CMU-CS-86-124

A Survey of Language Support for Programming in the Large

Thomas D. Newton

Department of Computer Science
Carnegie-Mellon University
Pittsburgh, Pennsylvania 15213

8 May 1986

Abstract

The support provided by a number of programming languages for the activity of "programming in the large" is examined, and their features are categorized with respect to decomposition of a system, import/export mechanisms, separate compilation, and version/configuration control. A comparison is made using this categorization. Eighteen languages are surveyed; ranging from Simula-67 to Modula-2 to Ada to BCPL, they exhibit a number of design philosophies.

Supported by the Defense Advanced Research Projects Agency, Department of Defense, ARPA Order 3597, monitored by the Air Force Avionics Laboratory under contract F33615-81-K-1539. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the U.S. Government.

Table of Contents

List of Figures

List of Tables

1. Introduction

Modular programming plays an important role in modern software engineering methodologies. Although modular programs can be written in any language, the full benefits of modular programming cannot be gained without at least some support from the programming language(s) in use. This paper examines the facilities provided by several languages with respect to the issues of "programming-in-the-large".

The particular languages chosen for study in this report are all typically compiled (as opposed to languages such as APL or Lisp) and include FORTRAN, ALGOL-60, BCPL, BLISS-11, Modula, Modula-2, Pascal, three extended Pascals, C, C++, TARTAN, Simula-67, PROTEL, CHILL, Mesa, and Ada. These are by no means the only "compiled" languages, but they illustrate several types and levels of support for the development of large programs. Except for cases where a language is strongly tied to a particular implementation, the features of any one compiler will not be discussed; however, the interaction between language features and implemention-level details may be discussed where it seems appropriate.

Several papers have presented comparisons that are similar in some ways to the one done in this report. Shaw *et al* ([Shaw 81]) compared FORTRAN, COBOL, Jovial, and the proposed (at the time) "Ironman" standard in the light of modern software engineering ideas. Their paper clearly identifies a number of issues that address the logical structure of large programs; however, the only "modern" language it covers is the preliminary version of Ada. Tichy ([Tichy 79]) categorized several languages with respect to the order in which they require type checking and compilation. His paper does not describe any language features that support modularization and information-hiding because he feels that these mechanisms belong in separate modular interconnection languages like the one described in his report. Applebe and Hansen ([Applebe 85]) surveyed a number of current programming languages with regards to their utility for systems programming. Although their report addresses the issues of encapsulation and environment support, it does not go into much detail, presumably since it also covers topics such as types and hardware environment specification.

The remainder of this paper is divided into four major sections. Section 2 contains definitions of some of the terms used in this paper together with some background information. Section 3 briefly describes each surveyed programming language. Section 4 categorizes and describes the mechanisms provided by the surveyed languages which are related to the support of programming-in-the-large; its results are briefly summarized in several tables which comprise section 5. Finally, section 6 wraps up the paper.

2. Definitions

There is a large body of literature supporting the idea that developing large programs is an activity that is both qualitatively different from and harder than developing small programs. As the number of programmers working on a project grows, communication and coordination costs consume an ever-increasing amount of the work effort, until a point is reached at which adding more programmers actually delays the project's completion ([Brooks 79]). Maintaining large programs is more difficult than maintaining small ones because humans have only a limited capacity to keep track of details.

As noted by Shaw *et al* ([Shaw 81]), modern software engineering methodologies share the view that it must be possible to *decompose* a project into smaller pieces that can be completed independently, *assemble* the results into a consistent, working system, and *maintain* the resulting system over a fairly long period of time. Programming languages and their compilers can provide support for these tasks in the form of support for modularization, information hiding, interface checking, separate compilation, reduction of recompilation costs, and version/configuration control.

The idea of *modular programming*, or breaking a system into modules which communicate through well-defined interfaces, dates back at least as far as 1970. Parnas ([Parnas 71a]) introduced the idea of *information hiding* in 1971. At the time, information hiding was often accomplished by hiding system documentation, but today the term is more commonly used when describing language features useful for producing the same effect of isolating design decisions and making them easier to change. DeRemer and Kron ([DeRemer 76]) introduced the terms *programming-in-the-large*, *programming-in-the-small*, and *modular interconnection language* (or *MIL*) to refer to the activity of putting together a collection of modules to form a system, the activity of programming each module, and the language used to describe the structure of a system, respectively.

Although some people take the point of view that modular interconnection languages should be independent of normal programming languages, it seems obvious that at least some of an environment's support for programming-in-the-large can only be feasibly provided by its compilers, and preferably should be provided by the programming language(s) in use. Tichy ([Tichy 79]) claims that it is "inappropriate to burden the languages for programming-in-the-small with them [information-hiding features]," but admits that "nevertheless, features like opaque record fields and write-protection must be supported by the languages themselves so that a compiler can enforce the restrictions on the required resources." Thus, languages which attempt to address the problems of programming-in-the-large (such as Mesa and Ada) are likely to be more suitable for writing large programs whether their modular interconnection facilities are used directly by the programmer or indirectly through a specialized tool.

3. The Languages

This section briefly describes each language surveyed for this report. The intent of each description is to provide the reader with an impression of the general "flavor" of a language, the special features of the language, and the circumstances surrounding its design. Thus, the descriptions may mention some features that are not directly relevant to programming-in-the-large; features which are relevant to programming-in-the-large are discussed in more depth in section 4. For more information on any of these languages, see the reference work(s) cited under the appropriate description.

3.1. FORTRAN

FORTRAN was one of the first "high-level" languages. It was designed to be able to express numeric computations symbolically. While it provides a set of numeric types and a way to declare multi-dimensional arrays of numeric values, it does not provide records, pointers, or any other method of declaring new types. In the way of control structures, it supplies a DO loop, a GOTO statement, a computed GOTO, and an IF statement (limited to executing one statement if its condition is satisfied). FORTRAN provides subroutines and functions, but they may not be recursive. A FORTRAN program consists of one or more separately-compiled subroutines; communication between them is done through subroutine calls or through COMMON blocks. FORTRAN was and still is widely used. Over the years it has evolved; the latest version (FORTRAN-77) has several features which are not in the version just described (the commonly-used FORTRAN IV).

Reference: *FORTRAN and WATFIV Language Manual*, by C. William Gear, Science Research Associates, Inc., 1978.

3.2. ALGOL-60

ALGOL-60 is an early programming language which has, directly or indirectly, influenced many of the programming languages in common use today. ALGOL was designed by an international committee with the intent that it would serve as an international standard. The preliminary report (prepared by committees of the ACM and the GAMM) appeared in 1958, the initial language report (by IFIP) in 1960, and a revised report in 1962. Although ALGOL-60 was never widely used, it is important because of its influence on other languages.

ALGOL-60 is a fairly sparse language. Like FORTRAN, it supplies a few predefined types and allows the declaration of multi-dimensional arrays, but doesn't support enumeration types, records, pointers, or dynamic allocation of memory. However, its control structures are much better than FORTRAN's.

It has an IF..THEN..ELSE statement, a FOR statement (which provides both counted loops and while loops), and a somewhat arcane way of selecting between multiple alternatives using GOTO and arrays of SWITCH (label) expressions. It supports blocks which consist of sequences of local declarations followed by sequences of statements. Procedures and functions can be called recursively, although ALGOL's call-by-name mechanism has proven to be less popular (in terms of the number of later languages which adopted it) than FORTRAN's call-by-reference mechanism.

Reference: *Introduction to ALGOL Programming*, by Torgil Ekman and Carl-Erik Froberg, STUDENTLITTERATUR, Lund, Sweden, 1967. (One appendix in this book is the *Revised Report on the Algorithmic Language ALGOL 60*, by Backus *et al*, IFIP, 1962.)

3.3. Simula-67

Simula-67 is a general-purpose programming language which was designed by O-J. Dahl, B. Myhrhaug, and K. Nygaard while they were staff members at the Norwegian Computing Centre in Oslo. It is meant to be of special use in writing simulation programs, and is "based on ALGOL 60 with the addition of record-oriented dynamic memory allocation, reference (pointer) structures, sets and queues, text- and character handling, sequential and direct access input-output, quasi-parallel sequencing (coroutines) and process (event) oriented simulation capabilities." One of Simula's most important features is its CLASS facility, which is useful for defining abstract data types and which has influenced other languages such as Smalltalk and C++. The DECSystem-10 implementation of the language includes a separate compilation facility based around PROCEDURES and CLASSES.

References: *DECsystem-10 SIMULA Language Handbook Part I* (second edition), by Graham Birtwistle and Jacob Palme, Swedish National Defense Research Institute, Sweden, 1975. *Simula Begin*, by G. M. Birtwistle *et al*, AUERBACH Publishers Inc., Philadelphia, Pa., 1973.

3.4. BCPL

BCPL was designed by Martin Richards as a compiler-writing tool and has been used for systems programming in general. It is much more "machine-oriented" than Simula-67 or FORTRAN; the machine word is its only data type, and the meaning of an expression depends only upon the context in which it is used. BCPL provides a number of bit-level operators, operators for taking the address of a variable and for indirectly accessing a memory location, and STRUCTURE declarations for associating names with "partial-word fields of variables, and individual words and partial-words of vectors." It also has a set of control structures similar to those in ALGOL.

References: *BCPL Reference Manual*, compiled by James E. Curry and the PARC staff, Computer

Sciences Laboratory, Xerox PARC, 1979. *BCPL -- The Language and its Compiler*, by Martin Richards and Colin Whitby-Strevens, Cambridge University Press, New York, 1979.

3.5. BLISS-11

BLISS-11 is a systems programming language for the PDP-11. It is related to BLISS-10, and is very similar to BCPL; the basic data type is a 16-bit word, and the language has a number of bit-level operators, a way of imposing "higher-level" structures on memory by associating names with address calculations, and a simple separate compilation facility. The most important contribution of the BLISS effort was not the language itself, but several optimization techniques which were used in its compiler (BLISS had a very highly optimizing compiler) and documented in technical reports.

References: *BLISS-11 Programmer's Manual*, published by Digital Equipment Corporation, Maynard, Massachusetts, 1972. *The Design of an Optimizing Compiler*, by Wulf *et al*, American Elsevier Publishing Company, Inc., New York, 1975.

3.6. Pascal and three variants

Pascal is a teaching language designed by Wirth which was influenced by Algol-60 and Algol-W, and which in turn has influenced several more recently developed languages. It postdates Algol-60 by about ten years; the first publication of the language was in 1971, and a revised report came out in 1973. Like ALGOL-60, Pascal is block-structured and has structured control statements. However, Pascal offers much more in the way of user-defined data types than ALGOL-60: in addition to arrays, it offers subranges, enumeration types, sets, records, and pointers (with dynamic storage allocation). Unlike almost all of the other surveyed languages, Pascal does not support separate compilation. This has been a motivating factor behind many efforts to extend the language; three extended versions of Pascal are described below.

Reference: *Pascal User Manual and Report* (second edition), by Kathleen Jensen, Springer-Verlag, New York, 1978.

PERQ Pascal is a variant of Pascal that runs on the PERQ workstations formerly manufactured by PERQ Systems, Inc. Unlike standard Pascal, it supports separate compilation -- programs always contain one PROGRAM file and may contain any number of separately-compiled MODULES. Each MODULE consists of a set of publically-readable declarations followed by the bodies of exported subroutines and all private declarations. PERQ Pascal also provides an exception-handling facility and a STRING data type.

Reference: *PERQ Pascal Extensions, Spice Programmers' Manual*, Department of Computer Science, Carnegie-Mellon University, 1984.

Celentano *et al* describe a variant of Pascal which supports a different style of separate compilation. Under their scheme, each MODULE declares the imported items which it uses and can thus be compiled independently of all other MODULES. The position of a particular MODULE within the tree representing a program is determined entirely at link time; this is meant to enhance reusability. In addition, it is possible to "partially specify" imported items (declaring objects to be of unknown type, or types to be of unknown representation) so long as the union of all the declarations for a given item form a complete, consistent declaration.

Reference: *Separate Compilation and Partial Specification in Pascal*, by Augusto Celentano *et al*, *IEEE Transactions on Software Engineering*, vol. SE-6, no. 4, July 1980.

