Global Program Analysis in an Interactive Environment

by Larry Melvin Masinter

SSL-80-1 JANUARY 1980

Abstract: See next page

This report reproduces a dissertation submitted to the Department of Computer Science and the Committee on Graduate Studies of Stanford University in partial fulfilment of the requirements for the degree of Doctor of Philosophy.

Key words and phrases: programming environments, cross reference, flow analysis, type inference, Lisp, program maintenance, natural language interface to data bases.

XEROX

PALO ALTO RESEARCH CENTER

3333 Coyote Hill Road / Palo Alto / California 94304

Abstract

This dissertation describes a programming tool, implemented in Lisp, called SCOPE. The basic idea behind SCOPE can be stated simply: SCOPE analyzes a user's programs, remembers what it sees, is able to answer questions based on the facts it remembers, and is able to incrementally update the data base when a piece of the program changes. A variety of program information is available about cross references, data flow and program organization. Facts about programs are stored in a data base; to answer a question, SCOPE retrieves and makes inferences based on information in the data base. SCOPE is *interactive* because it keeps track of which parts of the programs have changed during the course of an editing and debugging session, and is able to automatically and incrementally update its data base. Because SCOPE performs whatever re-analysis is necessary to answer the question when the question is asked, SCOPE maintains the illusion that the data base is always up to date—other than the additional wait time, it is as if SCOPE knew the answer all along.

SCOPE's foundation is a representation system in which properties of pieces of programs can be expressed. The objects of SCOPE's language are pieces of programs, and in particular, definitions of symbols—e.g., the definition of a procedure or a data structure. SCOPE does not model properties of individual statements or expressions in the program; SCOPE knows only individual facts about procedures, variables, data structures, and other pieces of a program which can be assigned as the definition of symbols. The facts are relations between the name of a definition and other symbols. For example, one of the relations that SCOPE keeps track of is **Call**; **Call**[FN_1,FN_2] holds if the definition whose name is FN_1 contains a call to a procedure named FN_2.

SCOPE has two interfaces: one to the user and one to other programs. The user interface is an English-like command language which allows for a uniform command structure and convenient defaults; the most frequently used commands are the easiest to type. All of the power available within the command language is accessible through the program interface as well. The compiler and various other utilities use the program interface.

Preface

This dissertation is based on work the author did as part of the INTERLISP system [Teitelman, et al. 1978], and in particular, the MASTERSCOPE facility. MASTERSCOPE was designed and implemented entirely by the author. The basic idea for MASTERSCOPE was originally suggested by Warren Teitelman and a preliminary non-incremental version (called INTERSCOPE) was implemented by Phillip C. Jackson; a tree structure display program (called PRINTSTRUCTURE) had previously been implemented by Danny Bobrow. MASTERSCOPE was first completed in 1975, and has been in use by many INTERLISP users since then. The system described in this dissertation, called SCOPE, is a generalization and extension of MASTERSCOPE. While MASTERSCOPE was designed to be a robust tool for use by a large community, the emphasis in the design of SCOPE has been on improved functional capabilities; some of the efficiency and robustness has been sacrificed for its additional capabilities. There are currently no plans to make SCOPE generally available.

This work would not have been possible without the help of many people. I would like to thank in particular:

Warren Teitelman, for his early willingness to set me free on a problem, and for being the source of many of the ideas which profoundly influenced this work;

Terry Winograd, for his patient and careful readings of multiple drafts, and his support and encouragement;

Danny Bobrow and Bruce Buchanan, as well as Peter Deutsch, Cordell Green, Ron Kaplan, and Beau Sheil, for listening and reading;

Bob Taylor and the Xerox Palo Alto Research Center for financial support and incentives to finally be done; and

Carol Masinter, for editing, proofreading, and sharing with me for what has been a very long time.

Thank you.

