is affected. If the duration is reduced, this change is non-compatible and could possibly necessitate further recompilations.

Thus, the programmer must be aware that, even though he has not explicitly specified the duration of a procedure, that duration is nevertheless a part of the declaration of that procedure. For this reason, the prototype of language L will allow the programmer either to specify the duration of a procedure, or to state explicitly that the duration is to be derived from context. If the duration is stated, the compiler can easily verify that it is not violated.

3.16 DURATION OF TYPES

It has already been mentioned that parameters to the type of a variable must not have lesser duration than that variable. A more general approach is to consider the type specification to be an expression of type TYPE, and to require that the duration of this type expression must not be less than the duration of the variable. The duration of a type expression is the least of the durations of the parameters and the type generator itself.

Built-in type generators such as INT, REAL, and ARRAY have infinite duration. The definition of the duration of a user-defined type is similar to that of procedures: the duration of a new type is that of the least duration referenced in the right-hand side of the type definition. The problem of separate compilation and the declaration of the duration of a type does not arise; type definitions may not be nested and do not contain a body or declarations which may be separately compiled.

This definition of the duration of a user-defined type removes a major impediment to bringing "external" files under control of the language system; records in files may be defined to have user-defined types provided that those types have permanent duration.

3.17 CONSTANTS AND "READ ONLY VARIABLES"

Languages which allow constants to be defined by the programmer usually permit only one of two kinds: the constant which must be equivalent to a literal, as in Pascal, and the constant which exists only for its visibility and therefore may take on any value which can be computed at entry to its visibility, as in Algol 68. Some languages support both facilities. In these languages, the former are referred to as constants, and the latter ironically as "read-only variables".

Both facilities are encompassed by a more general constant facility in L. A constant behaves exactly as a variable except that it may not be the target of an assignment; it maintains its initial value throughout its lifetime. A constant has storage, an address, a visibility, and a duration as does a variable.

Constants of permanent duration correspond to the first kind of constant described with the extension that the value of such constants may be the result of an expression provided that that expression has permanent duration. Constants of local duration are "read-only variables" as discussed above. Local constants may be given any value which can be computed at the declaration site. A constant may also have any intermediate duration. The following L declaration is like a Pascal constant declaration:

CONST E: REAL VALUE(2.718) ,

while the following is equivalent to an Algol 68 identity declaration:

LOCAL CONST X: Y+1 .

The following declaration is legal in L but not in Pascal; the expression 1+2 is evaluated at compile time:

CONST THREE: 1+2 .

3.18 INITIAL VALUES AND INVOCATION EXITS

As mentioned in section 3.15, storage is allocated for variables of a particular duration at the beginning of that duration. At this time, the INIT clause for that variable is evaluated. A side effect of this feature is that any procedure may gain control when any of its ancestors is invoked, merely by placing a call to an exit procedure in an INIT clause for a variable of the appropriate duration. If that duration is CONTAINING, the exit will be called when the father is started. If the duration is GLOBAL, the exit will be called when the program is initially loaded. If the duration is PERMANENT, the exit will be called at compile time.

This feature can be used to implement "optional features" as found in the Protel system [LAS79]. An optional feature is a feature which is implemented by a procedure which may or may not be linked with the program. If the procedure is linked with the program, it must let its father know that it is present by assigning itself to a procedure variable. This approach is taken so that there is not an unresolved reference if the procedure is not linked with the program. An example of an optional feature is presented.

```

PROC USER
  VAR EXTRA: PROC
  VAR EXTRAPRESENT: BOOL INIT(FALSE)

  PROC EXTRAFEATURE
    CONTAINING VAR JUNK: BOOL INIT(EXIT)
    CONTAINING PROC EXIT RETURNS R:BOOL
      EXTRAPRESENT := TRUE
      EXTRA := EXTRAFEATURE
      R := TRUE
    END EXIT

    --- implementation of extra feature ---

  END EXTRAFEATURE

  IF EXTRAPRESENT THEN CALL EXTRA

END USER

```

The procedure USER contains but does not reference EXTRAFEATURE. If EXTRAFEATURE is linked with USER, then USER calls it through the procedure variable EXTRA. If EXTRAFEATURE is not present, USER does nothing. A complete discussion of the use of this facility is found in [LAS79].

3.19 POINTERS AND SCOPE

Pointers allow the programmer to reference storage indirectly. A pointer value refers to an area of storage with specific attributes. Some languages, such as PL/I, do not restrict the type of storage to which a pointer may refer. This failure defeats static type security within the language, and requires extra information to be kept and extra work to be done at run time in order to ensure type security dynamically. A further problem exists because the duration of storage to which a pointer may refer is not constrained; it is entirely possible to reference storage whose duration has terminated. Such a reference is

known as a dangling reference. Once again, extra information has to be stored and extra checking done every time a pointer is used in order to detect dangling references.

Algol 68 has solved the type security problem by making the type of storage to which a pointer refers part of the type of that pointer. The duration problem has been recognised and relieved but still relies on run-time checking. The rule which is enforced in Algol 68 is that a pointer variable may not contain a pointer to storage whose duration is less than that of the pointer variable. This rule prevents dangling references but cannot be enforced statically. Thus every pointer value must also contain the duration of storage to which it refers. While run-time checking is still required, it need only be done when pointers are assigned rather than at every pointer reference. This saving is significant.

In L, the duration of the storage to which a pointer refers is considered to be part of that pointer's type as is the type of the value contained in that storage. For example, the programmer may declare:

```
VAR X: PTR(GLOBAL INT)
```

or

```
VAR Y: PTR(LOCAL REAL)
```

indicating that X points to global (HEAP in Algol 68) INTs and Y points to local REALs. The duration to which a pointer refers is independent of the duration of the pointer variable which contains the pointer with the exception that the latter must not exceed the former.

In L, a further storage attribute exists that does not in other languages. Storage may be variable or constant, as both variables and constants have storage allocated on their behalf. Algol 68 pretends that the storage for constants does not exist; a pointer cannot refer to constant storage. This restriction is severe in some cases [KOS75], such as when large arrays are forced to be copied in order to allow constant as well as variable arrays to be used as actual parameters to a procedure. The property of storage that determines whether it represents a variable or a constant is named accessibility. In L, the accessibility of storage to which a pointer refers is also part of the type of that pointer. A pointer may thus be declared to be a VAR pointer or a CONST pointer, with VAR or CONST indicating the accessibility. In order to be compatible with other languages the default duration and accessibility for pointers is GLOBAL VAR.

3.19.1 CONVERSION OF POINTER VALUES

It is possible to convert pointers which refer to one type of storage to pointers which apparently refer to a different type of storage. This conversion may be done if storage of the old type can, with security, impersonate storage of the new type. For example, a PTR(LOCAL VAR INT) may be converted to a PTR(LOCAL CONST INT), as variable storage may always be treated as CONST storage. Of course, VAR storage may be changed via a PTR(VAR) even while a PTR(CONST) refers to it.

Another conversion which is possible within the module which defines a new type is the conversion between a pointer to a NEW type and a pointer to the representation of that NEW type (within the module which

defines the NEW type, of course). This conversion must be explicitly requested using the 'REP syntax.

It is also possible in L to shorten the duration of a pointer. For example, a PTR(GLOBAL REAL) may be converted to a PTR(LOCAL REAL), as we know that no dangling reference could result; the pointer variable will disappear before the storage to which it refers.

Under some circumstances, it is possible to treat storage for a variable of one type as if it were storage of another type. One such example has been given: the ability to refer to storage of type T as if it were of type T'REP. Another instance is the ability to refer to storage which contains a type of a particular duration as if it contained a type with smaller duration. This pretense can only be made if the reference is a CONST reference. A CONST reference can be made as it is always safe to use a value of longer duration as if it had shorter duration. However, a VAR reference cannot be made as this would allow us to place into the storage a value of lesser duration than that storage is permitted to hold, thus potentially causing dangling references. In order to illustrate the difference between this and the previous rule, the two rules are stated precisely and the following example is given.

1. A pointer to storage of a certain duration may be interpreted as a pointer to storage of a lesser duration.
2. A pointer to constant storage which contains a value of a certain duration may be interpreted as a pointer to constant storage which contains a value of a lesser duration.

```

VAR A: PTR(GLOBAL VAR PTR(GLOBAL VAR INT))
VAR B: PTR(LOCAL VAR PTR(GLOBAL VAR INT))
VAR C: PTR(LOCAL VAR PTR(LOCAL VAR INT))
VAR D: PTR(LOCAL CONST PTR(LOCAL VAR INT))

A := B      (invalid because of first rule)
B := A      (valid because of first rule)
C := B      (invalid because of second rule)
D := B      (valid because of second rule)
D := A      (valid because of first and second rules)

```

In fact, one does not frequently use pointers to pointers. However, it is very common to use a pointer to a structure which has a pointer as one or more of its fields. These rules apply to every field in the structure. The same duration rules apply to pointers to procedure variables. All other values have infinite scope and are therefore not subject to the second rule.

3.20 DURATION OF PARAMETERS

As in Algol 68, parameters in L are passed by value. The duration of these values is local. The accessibility is CONST. If a formal parameter is to specify call by address, this may be done in two ways which are equivalent. The formal parameter

```
VAR X: INT
```

is equivalent to

```
X: PTR(LOCAL VAR INT) .
```

Call by address without allowing updating can be specified by:

```
CONST Y: INT
```

or

```
Y: PTR(LOCAL CONST INT) .
```

These alternatives are provided so that the programmer can have three modes of parameter passing without being familiar with the concept of pointers: call by value, call by const, and call by var.

The formal parameter may point to a value of greater duration. For example

```
GLOBAL VAR X: INT
```

or

```
X: PTR(GLOBAL VAR INT) .
```

In this case an integer variable of less than global duration may not be passed as an actual parameter. The benefit is that the address of this variable may be assigned to a GLOBAL pointer.

Under some circumstances, the duration of storage to which a parameter points is not known or is not the same for all invocations of the procedure. In such instances, the duration may be specified as an implicit parameter of type DURATION. For example, consider the following procedure which interchanges two GLOBAL pointer variables.

```
PROC EXCH(VAR P: PTR(GLOBAL INT), VAR Q: PTR(GLOBAL INT)):
```

```
  VAR TEMP: PTR(GLOBAL INT)
```

```
  TEMP := P
```

```
  P := Q
```

```
  Q := P
```

```
END EXCH
```

The above procedure is generalized to exchange two pointers to any scope:

```
PROC EXCH(VAR P: PTR(D:DURATION INT), VAR Q: PTR(D INT)):
    VAR TEMP: PTR(D INT)
    TEMP := P
    P := Q
    Q := P
END EXCH
```

The implicit parameter D is used in place of GLOBAL as the duration of P, Q, and TEMP. Run-time duration checking is required whenever a duration specified by a parameter is used in combination with another duration.

It is also possible to use implicit duration parameters in the type of pointer variables in the same way as other implicit parameters. Pointers whose duration is implicit are equivalent to the pointers in Algol 68:

```
TYPE REF(T:TYPE): PTR(D:DURATION T) .
```

Thus, unless the programmer specifically requests dynamic duration of pointers, all duration checking can be done statically using the above-mentioned rules.