Le Blanc and Fischer report on a modification to the UW-Pascal (University of Wisconsin Pascal) compiler to make it support separate compilation. The modification allows the top-level declaration of procedure stubs and separate compilation of procedure bodies. The scheme involves saving the compiler's run-time state to disk when processing a main program and restoring it when processing separately-compiled bodies, and the authors suggest that the technique might be of use in other compilers for block-structured languages. The choice to modify an existing Pascal compiler rather than to switch to a language with separate compilation facilities seems to have been driven by a need for backwards compatability.

Reference: *A Simple Separate Compilation Mechanism for Block-Structured Languages*, by Richard J. Le Blanc and Charles N. Fischer, in *IEEE Transactions on Software Engineering*, vol. SE-10, no. 3, May 1984.

3.7. C and C++

C is a general-purpose programming language which was originally designed by Dennis Ritchie for Unix on the PDP-11. Its major features are its terseness, its rich set of operators, the ease with which "low-level" operations can be done, and the ease with which it can be compiled into efficient machine code. C is not a very "high-level" language; it inherits many ideas indirectly from the language BCPL through the language B (by Ken Thompson). Unlike BCPL, C does have types, although its type-checking mechanism is weak and has a major hole (it isn't possible to declare a function's parameters and thus C doesn't check the number and types of expressions in function calls).

C++ is an extended version of the language created with the objective of giving C facilities for structuring medium-sized programs. It has a CLASS facility inspired by the one in Simula-67 and allows the declaration of function parameters so that a compiler can check function calls. Its design was subject to the constraint of not imposing any "hidden costs" which might tend to discourage its use within the C community; CLASSES are a compile-time facility, and programs using them suffer no loss of run-time efficiency.

References: *The C Programming Language*, by Brian Kernighan and Dennis Ritchie, Prentice-Hall, New Jersey, 1978. *Adding Classes to the C Language: An Exercise in Language Evolution*, by Bjarne Stroustrup, in *Software: Practice and Experience*, volume 13, pp. 139-161, 1983. *C++ Release E -- November 1984*, AT&T Bell Laboratories, Murray Hill, New Jersey.

3.8. PROTEL

PROTEL is a modular, typed language which was designed by Bell-Northern Research in 1975 to support the development of software for a family of telephone switches. Since telecommunications applications are usually highly customized, one requirement for the language and its environment was the ability to configure many slightly differing systems easily and at low cost. PROTEL offers both MODULES and a separate compilation facility based around subdivisions of MODULES which are called SECTIONS. The tools in its environment include an incremental, on-line loader and a library system with facilities for handling multiple versions of a program.

Reference: *Experience With a Modular Typed Language: PROTEL*, by Cashin *et al*, in the proceedings of the International Conference on Software Engineering, IEEE, 1981.

3.9. Mesa

Mesa was developed at XEROX PARC as "one component of a system intended for developing and maintaining a wide range of systems and applications programs." It is a strongly-typed, block-structured language which has a number of features: a MODULE construct with separate specification and implementation parts which is the basis for separate compilation, a specialized configuration language for building systems, a facility for handling error conditions, and facilities to handle both coroutines and parallel processes. Mesa also served as a basis for the Cedar language, which provided automatic garbage collection, a feature normally found in Lisp systems but not in "compiled" languages.

Reference: *Mesa Language Manual* (version 5.0), by James Mitchell, William Maybury, and Richard Sweet, Xerox PARC, 1979.

3.10. Modula and Modula-2

Modula is a special-purpose language designed by Wirth for experimenting with multiprogramming and device-handling. It is a rather small language in the same family as ALGOL and Pascal. Modula's name is derived from its MODULE facility, which provides a way of grouping sets of declarations for the purpose of restricting the visibility of names, and which served as a basis for the facility of the same name in Modula-2. Modula also provides process and device control facilities.

Modula-2 was designed by Wirth as a systems programming language for minicomputers. It is a strongly typed language belonging to the same family as ALGOL, Pascal, and Modula, and draws many of its features from the latter two. Modula-2 provides several features which have no counterpart in Pascal. It has a MODULE construct for grouping sets of declarations with initialization code and controlling the visibility of identifiers across group boundaries. It has a separate compilation facility based upon libraries of MODULES. The language also provides a coroutine mechanism and allows the declaration of procedure variables (a generalization of procedure parameters).

References: *Modula, a Language for Modular Multiprogramming*, by N. Wirth, in *Software-- Practice and Experience*, vol. 7, pp. 3-35, 1977. *Design and Implementation of Modula*, by N. Wirth, in *Software-- Practice and Experience*, vol 7, pp. 67-84, 1977. *The Programming Language Modula-2*, by N. Wirth, edited reprint of Report No. 36, Ifl, ETH Zurich, 1980.

3.11. TARTAN

TARTAN is a research language which was designed by Shaw, Hilfinger, and Wulf in an effort to determine whether or not a "simple" language could satisfy the "Ironman" requirements set down by the Department of Defense as part of the process that eventually led to the adoption of Ada. It is a strongly-typed language that in some ways is similar to Modula-2 and in some ways is similar to Ada. TARTAN provides a MODULE construct similar to the one in Modula-2 and facilities for defining GENERIC procedures or MODULES, overloading the names of procedures, and working with multiple processes. However, as a research language, it was never implemented or used.

Reference: *TARTAN: Language Design for the Ironman Requirement: Reference Manual*, by Mary Shaw, Paul Hilfinger, and Wm. A. Wulf, Technical Report CMU-CS-78-133, Department of Computer Science, CMU, 1978.

3.12. CHILL

CHILL is a language which was designed for programming telephone exchanges in accordance with goals set down by a CCITT committee. The goals included enhancing the reliability of programs, allowing the generation of efficient machine code, covering a wide range of hardware, and encouraging structured, modular programming ([CCITT 80]). From the materials gathered for this report, it is not clear whether or not the language has been completely standardized; there are indications that there have been several meetings on extending the 1980 draft recommendation in several areas. For the purpose of comparing CHILL with the other surveyed languages, it will be assumed that the proposals described in the paper referenced below form part of the language.

CHILL is a block-structured language in the same family as ALGOL with many additional features. It has data types similar to those in Pascal and facilities for working with dynamic arrays. It also provides procedure variables, MODULES for restricting the visibility of names, PROCESSES for concurrent execution, and several ways for PROCESSES to communicate.

References: *Draft Recommendation Z.200: Proposal for a Recommendation for a CCITT High Level Programming Language (CHILL)*, Geneva, 1980. *Separate Compilation and the Development of Large Programs in CHILL*, by Bishop *et al*, in *Proceedings IEE Fifth International Conference on Software Engineering for Telecommunications Systems*, Lund, Sweden, 1983.

3.13. Ada

Ada is the result of an effort sponsored by the Department of Defense to design a standard language suitable for programming both large systems and real-time systems. The requirements for the language were developed during the period from 1974 to 1978; four major language designs were produced, from which one served as the basis for Ada. Ada has undergone two revisions since the first reference manual was issued in 1980: one in 1982, and one in 1983.

Ada is a strongly-typed, block-structured language in the same family as ALGOL and Pascal. Its features include the ability to overload the names of procedures/functions/enumeration literals, a PACKAGE construct which serves a similar purpose to the MODULE construct in Modula-2, the ability to rename most entities, a tasking facility, separate compilation together with a library facility, the ability to handle exceptions, the ability to define GENERIC units (parameterized by objects, types, and subprograms), and the ability to affect the low-level representation of data types.

Reference: *American National Standard Reference Manual for the Ada Programming Language*, ANSI/MIL-STD-1815A-1983, ANSI, New York, 1983.

4. The Language Features

Now that all of the languages have been introduced, it is time to consider the role that they play in the development of large programs. As mentioned previously, developing large programs is both different from and harder than developing small ones. To overcome some of this difficulty, modern software engineering methodologies rely upon breaking programs into modules, in an application of the ancient principle of divide and conquer.

The development of a large program typically involves many overlapping life-cycles. Each life-cycle starts with a *design* phase in which the specifications for the program are determined, either by starting from scratch or by modifying previous designs. After (re-)design, work begins upon *coding* the program's pieces. The coding of each piece usually involves a number of cycles which are similar to the life-cycles of the overall development effort, but on a much smaller scale. The pieces of the program are put back together into a system in an *assembly* phase; depending upon the cost of assembly, this may happen every time a module is revised or only after many modules have been changed. Finally, the assembled system must undergo *testing and debugging*. The bugs found during this phase affect the amount of redesign needed in following life-cycles, which can range from none (best case) to a complete redesign of the system (worst case).

The degree of support that programming languages and environments provide for the activities of design, coding, assembly, and testing/debugging can have a major effect on the difficulty of developing a large program. Programming environments are beyond the scope of this report, but the contribution of a language alone can be significant. As an extreme example, Pascal does not support separate compilation; testing one small piece of a large program may require a day or more of compilation time, and this leads to (a) time spent coordinating the use of compilers (which defeats the goal of having teams work independently), and (b) a tendency to postpone testing pieces of code (which leads to more errors). Languages provide support for these four life-cycle phases with features that can be classified into one of the broad groups underlined below:

- **Design.** During the design phase, a system is decomposed into smaller pieces which communicate through relatively narrow interfaces and which can be completed more-or-less independently. If the pieces themselves are large, this process may continue recursively, with the complete set of designs forming a heirarchy. We will discuss decomposition of a system in section 4.1, and related issues of import/export mechanisms in section 4.2.

- **Coding.** During the coding phase, the pieces of a system are implemented in parallel by single programmers or small teams of programmers. Typically, each piece goes through several cycles of editing, compiling, and testing before release. We will discuss separate compilation in section 4.3.

- **Assembly.** During the assembly phase, individual pieces are put together to form a system. It is important to make sure that all the pieces are compatable. One source of incompatibility is errors made by programmers in the use of interfaces; another is the accidental use of the wrong version(s) of some module or set of modules. When more than one program is to be built from a set of pieces or some pieces have multiple implementations, it becomes necessary to keep track of the various configurations of the pieces. We will discuss version/configuration control in section 4.4.

- **Testing/Debugging.** During the testing and debugging phase, a program is tested in an effort to detect any bugs that have slipped through the design, coding, and assembly phases. Testing and debugging are not covered in this report because support for them is provided almost entirely by programming environments, rather than by languages.

Note that verification is not mentioned anywhere in the above list. While researchers have designed a few languages with the specific goal of making verification easier (such as EUCLID), verification techniques generally are not applied to large systems for at least two reasons. Producing formal specifications for a large system can be rather difficult and is obviously an error-prone process. And even if correct specifications are produced, verifying a large program generally takes too much time to be practical. Since verification techniques are not in widespread use, support for verification will not be one of the criteria used to compare the languages described in this report.

4.1. A word about words

Many of the languages surveyed for this report have constructs with names similar or identical to terms that are useful in a generic fashion. This has the potential of making discussions very confusing. To help prevent such confusion, SMALL CAPITALS will be used to denote language constructs with potentially confusing names; terms that appear in plain text can be assumed to have a generic meaning unless indicated otherwise.

4.2. Decomposition of a system

A language's support for decomposition of a system depends both upon its facilities for encapsulation and the overall program structure imposed by its facilities for separate compilation. The latter is important because it is desirable to place each "work assignment unit" into one or more separately-compiled text file(s) so that people working on logically different parts of a system do not need to spend a lot of time coordinating their use of basic tools such as editors and compilers.

Systems may be decomposed along the lines of their major algorithmic components (*functional* decomposition), along the lines of their major data structures (*data-oriented* decomposition), or along a combination of the two. Usually, functional decomposition refers to dividing a system into components which will be executed in some sequential order; this is the sense in which the term will

be used below. However, dividing a system into multiple threads of control can also be considered to be a form of functional decomposition. This second type of functional decomposition will be referred to below as *thread-of-control* decomposition to distinguish it from the first type.

4.2.1. Functional decomposition

The most important way in which languages support functional decomposition is by providing mechanisms for grouping and controlling access to sets of declarations. The grouping facilities provided by the surveyed languages tend to fall into one of three broad classes on a spectrum:

a. None. Although Pascal and ALGOL-60 are block-structured and have procedures, neither provides special grouping constructs or separate compilation.

b. Collections of individual declarations. FORTRAN, Simula-67, BCPL, BLISS-11, the three Pascal variants, C, and C++ have separate compilation facilities which can be used for the purposes of grouping, but for the most part they do not provide initialization mechanisms and provide only limited control over imports and exports.