Contents

List of Figures

Chapter 1—Introduction

1.1 MOTIVATION

> It is well known that software is in a desperate state. It is unreliable, delivered late, unresponsive to change, inefficient, and expensive. Furthermore, since it is currently labor-intensive, the situation will further deteriorate as demand increases and labor costs rise. Thus the industry faces one of two choices: either increase the productivity of highly trained, carefully selected specialists or reduce the training requirements through automation, thereby broadening the base of qualified users. [Balzer 1975]

Programming is costly, measured by almost any metric. In particular, the amount of money spent annually in the United States on software measures in the billions. Recent studies have shown that the major expense is in maintaining existing programs rather than in writing new ones [Lientz, et al. 1978]. Software is modified either to correct mistakes in the original implementation, to respond to new elements in the environment, or to improve performance or maintainability. Such activities are reported to consume as much as 75-80 percent of systems and programming resources. Regardless of these facts, many researchers interested in reducing the cost of software production do not address the issue of modification of complex existing programs but instead focus on initial program development.

Currently, there are two major themes in improving software production: improving the structure of the resulting software (to improve maintainability and reliability), and automating part or all of the task. As with other labor-intensive endeavors, it is thought that automation might improve the software production situation by reducing mistakes and increasing productivity. Efforts in program automation fall along a spectrum with respect to the degree of automation. While the goal of complete automation of the programming task is laudable, such an approach is far from producing practical results [Balzer 1975]. The alternative is to provide tools which *aid* the programmer in the production and maintenance of software. The set of tools available to a programmer form part of the *programming environment:*

> In normal usage, the word "environment" refers to the "aggregate of social and cultural conditions that influence the life of an individual." The programmer's environment influences, to a large extent *determines*, what sort of problems he can (and will want to) tackle, how far he can go, and how fast. If the environment is "cooperative" and "helpful"—the anthropomorphism is deliberate—then the programmer can be more ambitious and productive. If not, he will spend most of his time and energy "fighting" the system, which at times seems bent on frustrating his best efforts. [Teitelman 1969]

Whether a programmer is dealing with a toy problem or a highly complex one, there is widespread realization that, for any users of computers, the programming language and its compiler is only a small part of the environment with which the programmer must deal; a *complete* programming environment would include a variety of additional system aids and supportive facilities. INTERLISP is an example of a programming environment which attempts to be cooperative and helpful by providing facilities and aids which work with, not against, the programmer:

> The concept of a programming environment has added new dimensions to software research. With the advent of interactive use of computers a programmer can participate actively in software design and development. It is no longer realistic to view programming as a process of discrete steps starting at composition, then alternating between submittals and debugging the results. Instead it becomes a dynamic process with unclear demarcations. Recent programming systems specifically designed to operate interactively, the best example of which is INTERLISP, exemplify this concept by also taking an active role in the programming process. INTERLISP not only provides tools to the programmer, but it also "watches" over the process, giving aid where it can by detecting local errors and providing numerous "smart" commands to hide unnecessary programming details. Only a limited attempt is made, however, to "understand" the program. [Wilczynski 1975]

The goal of this work is to extend the INTERLISP environment to "understand" the program. The particular problem addressed is mainly that of maintenance of large systems—larger than can be comprehended in a single *gestalt*. The tools described here allow the programmer to interactively inquire about relationships between pieces of large programs without requiring the programmer to understand the whole. In this way, an attempt has been made to break the "complexity barrier" [Winograd 1975]: the limit of the size of the system with which a single programmer is able to deal. The same tools can also be used in several other ways. For example, some of the information they gather is also useful in improving compiler optimization.