3.21 COMPILE-TIME INVOCATION OF PROCEDURES

Many languages support a compile-time programming facility. This facility is another feature used to justify MACRO processors. In some languages, such as 370/ASSEMBLER [IBM75], the compile-time language does not resemble the run-time language. In other languages, such as PL/I, the syntax of the compile-time language is similar to that of the programming language, but the compile-time processor is nevertheless a distinct facility.

In L, the same procedures may be called at compile time as at run time. Indeed, variables containing such information as generation parameters and compilation date may be placed in variables shared by the compile-time and run-time routines. The restriction is that only procedures which are of permanent duration may be called at compile time. In fact, any expression whose duration is permanent is evaluated at compile time. This facility imposes an additional constraint on the order of compilation. must have been compiled previously.

3.22 CONCLUSIONS

These facilities for parameterization of procedures and types enable the programmer to define abstractions which are considerably more general than those feasible using conventional parameters. All of the facilities provided by abstract types, and many currently provided by macro facilities have been provided. The parameterization discussed can be compiled easily; no run-time symbol table or instantiation (of procedures or types) is required.

The concepts of duration and visibility of variables have been separated, allowing much more flexible storage control, including control over permanent disk storage. The duration of storage to which a pointer refers, and whether or not that storage may be modified, has been included within the type of a pointer. This inclusion allows all dangling references to be prevented at compile time unless the programmer specifically uses dynamic durations using the general pointer parameter mechanism presented.

Chapter IV

OVERLOADING AND AUTOMATIC CONVERSION

4.1 INTRODUCTION

Many languages allow the same symbol or name to denote many different functions. At a particular call site, the appropriate function is selected based on the types of its parameters and, in Ada, on the type of the desired result as well. These facilities are known as operator overloading and operator selection.

Many languages also support automatic conversions between various data types. However, all such conversions follow a set of predefined rules. It would be useful to allow the user to define conversions, especially if the language allows him to define his own data types. The inclusion of this facility is difficult, and is particularly difficult when operator overloading is also permitted.

Another issue which affects overloading and conversion is that of separate compilation. The facilities discussed in this chapter have been included in various combinations in existing languages, but are not all supported by a single language.

Some existing languages are now discussed.

4.2 OVERLOADING IN FORTRAN

Given the following FORTRAN statement:

$$Z = X / Y$$

a REAL division is performed if X and Y are REAL, whereas an INTEGER division is performed if X and Y are INTEGER. The type of Z is not considered. This omission frequently causes problems for the unwary programmer; the statement

$$Z = 1 / 3$$

gives Z the value zero regardless of its type.

FORTRAN also allows type conversion at the same time as overloading. In the previous example, if X or Y is REAL and the other INTEGER, the one which is INTEGER will be converted to REAL and REAL division will be performed. FORTRAN is able to support conversion and overloading because all operators and types are defined in the language; all possibilities may be tabulated within the compiler.

Thus, FORTRAN provides a useful facility with two drawbacks. The major drawback is the lack of user-defined types. The other drawback is that the type of the result is not considered in the selection process.

4.3 OVERLOADING IN PL/I

For built-in operators, PL/I provides the same facilities as FORTRAN. However, because of a richer set of predefined types, failure to consider the desired result in selection is more serious. The expression

$$10 + 1/3$$

yields the value 0.33333333333333 in PL/I because an inappropriate division is performed.

As well as the FORTRAN-like facility, PL/I does allow overloading of user-defined functions via the GENERIC statement. Each function has its own unique name. However, a group of functions may be assembled under one GENERIC name. The GENERIC statement specifies which of the functions should be selected when this name is referenced. This selection is accomplished by searching a list of possible parameter types; for each set of types on the list, the programmer specifies which function is to be used. Possible conversions are not considered during the selection process. After selection, however, conversions will be applied as necessary.

The major drawback of the PL/I system is that all possible functions and combinations of types must be tabulated in the GENERIC statement. Further problems are those of the FORTRAN system.

4.4 OVERLOADING IN ALGOL 68

The authors of Algol 68 have recognized that conversions and overloading interact and are difficult to provide at the same time. They have, therefore, chosen to provide two distinct kinds of functions. Operators are functions which may be overloaded but for which conversions will not be done in order to match their parameters. Procedures, on the other hand, may not be overloaded but automatic conversions will be applied in order to match actual parameters to formal parameters of procedures.

Algol 68's treatment of operators necessitates the definition of a very large number of operators. For example, in order to define a FORTRAN like multiplication operation which operates on INT and REAL values, it is necessary to define four multiplication operations: INT*INT, REAL*REAL, INT*REAL, and REAL*INT. Defining a new operation to apply to all N numeric types would require N^2 operator definitions. In a system which takes the type of the result into consideration, defining this new operation would require N^3 operator definitions.

4.5 OVERLOADING IN CLU

CLU was one of the first languages to allow the programmer to define a new data type along with the operations for objects of that new type within a CLUSTER. Overloading is permitted in CLU, as each CLUSTER may have its own version of a particular function. However, the selection rule is trivial; only the type of the first parameter is considered in selecting the appropriate CLUSTER and hence the appropriate function. The type of the result is certainly not considered. Furthermore, CLU allows no automatic conversion.

4.6 OVERLOADING IN ADA

Ada is similar to CLU in that it allows a programmer to define new types along with their access functions. All functions are overloaded and no automatic conversion is provided. The selection process is more sophisticated than that of CLU; not only are the types of all of the parameters considered during selection, the type of the result is considered also. It is necessary, therefore, to select all operators in

an expression simultaneously. Existing algorithms to perform the selection, and a new algorithm to do so are discussed in section 4.11.

4.7 THE LANGUAGE L

The language L supports all of the features described at the beginning of this chapter. Unlike those in Ada, all operators are not automatically overloaded. Selection of overloaded operators takes the types of all parameters and of the result into account. L also allows the programmer to define his own types, as discussed earlier. Along with user-defined types, user-defined automatic conversions are allowed, and these conversions are considered during the selection process. Ambiguities introduced by new conversions are resolved by the use of a "best fit" selection algorithm. These features are discussed in the context of the separate compilation system discussed previously.

4.7.1 OVERLOADED PROCEDURES IN L

Typically, procedures are not overloaded. A programmer may write routines called INIT, READ_RECORD, PROCESS_RECORD, PERFORM_CALCS, etc. where the selection algorithm is obvious; look up the name in the symbol table. However, in Ada, the entire symbol table must be searched to ensure that there is no other procedure with the same name. This overloading could cause programs to be unreadable; the reader would have to calculate the type of each parameter and work through the selection algorithm before he could assure himself that the following call actually referred to the procedure XYZ defined here.

```
XYZ: PROC(  ,  ,  ):
```

```
CALL XYZ(  ,  ,  );
```

This problem is particularly severe when separate compilation is permitted; all libraries must be searched in order to find all XYZs. Furthermore, one of the most important compatible modifications, the ability to add a procedure to the environment, is no longer a compatible change; adding a procedure to a library could cause a correct statement to become ambiguous. The problem is even more severe if a "best fit" selection algorithm is used. We may introduce a procedure which causes a better fit than the intended one, thus inadvertently changing the program.

Overloaded procedures are, of course, useful in such cases as the operations "+", NULL, etc.. For this reason, functions in L are not normally overloaded but the keyword OVERLOAD may be specified with the procedure definition indicating that this definition overloads an existing more global one, called the BASE procedure. It is not correct to specify OVERLOAD if there is no such BASE procedure. Overloading is transitive and so a set of overloaded functions may be defined by specifying OVERLOAD on the definitions of all but the base (most global) procedure.

This requirement enhances both readability and compilability. When we are reading a program, most procedures are not overloaded, thus we may look directly at the most local definition. In the case of overloaded procedures, we need only consider all procedures until the base procedure is encountered; no more global scopes and libraries need be searched.

Compatibility analysis is also simplified; an overload definition is considered to be a reference to the next most global procedure with the same name. If a new function with the same name is inserted more globally than the base function, the change is compatible and is not reflected in the CR. If the new function is less global than the base, a CR entry will be generated because the new function obscures a global reference (the overload reference).

4.8 USER-DEFINED AUTOMATIC CONVERSIONS

FORTRAN originally had no automatic conversions. This restriction led to awkward expressions containing many calls to built-in conversions routines such as FLOAT and IFIX. It also led to such absurdities as five functions to compute the maximum of a list of numbers: AMAX0, AMAX1, MAX0, MAX1, and DMAX1. With the introduction of user-defined types, these problems become even more severe, resulting in the definition of a combinatorial number of procedures being required to implement essentially one function.

Some languages, such as Mesa, provide automatic conversions only for built-in types. It is argued that automatic conversions (and overloading as well) are really only useful for numeric types. With certain exceptions this is true, but it is also true that a large number of user-defined types will be numeric. For example, we may define counters, quantities, distances, and addresses, all of which are manipulated arithmetically.

One major occasion when data conversions are required is when operations are done on objects which have varying precision. For example, if we concatenate two strings of maximum possible length ten, the result is a string of maximum possible length twenty. However, it is common to assign that result to a variable of maximum possible length ten, requiring truncation.

Another exception to the "conversions for numerics only" rule is that of literals for user-defined types. CLU does not support literals for user-defined types. This restriction is very cumbersome. Furthermore, it encourages the implementor to add extra built-in types to the compiler instead of adding type definitions to a library, a more flexible approach.

Ada attempts to address this problem by having two independent ways of creating user-defined types. The modular version of a user-defined type does not allow the programmer to use literals. The other form of user-defined type is the NEW type. A NEW type is defined thus:

```
TYPE X = NEW Y
```

X inherits all literals which are valid for Y, as well as all functions defined which have a parameter or result of type Y. While NEW types appear to give the programmer a useful facility without automatic conversions, several serious drawbacks exist.

NEW types do not support data hiding. NEW types are useful in a limited number of applications where dimensional consistency is required (e.g. preventing the addition of apples and oranges). A complete system to perform dimensional analysis is more useful and can be

implemented using the tools of L. NEW types, as defined in Ada, make a large number of expressions ambiguous. NEW types, as re-defined to avoid ambiguity, could be useful but require a selection algorithm and symbol-table-manipulation algorithm far more complicated than that required for user-defined automatic conversions.

4.9 USER-DEFINED AUTOMATIC CONVERSIONS IN L

L provides a general automatic conversion facility. At any place, a special procedure may be defined which accepts a parameter of one type and returns another. Whenever necessary, this procedure will be called in order to match the result of an expression to the target type. However, between a given source and target, at most one such conversion will be applied.

This restriction can be a nuisance to the programmer. For example, he may define conversions between REAL and LONG REAL and between LONG REAL and LONG COMPLEX. Why should this not automatically define a conversion between REAL and LONG COMPLEX?

First, were the programmer to define the conversion between REAL and LONG COMPLEX directly for efficiency reasons, any expression requiring the conversion would be ambiguous. One could choose the single conversion over the double. But what if the single conversion were defined at a more global scope than both conversions involved in the double conversion? The same problems arise as with automatic overloading: readability is impaired; we must search the entire symbol table in order to ensure that a better conversion does not exist; adding any conversion would be a non-compatible change.