   Since the unit of grouping in these languages is the compilation unit, it is appropriate to briefly examine their compilation facilities. BCPL, BLISS-11, PERQ Pascal, C, and C++ let compilation units contain mixtures of constants, types (or structures), variables, and procedures. FORTRAN is somewhat more restrictive; its compilation units are subroutines. Simula-67 provides for separate compilation of procedures and CLASSES. The other two extended Pascals allow the compilation of groups of procedures (and private variables in the case of UW-Pascal), but the structure of their programs is different from that of the other languages, as will be seen in section 4.2.5.

   Although they are best suited for defining abstract data types, the class mechanisms in Simula-67 and C++ provide another way to group procedures and variables. Consider a class with one instance. The fields belonging to that instance serve the same purpose as module variables, and the operations defined for its class correspond to module procedures. In a similar fashion, the body (or 'new' function) of that class corresponds to the initialization section of a module. Nested classes can even serve the purpose of types or of nested modules to the extent that they do not need to access the private declarations of enclosing classes (although C++ further restricts their utility by not allowing them to be exported). And the CLASSES in C++ provide one feature which would be nice to have in module constructs: a finalization service.

c. Module constructs. PROTEL, Mesa, Modula, Modula-2, TARTAN, CHILL, and Ada provide special constructs for grouping and initialization; with the exception of Modula, all provide separate compilation. They also tend to provide more control over imports and exports than the languages in the previous category. Note that while two of the Pascal variants (PERQ Pascal and Celentano's Pascal) have constructs called MODULES, they do not provide facilities for initialization and have not been placed in this group. On the other hand, while Mesa's MODULES are identical to its compilation units, they do provide initialization and a fairly large amount of control over imports and exports, and thus belong in this group.

In addition, some languages/systems provide *subsystem* facilities which can be used to group together a number of separately-compiled modules in a similar fashion to the way that modules group together individual language entities such as type declarations and procedures. A subsystem facility is especially useful for programs which are large enough to have heirarchies of designs. Two of the surveyed languages provide subsystem facilities: PROTEL (which has a special AREA construct) and Mesa (which provides subsystems as one facet of a configuration language called C/Mesa).

Two other mechanisms are useful for functional composition: nesting and procedure parameters/variables. Many languages allow nesting procedures, but C and C++ do not allow such nesting, while BCPL, BLISS-11, and TARTAN allow nesting but prohibit up-level adressing. Except for Mesa and PROTEL, all of the surveyed languages which provide module constructs allow them to be nested. Procedure parameters or variables can be useful in situations where a complicated control structure iterating over some data is logically independent of the operations to be performed on that data. FORTRAN, Pascal, and the Pascal variants covered in this report provide procedure parameters. As will be seen in section 4.5, procedure variables can also be of use in writing programs which have multiple configurations; the list of languages which provide them can be found there.

4.2.2. Data-oriented decomposition

Languages support data-oriented decomposition by providing mechanisms which can be used for defining abstract data types. Three capabilities are useful for defining abstract data types: a way to guarantee the initialization and finalization of each instance of a type, a way to limit access to the data structure(s) used to implement an abstract type, and a way to control which operations are available to the users of a type. Just considering the first two items for the moment (the third will be discussed in section 4.3.3), the facilities provided by the surveyed languages tend to fall into one of four categories[1]:

a. None. A number of languages, including FORTRAN, ALGOL-60, BCPL, BLISS-11, Pascal, PERQ Pascal, UW-Pascal, and C, do not provide mechanisms to hide the implementation details of types from their users or do not provide types.

b. Information-hiding mechanisms which allow importers to restrict their access to a type's data structure. Such mechanisms provide some protection, but they place the exporter of a type at the mercy of the "good will" of its importers. Celentano's Pascal provides this type of facility. In a manner of speaking, BCPL, BLISS-11, and C, which are listed above, also provide this type of facility -- they allow pointers to unknown structures or types to be passed around very easily.

c. Information-hiding mechanisms based around modules which provide full access to a

[1] [Cashin 81] does not provide enough information to allow classifying PROTEL's facilities.

type's data structure within the module defining it but which allow at most limited access elsewhere. Several languages provide such mechanisms, including Mesa, Modula, Modula-2, TARTAN, CHILL, and Ada; their facilities vary in the ease with which record fields can be selectively hidden, among other things. Like the languages mentioned in previous categories, the languages which fall into this category generally lack mechanisms to enforce initialization and finalization of instances of a type.

d. Class constructs which allow the definition of abstract data types by providing initialization/finalization operations and the ability to selectively hide parts of a type definition. Whereas a module construct provides a means of grouping and controlling access to individual entities, which may include both the declarations of types and the declarations of procedures and functions operating upon them, a class construct is a form of type declaration. But classes differ from "ordinary" types in that they encapsulate both a type's data structure and the operations defined upon it. Both Simula-67 and C++ provide class constructs; although the CLASSES of the former lack a finalization mechanism, they fit the rest of the description just given.

Several languages with class constructs, including Simula-67 and C++, provide *prefixing* or *subclass* mechanisms useful for composing abstract data types. With these mechanisms, it is possible to define a class which (for example) performs stack operations and then give any number of other classes the property of being stackable by simply mentioning the name of the "stack" class in their declarations. This both increases the clarity of a program (by separating logically independent concepts) and reduces the size of its source code.

Another composition mechanism found in some languages (TARTAN and Ada) is the ability to define *generic* units which are parameterized by objects, types and procedures, and which can be *instantiated* to produce non-generic units which can be used directly. Generic units seem to be almost the mirror image of classes. A class declaration defines a new data type whose details are known within the class but not outside. A generic unit hides the details of an outside type from its internal declarations; it does not define a new type *per se*. Thus, generic facilities must be combined with type declaration mechanisms if they are to be used for composing data types. For example, the rough Ada equivalent of a "stack" class (used as a prefix) would be a generic package which could be instantiated with a type ELEM to produce a normal package exporting the type "stack of ELEM".

4.2.3. Thread-of-control decomposition

While a single thread of control is sufficient for many programs, some types of programs may benefit from having their control structures decomposed into multiple threads. The surveyed languages fall into two groups based upon the control structures which they support[2]:

---

[2] [Cashin 81] does not provide sufficient information to classify Protel one way or the other.

a. Single thread of control. Languages in this category view all programs as consisting of a single thread of control; they include FORTRAN, ALGOL-60, BCPL, Pascal, PERQ Pascal, UW-Pascal, Celentano's Pascal, and C. One advantage to this approach is that it does not complicate a language with facilities for multiple threads of control which may be infrequently used or which may duplicate services available through the use of system libraries. The corresponding disadvantage is that there is no guarantee that such alternate services will be available when they are needed.

b. Multiple threads of control. Languages in this category allow programs to consist of more than one thread of control. They include Simula-67, BLISS-11, Mesa, Modula, C++, TARTAN, CHILL, and Ada; many of these languages are meant to be useful for programming embedded systems or writing simulation programs, applications which often involve a great deal of 'natural' concurrency. One advantage of building concurrency control into a language is to guarantee its availability. However, this must be weighed against the extra complexity thus introduced. The mechanisms provided by the above-mentioned languages can be classified into two groups: *coroutines* and *processes*.

Under a coroutine mechanism, only one thread of control is active at a time; transfers of control are made explicitly. In the absence of processes (some languages support both), mutually exclusive access to variables is provided for free. Manual scheduling also allows for very tight control over timing, but scheduling the execution of various threads can be a cumbersome job. Bliss-11, Mesa, and Modula-2 provide straightforward coroutine mechanisms. Simula-67 provides both coroutines based around CLASSES and a special SIMULATION CLASS which uses them internally to provide semi-automatic scheduling. And C++ provides a tasking facility which is very similar in appearance to a process mechanism but which depends upon having each task call one of the tasking system routines periodically (and thus appears to be built on top of a coroutine mechanism).

Under a process mechanism, multiple threads of control can be (conceptually) active at once; transfers of control are made implicitly. Coordinating two processes thus requires mechanisms for synchronization and communication. One mechanism that appears frequently and in many different forms is the *monitor*. Monitors are used to group procedures and private data and look somewhat like modules; they differ in that only one task may be executing a monitor procedure at any given time. Usually there is some mechanism by which a monitor procedure can wait for data to arrive or for a condition to become true; while the procedure is delayed, the monitor lock is released so that another task may execute inside the monitor. Languages providing process facilities include Mesa, Modula, TARTAN, CHILL, and Ada. Of these five, all but TARTAN have high-level coordination mechanisms. Mesa, Modula, and CHILL provide monitors (among other facilities), and Ada allows two tasks to *rendezvous*, but TARTAN only provides simple LATCHES (semaphores with associated process queues).

4.2.4. Override mechanisms

When decomposing a system, it may sometimes be useful to have several modules (or classes) which know each other's implementation details but which present a much narrower interface to the rest of the world. For example, there might be compelling efficiency reasons for letting two heavily-used modules access each other's internal data, but no compelling reason for compromising their abstractions with regards to other modules. A somewhat different justification is given in [Stroustrup 83]:

> . . . In many cases the public/private distinction was too sharp; some abstractions are best represented by a set of mutually dependent classes, rather than a single class. . . .
>
> For example, class *qhead* and class *qtail* together present the two ends of a queue to users of a task system implemented using classes. The attributes of the queue itself are hidden from the user and shared by *qtail* and *qhead*. One can, for example, *put* elements on a *qtail* and *get* elements from a *qhead*, but not for example *get* from a *qtail*. Some operations, like inserting a filter into the stream of data passing through the queue, exist for both classes, but are interpreted differently in the two cases. To implement these facilities each *qhead* has to pass information to its corresponding *qtail* (and vice versa), and in the simplest most common case this is most easily done by assignment to the *qtail*'s private variables.

However, languages which have strong encapsulation mechanisms normally protect the implementation of a module or class from all of its users. There are two obvious ways of sharing implementation details in this situation; both have drawbacks. Combining several modules or classes has the disadvantage of producing unnecessarily complex interfaces; making implementation details public has the disadvantage of making them too widely available and negates the advantages gained from using strong encapsulation constructs. Two of the surveyed languages provide override mechanisms as a third alternative. Mesa allows MODULES to gain access to the private declarations of imported MODULES using a SHARES clause, and C++ allows CLASS boundaries to be overridden using FRIEND declarations.

4.2.5. Overall program structure

Another factor affecting the decomposition of a system is the overall program structure imposed by the compilation mechanism(s) of the language(s) in which it is written. Most of the languages surveyed have separate compilation mechanisms which support either a top-down (parent unit and subunits) or a bottom-up (main program and library units) approach to program development. The top-down approach is characterized by compilation units (subunits) which name the sole unit that imports them and which are compiled in the context of that unit; the overall structure of a program is that of a tree. The bottom-up approach is characterized by compilation units which can be imported by any number of other compilation units, which do not know the names of their importers, and which

are compiled in a context containing only standard and imported entities. Under the bottom-up approach, the overall structure of a program may look like a tree but in general will be a graph.

One aspect of program structure that is relative to both types of compilation mechanisms is whether or not a language provides separate specifications and implementations for the pieces comprising a program. Languages with top-down compilation provide separate specification and implementation parts in the form of stubs and subunits. Languages with bottom-up compilation vary, but there seems to be a high degree of correlation between languages that provide full checking across compilation unit boundaries and those that provide separate specification and implementation parts. Note, however, that just because a language provides separate specifications and implementations for entities such as procedures and modules does not imply that it provides them for types, as we will see later.

As can be seen from the descriptions below, most of the surveyed languages provide bottom-up compilation facilities; a few provide top-down compilation facilities, and a few don't support separate compilation at all.

- ALGOL-60, Standard Pascal, and Modula do not support separate compilation.

- FORTRAN supports a bottom-up approach. A FORTRAN program consists of one or more separately-compiled subroutines, which may share data using COMMON blocks. Global naming space is flat and is determined at link time, so the names of library subroutines must be chosen carefully to avoid massive naming conflicts (if subroutine FOO uses subroutine BAR, programs using FOO must be careful to avoid naming some other subroutine BAR).