1.2 Overview

This dissertation describes the implementation and characteristics of a programming tool called SCOPE. The basic idea behind SCOPE can be stated simply: SCOPE analyzes a user's programs, remembers what it sees, is able to answer questions based on the facts it remembers, and is able to incrementally update the data base when a piece of the program changes. A variety of program analysis techniques are used to extract different kinds of information from programs; examples include cross reference information, flow analysis, data type inference, and program maintenance history. Facts about programs are stored in a data base; question answering takes the form of retrieval and inference based on information in the data base. The interactive nature of the system is maintained because SCOPE keeps track of which parts of the programs have changed during the course of an editing/debugging session, and is able to automatically and incrementally update the data base. SCOPE maintains the illusion that the data base is always up to date, because SCOPE performs whatever re-analysis is necessary to answer the question whenever a question is asked. Other than the additional wait time, it is as if SCOPE knew the answer all along.

SCOPE's foundation is a representation system in which properties of pieces of programs can be expressed. Representation systems are characterized by the entities they describe, the kind of facts they can contain and the manner in which the facts are derived. The objects with which SCOPE deals are pieces of programs, and in particular, definitions of symbols—e.g., the

definition of a procedure, record type or macro. SCOPE does not model properties of individual statements in the program, the micro-syntax of symbols, or the presence of formatting; SCOPE knows individual facts about procedures, variables, data structures, and other pieces of a program which can be assigned as the definition of symbols. The facts are relations between the name of a definition and other symbols. For example, one of the relations that SCOPE keeps track of is **Call**: **Call**[FN_1,FN_2] holds if the definition whose name is FN_1 contains a call to a procedure named FN_2. The class of facts which SCOPE can remember is general enough to encode the results of many kinds of program analysis. However, it is not the most general imaginable; for example, interactive verification systems [Moriconi 1978, Deutsch 1973] often allow assertions which involve quantified expressions.

SCOPE employs several different kinds of program analysis techniques to extract information from the user's programs. While program analysis is itself an important topic of investigation, the emphasis in this dissertation is on the mechanism for providing assistance to programmers, rather than on the analysis techniques themselves.

Interface to SCOPE

The SCOPE system operates within the INTERLISP environment. During a working session, as the user is editing and debugging a program, the user communicates to SCOPE via a command language (Figure 1-1). SCOPE is able to analyze the program the user is debugging and store a data base of facts about it. SCOPE uses the data base to answer the user's questions.

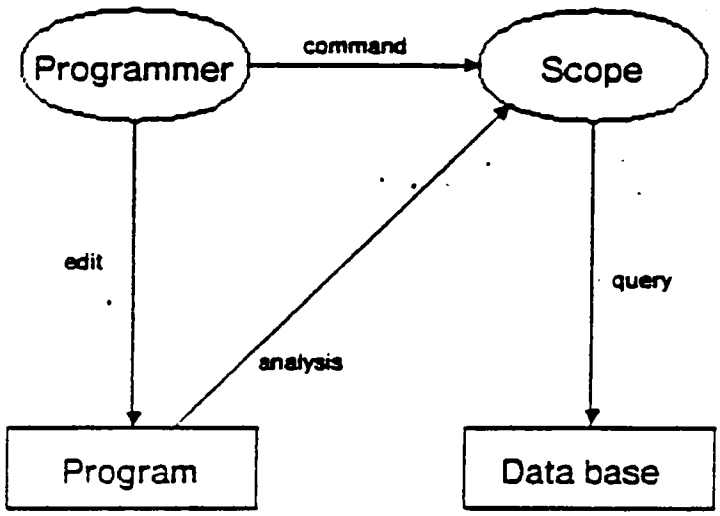

Figure 1-1—Overview of SCOPE

SCOPE has two interfaces: one to the user and one to other programs. The user interface is an English-like command-language which allows for a uniform command structure and convenient defaults; the most frequently used commands are the easiest to type. All of the power available within the command language is accessible through the program interface as well. The compiler and various other utilities use the program interface.

1.3 WHAT SCOPE CAN DO

SCOPE makes available several different kinds of information about programs, such as cross reference information, data flow information (including summary information about variables, side effects, and data types), and filing information. The information SCOPE provides can be used in several ways. For example, SCOPE can help the programmer to understand an unfamiliar program or to check for programming errors. This section is intended to give the reader an overview of the kinds of information that SCOPE provides and of applications of that information.