Suppose we were to choose the most local conversion path. We could then choose an indirect conversion when the direct one is appropriate; a programmer could inadvertently override a built-in conversion by a series of user-defined conversions. For example,

```
TYPE X
  CONVERT INT -> X
  CONVERT REAL -> X
  CONVERT X->REAL
```

the above statements would override the system conversion between INT and REAL, which is not what was intended.

Inserting a conversion into the library would not be compatible using this method either. For example we may have:

```
BEGIN
  CONVERT A -> C
  BEGIN
    BEGIN
      CONVERT B -> C
      VARIABLE AA: A, BB: B, CC: C
      C := A
    END
  END
END
```

The above assignment invokes the conversion defined in the outermost block. If, however, the statement "CONVERT A -> B" were inserted into the second block, it would be used in combination with the conversion in

the innermost block. The cross-reference information for B gives no indication that this insertion would cause problems. Therefore all insertions of conversions would have to be regarded as non-compatible.

Thus, all conversions in L must be defined directly. If a complex conversion is required, a cast may be used. For example,

```
VARIABLE LC: LONG COMPLEX, R: REAL
```

```
LC := LONG REAL R
```

```
LC := COMPLEX R
```

Defining all possible conversions is therefore straightforward, although a bit tedious.

When user-defined conversions are permitted, many expressions become ambiguous. For example, given the conversions `INT -> REAL` and `REAL -> INT`, and the procedures `INT * INT -> INT` and `REAL * REAL -> REAL`, the expression

```
VARIABLE I: INT
```

```
I := I * I
```

is ambiguous; we do not know which `"*"` to select. The programmer clearly intends that the `INT` version be chosen. Thus, requiring an unambiguous selection is unrealistic. Rather, operators in an expression must be selected according to "best fit".

In the above example, selecting the `REAL "*"` would require three conversions whereas the `INT "*"` would require none. Thus, the number of conversions could be used as a cost function to be minimized.

However, this algorithm fails to take into account the fact that some conversions are less desirable than others. Widenings, conversions which lose no information, are infinitely preferable to narrowings, those which do lose information. Our selection criteria could then be: 1. minimize narrowings, 2. minimize widenings among those selections which are tied for the least number of narrowings.

This algorithm still has a number of weaknesses. Assuming that $\text{LOG}(X)$ exists for REAL X and LONG REAL X , and returns the same result, the statement

```
VARIABLE LR: LONG REAL, R: REAL
```

```
LR := LOG(R)
```

is ambiguous because either the operand or the result may be widened to LONG REAL. In most cases the choice is arbitrary but consider $\text{LOG}(123456)$ and assume that REAL carries six digits of precision and LONG REAL twelve. The result to twelve digits is 11.7236400962, which is in fact the result produced if the parameter to LOG is widened. If instead the result is widened we get 11.7236000000, which is closer to $\text{LOG}(123451.049975)$ than to $\text{LOG}(123456)$.

The problem becomes more extreme when we consider operators with two parameters. Consider the choice between INT and REAL multiplication in the following statement:

```
VARIABLE I,J: INT, R: REAL
```

```
R := J * I
```

or between INT and REAL division here:

```
R := J / I.
```

Fewer widenings would be required if the INT operations were used. However, this choice could easily lead to truncation whereas no information could be lost if the REAL operations were used.

This selection can be made by modifying the COST function so that widening the result of a function is considered to implicitly widen not only the result, but each of the parameters as well. That is, the cost of a particular widening is equal to the number of nodes in the parse tree for the expression whose result is widened, including non-terminal nodes.

Similar analysis may be applied to the cost function for narrowings. Consider the following expression

$$R := X - Y$$

where X and Y are LONG REALs containing 123457.289012 and 123456.510987 respectively. If LONG REAL subtraction is used we get the correct result, 0.578025 but if REAL subtraction is used, we get 0.

Each operation applied to a narrowed value compounds the information loss. Therefore narrowings, when required, should be done as late as possible. This criterion may be incorporated into the cost function for narrowings by counting each narrowing plus the number of functions through which the narrowed value passes in contributing to the result. That is, the cost of a particular narrowing is multiplied by its depth in the parse tree.

The selection criteria may be summarized as follows.

All conversions are avoided where possible. Widenings are preferable to narrowings. Widenings are done as early as possible in the evaluation of the expression and narrowings are done as late as possible.

4.10 IMPLEMENTATION OF OVERLOADING IN CONVENTIONAL LANGUAGES

Selection which does not consider result types is straightforward. The algorithm may be expressed as a recursive function which selects the appropriate choice for the operator at the root of the parse tree.

SELECT(PARSE_TREE) RETURNS FUNCTION

 X := null list of functions

 for all sub-trees S of PARSE_TREE do

 concatenate SELECT(S) to X

 FUNCTION := the function with the name in ROOT(PARSE_TREE) which
 has formal parameters whose types correspond to X

END SELECT

The decision as to whether or not the formal parameters of a function match X differs among languages. However, there is no facility for applying an overall cost function to the expression.

4.11 AN ALGORITHM FOR THE SELECTION OF OVERLOADED FUNCTIONS IN ADA

In Ada, as in L, overloaded functions are selected based on the types of their results. However, Ada allows no automatic conversion and therefore the selection process is simpler than in L. Existing solutions for this simpler problem, as well as a new solution, are discussed in this section. In the next section, the new solution is enhanced to handle the selection process in L.

In conventional languages, the appropriate functions may be selected by a single bottom-up traversal of the parse tree. The preliminary Ada rationale [ICH79] presents an algorithm which selects functions using a number of top-down then bottom-up traversals. Persch et al. [PER80] show that this algorithm requires at most three traversals: top-down, then bottom-up, then top-down. They then show that the selection can be done with two traversals: bottom-up then top-down.

Both algorithms are complex and difficult to state precisely, and require the manipulation of sets whose size is unknown. The processing time required depends on the total number of set elements processed, rather than the number of traversals. Furthermore, storage is more likely to be a constraint than execution time. Thus it is not obvious that the [PER80] algorithm is superior to the [ICH79] algorithm in terms of either memory usage or execution time.

This section presents a new recursive algorithm which selects functions in Ada. Only stack (local) storage is required, and the stack depth will never exceed the depth of the parse tree being processed. In our examples, processing time has been found to be negligible. Nevertheless, three optimizations are presented. The first two require no extra storage but can save a great deal of time. The third requires some storage but reduces processing complexity to less than [ICH79].

4.11.1 AN ALGORITHM TO COUNT THE NUMBER OF INTERPRETATIONS OF AN EXPRESSION

The following algorithm is expressed in the form of a recursive procedure, SELECT, which accepts as parameters a parse tree for an expression and the desired type of the expression. It returns the number of combinations of functions which may be selected such that the expression has a result of the desired type.

```

SELECT(DESIRED_TYPE,PARSE_TREE) returns NUMBER_SOLUTIONS;

  if PARSE_TREE is a leaf then
    if type of PARSE_TREE is DESIRED_TYPE then
      NUMBER_SOLUTIONS := 1
    else NUMBER_SOLUTIONS := 0
  else
    NUMBER_SOLUTIONS := 0
    let Q = name of proc at root of parse tree
    let R = list of actual parameters to Q (as parse trees)
    for every proc QQ with name Q which yields DESIRED_TYPE do
      let S = list of formal parameters to QQ
      if R matches S in terms of length, parameter names, and
        default values
      then
        PARM_COMBOS := 1
        for I from 1 to length of R do
          PARM_COMBOS := PARM_COMBOS * SELECT(type of S[I],R[I])
        NUMBER_SOLUTIONS := NUMBER_SOLUTIONS + PARM_COMBOS
    END SELECT

```

If `NUMBER_SOLUTIONS` is 1, the expression is correct; if it is greater than 1, the expression is ambiguous; if it is 0, there is no consistent way of selecting the functions in the expression.

Note that all variables (`NUMBER_SOLUTIONS`, `PARM_COMBOS`, `QQ`, and `I`) are of fixed size and that the number of levels of recursion is equal to the depth of the parse tree. Furthermore, no functions are investigated unless their result could potentially be used in the expression. This statement also holds for [ICH79] but is not true for [PER80]. This fact raises doubts as to whether or not [PER80] is faster than [ICH79].

4.11.2 OPTIMIZATIONS TO THE ALGORITHM

4.11.2.1 OPTIMIZATIONS WHICH REQUIRE NO EXTRA STORAGE

Clearly, the loop beginning "for `I`" can be aborted if `PARM_COMBOS` ever becomes zero. This means that once a parameter of a function cannot be matched, that function is discarded from further consideration; we need not attempt to match the remainder of its parameters. Note that we may NOT terminate the loop when `PARM_COMBOS` exceeds one.

This modification reduces the processing required to reject invalid `QQ`'s (note that, in any valid expression, all but one `QQ` are invalid). If there is one invalid parameter, we have to examine, on average, only half of the parameters before the invalid one is encountered. If more than one parameter is invalid, as will often be the case, shorter searches occur. This optimization applies at each level of recursion and thus causes exponential improvement with the depth of the parse tree. Further heuristics may be applied, such as matching parameters of

uncommon types first, or matching parameters with smaller parse trees first. These techniques are not investigated in this thesis.

A similar optimization may be applied to the loop beginning "for every QQ". The loop may be aborted when NUMBER_SOLUTIONS exceeds 1; once the expression is found to be ambiguous we don't care how ambiguous it is. Note that this optimization changes the result of SELECT; we no longer receive a true count but can still determine whether the expression was inconsistent, correct, or ambiguous. As with the previous example, heuristics may be applied to the order of the loop; as with the previous example, these heuristics have not been pursued.

4.11.2.2 AN OPTIMIZATION WHICH USES EXTRA STORAGE

With the above optimizations, the algorithm investigates fewer possibilities than [ICH79] and many fewer than [PER80]. However, many recursive calls may be made with exactly the same arguments. For example, we may have:

```
PROCEDURE F(A:INTEGER; B:REAL);
PROCEDURE F(A:INTEGER; B:INTEGER);
```

In resolving the expression

```
F(1+2,3)
```

SELECT is called twice with DESIRED_TYPE = INTEGER and the parse tree for "1+2". This duplication can be avoided by attaching to each node of the parse tree a list of each DESIRED_TYPE for which SELECT has been called. The result of the calls must be stored also. The storage

required would be equal to that required by [ICH79] if the optimizations described in section 11.2.1 were not applied. If the optimizations were included, the storage required would be considerably less. Furthermore, running out of storage merely defeats the optimization; it need not cause termination of the algorithm.