- BCPL, BLISS-11, C, and C++ support a bottom-up approach. Their programs consist of separately-compiled mixtures of entities such as variables, type or structure declarations, and procedures. Typically, import declarations are placed into *include files*, which are not compiled themselves. As in the case of FORTRAN, the flat global naming space used for exported procedure and variable names means that such names must be chosen carefully.

- Simula-67 supports a bottom-up approach. It allows procedures and CLASSES to be compiled separately from the main program. Separately-compiled units may import each other, although every separately-compiled unit must also be imported directly by the main program. Although Simula-67 does not provide separate specifications and implementations, the DECSystem-10 compiler is sufficiently intelligent to recognize that changes to the bodies of procedures of a separately-compiled CLASS do not require the recompilation of its importers.

- UW-Pascal supports a top-down approach. Its programs consist of a *main segment* together with any number of *extension segments* and *deferred segments*. Extension segments act as additions to the main segment; they exist to allow adding new

declarations to a program without requiring the recompilation of deferred segments that do not use the new declarations. Deferred segments contain procedure bodies corresponding to stubs in the main and extension segments; they may also contain declarations of private variables.

- Celentano's Pascal supports a top-down approach. Its programs consist of collections of MODULES which can be compiled in any order and which are arranged at link-time into tree structures specified in linker command files. Each MODULE consists of one or more procedures together with a set of import/export declarations. To make procedure nesting for any given MODULE heirarchy obvious, MODULES with more than one procedure are only allowed to be leaves of program trees.

- PERQ Pascal, Mesa, and Modula-2 support a bottom-up approach. Unlike such languages as FORTRAN and BCPL, they allow the building of layered libraries in which names exported from lower layers are "invisible" to programs which don't want to see them. Programs written in one of these languages consist of MODULES importing each other by name. Despite having similar names and purposes, the three MODULE facilities are quite different, as will be seen later. One obvious difference: top-level MODULES in Mesa and Modula-2 are divided into separately-compiled specification and body parts, while PERQ Pascal MODULES are logically, but not physically, divided. Mesa's MODULES are somewhat unusual in that one PROGRAM MODULE may implement several interfaces, and several PROGRAM MODULES may cooperate to implement a single interface.

- PROTEL supports a mixed approach which is predominantly bottom-up. PROTEL programs consist of MODULES importing each other in bottom-up fashion. Each MODULE may contain several separately-compiled INTERFACE and IMPLEMENTATION SECTIONS, structured in top-down fashion.

- TARTAN allows, but does not require, support for a top-down approach. Both MODULES and ROUTINES may be placed in files separate from the one containing the main program; implementations are free to treat these files either as *include* files (when compiling the main program) or as separate compilation units.

- CHILL supports a top-down approach. A CHILL program is conceptually one piece of text, but may be broken up into pieces to facilitate compilation, where each separately-defined piece is a nested MODULE. Since CHILL allows implementations to choose to compile nested MODULES completely independently from their parents, subunits typically declare the context they inherit from their parents rather than naming their parents. CHILL also has a rather unusual qualification mechanism, discussed later, that allows a program's naming scheme to be decoupled from its real structure.

- Ada supports bottom-up, top-down, and mixed approaches. An Ada program consists of one or more *compilation units*, of which at least one must contain a single top-level procedure. A compilation unit may consist of a subprogram specification or body, a PACKAGE specification or body, a GENERIC declaration or body, a GENERIC instantiation, or a subunit. Subunits can be nested and may contain subprogram bodies, PACKAGE bodies, GENERIC bodies, and task bodies. Like Mesa and several others, Ada allows the construction of layered libraries.

4.3. Import/export mechanisms

In addition to grouping sets of related entities, encapsulation constructs such as the ones described earlier provide facilities for controlling which names are exported from a construct and which names are imported into it. They may also provide information-hiding mechanisms for exporting the name of a type without exporting all of that type's characteristics.

The mechanisms which languages provide for importing and exporting entities vary in several ways. One major distinction between the various schemes is the method by which they gather the information on imported entities needed for compilation and linking; section 4.3.1 introduces the alternatives, which have effects on separate compilation that will be discussed in section 4.4.1. Languages also differ in the type of control provided over the importing and exporting of entities. Section 4.3.2 discusses control over exports, section 4.3.3 discusses the subcase of control over exported types, section 4.3.4 discusses control over imports, and section 4.3.5 discusses control over imported types. Finally, section 4.3.6 discusses mechanisms for handling naming conflicts which can arise when importing entities from multiple sources.

4.3.1. Gathering information about imported entities

One way in which languages differ is in the methods they use to gather information about entities imported across compilation unit boundaries. The source and quantity of information gathered about imported entities may influence (or be influenced by) the approach a language takes towards issues as diverse as data-oriented decomposition and separate compilation.

The surveyed languages which support separate compilation (this excludes ALGOL-60, Pascal, and Modula) tend to gather information using one or more of the following methods:

a. The properties of automatically-imported entities may be guessed based upon the context in which the undeclared identifier which triggered their import appeared. This method has all the disadvantages of the following one, plus an increased probability that the imported information is simply wrong.

   FORTRAN, C, and C++ use this method. FORTRAN assumes that any name followed by a left parenthesis which was not declared as an array denotes an external subprogram or function; the assumed type of a function is based on the first letter of its name. C and C++ assume that undeclared identifiers used as function names refer to external functions returning integers. FORTRAN and C do not check the number and types of arguments in external function calls, but C++ requires all calls to automatically-imported functions to have the same profile as the calls which triggered their import.

b. The properties of imported entities may be declared in the unit(s) which import them. This may be done either by physically including the text in the importing unit(s) or with the help of *include files* (separate text files which are usually shared between all importers of the items which they declare). Imports generally are not associated with any particular

exporter until link time; this has several implications. One is that the only way to enforce consistent use of interfaces is to enlist the aid of a linker; most languages don't bother. Another is that in the absence of explicit instructions, about all a linker can do in the way of connecting importers and exporters is to resolve references using a flat naming space; while naming conflicts between locally-declared and imported names will be detected at compile time, other naming conflicts may not be discovered until link time.

FORTRAN, BCPL, BLISS-11, C, C++, and Celentano's Pascal use this method. The first five use flat link-time naming spaces to resolve references to entities such as COMMON blocks, external routines, and external variables; Celentano's Pascal provides a multi-level naming scheme based on descriptions of "module" heirarchies which are given to a special linker. CHILL implementations which independently compile MODULES also use this method; the properties of entities in parents and subunits are declared in *context specifications* and *grant specifications* respectively. Celentano's Pascal and the CHILL implementations described above both require link-time checks to enforce the consistent use of interfaces. The other languages do not require such checks, although BCPL, BLISS-11, C, and C++ provide include file facilities which can be helpful for maintaining consistency. One potential problem with include files is that including the same file twice (as may happen when include files include each other) can cause false naming conflicts. The macro facilities in C and C++ can be used to avoid such conflicts, and the Nova implementation of BCPL avoids such problems by ignoring all but the first GET statement on any particular file.

c. The properties of imported entities may be determined by examining the unit(s) which export them. Many languages in this category take advantage of the association between importers and exporters to enforce consistent interface usage (by requiring importers to be compiled after the units they import) and to offer a multi-level global naming space. There are at least two subissues: (1) whether or not units have separate specifications and implementations, and (2) the degree of control that is provided over the importation and exportation of individual entities within a unit.

   Simula-67, PERQ Pascal, PROTEL, Mesa, Modula-2, and Ada use this method. All are strongly-typed languages which do type checking across compilation unit boundaries; with the exception of PERQ Pascal, all impose restrictions on compilation order sufficient to guarantee consistent use of interfaces. PROTEL, Mesa, Modula-2, and Ada provide separately-compiled specifications. PERQ Pascal divides its MODULES into exported and private parts, but the the two parts are not physically separate (in fact, a MODULE without procedures can be imported but not compiled). Simula-67 does not provide separate specifications and implementations for its compilation units (which are PROCEDURES and CLASSES).

d. The properties of subunits may be declared in their parent units, with the properties of parent units determined by examining them. If this sounds confusing, consider that the relationship between a parent unit and one of its subunits actually involves two imports. The parent unit imports useful properties from its subunit by requiring them to be declared as part of its (the parent's) text. And the subunit imports its context from its parent. One way to import this context is to explicitly declare it (as in Celentano's Pascal); in this case, both imports fall under category (b). But when this context is obtained by examining the parent unit, the relationship between a subunit and its parent does not fall completely under category (b) or category (c), but involves elements of both.

UW-Pascal and Ada use this method. Both languages require the specifications of subunits to be part of the text of the corresponding parent units. Both also require that subunits be compiled after parent units; this guarantees that a subunit's parent will always be available for examination. TARTAN's subunit mechanism is fairly similar; the major differences are that only the name of a subunit must be declared in a main program, and that implementations may choose not to support separate compilation at all (the manual says only that if an implementation supports separate compilation, it must do full checking across compilation boundaries). CHILL implementations which support separate but not independent compilation may also use this method (or something similar to it -- the CHILL separate compilation proposal is fairly vague on this topic).

Separation of specification and implementation is an issue which directly affects the degree to which a language and its implementation can succeed at several conflicting design goals. Abstraction concerns would seem to be best served by allowing the complete separation of specifications from their implementations. A complete separation would also serve the goal of reducing recompilation costs. However, a compiler which does not have any information about the implementation of an imported entity is likely to be harder to write and to produce less efficient object code than a compiler which has full information. It may be necessary to build a very complex and time-consuming linker to handle the output of a minimum-information compiler. Taking the other extreme reverses the disadvantages, sacrificing abstraction and recompilation time in the name of object code efficiency. We have seen already that several languages provide separate specification and implementation parts for entities such as procedures and modules. As will be seen later, most of these languages require some type implementation details to be put into specifications for the benefit of their compilers, and thus provide only partial separation when everything is considered.

4.3.2. Controlling exports

Languages which provide module constructs or which provide bottom-up separate compilation generally provide control over which declarations in a grouping unit (henceforth referred to as a "module") are exported. The most basic of these, without which the others are useless, is the mechanism by which entire entities are classified as public or private:

a. Entities may be marked as being either public or private. This approach has the disadvantage of mixing specifications with implementations, which has two consequences. Finding all the entities exported from a "module" may require looking through many pages of implementation source. For languages which gather information about imported entities from their exporter(s), the mixing of specification and implementation may result in many extra recompilations.

   BCPL, BLISS-11, C, and C++ take this approach when handling global variables and procedures. Unlike the other three languages, C exports these entities unless they are explicitly declared local to a compilation unit.

b. Entities may be private unless named in an export list. This approach also has the disadvantage of mixing specifications and implementations, with the attendant consequences. However, for the purpose of determining which names are exported by a "module", it is easier to read an export list than an entire source file.

   FORTRAN takes this approach when handling global variables. Each subprogram may contain COMMON declarations which associate a set of variables with offsets into data blocks that are shared with all other subprograms declaring COMMON blocks of the same name. However, there is no guarantee that all the subprograms using a COMMON block will divide it up in the same way, since really just the name and size of the block are exported, rather than the individual variable names.

   Modula and TARTAN take this approach; both use export lists to control which entities are exported from their respective MODULES. Modula only allows simple identifiers to appear in export lists, while TARTAN allows the use of qualified names. TARTAN allows MODULE names to be exported; although the language definition does not specify what this means, presumably it provides a way to re-export all of the names exported by a MODULE as a group.

   Modula-2 uses this approach for nested MODULES, but not for MODULES at the compilation unit level. Names can be exported in either unqualified or qualified form.

   CHILL also uses export lists. GRANT statements export names from a MODULE or REGION to the surrounding scope. Names can be granted with the PERVASIVE property, which automatically makes them available in every MODULE or REGION that could import them. Names can also be granted with arbitrary qualifiers, and names which are already qualified can be re-granted with different qualifiers than they have in the exporting MODULE. CHILL also provides a way to export all of the "strongly-visible" names defined in a MODULE or REGION without listing them all.

c. Entities may be placed in different sections of a program to indicate their export status. This approach has the advantage of separating specifications and implementations, which makes it easier to locate exported entities, and languages which use it often couple it with a separate compilation mechanism that allows the implementation of a "module" to be recompiled without affecting the clients of that "module". (However, separation of specification and implementation is often compromised with respect to types, presumably for reasons of runtime efficiency.)