4.11.3 YIELDING THE SELECTED FUNCTIONS

If the expression is unambiguous, it is necessary to set a pointer to the appropriate symbol-table entry for each function in the parse tree. The algorithm, as discussed so far, has merely indicated whether or not an unambiguous filled-in tree exists. A trivial method of obtaining the filled-in parse tree would be for SELECT to return a filled-in subtree from every successful invocation. However, this section describes a modified algorithm which sets the pointers in the original parse tree,

thus avoiding the storage cost of the previously mentioned modification.

```

SELECT(DESIRED_TYPE,PARSE_TREE,MARK) returns NUMBER_SOLUTIONS;

  if PARSE_TREE is a leaf then
    if type of PARSE_TREE is DESIRED_TYPE then
      NUMBER_SOLUTIONS := 1
    else NUMBER_SOLUTIONS := 0
  else
    NUMBER_SOLUTIONS := 0

    let Q = name of proc at root of parse tree
    let R = list of actual parameters to Q (as parse trees)
    for every proc QQ with name Q which yields DESIRED_TYPE do
      let S = list of formal parameters to QQ
      if R matches S in terms of length, parameter names, and
        default values
      then
        PARM_COMBOS := 1
        for I from 1 to length of R do
          PARM_COMBOS := PARM_COMBOS * SELECT(
            type of S[I],R[I],MARK and NUMBER_SOLUTIONS=0)
        if MARK and PARM_COMBOS > 0 and NUMBER_SOLUTIONS = 0 then
          set pointer in root of PARSE_TREE to point to QQ
        NUMBER_SOLUTIONS := NUMBER_SOLUTIONS + PARM_COMBOS
    END SELECT

```

We have added a boolean parameter, MARK, which indicates whether or

not `PARSE_TREE` should be marked with pointers to the appropriate symbol table entries. The pointers continue to be filled in until a solution is found. Subsequent calls merely check for ambiguity and therefore do not request that the parse tree be marked. If there is one solution, the tree is filled in appropriately; if there is more than one solution, the first one is filled in; if there are no solutions, the pointers which are filled in in the parse tree are not meaningful.

The optimizations discussed in section 11.2.1 can be applied to this algorithm as easily as to the first. The optimization described in section 11.2.2 may be applied also, but the symbol table pointers for the root of the tree must be stored along with the list of desired types for which `SELECT` has been called.

4.11.4 CONCLUSIONS

The major advantage of the algorithm presented is simplicity, both as a statement of the rules for overloading and as a guide for implementation. It is suggested that the execution time of the algorithm is not important. Nevertheless, the speed of three algorithms has been analysed. [PER80] follows different paths than [ICH79] or `SELECT` (the algorithm presented herein). However there is little reason to believe that it is faster or uses less storage than [ICH79]. Three optimizations to `SELECT`, which are mechanical and do not change the algorithm, make `SELECT` considerably faster than [ICH79].

`SELECT` is clearly superior in what we consider to be a more important constraint, namely, memory size. Two of the optimizations do not affect

memory usage, while the third may be selected at run time whenever there is surplus storage available.

The algorithm (without optimizations) has been implemented using PL/I. The effort required was very small and the algorithm was found to work correctly on a number of test cases..

4.12 IMPLEMENTATION OF USER-DEFINED CONVERSIONS

A user-defined conversion routine is specified and compiled much like an ordinary procedure or operator. Conversions, when used, are not necessarily done in line, but are subject to the general rules for in-line vs. dynamically called routines.

The pertinent information about the conversion routine is stored in the symbol table. However, the key used to access the symbol table entry is not the name of the conversion routine, but a unique name derived from the type of the parameter and the type of the result. Normal block structure applies; a more local conversion may hide a more global one with the same "name".

When a conversion is required, the appropriate routine may be found directly in the symbol table; no searching of possible conversions is required. Separate compilation is not affected as the same compatibility rules apply as to ordinary procedures. The concept of overloading is not meaningful as applied to conversions.

4.13 BEST FIT SELECTION WHERE CONVERSIONS ARE PERMITTED

This section describes an algorithm to select the best combination of overloaded functions in an expression. The cost function described previously is used to determine the best choice. First, an algorithm to determine the minimum cost and the number of choices which produce this minimum cost is presented. Modifications are then described which make the algorithm mark the best choice in the parse tree.

The algorithm is expressed as a recursive procedure which accepts three parameters: the type of the desired result, the parse tree for the expression, and the depth of the root of the parse tree. For complete expressions, this depth is 1. Only for recursive calls which pass sub-trees is the depth greater than 1.

Three results are returned: the number of solutions of minimum cost, the cost of widenings in the best solution, and the cost of narrowings in the best solution. The algorithm is as follows:

```

SELECT(DESIRED,TREE,DEPTH,#SOLNS,BEST_W,BEST_N)
  IF TREE IS A LEAF THEN
    IF TYPE IS DESIRED THEN #SOLNS,W COST,NCOST := 1,0,0
    ELSE IF TYPE WIDENS TO DESIRED THEN #SOLNS,W COST,NCOST := 1,1,0
    ELSE IF TYPE NARROWS TO DESIRED THEN
      #SOLNS,W COST,NCOST := 1,0,DEPTH
    ELSE #SOLNS,W COST,NCOST := 0,9999,9999
  ELSE
    LET Q = NAME OF PROC AT ROOT OF TREE
    LET R = ACTUAL PARAMETERS OF Q
    #SOLNS,BEST_W,BEST_N := 0,9999,9999
    FOR EVERY PROC QQ WITH NAME Q DO
      LET S = LIST OF FORMAL PARMS TO QQ
      IF R MATCHES S STRUCTURALLY THEN
        IF TYPE OF QQ CAN CONVERT TO DESIRED THEN
          IF TYPE OF QQ IS DESIRED THEN W,N = 0,0
          ELSE IF TYPE OF QQ WIDENS TO DESIRED THEN
            W,N := COUNT(TREE),0
          ELSE IF TYPE OF QQ NARROWS TO DESIRED THEN
            W,N := 0,DEPTH
        PC := 1
        FOR I FROM 1 TO LENGTH OF R DO
          CALL SELECT(TYPE(S(I)),R(I),DEPTH+1,A,B,C);
          PC := PC * A;
          W := W + B;
          N := N + C;
        IF N < BEST_N OR N = BEST_N AND W < BEST_W THEN
          #SOLNS,BEST_W,BEST_N := PC,W,N
        ELSE IF N = BEST_N & W = BEST_W THEN
          #SOLNS := #SOLNS + PC
      END SELECT
  END SELECT

```

4.13.1 OPTIMIZATIONS

The loop beginning "FOR I FROM 1" may be terminated if N exceeds BEST_N or if N equals BEST_N and W exceeds BEST_W. This optimization is similar to the first optimization discussed in the previous section.

A further optimization may be accomplished if another parameter, REQUIRED_COST, is added to SELECT. This parameter specifies the worst possible fit which could still contribute to a best overall solution.

BEST_W and BEST_N are initialized to REQUIRED_COST instead of infinity, while #SOLNS is initialized to zero. Thus the loop "FOR I" will terminate if the maximum acceptable cost cannot be obtained or if a previous selection has yielded a lower cost. When SELECT is called recursively in this loop, the value passed as REQUIRED_COST will be BEST_W - W and BEST_N - N.

The loop beginning "FOR EVERY PROC QQ" may not be optimized as it valid for expressions to be ambiguous; we must examine all possibilities to find out if there is a better choice for the particular function that this loop is attempting to select.

The optimization which requires storage may also be made. A list of all desired types along with the least cost of selecting each is stored with each node in the parse tree.

4.13.2 YIELDING THE SELECTED FUNCTIONS

The marking of which functions yield the lowest cost may be added to the above algorithm thus:

```

SELECT(DESIRED,TREE,DEPTH,#SOLNS,BEST_W,BEST_N,MARK)
  IF TREE IS A LEAF THEN
    IF TYPE IS DESIRED THEN #SOLNS,W COST,N COST := 1,0,0
    ELSE IF TYPE WIDENS TO DESIRED THEN #SOLNS,W COST,N COST := 1,1,0
    ELSE IF TYPE NARROWS TO DESIRED THEN
      #SOLNS,W COST,N COST := 1,0,DEPTH
    ELSE #SOLNS,W COST,N COST := 0,9999,9999
  ELSE
    LET Q = NAME OF PROC AT ROOT OF TREE
    LET R = ACTUAL PARAMETERS OF Q
    #SOLNS,BEST_W,BEST_N := 0,9999,9999
    FOR EVERY PROC QQ WITH NAME Q DO
      LET S = LIST OF FORMAL PARMS TO QQ
      IF R MATCHES S STRUCTURALLY THEN
        IF TYPE OF QQ CAN CONVERT TO DESIRED THEN
          IF TYPE OF QQ IS DESIRED THEN W,N = 0,0
          ELSE IF TYPE OF QQ WIDENS TO DESIRED THEN
            W,N := COUNT(TREE)
          ELSE IF TYPE OF QQ NARROWS TO DESIRED THEN
            W,N := 0,DEPTH
        PC := 1
        FOR I FROM 1 TO LENGTH OF R DO
          CALL SELECT(TYPE(S(I)),R(I),DEPTH+1,A,B,C,FALSE);
          PC := PC * A;
          W := W + B;
          N := N + C;
        IF N < BEST_N OR N = BEST_N AND W < BEST_W THEN
          #SOLNS,BEST_W,BEST_N := PC,W,N
          BEST_QQ = QQ
          BEST_S = S
        ELSE IF N = BEST_N & W = BEST_W THEN
          #SOLNS := #SOLNS + PC
      IF MARK AND #SOLNS = 1 THEN
        FOR I FROM 1 TO LENGTH OF BEST_S DO
          CALL SELECT(TYPE(S(I)),R(I),DEPTH+1,A,B,C,TRUE)
        SET ROOT OF TREE TO POINT TO QQ AS SELECTED FUNCTION
    END SELECT

```

Once again, an additional boolean parameter, MARK, is added to SELECT. When this parameter has the value TRUE, SELECT will fill in to the parse-tree pointers to the appropriate functions.

The algorithm, like the previous one, finds the cost of selecting a particular function. If that cost is the least so far, a pointer to this function (BEST_QQ and BEST_S) is set. After all the functions have

been tested, BEST_QQ contains a pointer to the best choice. Then, if MARK is TRUE and the best solution is unambiguous the root of the parse tree is filled in with BEST_QQ. Furthermore, SELECT is called for each of the parameters to BEST_QQ (BEST_S) in order to mark the subtrees of TREE.

The algorithm which does the selection can be somewhat slower than the algorithm which merely determines the cost. The fraction of extra time required depends on the number of choices for each overloaded function; the more choices there are the less overhead the selection involves. Let us consider the case where there are two choices for each function in an expression. At any given position in the parse tree, SELECT is called recursively twice as many times as there are parameters. The modified version of SELECT is called recursively three times as many times as there are parameters when MARK is true and twice as many when MARK is FALSE. Were there five choices for each, function, the extra calls would only amount to twenty percent at a given level, when MARK is TRUE.

The extra recursive calls are each more expensive, owing to the fact that they are expected to mark the parse tree. Thus, the time factor involved in marking the parse tree will be

$$1+(1/n)(1+(1/n)(1+(1/n)(\dots \dots)))$$

or

$$1+(1/n)+(1/n)**2+(1/n)**3+ \dots +(1/n)**d$$

where d is the depth of the parse tree. The sum of this series is

$$(1-(1/n)**d)/(1-1/n).$$

For the worst case of $n=2$ and d infinite, the value of this expression is 2. The marking algorithm, therefore, cannot be more than twice as slow as the non-marking algorithm.

The same optimizations can further pare the cost of marking. The optimization involving the extra parameter, "required cost", will be particularly effective on the extra calls as the actual best cost may be passed thus avoiding all paths which would exceed this best cost.

The optimization involving extra storage would eliminate the extra cost of marking. Every time SELECT is called, the parameters and results are added to a list pointed to by the root of TREE. Along with this information, a pointer to the best choice for the root is stored. Selection of the appropriate functions would then be equivalent to one traversal of the parse tree.

4.14 BEST FIT AND SEPARATE COMPILATION

When an expression is compiled, the set of functions which yields the least cost is selected. If conversions are later added or deleted, this selection may no longer yield the lowest cost and therefore no longer be correct. In this section the compatibility information which will indicate such situations is discussed.

Conversions which are used in the selected expression are considered to be referenced in the normal way. If one of these conversions is deleted, modified, or obscured, the program will be recognised as needing recompilation.

Recognising which conversions, if added, would cause a different selection to be made is more difficult. In general, we can write a logical expression which describes all conditions under which the current selection would not be optimal. If this logical expression were stored as part of the compatibility information, it could be used to determine whether or not a segment needed to be recompiled due to the addition of a conversion.

Unfortunately, this logical expression will be prohibitively large for all but the simplest expressions. Nevertheless, some aspects of dealing with these expressions are now discussed.

The variables in the boolean expression represent the existence of conversions which could be pertinent to the selection process. These conversions may be enumerated by considering, for any subtree in the original expression, the conversions between result types of that subtree and all possible desired types. The union of all such sets, one per subtree, is the set of variables in the logical expression.

Classical methods of boolean manipulation and simplification require the generation of all the minterms in a logical expression [MCL65]. These expressions could easily contain one hundred variables and therefore 2^{100} minterms. It is therefore impossible to generate the logical expression by trying all possible combinations of conversions to see which yield a cheaper solution.

A somewhat more practical approach would be to try all combinations of overloaded functions in the expression tree and to determine which

conversions would be required for each combination. Those combinations for which the overall cost is less than or equal to the actual cost would be added to the boolean expression. This approach would be within the realm of possibility for small expressions.

Assuming that the logical expression were to be generated in this fashion, it would have to be simplified. Simplification methods involving the generation of minterms or of prime implicants are hopeless. We can, however, do symbolic simplification if we take advantage of the fact that there are no complemented terms in the expression; all terms are combinations of conversions which must be present, not absent.

The initial step of simplification is to remove any terms which are covered by another; that is, all symbols in the covering term are also in the covered term. Since there are no complemented variables, we know that this simplification yields the simplest possible sum-of-products form of the expression.

This simplified expression is not simple to incorporate into a compatibility record. Nor is it the smallest possible or the easiest to evaluate. In order to minimize size and evaluation effort, it is possible to factor the expression so as to minimize the total number of boolean operations in the expression. This criterion is the same as for the minimization of gates in realising a boolean expression in hardware. Finding the optimum factorization is quite difficult [DAV78].

4.14.1 WEAKENING THE LOGICAL EXPRESSION

Even the simplified expression would be complex and time-consuming to evaluate. Since the objective is to avoid unnecessary compilation time, there is no benefit if more time is spent in this evaluation than that which is saved.

Since there is no harm in occasional spurious recompilations, we can weaken the logical expression in order to make it simpler. The weakened expression must still be true in all cases where recompilation is necessary; it could also be true when recompilation is not necessary.

One possible weakening is to use TRUE as the logical expression; all subprograms would be recompiled whenever a conversion was added to the environment. Another possible weakening is to use the disjunction of the variables corresponding to all possible conversions as defined previously. This set is easy to compute and may be added directly to the compatibility list as it is just a list of items which cause recompilation and does not require a data structure to represent a logical expression. However, the list may be excessively large. For example, in an arithmetic expression with six arithmetic types, all sixty possible widenings and narrowings would appear in the list.

Certain conversions may be removed from the list. These conversions are those which do not appear in any of the terms of the simplified unwidened logical expression. Clearly, the expression

$$ABC + ABD + BDE$$

is covered by

$$A + B + C + D + E.$$

However, the expression is also covered by

$A + B$ $A + E$ $B + D$ $B + E$

or

 $C + D,$

which are all shorter and would yield fewer spurious recompilations. Any disjunction of single variables which contains at least one symbol from each term will cover the original expression. Further discussion in this thesis will concern finding the minimum such disjunction of single symbols. Since calculating the original expression and then selecting the minimum cover would be prohibitively expensive, methods are developed here which approach this cover directly, without generating the logical expression.

4.14.2 FINDING THE MINIMUM COMPATIBILITY LIST

As mentioned previously, a possible compatibility list is the list of all conversions between result types and desired types at all positions in the expression tree. It is possible to reduce this list by isolating those terms which must appear, those which may appear, and those which do not appear in the minimum list.

4.15 A GRAPHICAL REPRESENTATION OF THE OPERATOR SELECTION PROBLEM

In order to analyse possible reductions to the compatibility list, a graphical representation of the expression and possible operations and conversions is needed. First, a simplified model is presented in which operators may have zero or one parameter and zero or one result. Then the representation is generalized to handle operators with more than one parameter.

4.15.1 A REPRESENTATION FOR ONE-PARAMETER FUNCTIONS

An expression tree involving only these kinds of functions is a line. At the root is a procedure with no result and at the leaf is a procedure with no parameters. The target of an assignment can be viewed as a procedure with one parameter and no result; a variable or literal can be viewed as a procedure with a result and no parameters. The rest of the functions will have one parameter and one result.

Consider the following expression.

$X := F(G(H(Y)))$

PAGE 118 OMITTED IN PAGE NUMBERING

Then, between pairs of result and parameter groups corresponding to adjacent functions, define an edge between vertices representing types between which a conversion exists. This edge has a weight of zero if the two types are the same; it has a cost of a widening or a narrowing at this position in the parse tree if such conversion exists.

Assume that, in the above example, the following procedures and conversions exist.

TYPE A,B,C,D,E

VAR X:A, Y:B

FUNC F(A) RETURNS B

FUNC F(A) RETURNS C

FUNC F(D) RETURNS D

FUNC G(C) RETURNS D

FUNC G(E) RETURNS A

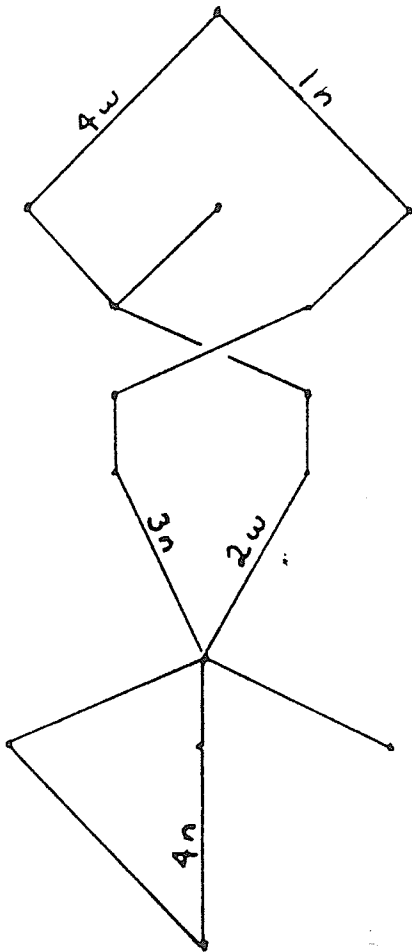
FUNC H(B) RETURNS B

FUNC H(C) RETURNS B

WIDEN B TO A, C TO D, B TO E

NARROW B TO C, D TO A

The resulting graph would be:



<u>FUNCTION</u>	<u>TYPES</u>
X	A
F (result)	B,C,D
F (parm)	A,D
G (result)	D,A
G (parm)	C,E
H (result)	B
H (parm)	B,C,D
Y	B

unlabelled edges have weight 0

The graph may be represented as seven cost matrices.

$$C1 \quad (0 \quad 4n \quad \infty)$$

$$C2 \quad \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}$$

$$C3 \quad (3n \quad 2w)$$

$$C4 \quad \begin{pmatrix} 0 & \infty \\ \infty & 0 \end{pmatrix}$$

$$C5 \quad \begin{pmatrix} \infty & 0 \\ 0 & \infty \end{pmatrix}$$

$$C6 \quad \begin{pmatrix} 0 & 0 & \infty \\ \infty & \infty & 0 \end{pmatrix}$$

$$C7 \quad \begin{pmatrix} 4w \\ \infty \\ 1n \end{pmatrix}$$

The minimum possible cost of selecting functions may be determined by computing

$$C = C_1 \underline{.} + C_2 \underline{.} + C_3 \underline{.} + C_4 \underline{.} + C_5 \underline{.} + C_6 \underline{.} + C_7$$

where $\underline{.}$ is the APL operator which is defined as

$$X = A \underline{.} B \text{ is equivalent to}$$

$$X(i,j) = \min_k A(i,k) + B(k,j)$$

That is, the operation $\underline{.}$ is matrix multiplication with the minimum and addition operations used in place of addition and multiplication respectively.

Once the minimum cost is computed, we can find out which functions contributed to this cost by solving the matrix equation

$$C_1 L + C_2 L + \dots L + C_{i-1} L + U_i L + C_{i+1} L + \dots C_7 = C$$

for each U_i ($1 \leq i \leq 7$). We know that a given function contributes to the best solution if

$$C_i(m,n) = U_i(m,n)$$

where $C_i(m,n)$ is the entry in the cost matrix representing the function in question. If for a given C_i , this is true for more than one pair of m and n , the function is ambiguous. Similarly, the C_i and U_i matrices which correspond to conversion paths indicate which conversions are used in the solution.

4.15.1.1 SOLVING THE MATRIX EQUATION $A L + X = B$ FOR X

The above equation is equivalent to

$$B(i,j) = \min_k A(i,k) + X(k,j)$$

We can see that, in order for this equation to hold, $A(i,k) + X(k,j)$ must exceed or equal $B(i,j)$ for all i, j , and k . That is,

$$X(k,j) \geq \max_i B(i,j) - A(i,k)$$

or

$$X \geq (-A)^T L + B \text{ where}$$

$$D = E L + F$$

is defined as

$$D(i,j) = \max_k E(i,k) + F(k,j)$$

Thus we have shown that

$$X = (-A)^T \bar{+} B$$

is the minimum possible solution to the matrix equation

$$A \underline{+} X \geq B$$

and is therefore a solution to

$$A \underline{+} X = B$$

if a solution exists. Similar argument shows that

$$(-A)^T$$

is also the right inverse of A with respect to $\underline{+}$, provided that the dual operation, $\bar{+}$ is used when multiplying by these inverses.