   PERQ Pascal uses this approach. Its MODULES consist of an EXPORTS part and a PRIVATE part. Both parts are placed in the same file; this can be the cause of unnecessary recompilations when using environment tools such as 'make' which depend upon file modification dates.

   PROTEL also uses this approach. Its MODULES may consist of a number of separately-compiled INTERFACE and IMPLEMENTATION SECTIONS. As the names imply, items declared in INTERFACE SECTIONS are public, while those declared in IMPLEMENTATION SECTIONS are private.

   Mesa combines this approach with a public/private marking scheme. Mesa has two types of MODULES: DEFINITIONS MODULES and PROGRAMS; by default, all items declared in

the former are public, while all items declared in the latter are private. However, Mesa allows names in either type of MODULE to be marked PUBLIC or PRIVATE; this mechanism extends to such things as record field names, and it serves as a basis for type information-hiding.

Modula-2 combines this approach with export lists at the compilation unit level and uses export lists alone for nested MODULES. At the compilation unit level, Modula-2 provides three types of MODULES: DEFINITION MODULES, IMPLEMENTATION MODULES, and main programs; only the former can export items. DEFINITION MODULES are restricted to exporting qualified identifiers; nested MODULES may also export unqualified identifiers. Exporting a record type or an enumeration type automatically exports the names of the appropriate field identifiers or enumeration constants; exporting a MODULE has the effect of exporting the names in its export list as qualified names.

Ada also takes the approach of separating specifications and implementations. The specifications of PACKAGES, tasks, and GENERIC units are always separate from their bodies, and subprogram specifications can be declared separately. PACKAGE specifications are further divided into public and PRIVATE parts. Since the implementation of a PACKAGE cannot be divided among several bodies, PRIVATE parts serve little purpose other than as a repository for the full type declarations of PRIVATE types, as discussed in the next section.

Five of the surveyed languages do not appear in the above list. Of these, two do not provide grouping mechanisms (ALGOL-60 and Pascal), two provide compilation units which are arranged in top-down fashion to form programs but do not provide module constructs (UW-Pascal and Celentano's Pascal), and only one provides a grouping mechanism of the form mentioned above. This language is Simula-67. It provides separate compilation but does not provide a module construct; since its compilation units are single entities (PROCEDURES and CLASSES), export declarations would be redundant.

Languages may also provide mechanisms for making distinctions finer than just "public" or "private". Some languages have provisions for exporting variables from a module in "read only" mode; this can be used to avoid creating trivial functions which simply return the value of a variable that needs to be protected from "unauthorized" updates. This type of exporting is provided by Modula, which forces it upon programmers, and Mesa, in which it is optional. Some languages have provisions for selectively exporting parts of a type definition; control over exported types is discussed separately below.

4.3.3. Controlling exported types

As mentioned above, one aspect of control over exports is the degree to which a language allows control over exported types. However, control over types is also directly related to data-oriented decomposition as discussed in section 4.2.2. This section should be considered as an adjunct to both of those just mentioned; it expands upon references that they make.

An important motivation for type control mechanisms is that abstract data types can be very useful for decomposing a system. An abstract data type consists of two parts: a data structure (of which some parts may not be intended for public use) and a set of operations defined upon it. Thus, it is useful to consider both mechanisms for hiding the details of a data structure and mechanisms for controlling operations upon it.

The surveyed languages provide various degrees of control over the selective exporting of data structures; where control is provided, it may be linked to a module or to a class, as detailed below[3]. Note that a language may provide a way to hide the structure of a type without providing separate specification and implementation parts for it; the idea behind this is often that compilers can produce more efficient object code when they have access to details about type representations and that it is not too harmful to let a compiler use information about an abstract type's representation (since any "bad assumptions" a compiler makes when generating object code can easily be fixed by recompilation after the type representation changes).

a. Some languages do not provide user-definable types, do not provide any form of grouping or class construct, or have only top-down compilation mechanisms (and provide no control since subunits are logically part of their parents). Naturally, they do not provide separate specifications and implementations for types. Languages in this category include FORTRAN, ALGOL-60, Pascal, UW-Pascal, and Celentano's Pascal; the latter allows importers of a type to restrict their knowledge about that type, as will be seen later.

b. Some languages have bottom-up compilation mechanisms but provide no control -- a type must be completely exported or completely private. As in the above case, this has the severe drawback that it is impossible to prevent the users of an abstract data type from improperly relying upon implementation details. Languages in this category include BCPL, BLISS-11, C, and PERQ Pascal; like the languages listed above, they do not provide separate specifications and implementations for types.

c. Some languages allow the representations of exported types to be completely hidden. This solves the problem of improper access to internal details, but constructing a data structure (such as a record) which has some public parts and some private parts may require a number of type declarations. Languages in this category include Modula,

---

[3] [Cashin 81] does not provide enough information to classify PROTEL's facilities.

Modula-2, and Ada; as will be seen below, none of these languages provides fully separate specifications and implementations for types.

Modula enforces a rather strict rule: the structure of any type exported from a MODULE is unavailable outside that MODULE. While this rule ensures that protection is provided when desired, it also prevents the use of the MODULE facility as a mechanism for grouping public type declarations. Modula does not support separate compilation, much less separate specification and implementation parts for types; thus, even the smallest change to a type declaration is sufficient to force full recompilation of a program.

Modula-2 abandoned Modula's restrictive rule in favor of *opaque* types. Types exported from modules normally come with full structural information. However, when writing DEFINITION MODULES (compilation units which serve the same purpose as Mesa's INTERFACE MODULES or Ada's package specifications), it is possible to simply declare a type's identifier. This completely hides the properties of the type from its importers. The full type declaration of an opaque type appears in the corresponding IMPLEMENTATION MODULE, and thus it is always possible to change the implementation of an opaque type without requiring recompilation of the users of the MODULE to which it belongs. However, the full type must be a subrange of a standard numeric type or a pointer type; this restriction seems to be for the benefit of language compilers (typically both numeric types and pointers can be accomodated in one or two words of memory).

Ada allows package specifications to declare *private* types. A private type declaration contains the name of a type and may contain a discriminant part; it must be accompanied by a full type declaration in the private part of the same package specification. Outside the package which contains the private type declaration, none of the structure declared in the full type declaration is visible, although some information (such as the number of bits required to store a value of the type) is still available through attributes. While Ada's private type mechanism allows more implementation choices than Modula-2's opaque type mechanism, this advantage is not without cost. Both languages make compromises in their information-hiding mechanisms to allow generation of efficient object code, and Ada's compromise is that it allows compilers to rely upon information about the implementation of private types. This translates directly into added recompilations.

d. Some languages go further than the above solution by allowing the declaration of record types (or classes) in which fields are individually marked as being public or private. This approach adds flexibility without losing protection. Languages in this category include Simula-67, C++, Mesa, TARTAN, and CHILL. Like Ada, these five languages do not provide separate specification and implementation parts for types and data fields of classes. Compilers can rely upon having full information about all imported types; while this allows generating efficient object code, it can also result in extra recompilations.

Simula-67 allows CLASS attributes to be protected by listing either the attributes which are to be protected or the attributes which are to be left unprotected in a PROTECTED or HIDDEN PROTECTED declaration. Two of the possible types of attributes, simple variables and arrays, correspond to the fields of a record; other attribute types are procedures, functions, and CLASSES. PROTECTED declarations limit the visibility of attributes to the CLASS in which they are defined, its subclasses, and blocks which are prefixed with the name of the CLASS. The keyword HIDDEN further restricts visibility to the defining CLASS itself; subclasses are allowed to declare PROTECTED attributes in their parents to be

HIDDEN within themselves and their descendants.

C++ divides CLASS declarations into two parts; only members declared after the keyword PUBLIC can be accessed outside the defining CLASS. Derived CLASSES cannot access the private members of their base classes; private members are analogous to Simula-67's HIDDEN PROTECTED attributes, and there is no equivalent to Simula-67's PROTECTED declaration.

Mesa allows PUBLIC and PRIVATE access attributes to be specified in a large number of places ("anywhere a name can be declared" and "preceding the TypeSpecification in a type definition" -- [Mitchell 79]). This allows such things as declaring an exported type with a completely private representation, or declaring an exported record type which has some fields marked PRIVATE and other fields marked PUBLIC.

TARTAN uses MODULE export lists to provide control over which parts of a type declaration are exported. Record field names are exported separately from the names of record types, so it is easy to export only selected fields of a record. Export lists are also used to grant permission to use the infix operators, subscripts, constructors, ".all" operation, or "create" function associated with a type outside its exporting MODULE.

CHILL also uses MODULE export lists to provide information-hiding. It does not provide a direct way to hide the implementations of arbitrary types, unlike Mesa and Ada, but it allows the fields of record types to be selectively hidden using FORBID clauses in GRANT statements. A FORBID clause may either list the fields to be hidden or specify ALL, in which case all of the field names of that record type are hidden.

Most languages provide an implicit set of operations upon user-defined data types; this typically includes at least assignment and equality. For some abstract data types, these operations may be undesired or unsuitable (as in the case of a data type implementing fractions, where a standard bit-by-bit equality operator would come to the conclusion that 1/2 is not equal to 3/6). Yet assignment and (in)equality are usually exported even for types whose representation is hidden. A few languages provide facilities for preventing the export of standard operations and/or replacing them with user-supplied code:

- Celentano's Pascal does not allow either assignment or (in)equality to be applied to variables of unknown types.

- C++ allows the overloading of most operators, subject to the restriction that user-defined operators must take a class object as one of their arguments (to prevent things like redefining integer addition). Although the assignment and "address of" operators cannot be directly suppressed, they can be overloaded and can thus be disabled using appropriately-written function declarations.

- Modula does not allow using assignment or (in)equality tests on objects of a type outside the MODULE in which that type is defined ([Smedema 83]).

- Modula-2 does not allow using (in)equality tests on objects of an opaque type outside of

the MODULE in which that type is defined. Unfortunately, the language provides no way to suppress assignment ([Gleaves 84]).

- TARTAN allows programs to overload standard operators; this seems to include assignment. As mentioned above, the language also provides control over the kinds of operations available for a type outside its defining MODULE.

- Ada allows most operators (unfortunately, not including assignment) to be overloaded, and provides *limited private* types for which assignment and predefined (in)equality are not available outside the exporting PACKAGE.

4.3.4. Controlling imports

The surveyed languages provide several ways of controlling precisely which external entities are imported into a module or compilation unit:

- Entities which are used but not declared may be implicitly imported into a "module" (example: C assumes that an undeclared identifier used as "id(..)" denotes an external function returning an int). This mechanism isn't very desirable, even though it may cut down on the number of import declarations that need to be written. A mistyped identifier in a program would not be caught at compile time and might not even be caught at link time (consider the case where some other, unrelated part of the program exports something with the same identifier), going on to cause rather "interesting" bugs.

- Entities may be imported on an item-by-item basis (example: the IMPORT declaration in Modula-2, when used with anything other than module names). Assuming that programmers do not want to type any more import declarations than necessary, this has the advantage of making connections between modules very explicit. This can be of use both to humans maintaining a program and to a "smart" compiler trying to reduce recompilations. One disadvantage is that importing everything exported in another "module" is rather cumbersome. Whether an imported entity must be referred to using a qualified name or can be referred to using an unqualified name has some bearing on naming conflicts (discussed later) and on the ease of using imported names, but doesn't really affect the ease of finding connections between "modules".

- Entities may be imported as a group with qualified names only (example: an Ada WITH clause used to import a package specification). This method makes it very easy to import all of the items exported from another "module". It also makes the connections between "modules" somewhat less explicit than with the above approach. The mitigating factor which partially makes up for group importing is the necessity to qualify references to imported entities. Names which are qualified tend to stand out visually and syntactically, even though they do take longer to type and are thus harder to use than unqualified ones.

- Entities may be imported as a group with unqualified names (example: an Ada WITH/USE combination used to import a package specification). This makes the connections between "modules" rather hard to find, since import declarations only name groups and the places where imported entities are used do not stand out visually. It can also make life much harder for tools in a programming environment, especially for languages like Ada which have complex semantics. The advantage of this approach is that it is very easy to import and use all of the entities exported from another "module".