4.15.1.2 CALCULATING ALL THE U_i

For an expression represented by n C_i , we need to solve the following equation for all U_i ($1 \leq i \leq n$).

$$C_1 \underline{+} C_2 \dots \underline{+} U_i \underline{+} \dots C_{n-1} \underline{+} C_n = C$$

In order to do this, we compute for every i

$$U_i = (-L_{i-1})^T \bar{+} C \bar{+} (-R_{i+1})^T$$

where

$$L_i = C_1 \underline{+} C_2 \underline{+} \dots C_i$$

and

$$R_i = C_i \underline{+} C_{i+1} \underline{+} \dots C_n.$$

Note that L_i and R_i are vectors. All the L_i may be computed and stored using $n-1$ vector-matrix multiplications. Then the R_i and U_i may be computed using another $n-1$ matrix-vector multiplications.

4.15.1.3 MINIMIZING THE COMPATIBILITY LIST USING MATRICES

We can define a set of cost matrices, P_i , which parallel the C_i matrices. A particular element in P_i indicates the best possible conversion cost which could occur if the appropriate conversion were added to the environment. If the two types are the same, this cost is zero. Otherwise, the cost is that of a widening at the i th position in the expression tree.

If any element of a P_i is less than or equal to the corresponding element in U_i , the conversion which these elements represent must appear in the compatibility list. These conversions are referred to as MUSTs as they always necessitate recompilation and therefore must be present in the weakened logical expression.

Using the P_i and the C_i , it is also possible to determine which conversions could never affect the selection made and therefore necessitate recompilation, even if all other possible conversions existed. Such conversions are referred to as NEVERs. We can define another set of matrices, W_i , which indicate the most expensive conversion at any given position which could contribute to a better solution. To compute W_i , we must solve the matrix inequality

$$P_1 \cup P_2 \cup \dots \cup P_{i-1} \cup W_i \cup P_{i+1} \dots \cup P_n \geq C.$$

The W_i may be computed using the same approach as when calculating the U_i .

Any element of a P_i which is less than the corresponding element in W_i could not possibly contribute to a selection which is better than the existing one. However, it is possible to find more conversions which

may be deleted from the compatibility list. These are conversions which, if they existed, could contribute to a selection which is better. However, no such solution would be optimal; the other conversions involved would combine with other existing conversions to produce an even better selection.

Such conversions may be recognized using the matrices. The matrices K_{ij} are defined as

$$K_{ij} = C_i L + C_{i+1} L + \dots \dots L + C_j \quad (1 \leq i < j \leq n)$$

If a conversion represented by an element of P_1 could not contribute to any element of some K_{ij} ($i \leq 1 \leq j$), then that conversion is one of those just described.

We therefore define X_{ij1} as being the minimum solution to

$$P_i L + P_{i+1} L + \dots P_{1-1} L + X_{ij1} L + P_{1+1} L + \dots P_j \geq K_{ij}$$

that is,

$$X_{ij1} = (-P_{1-1}^T) L + \dots (-P_i^T) L + K_{ij} L + (-P_j^T) L + \dots (-P_{1+1}^T) L$$

A stronger W_i is thus defined to be

$$W_i(j,k) = \min_{x,y} X_{xyi}(j,k) .$$

All conversions whose cost exceeds the corresponding element of W_i can be discarded from the compatibility list.

Evaluation of the new W_i in the straightforward manner would require of the order n^2 matrix multiplications. Therefore, the problem is now rearranged in order to reduce computational effort. The definition of X_{ij1} may be reassociated as follows.

$$X_{ij1} = B_{i,1-1} L + C_1 L + T_{1+1,j}$$

where

$$B_{ij} = -(P_i \underline{L} + \dots P_j)^T \underline{I} + (C_i \underline{L} + \dots C_j)$$

and

$$T_{ij} = (C_i \underline{L} + \dots C_j) \underline{I} + -(P_i \underline{L} + \dots P_j)^T.$$

These definitions may be rearranged.

$$B_{ij} = ((-P_j)^T \underline{I} + (\dots ((-P_i)^T \underline{I} + C_i) \underline{L} + \dots) \underline{L} + C_j)$$

$$T_{ij} = (C_i \underline{L} + (\dots (C_j \underline{I} + (-P_j)^T) \underline{I} + \dots) \underline{I} + (-P_i)^T)$$

These matrices may be computed using the following recursions:

$$B_{ij} = (-P_j)^T \underline{I} + B_{i,j-1} \underline{L} + C_j \quad (j \geq i)$$

$$B_{ij} = I \quad (j < i)$$

and

$$T_{ij} = C_i \underline{L} + T_{i-1,j} \underline{I} + (-P_i)^T \quad (j \geq i)$$

$$T_{ij} = I \quad (j < i)$$

where

$$I = \begin{pmatrix} 0 & & & & & \\ & 0 & & & & \\ & & 0 & & & \\ & & & \ddots & & \\ & & & & 0 & \\ & & & & & 0 \\ & & & & & & 0 \\ & & & & & & & 0 \\ & & & & & & & & 0 \end{pmatrix}$$

The formula for W_i , expressed in terms of B and T is

$$W_i = \min_{j,k} B_{j,i-1} \underline{L} + C_i \underline{L} + T_{i+1,k}$$

Reassociating this expression yields:

$$W_i = (\min_j B_{j,i-1}) \sqcup C_i \sqcup (\min_k T_{i+1,k})$$

Using the recursions previously defined, we have

$$(\min_j B_{j,i-1}) = \min(I, (-P_{i-1})^T \sqcup (\min_j B_{j,i-2}) \sqcup C_{i-1})$$

and

$$(\min_k T_{i+1,k}) = \min(I, C_{i+1} \sqcup (\min_k T_{i+2,k})^T \sqcup (-P_{i+1})^T).$$

Thus all the new W_i may be calculated using of the order n multiplications. After each multiplication the diagonal elements, e , are set to $\min(e, 0)$.

Comparing W_i to the corresponding P_i indicates all of the conversions which could never contribute to a solution which tied or bettered the current best. As well as these conversions, we may exclude all conversions already added to the compatibility list in order to lengthen the list of NEVERs by calculation of an even stronger set of W_i . Conversions may already exist in the compatibility list for one of two reasons: they may have been selected as part of the optimal solution, or they may have been found to be MUSTs. The P_i used in calculation of the new W_i are modified in the following way. The elements which correspond to the selected conversions are set to infinity; if one of these conversions were modified, that fact alone would cause recompilation and so any MAYBEs which depend on that can safely be considered NEVERs. Second, the elements of a P_i which correspond to MUSTs are set to the cost of the next most expensive conversion. That

is, if a widening is a MUST, the corresponding P_i is set to the narrowing cost; if a narrowing is a MUST, the corresponding P_i becomes infinite.

After this refinement, the set of conversions which have been neither selected nor rejected has the following property. The set M contains all conversions, C , such that for each such C there exists a set of other conversions, S , such that the cost of the optimal solution with S added to the environment is not less than the original cost but if S and C are added the optimal solution is less than or equal to the original cost. Thus none of the conversions in M causes recompilation by itself but each could in combination with other elements of M .

The above statement assumes that the conversions in each position of the parse tree can be present or absent independently. In fact, all conversions between the same pair of types must be present or absent in common.

4.15.1.4 FURTHER REDUCTIONS TO THE COMPATIBILITY LIST

Further MAYBES can be eliminated from the list. For example, any single MAYBE may be considered a NEVER as it will never necessitate recompilation by itself. After this elimination, the C matrices would have to be updated to assume that this conversion was present, and the set of MUSTs recomputed. Then another MAYBE could be eliminated. This technique would reduce the size of the compatibility list but would not necessarily lead to the shortest possible one. A second problem is that re-evaluation of the U matrices could consume a great deal of time.

In order to approach a minimal solution, the order of elimination may be chosen using various heuristics. Speed can be enhanced by noticing that eliminating a conversion at one position cannot cause another conversion at the same position (represented as an element in the same matrix) to become a MUST. Thus, all the conversions at a particular position could be eliminated at one time. However, conversions which act as MAYBES in more than one position could not be eliminated.

4.15.1.5 A DUAL APPROACH TO REDUCTION

There is a dual approach to reducing the list which is not so sensitive to conversions which appear in more than one place. That approach is to select one or more MAYBES and add them to the compatibility list as if they were MUSTs. The P matrices may then be updated as described earlier, and the new Ws calculated. These new Ws could create new NEVERs. This process is continued until there are no more MAYBES. The size of the resulting compatibility list is again dependent on the order in which the conversions are selected. Once again, conversions may be selected in batches. However, there is no restriction that these batches of conversions be picked from the same position in the parse tree. Furthermore, it is beneficial to select conversions which appear in more than one place.

Nevertheless it is still desirable to select all the conversions at one place. One reason is that selecting a conversion could not cause another at the same position to become a NEVER. Another reason is that a matrix of conversions could be represented compactly as a single entity in the compatibility list. Furthermore, changing all the

elements in a P is more likely to affect many W s and hence cause more new NEVERs than making the same number of changes at random positions.

The order in which the conversion matrices are selected for addition to the compatibility matrix is still the subject of experimentation, but selecting that position where the most expensive conversion is used seems to be a reasonable choice.

4.15.2 FUNCTIONS WITH MORE THAN ONE PARAMETER

The previous sections described the representation of an expression as a graph consisting of groups of vertices. The edges on the graph connected only members of adjacent groups. The groups of vertices formed a line themselves because the original parse tree was a line.

In general, the graph of an expression is a tree. We can represent the selection graph as a set of groups of vertices. These groups are arranged in the same structure as the parse tree for the expression. Each group contains one vertex for each possible parameter or result type. Between each adjacent result type and formal parameter type an edge exists if the corresponding conversion exists. Groups of parameters and result types which correspond to an existing function are connected by an "edge". This "edge" is not really an edge as it connects more than two vertices. It is, however, represented by an entry in an n -dimensional matrix, C , where n is the number of parameters plus the number of results.

The selection process consists of solving a tree-like equation involving these n -dimensional matrices. In order to manipulate the

multi-dimensional matrices, generalized matrix multiplication and inversion are now discussed.

4.15.2.1 MATRIX MULTIPLICATION IN N DIMENSIONS

In this section, a generalized matrix operator is discussed. The syntax of this compound operator, $\circ$, is

```

 $\circ ::= \langle \text{undotted} \rangle$ 
 $\circ ::= \langle \text{dotted} \rangle$ 
 $\langle \text{dotted} \rangle ::= . \langle \text{undotted} \rangle$ 
 $\langle \text{undotted} \rangle ::= \langle f \rangle$ 
 $\circ ::= \langle f \rangle \langle \text{dotted} \rangle$ 
 $\langle f \rangle ::= + \mid - \mid * \mid M \mid X \mid \dots$ 

```

The semantics of $\circ$ are as follows.

In evaluating the expression $A \circ B$, we first compute the value R as defined by:

$$R(i_1, i_2, i_3 \dots i_n, 1, j_1, j_2, j_3 \dots j_m) = A(i_1, i_2, i_3 \dots i_n)$$

$$R(i_1, i_2, i_3 \dots i_n, 2, j_1, j_2, j_3 \dots j_m) = B(i_1, i_2, i_3 \dots i_n)$$

That is, R has one more dimension than the sum of the number of dimensions of A and B . The list of dimensions of R is the list of dimensions of A concatenated with a 2 followed by the list of dimensions of B . The extra dimension is referred to as the CENTRE dimension.

Whenever a $\langle \text{undotted} \rangle$ is recognized (the rightmost operator is recognized first), the operation f is applied across the centre dimension of R , thus eliminating that dimension. Whenever a $\langle \text{dotted} \rangle$ is recognized, the two dimensions adjacent to where the centre used to be

are combined by taking the diagonal into a new centre dimension. This process is continued until 0 has been recognized. At this time R represents the result of $A \circ B$.

Using APL, these three steps may be expressed as follows.

I.
$$R \leftarrow ((1 \rho \rho A), (1 + \rho \rho A) + 1 \rho \rho B), 1 + \rho \rho A \quad \text{Q} \quad ((\rho A), (\rho B), 2) \rho (, A), , B$$

II. for every function, f

$$R \leftarrow f / [2 + (\rho \rho A) - I] R$$

for the I^{th} f from the right of 0

III. for every .

$$R \leftarrow ((1 + (\rho \rho A) - I), ((\rho \rho A) - I) + 1 (\rho \rho B) - I) \quad \text{Q} \quad R$$

for the I^{th} . from the right of 0

The operation $+.*$ is normal matrix multiplication. $_{-}.+$ and $|.+$ are the same as have been used previously in this chapter. However, these operators may be used on matrices of higher dimension. It should be noted that although the above semantic description is convenient, an actual multiplication may be optimized to avoid generating the large intermediate result, R . For example the function $R = A F.G.H B$ is

$$R(i_1, i_2 \dots i_{n-2}, j_3, j_4 \dots j_m) =$$

$$\begin{matrix} F & G & A(i_1, i_2 \dots i_{n-2}, p, q) & H & B(q, p, j_3, j_4 \dots j_m) \\ p & q & & & \end{matrix}$$

for A n dimensional and B m dimensional.

The minimal cost of selecting functions in an expression may be expressed as an equation using matrix products. A notation is now presented so that such equations may be written.

4.15.2.2 A NOTATION FOR MULTI-DIMENSIONAL MATRIX EQUATIONS

A multi-dimensional matrix is represented by a symbol such as M . Each dimension of M is represented by a dot. A column vector is denoted $*M$ and a row vector M^* . The dot indicates that these vectors may be multiplied on the left and on the right, respectively. When a product is formed, the operation is represented by a line joining two dots. For example, the dot product of row vector A and column vector B is

$$A \text{----} B$$

The line merely shows the structure of the expression. In order to specify which operation is performed, we may write the operator above the line.

$$\begin{matrix} +.* \\ A \text{-----} B \end{matrix}$$

In this case, the product is a scalar, as indicated by the fact that there are no remaining dots.

A conventional matrix is represented as $*M*$. The positions of the dots indicate that $*M*$ may be multiplied either on the right or on the left. The various conventional multiplications are listed in the following table.

$*M---V$ (matrix times column vector yields column vector)

$V---M*$ (row vector times matrix yields row vector)

$*M---M*$ (matrix times matrix yields matrix)

In addition to matrix multiplications, outer products as defined in APL may be formed. These products are represented by a single simple operator and do not consume any dots. For example, the outer product of two matrices yields a four dimensional result:

$* * *$
 $*A+B*$

The compound multiplications discussed earlier may also be applied.

$A \overline{\overline{L.F.}} + B$

would be represented by

$\overline{\overline{L.F.}}$

$A===B$. The double line consumes two pairs of dots. As the result has no dots, we can see that it is a scalar. The order in which the dimensions are used is: the right most dimension of the left operand and the left most dimension of the right operand are combined first; then the dimensions of the right operand are used counter clockwise and of the right operand, clockwise.

Objects of larger dimension are represented using a dot for each dimension. These dots are arranged in a semicircle above the symbol. A three dimensional matrix would be

$$\begin{array}{c} * \\ *M* \end{array}$$

a four dimensional matrix would be

$$\begin{array}{c} * * \\ *M* \end{array}$$

and so on. Multiplication of two three dimensional matrices yields a four dimensional result:

$$\begin{array}{c} * \quad * \\ *A---B* \end{array}$$

An outer product yields a six dimensional result:

$$\begin{array}{c} **** \\ *A \quad B* \end{array}$$

A double multiplication yields a two dimensional matrix:

$$*A==B*$$

Multi-dimensional matrices may be multiplied in other orientations:

$$\begin{array}{c} * \\ *B \\ | \\ *A* \end{array}$$

or

$$\begin{array}{c} * \\ B* \\ | \\ *A* \end{array}$$

or

$$\begin{array}{c} * * \\ B-A* \\ * \end{array}$$

In these examples, the orientation of B may be deduced from the positions of the dots and lines. However, consider the case of a two dimensional matrix, M, being multiplied by a three dimensional matrix, A.

$$\begin{array}{c} * \\ M \\ | \\ *A* \end{array}$$

It is clear that M is being multiplied by the centre dimension of A. However, we cannot determine whether M is being multiplied on the right or on the left by A. In order to resolve this ambiguity, we can, wherever necessary, mark the bottom of a matrix using an arrow which points from the first dimension to the left. That is, a three dimensional matrix in normal orientation would be:

$$\begin{array}{c} * \\ *A* \\ \rightarrow \end{array}$$

The above expression could then be written as

$$\begin{array}{c} * \\ M \uparrow \\ | \\ *A* \\ \rightarrow \end{array}$$

or as

$$\begin{array}{c} * \\ \downarrow M \\ | \\ *A* \\ \rightarrow \end{array}$$

the first representing M on the right and the second, M on the left.

This notation is also used to represent the transpose of a matrix.

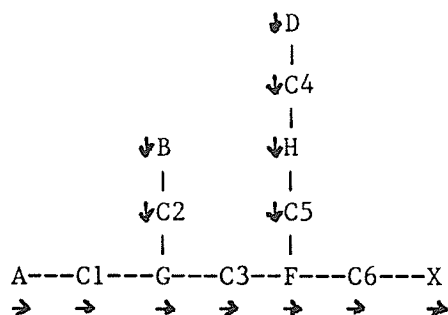
$$\begin{array}{c} T \\ A = A \\ \rightarrow \quad \leftarrow \end{array}$$

4.15.2.3 REPRESENTING THE SELECTION PROCESS AS A MATRIX EXPRESSION

Consider the following expression.

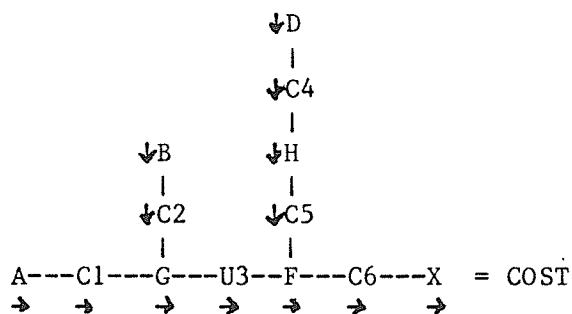
$$x := f(g(a,b),h(d))$$

The cost of selection can be calculated by evaluating



where X, F, G, H, A, B, and D are the C matrices representing x, f, g, h, a, b, and d respectively, and C1 through C6 represent the appropriate conversions.

In addition to the C matrices, the U matrices may be computed. For example, U3 is determined by solving the matrix equation



which is equivalent to the equation

$$L3 \text{---} U3 \text{---} R3 = L3 \text{---} C3 \text{---} R3$$

where

$$\begin{array}{c}
 \downarrow B \\
 | \\
 \downarrow C2 \\
 | \\
 L3 = A \text{---} C1 \text{---} G^* \\
 \rightarrow \quad \rightarrow \quad \rightarrow
 \end{array}$$

and

$$\begin{array}{c}
 \downarrow D \\
 | \\
 \downarrow C4 \\
 | \\
 \downarrow H \\
 | \\
 \downarrow C5 \\
 *F \text{---} C6 \text{---} X \quad . \\
 \rightarrow \quad \rightarrow \quad \rightarrow
 \end{array}$$

Multidimensional U_s which indicate which functions are involved in the optimal solution may also be calculated. For example, the matrix U_f which corresponds to the F is calculated.

$$\begin{array}{c}
 \downarrow D \\
 | \\
 \downarrow C4 \\
 | \\
 \downarrow B \quad \downarrow H \\
 | \quad | \\
 \downarrow C2 \quad \downarrow C5 \\
 | \quad | \\
 A \text{---} C1 \text{---} G \text{---} C3 \text{---} U_f \text{---} C6 \text{---} X = \text{COST} \\
 \rightarrow \quad \rightarrow \quad \rightarrow \quad \rightarrow \quad \rightarrow \quad \rightarrow \quad \rightarrow
 \end{array}$$

The above equation is equivalent to

$$\begin{array}{c}
 \downarrow L2f \\
 | \\
 L1f \text{---} U_f \text{---} Rf = \text{COST} \quad . \\
 \rightarrow \quad \rightarrow \quad \rightarrow
 \end{array}$$

Solving for U_f we have

$$\begin{array}{c}
 * \\
 \uparrow (-L2f) \\
 + \\
 U_f = *(-L1f) + \text{COST} + (-Rf)* \quad . \\
 \leftarrow \quad \quad \leftarrow
 \end{array}$$

All of the L vectors may be computed in advance. After the L s have been computed, the R s may be computed as well. As an example, consider the

computation of L3 and R3.

$$\begin{array}{c}
 \downarrow B \\
 | \\
 \downarrow C2 \\
 | \\
 L3 = A \text{---} C1 \text{---} G^* \\
 \rightarrow \quad \rightarrow \quad \rightarrow \\
 \\
 \downarrow L2g \\
 | \\
 L3 = L1g \text{---} G^* \\
 \rightarrow \quad \rightarrow
 \end{array}$$

L3 has been recursively defined in terms of other L which are products of fewer matrices. Generalizing this approach allows us to calculate all of the Ls.

R3 can be computed in a similar manner.

$$\begin{array}{c}
 \downarrow L2f \\
 | \\
 R3 = *F \text{---} Rf \\
 \rightarrow \quad \rightarrow
 \end{array}$$

Note that L is defined recursively independently of R but R depends on the value of L for its recursive definition.

In order to minimize the compatibility list it is necessary to calculate the W. Each W_i is computed by solving this equation:

$$W_i = B_i \text{---} C_i \text{---} T_i$$

where B_i and T_i are defined recursively. As an example, the calculation of W_3 is given.

$$\begin{array}{c}
 W3 = B3 \text{---} C3 \text{---} T3 \\
 \rightarrow \quad \rightarrow \quad \rightarrow \\
 \\
 B3 = \text{MIN} \left(\begin{array}{c} \uparrow \quad \uparrow + \\ *(-G) \text{---} Z^* \\ \rightarrow \quad \rightarrow \end{array} , I \right)
 \end{array}$$

$$\begin{array}{c}
 * \\
 \downarrow B2g \\
 | \\
 Z = \begin{array}{c} *B1g \text{ --- } G* \\ \rightarrow \quad \rightarrow \end{array} \\
 \\
 T3 = \text{MIN}(\begin{array}{c} \uparrow \quad \uparrow + \\ *Y \text{ ===== } (-F)* \\ \rightarrow \quad \rightarrow \end{array}, I) \\
 \\
 * \\
 \downarrow B2f \\
 | \\
 Y = \begin{array}{c} *F \text{ --- } Tf* \\ \rightarrow \quad \rightarrow \end{array}
 \end{array}$$

Because the multidimensional matrices are sparse and contain only two possible values, no such matrices need be formed in evaluating the above expressions. A program which computes the minimal compatibility list as defined in this chapter has been implemented using PL/I.