Many of the surveyed languages provide mechanisms for more than one type of importing, as noted in the following list (as expected, ALGOL-60 and Pascal do not appear due to their lack of grouping mechanisms):

- FORTRAN supports both implicit and item-by-item importing. Subprograms and functions are usually imported implicitly; this is true even in the presence of function type declarations, which look exactly like variable declarations. These implicit imports are triggered by using a name in a CALL statement (subprograms) or by putting an argument list after an identifier which does not denote an array (functions). Functions may also be imported explicitly using EXTERNAL declarations; this is necessary when a function is to be passed as a parameter. Variables can be imported/exported explicitly using COMMON blocks; importing only selected variables from COMMON blocks generally requires declaring junk variables for padding purposes.

- Simula-67 supports item-by-item importing of PROCEDURES and CLASSES. Indeed, since it does not have a module construct and since its compilation units consist of single entities, it is hard to imagine how Simula-67 might provide any other style of importing.

- BCPL, BLISS-11, C, and C++ allow procedures and variables to be imported on an item-by-item basis (individual "extern" declarations) or as a group with unqualified names (by using include files full of such declarations). They do not provide ways to export entities like type or structure declarations; nonetheless, programs written in these languages commonly use include files as a way of "exporting" and "importing" items from and to each source file. This usage falls under the category "importing groups of unqualified names." And C and C++ assume that undefined identifiers used as function names in the context of function calls refer to external functions; as mentioned before, this can be rather undesirable.

- PERQ Pascal supports importing all the names exported from a MODULE as a group with unqualified names.

- Celentano's Pascal has programs with a top-down structure, but it supports item-by-item importing so that it can provide independent compilation. As will be seen in the next section, it provides a fair amount of control over the importing of types.

- UW-Pascal has programs with a top-down structure, but it provides a degree of control over imports as a side-effect of its compilation mechanism. The major purpose for its *extension segments* is to allow new declarations to be added to a main program without forcing recompilation of all its subunits. Subunits do not have any knowledge about extension segments "beyond" the one that they name, so they do not need to be recompiled when new ones are added. This "lack of knowledge" effect can be deliberately used for import control, although the amount of control thus provided is obviously limited.

- PROTEL allows groups of names to be imported, and it allows names to be qualified. Unfortunately, [Cashin 81] is not very specific about the importing mechanisms available in the language.

- Mesa has a fairly flexible importing mechanism. In its most basic form, all of the names exported from a MODULE are imported with qualified names. OPEN clauses allow unqualified references to names exported from a MODULE. USING clauses are basically import lists; they tell the compiler both to prevent access to unlisted names in the appropriate MODULE, and to generate warnings for listed names which are not used. The warnings serve a fairly useful purpose: they help programmers keep USING lists up-to-date, which in turn makes it easier to understand the connections between MODULES.

- Modula-2 uses import lists and has fairly restrictive scoping rules. The entities in the scope immediately surrounding a MODULE may be imported on an item-by-item basis. By default, no identifiers (except the "standard" ones) cross MODULE boundaries; this is in sharp contrast to Ada, where a PACKAGE automatically inherits entities in surrounding scopes according to standard block-structuring rules. One consequence of this rule is that if a nested MODULE needs to use a library MODULE, the library MODULE must be imported at the top level and passed down through all intermediate scopes. Modula-2 allows entities to be imported both on an item-by-item basis and, by importing MODULE identifiers, as a group with qualified names. Modula-2 does not allow import lists to contain qualified names, but by using FROM clauses, it is possible to import selected parts of a MODULE.

- Modula and TARTAN use import lists to control the names imported by their respective MODULES; like Modula-2, both use closed scoping. TARTAN requires that all imported objects have "global extent"; this means that a MODULE declared inside a procedure cannot import any of the locals of that procedure. TARTAN, like Modula-2, allows importing the names of MODULES; presumably this provides a form of group importing.

- CHILL supports several styles of importing. Like Modula-2 and unlike Ada, CHILL has closed scoping -- nested MODULES do not automatically inherit names in their surrounding contexts. Names exported with the PERVASIVE attribute are an exception to this rule, and are inherited automatically. Using SEIZE statements, a MODULE may import individual names from its surrounding context or groups of names from other visible MODULES. Whether or not imported names are qualified depends both upon their export declarations (if any) and their import declarations; SEIZE statements are allowed to unqualify names or to locally rename their prefixes, although they need do neither of these operations.

- Ada supports several styles of importing, all of which involve the use of its WITH clause. By itself, a WITH clause imports a library unit (which can be a subprogram/PACKAGE specification, a GENERIC declaration, or an instantiation) with an unqualified name; for a PACKAGE, this has the effect of importing all of its exported entities with qualified names. USE clauses allow unqualified references to all entities exported from a PACKAGE (modulo naming conflicts), and renaming declarations can be used to selectively unqualify the names of objects, subprograms, exceptions, and PACKAGES.

4.3.5. Controlling imported types

Most languages which provide mechanisms to hide the implementation of a type from its users place control over hiding in the module which contains a type definition or the class which defines a type. But a few languages allow the importers of a type to voluntarily restrict their knowledge of its structure. These languages generally do not provide control over type exports, and while the hiding mechanisms they provide are better than none at all, they are not as secure as the ones discussed in section 4.3.3, since the exporter of a type is completely at the mercy of the "good behavior" of its importers.

The following five languages provide ways to hide the implementation of imported types:

- Celentano's Pascal allows import declarations (and local type declarations corresponding to import declarations) to *partially specify* types. Types may be specified to have completely unknown representations. Also, record types may be specified to have some fields of known type and some of unknown type; the latter are basically unusable within the MODULE declaring them.

- BCPL, BLISS-11, and C, while not providing explicit mechanisms to hide the details of an imported type or structure, make it very easy to use pointers to unknown types or structures. This allows source files to cooperate to provide a form of hiding similar to that provided by Modula-2's *opaque* types.

- C++ also provides "support" for hiding using the approach outlined above for C. However, since C++ provides a CLASS mechanism (discussed previously) that is much more suitable for hiding a type's implementation details, there isn't much point in resorting to the use of "generic" pointers.

4.3.6. Handling naming conflicts

Sometimes, different modules or compilation units export entities which have the same name. If two (or more) such "modules" are imported by a client, conflicts between the names of the entities they export can be a problem. Two facilities useful for handling naming conflicts are *qualification* (specifying in which context a reference is to be bound) and *overloading* (having a compiler determine which entity is being referenced based upon a process of elimination).

Provisions for qualification vary:

- FORTRAN, ALGOL-60, BCPL, BLISS-11, C, Pascal, PERQ Pascal, UW-Pascal, Celentano's Pascal, and Modula provide no facilities for qualifying names. Long identifiers can sometimes be used to achieve a similar effect to always-qualified names, but they are cumbersome both to use and to retrofit.

- Simula-67 and C++ provide implicit qualification of the names of class operations based upon an object's type. In both languages, calls upon class operations are written in the form "*object.operation*", and since variable names cannot be overloaded, the meaning of

such a call is clear no matter how many classes use the same name for an operation. This type of qualification can provide some of the convenience of overloading facilities in practice. As far as qualifying other names exported from compilation units goes, C + + is as lacking in facilities as C.

- PROTEL, Mesa, Modula-2, TARTAN, CHILL, and Ada provide explicit qualification of names exported from modules, generally using module names as qualifiers. In Modula-2 and CHILL, which use export lists, whether or not exported names are qualified is under the control of the exporter -- if an exporter chooses to export unqualified names, naming conflicts in an importer will result in an error. TARTAN also uses export lists, but when imported names conflict, their unqualified forms simply cancel each other out. Mesa and Ada always export names in qualified form; presumably, PROTEL does the same. Modula-2 allows exported names to be qualified but does not allow qualified names to appear in export lists; this interacts with nested MODULES such that if two inner MODULES export items with the same name, the outer MODULE cannot re-export just those two items; it must export at least one of the inner MODULES. CHILL's qualification mechanism is both flexible and unusual; the qualifiers attached to names exported from a MODULE need bear no relation to the name of that MODULE, and qualifiers can be renamed at any import or export statement. This allows hiding the structure of a MODULE from its importers, among other things.

Several languages provide overloading facilities which allow the declaration (or importing) of several procedures or functions with identical names. The binding of a particular procedure or function to a name in a call is made by using contextual information about each call together with the parameter and result types declared for procedures and functions to eliminate "impossible" choices. As long as there is enough information to determine which entity is meant, naming conflicts between overloadable entities do not cause problems. In fact, overloading can simplify naming schemes by allowing the use of the same name for routines which perform similar operations on different types; a good example of this is Ada's TEXT – IO package, which provides GET and PUT for several standard types. However, overloading may also contribute quite significantly to the complexity of a language's semantics. Overloading may occasionally combine with other language features to produce nasty surprises, as in the case of the Ada code in Figure 4-1, where adding a procedure FOO to the package IMPORT2 silently changes the meaning of a call in procedure MAIN.

In this example, before IMPORT2.FOO is declared, there is only one possible interpretation of the call in MAIN: IMPORT1.FOO(1), which involves one implicit conversion (from UNIVERSAL INTEGER to REAL). Once IMPORT2.FOO is declared, there are two interpretations: IMPORT1.FOO(1) (which requres one implicit conversion) and IMPORT2.FOO(1) (which does not require a conversion). Since there are two functions named FOO for which the argument (1) is valid, it might seem that the call in MAIN is ambiguous. However, in this case, Ada's overloading rules give preference to the interpretation which requires the smallest number of implicit conversions, namely IMPORT2.FOO(1).

Figure 4-1: Pathological Ada Overloading Example

```
package IMPORT1;
    -- original FOO, the procedure MAIN intends to call
    procedure FOO(X: REAL);          -- one implicit conversion
end IMPORT1;

package IMPORT2;
    procedure OLD_STUFF;

    -- adding this FOO silently breaks the call in MAIN
    procedure FOO(X: INTEGER);       -- no implicit conversions
end IMPORT2;

with IMPORT1; use IMPORT1;
with IMPORT2; use IMPORT2;
procedure MAIN is
begin
    FOO(1);     -- calls IMPORT1.FOO until IMPORT2.FOO appears
end;
```

Thus, the meaning of the call in MAIN changes in an unintended way without so much as the slightest hint that anything has happened.

Languages which provide overloading include TARTAN, Ada, and C++. The former two always allow procedure and function names to be overloaded, whereas C++ requires functions which are overloadable to be explicitly declared as such; this provides backwards compatability with C and helps to prevent accidental overloading. But it's still possible to construct examples in C++ where the meaning of a program changes silently, as in the Ada example above, although the mechanism involved is somewhat different.

4.4. Separate compilation

Most of the languages surveyed for this report support separate compilation. Separate compilation facilities are necessary for developing large programs, since fully recompiling a large program would take too much time to be done often. The degree to which a separate compilation facility can reduce compilation costs depends both upon its design (the dependency relationships and safety restrictions built into a language) and its implementation (the ability of a particular language implementation to reduce recompilations resulting from minor changes).

4.4.1. Safety and order of compilation

Separate compilation facilities can be categorized according to two factors: whether or not pieces of a program can be compiled independently, and whether or not a program consisting of several compilation units is checked as well as one consisting of a single compilation unit:

a. Unsafe independent compilation. Compilation units can be compiled in any order, interface changes must be propagated by hand, and when several units are put together to form a program, there is little or no checking to make sure that interfaces were used consistently. Implementations typically rely upon standard linkers and do not use compilation databases. Languages in this category include FORTRAN, BCPL, BLISS-11, C, and C++.

b. Safe independent compilation. Compilation units can be compiled in any order and interface changes must be propagated by hand, but when several units are put together to form a program, potentially extensive checking is done to make sure that interfaces were used consistently. Implementations typically use non-interconnected databases and rely upon special linkers. Languages in this category include Celentano's Pascal and CHILL.

c. Unsafe dependent compilation. Compilation units should be compiled in some partial order; when several units are put together to form a program, little or no checking is done to make sure that interfaces were used consistently. Implementations typically do not use either databases or special linkers. PERQ Pascal falls into this category only because it does not enforce the compilation of MODULES after changes to the interfaces of the MODULES they import; this in turn can be traced to its lack of physical separation for specifications and bodies. When used with include files, BCPL, BLISS-11, C, and C++ also fall into this category (header files usually are not compiled but do impose dependencies).

d. Safe dependent compilation. Compilation units should be compiled in some partial order; when several units are put together to form a program, checking is done to make sure that interfaces were used consistently. Implementations typically use interconnected compilation databases to store symbol table information and to enforce compilation order; special linkers are generally not required. Languages in this category include Simula-67, UW-Pascal, PROTEL, Mesa, Modula-2, TARTAN, CHILL, and Ada. CHILL is listed both here and above because CHILL implementations are allowed to compile programs using either an independent, dependent, or hybrid approach, so long as whatever compilation method is used is safe.