4.16 CONCLUSIONS

Although the algorithm to reduce the compatibility list is complex for the general case, the program which implements the algorithm used neither an excessive amount of CPU time nor an excessive amount of storage in several test runs. If a less complex solution is required, many subsets of the algorithm could possibly be used. The most simple is to use TRUE for the logical expression. This approach is no worse than that used currently. Another simple approach is to include in the compatibility list only those conversions whose cost is less than the overall cost of the expression when compiled. The weaker version of the W_i may be computed quite simply without requiring matrix storage and

could be added to the original selection algorithm. Also, the approach of selecting MAYBES to become MUSTs may be applied in combination with any of these less powerful approaches.

Chapter V

CONCLUSIONS

5.1 IMPLEMENTATION AND USE OF L

All of the major components of a type-secure language and of a compilation system which allows the programmer to develop abstractions have been presented. In all cases, design decisions have been made with straightforward implementation in mind. Furthermore, abstractions and separate compilation can be used to implement many of the features which are usually built into a compiler, greatly reducing its complexity. It is essential at this point in programming language evolution that systems incorporating these new features be implemented so that programmers may develop a methodology which makes use of these features. Only in the light of this experience may these ideas be fully judged and then improved. Some of the aspects of L, such as the procedure selection algorithms discussed in chapter 4, have been implemented. The implementation of L will be completed in order that research into the use of L may proceed; it is the use rather than implementation that will be significant future research.

5.2 BLOCK STRUCTURE

L is a block-structured language. Conventional block structure has come under criticism recently for three major reasons.

1. In order that two procedures may share data, they must share that data with the rest of the world.
2. In order for a variable to be persistent, it must be global, but if it is global, it is freely accessible to every procedure.
3. Nested procedures cannot be separately compiled.

On the other hand, the following points in favour of block-structure exist.

1. A hierarchical nesting of environments is natural; we have, for example, an operating system, several sub-systems, a user environment, the user's global data, and the user's local data.
2. It is impossible to do away with sharing of data. Any two procedures which interact must share a common environment. At the very least, they must share the definitions of the types of their parameters.

It is felt that L takes advantage of block structure while addressing the three drawbacks. The MODULE facility restricts the visibility of names shared among a set of routines; the separation of a variable's visibility and its duration addresses the second criticism; and the separate compilation system allows nested procedures to be recompiled. Furthermore, the separate compilation system encourages the hierarchical

approach to environments by allowing user libraries and therefore local modifications to programs.

This argument is not meant to imply that many of the features presented in this thesis could not be applied to non-block-structured languages. The separate compilation system could be used for a language such as Ada, Mesa, or Protel, where program segments can import declarations from others. The compatibility analysis can be used to prevent all segments which import information from another (exporting) segment from having to be recompiled when that segment is recompiled. An ODB very similar to that presented in chapter 2 would be used with the difference that each segment would not necessarily have a unique parent. There would have to be one father pointer, and one FTS for every segment imported. The compatibility checking algorithm would be applied to every father of the segment. Reaching the same module through different paths would not be a problem; only the first time a segment is reached could compatibility maintenance be applied as the TS would be updated at this time.

The procedure, parameter, and typing mechanisms could be included as extensions to many languages. So could operator overloading and automatic conversion. The concept of DURATION, however, would have to be modified. The durations LOCAL, GLOBAL, and PERMANENT would still be meaningful (and useful). However, specifying intermediate durations would be more difficult. In order to specify an intermediate duration the programmer could specify the name of a procedure whose duration would be the same as his variable. Such a specification would be valid

only if, when the call graph was examined, it were found that any path by which the procedure containing the variable declaration contained the procedure whose duration was selected. It is not obvious that such an intermediate duration would be useful.

5.3 OPTIMIZATION

As the programmer uses more levels of abstraction, the number of procedures called for each statement compiled will increase. At the same time, the number of features which are built into the compiler decreases. Because of the number of procedure calls, any significant optimization technique will have to work across procedure boundaries, and decrease the linkage overhead of calling procedures.

Two approaches may be pursued. One may add extra information (similar to compatibility information) to the object modules in order to facilitate optimization within procedures. Or one may do optimization at binding time yielding an executable code which is no longer eligible for separate compilation.

The first approach has been investigated by [CON81], who suggests what information should be maintained about each compilation unit. The information already contained in the ODB forms a sizable subset of this information. However, if we allow the optimizer to make assumptions about procedures which another calls, then any change to the executable code of a procedure could necessitate recompilation of all procedures which call that procedure. A timestamp approach could be developed to prevent the traversal of the entire graph. This approach would be

similar to the timestamp approach used in the ODB, which prevents traversal of the static visibility graph.

But such optimization will be of limited use; there is little point in, say, allocating registers optimally, when linkage to procedures will cause storage and reloading of the register set many times. A more promising approach is to generate in-line code for the many levels of called procedures. Unfortunately, the generation of in-line object code precludes separate compilation; if a procedure has been generated in line, it can no longer be replaced by the linker. The Mesa language provides an in-line procedure facility. In Mesa, procedures must be declared as "in-line" at their definition site. The body of an in-line procedure becomes part of its type; any change causes recompilation of all units to whom the in-line procedure is visible. The compatibility analysis necessary to recompile only those procedures which actually used the in-line procedure could be easily added to the ODB.

If this information is to be added to the ODB, we may free the programmer of having to specify at the definition site whether or not a procedure is in-line. At the call site, the programmer may ask to have object code generated in line for any procedure. Or, the optimizer could decide when the time-space tradeoff would justify in-line code. If maintenance were likely to be applied frequently, the programmer could request that no object code be generated in line, thereby making his program as maintainable as possible. Finally, if a procedure is generated in line, the optimizer may take advantage of any aspect of that procedure whatsoever; if any change is made the program will be recompiled and the optimization re-done.

The decision whether to load dynamically a procedure when it is called, or to link it with the calling routine, is a more global instance of the same in-line optimization technique. If a procedure is linked in, the program must be re-linked in order to be maintained; if a procedure is loaded dynamically, it will benefit from maintenance immediately. In many systems, this decision has to be made by the programmer; he must indicate at the source level which technique is to be used. It has been mentioned in chapter 2 that the linker/loader can make this decision automatically, making use of the call graph to which it has access. Whichever method is used, the effect is the same.

If the proper object code were generated by the compiler, the linker could also make the appropriate decisions as to which procedures should be called in line. This approach would remove the need for extra compatibility information; the program is re-linked whenever a modification is made. The design of an appropriate object code to make such optimization easy should be investigated.

REFERENCES

- ANS66 USA Standard FORTRAN. ANSI, USAS X3.9-1966, New York, 1966.
- BER78 Bert D., Jaquet P., "Some Verification Problems with Parameterized Types and Generic Functions", Proc. Third International Symposium on Programming. Dunod, Paris, 1978.
- BOU74 Bourne S., Birrell A., Walker I., Algol 68C Reference Manual. Cambridge University, 1974.
- CON81 Conradi R., Inter-procedural Optimization of Object Code. Carnegie Mellon University, Pittsburgh, 1981.
- DAH70 Dahl O.J., Myhrhaug B., Nygaard K., The SIMULA 67 Common Base Language. Norwegian Computing Centre, Publication S-22, Oslo, 1970.
- DAV78 Davio M., Deschamps J., Thayse A., Discrete and Switching Functions. McGraw Hill, New York, 1978. pp. 546-587
- FOX79 Foxall D.G., et al., 'Protel: A High Level Language For Telephony", Proc. 3rd International Computer Somftware and Applications Conference. November 1979.
- ICH79a Ichbiah J.D., et al., "Rationale for the Design of the Ada Programming Language", SIGPLAN Notices. V14, #6, June 1979.
- ICH79b Ichbiah J.D., et al., "Preliminary Ada Reference Manual", SIGPLAN Notices. V14, #6, June 1979.
- ICH80 Ichbiah J.D., et al., Reference Manual for the Ada Programming Language. U.S. Department of Defense, Washington D.C., 1980.
- IBM75 OS/VS-DOS/VS-VM/370 Assembler Language. IBM, Manual GC33-4010, 1975
- IBM76 IBM, OS PL/I Checkout and Optimizing Compilers: Language Reference Manual. IBM, Manual GC33-0009, 1976.
- IBM78 APL Language. IBM, Manual GC26-3847, 1978.
- IBM80 OS/VS System Modification Program System Programmer's Guide. IBM, Manual GC28-0673, 1980.
- IVE62 Iverson K.E., A Programming Language. Wiley, New York, 1962.

- JOL81 Joliat M., "Experience with a Modular Type Language: Protel", Proc. Fifth Conference on Software Engineering. IEEE, Long Beach, 1981.
- KOS75 Koster C.H.A., "The Mode System in Algol 68", New Directions in Algorithmic Languages. IRIA, Paris, 1975.
- LAS79 Lasker D.M., "Module Structure in an Evolving Family of Real Time Systems" Proc. Fourth Conference on Software Engineering. IEEE, Long Beach, 1979.
- LAU79 Lauer H, Satterthwaite E, "The Impact of Mesa on System Design", Proc. Fourth Conference on Software Engineering. IEEE, Long Beach, 1979.
- LIS75 Liskov B., "An Introduction to CLU", New Directions in Algorithmic Languages. IRIA, Paris, 1975.
- LIS79 Liskov B., et al., CLU Reference Manual, MIT, 1979.
- MCL65 McCluskey E.J., Introduction to the Theory of Switching Circuits. McGraw Hill, New York, 1965. pp. 114-178
- MIT79 Mitchell J., et al., Mesa Language Manual. Xerox, Palo Alto, 1979.
- NAU62 Naur, P.(ed.), Revised Report on the Algorithmic Language Algol 60. Regnecentralen, Copenhagen, 1962.
- PER80 Persch G., et al., "Overloading in Preliminary Ada", Proc. ACM-SIGPLAN Symposium on the Ada Programming Language. ACM SIGPLAN Notices, V15, #11, November 1980.
- SIT72 Sites R.L., Algol-W Reference Manual. Stanford University, Manual STAN-CS-71-230, Stanford, 1972.
- STE81 Steensgaard-Madsen J., "A Statement Oriented Approach to Data Abstraction", Transactions on Programming Languages and Systems. ACM, Vol. 3, No. 1, January 1981.
- WIJ76 van Wijngaarden A., et al., Revised Report on the Algorithmic Language Algol 68. Springer Verlag, New York, 1976.
- WEG74 Wegbreit B., "The Treatment of Data Types in ELI", CACM. V17, #5, May 1974.
- WEI67 Weissman C., LISP 1.5 Primer. Dickenson, Belmont, 1967
- WIR71 Wirth N., "The Programming Language Pascal", Acta Informatica. V1, #1, 1971.