4.4.2. Recompilation strategies

When some compilation unit or set of compilation units has changed, it becomes necessary to recompile the changed unit(s) and any other units affected by the change. Recompilation can often be an expensive step in terms of both time and machine resources, and minimizing the amount of recompilation required as the result of a change is often desirable. Although recompilation strategies are often implementation-dependent, it seems worthwhile to list three general cases:

- Recompile the world. This approach to recompilation is very expensive, and in many

situations is to be regarded as a last resort. However, recompiling all of a program's compilation units does have the advantage of getting rid of strange problems that can arise from linking together inconsistent object files (as can happen when using languages with unsafe separate compilation).

- Recompile "potentially-affected" units, where "potentially affected" is recursively defined to include (a) units which have changed since they have last been compiled, and (b) units which depend upon "potentially affected" units (in the sense of order-of-compilation dependency). This approach to recompilation is generally taken by implementations of languages which provide safe dependent compilation; these languages generally provide at least some separation of specification and implementation so that changes to things like the bodies of procedures do not cause lots of recompilations. If this method is consistently applied, it provides just as much safety as recompiling the world, usually at less cost. But recompilation costs may still be unacceptable, since changes to widely-used low-level interfaces can have "ripple" effects resulting in the entire recompilation of a large system.

- Try to recompile just "actually-affected" units -- potentially-affected units which are affected semantically by a change or whose intermediate/object code representations need to be updated. Obviously, the definition of which units are "actually-affected" varies from implementation to implementation. If the cost of determining which recompilations can be skipped is not too high, this approach can save a large amount of recompilation time. It can often be applied manually when using languages with unsafe separate compilation by taking advantage of the way that a particular compiler is known to operate, and the savings may be attractive enough to be worth risking the occasional full recompilation. But when using languages with safe dependent compilation, it generally must be supported by "smart" compilers which attempt to determine the differences between the "old" and "new" versions of a compilation unit (such as the PROTEL and Mary-2 implementations described in [Cashin 81] and [Rain 84], respectively) if it is to be available.

4.5. Version/configuration control

Although version control and configuration management are to a large extent environment issues, a few languages provide facilities which may be of some use in these areas. These facilities include single-version control, configuration languages, conditional compilation, and procedure variables; for the most part, they are better viewed as supplements to environment tools than as competitors to them.

Single-version control is often found in languages which have strong typing and which do type-checking across compilation boundaries by examining the exporters of imported items. It can be useful in the absence of environment tools designed to support multiple versions of a system. By definition, all of the languages listed in section 4.4.1 as having safe separate compilation facilities

provide at least single-version control[4]. Languages which do not support separate compilation also provide single-version control in a trivial sense, although large programs written in one of them are likely to present extreme version-control problems.

Configuration languages were fairly rare among the surveyed languages; this is not too surprising since they direct actions at the linker/binder level. Mesa provides a configuration language known as C/Mesa which can be used for building subsystems, specifying that several modules cooperate to supply an interface, and specifying which module is to act as the representative of a group of modules, among other things ([Mitchell 79]). And PROTEL's incremental loader has a command language said to be similar in purpose to C/Mesa ([Cashin 81]).

Two other facilities that can be useful when working with multiple configurations are conditional compilation and procedure variables. Conditional compilation facilities can be found in PERQ Pascal, C, and C++; they are most useful when differences are slight and do not involve heavily-used interfaces, since one of their major drawbacks is that reconfiguring systems written using them may require too much recompilation to be practical. Procedure variables provide a different and perhaps more useful approach, as discussed in [Cashin 81]:

> One of the requirements for DMS was the ability to tailor large numbers of slightly differing systems easily and inexpensively. This rules out any configuration scheme based on conditional compilation as the amount of time required to recompile a system of this size is large. Instead, we adopted a simple configuration process based on selection of subsystems from a common library. . . . When an optional module is loaded, it "binds" itself (i.e. initializes procedure variables to point at its procedures) through the system and is subsequently invoked through these procedure variables. Operating system and language support is required to identify and execute module initialization code. It is interesting to note that ADA does not support procedure variables whereas software in both Mesa and PROTEL makes extensive use of this feature.

In addition to PROTEL and Mesa, a number of other languages provide procedure variables, including BCPL, BLISS-11, C, C++, Modula-2, and CHILL.

5. Summary

The tables on the next few pages summarize the categorizations made in section 4. They do not contain any new information, and thus may be skipped without penalty by readers who want to move on to section 6.

---

[4]PROTEL's library manager provides control over multiple versions, but should be considered as an environment tool rather than as part of the language itself.

Table 5-1: Facilities for Decomposition of a System

Language	Functional Decomposition	Procedure Parms/Vars	Data-oriented Decomposition	Coroutines	Processes	Program Structuring
FORTRAN	comp. units	yes	no support	no	no	bottom-up
ALGOL-60	no support	no	no support	no	no	---
Simula-67	comp. units	no	classes [c]	yes	no	bottom-up
BCPL	comp. units	yes	no support	no	no	bottom-up
BLISS-11	comp. units	yes	no support	yes	no	bottom-up
Pascal	no support	yes	no support	no	no	---
PERQ Pascal	comp. units	yes	no support	no	no	bottom-up
UW-Pascal	comp. units	yes	no support	no	no	top-down
Cel. Pascal	comp. units	yes	import ctrl	no	no	top-down
C	comp. units	yes	no support	no	no	bottom-up
C++	comp. units	yes	classes [c]	yes	no	bottom-up
PROTEL	modules [a]	yes	-------	---	---	bottom-up [e]
Mesa	modules [a]	yes	export ctrl	yes	yes	bottom-up
Modula	modules [b]	no	export ctrl	no	yes	---
Modula-2	modules [b]	yes	export ctrl	yes	no	bottom-up
TARTAN	modules [b]	no	export ctrl [d]	no	yes	top-down
CHILL	modules [b]	yes	export ctrl	no	yes	top-down [f]
Ada	modules [b]	no	export ctrl [d]	no	yes	both types

[a] has subsystem facility
[b] allows nested modules
[c] has subclasses
[d] has generics facility
[e] top-down within MODULE
[f] flexible qualification

Table 5-2: Mechanisms for Importing and Exporting Whole Entities

Language	Gathering Information A	B	C	D	Export Control mark items	list items	spec part	Import Control auto import	name items	group qual	group unqu	Naming Conflicts qualified names?	overload names?
FORTRAN	X	X				X		X	X			no	no
ALGOL-60		---				---			---			no	no
Simula-67			X			---			X			class	no
BCPL		X			X		*		X		*	no	no
BLISS-11		X			X		*		X		*	no	no
Pascal		---				---			---			no	no
PERQ Pascal			X				X				X	no	no
UW-Pascal				X		---			---			no	no
Cel. Pascal		X				---			X			no	no
C	X	X			X		*	X	X		*	no	no
C++	X	X			X		*	X	X		*	class	yes
PROTEL			X				X			?	?	module	no
Mesa			X		X		X		X	X	X	module	no
Modula			X			X			X	X		no	no
Modula-2			X			X	X		X	X		module	no
TARTAN				X		X			X	X		module	yes
CHILL		X		?		X			X	X	X	module	no
Ada			X	X			X			X	X	module	yes

Legend:

A = guess properties of imports, B = declare properties in importer,
C = examine the exporting unit, D = parent/subunit relationship (b & c)
* - has include files, which can be used to serve a similar purpose

Table 5-3: Information Hiding, Compilation, and Configuration Facilities

	Control Over Data Types					Separate Compilation		Misc. Facilities		
Language	Location if avail	Hide Any Type?	Hide Fields Selectively?	Replace/Suppress Assign?	(In)Eq?	safe?	dep.?	Cnfg Lang	Cond Comp	Proc Vars
FORTRAN						no	no			X
ALGOL-60						---				
Simula-67	exporter	[1]	X			yes	yes			
BCPL						no	[2]			X
BLISS-11						no	[2]			X
Pascal						---				
PERQ Pascal						no	yes		X	
UW-Pascal						yes	yes			
Cel. Pascal	importer	X	X	X	X	yes	no			
C						no	[2]		X	X
C++	exporter		X	X	X	no	[2]		X	X
PROTEL	(do not know if type control is available)					yes	yes	X		X
Mesa	exporter	X	X			yes	yes	X		X
Modula	exporter	X		X	X	---				
Modula-2	exporter	X			X	yes	yes			X
TARTAN	exporter		X	X	X	yes	yes			
Chill	exporter		X			yes	[3]			X
Ada	exporter	X		X	X	yes	yes			

[1] yes, by virtue of CLASSes being the only type-definition mechanism
[2] technically independent, but header files are often used and impose obvious dependencies
[3] implementations may use independent, dependent, or hybrid compilation strategies

6. Conclusions

In the process of writing this report, I noticed several relationships and trends involving the surveyed programming languages, as described below.

Generally speaking, languages which are typically compiled (such as the ones described in this report) have been improving over the years in terms of their support for programming-in-the-large. FORTRAN and ALGOL-60 are two very early languages and they show their age -- of the surveyed languages, they were the only two which did not provide user-definable data structures. Except for Simula-67, all of the surveyed languages which provide strong mechanisms for either functional or data-oriented decomposition are of fairly recent origin. The mid-70's seems to be a turning point before which few languages supported any kind of decomposition well and after which a number of languages with module constructs appeared. These latter languages can be seen as a response to the concerns raised in the early-to-mid-70's in papers such as [Parnas 71b] and [DeRemer 76].

Among the languages which supported separate compilation, there was a very strong correlation between the method(s) used to gather information about imported entities, strong typing, and safe separate compilation. Languages which guessed the properties of imported entities or obtained them from the importing units fell into two groups: most were weakly typed (or untyped) and had unsafe compilation facilities, although two were strongly typed and had safe compilation facilities which depended upon link-time semantic checks. Languages which obtained information from exporting units (either library units or parents of subunits) were uniformly strongly typed, and all but one had safe compilation facilities (the exception was PERQ Pascal). Although several languages used more than one method of gathering information, only CHILL gathered information using both types of methods just discussed, so the method a language uses to gather information about imported entities would seem to be a fairly good predictor of the strength of its typing and the safety of its compilation facilities.

This combined with other information suggests an interesting way to categorize the surveyed languages, namely:

- *Languages which are completely unsuitable for programming-in-the-large* -- those without separate compilation. This category includes ALGOL-60, Pascal, and Modula.

- *Languages which provide generally poor support for programming-in-the-large* -- those which are weakly typed or untyped, have flat name spaces, do not provide control over exported types, and have unsafe separate compilation facilities. This category includes FORTRAN, BCPL, BLISS-11, and C. The one advantage that these languages have over many of those in the next category is that they make it very easy to skip recompilations known to be unnecessary.

- *Languages which provide generally good support for programming-in-the-large* -- those which are strongly typed, have multi-level name spaces with some form of qualified names, provide control over exported types, and have safe separate compilation facilities. This category includes Mesa, Modula-2, TARTAN, CHILL, and Ada. It probably includes PROTEL, which fits the description in all ways for which [Cashin 81] provides any information. Simula-67 also fits this category in many ways, although it has a fairly flat program structure (all of the units comprising a program must be imported by the main program). Note that all but one of these languages provide modules (the remaining one has classes) and that only one modular language (Modula) is not in this category.

- *Languages which do not fit neatly into one of the above categories, because they are extensions of other languages.* This category includes PERQ Pascal, UW-Pascal, Celentano's Pascal, and C++. These languages are generally more suitable for programming-in-the-large than those on which they were based; however, their facilities are generally not as extensive as the facilities of the languages in the third category. This is not too surprising, as the goal of extension efforts is often specific improvement rather than full revision.

Even among languages in the third category, there are differences in design philosophy, as evidenced by the styles of programming each supports. Modula-2 illustrates one approach; its closed scoping and import/export lists force programmers to make connections between modules explicit and detailed, at the expense of convenience. The language is not very large and obviously was designed to permit easy implementation. Ada illustrates a quite different approach; it has open scoping and makes it very easy for programmers to import and use items from many different sources, although it may be fairly hard to track down the uses of any particular item. Furthermore, the language contains many features of varying degrees of complexity. Mesa and CHILL hedged their bets, at least on the issue of ease of importing versus accountability, by letting programmers have greater freedom to select the style of importing. At this point in time, it is not clear which approach will ultimately win out.

Moving on to a different topic, one thing that was somewhat surprising was a strong correlation between languages that provided modules or classes and languages that provided multiple threads of control. All languages which provided the former provided the latter, with the possible exception of PROTEL ([Cashin 81] does not mention whether or not it supports multiple threads). Conversely, only one of the languages which did not have modules or classes provided multiple threads. The reason that both languages with classes had coroutines is fairly obvious; the extensions that turned C into C++ were largely inspired by Simula-67, and Simula-67 has a coroutine mechanism to facilitate event-driven simulations. While this sample is hardly sufficient to generalize about all languages with classes, it is interesting to note that Smalltalk also supports event-driven simulations. The connection, if any, between modules and multiple threads of control seems to be rather less obvious;

perhaps the only connection is that the languages which have modules are fairly recent ones, and have facilities for multiple threads only because of separate interest in parallel processing at the time that they were invented.

None of the surveyed languages provided both modules and classes. This seems odd. While classes can be used to imitate modules to some extent, declaring both a class and an instance of that class would seem to be less convenient than declaring a module, especially when the combination "class + instance" cannot be separately compiled (the case for Simula-67). And while all of the languages with modules (with the possible exception of PROTEL) also provided mechanisms to hide type details, only two provided any form of composition mechanism which could begin to compete with the subclasses found in Simula-67 and C++. It would appear that combining modules and classes in one language, if done properly, would produce a language which could support decomposition better than most or all of the languages that were examined for this report. However, it may be worthwhile to briefly consider two languages which seem to have taken the opposite approach of taking one construct and building around it heavily.

One of these languages is Ada. Ada provides PACKAGES (modules) which are useful for functional decomposition and PRIVATE types which are declared inside PACKAGES and are useful for data-oriented decomposition. In this respect, it is not too much different from languages such as Modula-2 or Mesa. But Ada has other features which make it easier to use than many other modular languages for the purpose of programming in the style normally associated with classes. Generic units provide a form of data type composition, and overloading of procedure and function names can be used to simulate the effect of implicit type qualification. This is not to say that these features work in the same way as the class-related features which they can be used to simulate. In fact, like much of the rest of the language, they are fairly complicated to implement. The complexity of Ada in general (not just the parts discussed herein) has led to a number of criticisms of the language and the suggestion that it is basically the final language in a particular line of development.

The other language is Smalltalk. Smalltalk was not surveyed for this report, and is quite different from the languages previously mentioned. The typical (read Xerox) implementation consists of an entire environment which is built around the language and which is highly interactive; it is much closer to the environment provided by a Lisp Machine or by an APL interpreter than to the environments provided for most implementations of the languages surveyed for this report. Smalltalk provides a class construct which in the latest public incarnation provides subclassing similar to that of Simula-67 (there have been at least three major versions of the language and the 1976 version of the language may have allowed classes to inherit properties from multiple superclasses, as hinted at in

[Ingalls 78]). But what is interesting about Smalltalk is that all Smalltalk programs, as well as the Smalltalk environment itself, are based around the idea of defining classes and performing operations upon instances of classes. Even such basic expressions as "3 + 4" reflect this orientation (conceptually, a message is being sent to the number 3 asking it to add 4 to itself and return the result, although the actual language implementation is intelligent enough to recognize this special case). The entire programming paradigm is quite different from the ones presented by languages such as FORTRAN, Pascal, or Ada. In fact, it is even quite different from the paradigm presented by Simula-67 and C++, for while neither of these languages provides module constructs, it is possible to write programs in Simula-67 or C++ which are not in the least data-oriented and which use classes merely as record types if at all. So the Smalltalk approach is to get programmers to think about their programs in a different way, rather than providing features which can be used to more easily simulate modules. But Smalltalk may also influence the way in which programmers use modular languages, as we will see below.

First, consider that most of the surveyed languages which provided support for programming-in-the-large provided safe, dependent compilation. While this approach has some advantages over the alternatives, it tends to impose extra recompilation costs, since it often is not possible to manually skip the recompilation of a potentially-affected unit and typical implementations are not sufficiently intelligent to skip the recompilation of any unit in a potentially-affected set. This tendency is magnified by the typical lack of separation of type specifications and implementations, which allows compilers to generate code that depends upon the representations of imported abstract types and becomes obsolete when these representations change. While the code thus produced may be efficient, recompilation time is often much more critical than object code efficiency during most of the development cycle. And this brings us to the final point.

The desirability of using languages which are designed to support building large systems combined with concern over recompilation costs may have a major effect on future implementations of such languages. In particular, future implementations may be more likely to provide integrated environments and to do incremental compilation or even interpretation of programs throughout the development process, with traditional compilation runs performed only when a program is to be released and its executable form must be fairly efficient. While the benefits of strong environment support have been enjoyed for years by people in the Lisp and Smalltalk communities, strong environment support for the use of "compiled" languages has generally not been available, perhaps due to a general emphasis on the efficient use of machine time and the difficulty of providing nice environments for such languages. But as machine time becomes less important and programmer time becomes more important, good environments are becoming less of a "luxury" and more of a

"necessity" for software development, and thus we can expect to see more of them in the future.

References

[ANSI 83] *American National Standard Reference Manual for the Ada Programming Language*
New York, 1983.
ANSI/MIL-STD-1815A-1983.

[Applebe 85] Applebe, William F. and Hansen, Klaus.
A Survey of Systems Programming Languages: Concepts and Facilities.
Software -- Practice and Experience 15(2):169-190, February, 1985.

[AT&T 84] *C + + (Release E)*
AT&T Bell Laboratories, 1984.

[Birtwistle 73] Birtwistle, G. M., Dahl, O-J, Myhrhaug, B., and Nygaard, K.
Simula Begin.
AUERBACH, Philadelphia, 1973.

[Birtwistle 75] Birtwistle, Graham and Palme, Jacob.
DECsystem-10 SIMULA Language Handbook Part I (Second Edition)
Swedish National Defense Research Institute, 1975.
Published through DECUS Program Library (# 10-223B).

[Bishop 83] Bishop, R., Bordelon, E., Cheung, R., Feay, M. R., Louis, G., and Smedema, C. H.
Separate Compilation and the Development of Large Programs in CHILL.
In *Proceedings of the Fifth International Conference on Software Engineering for Telecommunications Systems*, pages 80-86. IEE, 1983.

[Brooks 79] Brooks (Jr.), Frederick P.
The Mythical Man-Month: Essays on Software Engineering.
Addison-Wesley Publishing Company, Reading, Massachusetts, 1979.

[Cashin 81] Cashin, P. M., Joliat, M. L., Kamel, R. F., and Lasker, D. M.
Experience With a Modular Typed Language: PROTEL.
In *Proceedings of the International Conference on Software Engineering*, pages 136-143. IEEE, 1981.

[CCITT 80] *Draft Recommendation Z.200: Proposal for a recommendation for a CCITT High Level Programming Language (CHILL)*
CCITT, 1980.
Temporary Document No. 39-E.

[Celentano 80] Celentano, Augusto, Vigna, Pierluigi Della, Ghezzi, Carlo, and Mandrioli, Dino.
Separate Compilation and Partial Specification in Pascal.
IEEE Transactions on Software Engineering SE-6(4):320-328, July, 1980.

[Curry 79] Curry, James E. and PARC staff.
BCPL Reference Manual
Computer Sciences Laboratory, Xerox Palo Alto Research Center, 1979.

[DeRemer 76] DeRemer, Frank and Kron, Hans H.
Programming-in-the-Large Versus Programming-in-the-Small.
IEEE Transactions on Software Engineering SE-2(2):80-86, June, 1976.

[Ekman 67] Ekman, Torgil and Froberg, Carl-Erik.
Introduction to ALGOL Programming.
STUDENTLITTERATUR, Lund, Sweden, 1967.

[Gear 78] Gear, C. William.
FORTRAN and WATFIV Language Manual.
Science Research Associates, Inc., 1978.

[Gleaves 84] Gleaves, Richard.
Modula-2 for Pascal Programmers.
Springer-Verlag, New York, 1984.

[Goldberg 83] Goldberg, Adele and Robson, David.
Smalltalk-80: The Language and Its Implementation.
Addison-Wesley Publishing Company, Menlo Park, CA, 1983.

[Ingalls 78] Ingalls, Daniel H.
The Smalltalk-76 Programming System: Design and Implementation.
In *Proceedings of the Fifth ACM Symposium on Principles of Programming Languages*, pages 9-16. ACM, 1978.

[Jensen 74] Jensen, Kathleen and Wirth, Niklaus.
Pascal User Manual and Report (second edition).
Springer-Verlag, New York, 1974.

[Kernighan 78] Kernighan, Brian W. and Ritchie, Dennis M.
The C Programming Language.
Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1978.

[Le Blanc 84] Le Blanc, Richard J. and Fischer, Charles N.
A Simple Separate Compilation Mechanism for Block-Structured Languages.
IEEE Transactions on Software Engineering SE-10(3):221-226, May, 1984.

[Mitchell 79] Mitchell, James G., Maybury, William, and Sweet, Richard.
Mesa Language Manual version 5.0
System Development Department, Xerox Palo Alto Research Center, 1979.

[Parnas 71a] Parnas, D. L.
Information Distribution Aspects of Design Methodology.
Technical Report, Department of Computer Science, Carnegie-Mellon University, February, 1971.

[Parnas 71b] Parnas, D. L.
On the Criteria to be Used in Decomposing Systems Into Modules.
Technical Report CMU-CS-71-101, Department of Computer Science, Carnegie Mellon University, August, 1971.

[Rain 84] Rain, Mark.
Avoiding Trickle-down Recompilation in the Mary2 Implementation.
Software -- Practice and Experience 14(12):1149-1157, December, 1984.

[Richards 79] Richards, Martin and Whitby-Strevens, Colin.
BCPL -- The Language and its Compiler.
Cambridge University Press, New York, 1979.

[Shaw 78] Shaw, Mary, Hilfinger, Paul, and Wulf, Wm. A.
TARTAN -- Language Design for the Ironman Requirement: Reference Manual.
Technical Report CMU-CS-78-133, Department of Computer Science, Carnegie-Mellon University, June, 1978.

[Shaw 81] Shaw, Mary, Almes, Guy T., Newcomer, Joseph, Reid, Brian, and Wulf, Wm. A.
A Comparison of Programming Languages for Software Engineering.
Software -- Practice and Experience 11:1-52, 1981.

[Smedema 83] Smedema, C. H., Medema, P., and Boasson, M.
The Programming Languages: Pascal, Modula, CHILL, and Ada.
Prentice-Hall International, Englewood Cliffs, New Jersey, 1983.

[Spice 84] *PERQ Pascal Extensions, Spice Programmer's Manual*
Department of Computer Science, Carnegie-Mellon University, 1984.

[Stroustrup 83] Stroustrup, Bjarne.
Adding Classes to the C Language: An Exercise in Language Evolution.
Software -- Practice and Experience 13:139-161, 1983.

[Tichy 79] Tichy, Walter F.
Software Development Control Based on Module Interconnection.
In *Proceedings of the International Conference on Software Engineering*, pages 29-41. IEEE, 1979.

[Wirth 77] Wirth, N.
Modula: a Language for Modular Multiprogramming.
Software -- Practice and Experience 7:3-35, 1977.

[Wirth 80] Wirth, N.
The Programming Language Modula-2
Institut for Informatik, ETH Zurich, 1980.
This is an edited reprint of Report Nr. 36, IfI, ETH Zurich.

[Wulf 72] Wulf, William A. et al.
BLISS-11 Programmer's Manual
Digital Equipment Corporation, 1972.

[Wulf 75] Wulf et al.
The Design of an Optimizing Compiler.
American Elsevier Publishing Company, Inc., New York, 1975.