Cross reference

Information about the location of references to symbols is called cross reference information. Such information is useful when trying to understand or modify a program. For example, a programmer who has changed a procedure BRK might want to find the places where BRK is used. In this situation, the programmer can merely ask the question:

```
←. WHO CALLS BRK
```

and receive the response

```
(COMMAND SPACE LEADBL PUTWRD)
```

which lists the places where BRK is called. At no time during an interactive session is the user required to do anything special to make sure that the results are up-to-date. The only visible effect that changing the program has is that the response to a command to SCOPE might be returned more slowly if much of the program has changed since the last time a question was asked. Thus, if the user edits SPACE and changes it so that it no longer calls BRK, SCOPE would subsequently respond with (COMMAND LEADBL PUTWRD).

Cross reference information can be used to drive the INTERLISP editor so that, if one wants to change the way a piece of program works, it is simple to make sure that all of the uses of that piece are caught. Changing a data structure type is simplified by the ability to direct the editor to those places which reference the parts of that data structure. For example,

```
←. EDIT WHERE ANY FIELD OF COMTYPE IS USED
```

will invoke the INTERLISP editor sequentially at those places which reference any field of the data structure type named COMTYPE, giving the user the opportunity to explore or modify the piece of program which contains the reference:

```
GETVAL:
(create COMTYPE TYPE ← ARGTYP N ← (CTOI BUF))
tty:
* ...interactive edit session ...
*OK
FORMATSET:
(fetch TYPE of VAL)
tty:
* ...interactive edit session ...
*OK
(fetch N of VAL)
tty:
* ...interactive edit session ...
*OK
```

The user is led sequentially through all of the references to the fields of COMTYPE; at each location, the editor pauses to allow the user to explore the surroundings, modify the program, or perform other actions—even to (recursively) invoke SCOPE.

Flow information

> ... the applications for interprocedural data flow analysis which are unrelated to optimization are of far greater importance than code improvement. Most of these applications relate to the detection of programming errors, program documentation, and improved language design. [Barth 1977]

Another kind of information of which SCOPE keeps track relates to program flow. Flow information reflects the dynamic properties of the execution of programs, while cross reference information relates to the static interrelations of the structure of pieces of programs independent of program execution. (It is possible to "understand" cross reference even for non-executable languages, e.g., one data structure type can reference another.) The flow information which SCOPE computes includes the ways in which one procedure might call another, and the location where variables are bound, used, and assigned. Flow information has many applications: for example, flow properties can be used for detecting programming errors, in aiding compiler optimization, and to provide useful information to the programmer.

One common error in INTERLISP programs arises from misuse of free variables. A free variable is used in one procedure and declared in another; the identity of the variable is determined by the run-time context of the use. Detecting free variable errors is difficult for a programmer because it often involves examination of large portions of the program. SCOPE's flow information, which includes the location where variables are used freely, where they are

bound, and the possible calling chains, is sufficient to detect the possibility of a free variable error. At any time during the program development process, the programmer can ask SCOPE to check for free variable errors using the CHECK command. For example, the command

```
←. CHECK FORMAT
```

might result in the warning:

```
BLANKS is used freely by SKIPBL, which can be reached from INDENT,
an entry, without BLANKS being bound.
```

This warning message means that there is a possible dynamic calling path which can reach the procedure SKIPBL in which the variable BLANKS is not defined.

Side effect information

A particular kind of flow information which SCOPE provides is a summary of the *side effects* of procedures: SCOPE can determine, for a procedure, what types of data structures might be changed as a result of a call to that procedure. The classical use of side effect information is in program optimization. Many code transformations in an optimizing compiler have preconditions which are expressed in terms of side effects and uses. In a language such as LISP which is strongly oriented toward short procedures, interprocedural information is important when making code improvements.

For example, the program fragment:

```
(VAL←(GETVAL BUF))
(CT←(COMTYP BUF))
(DOCOMMAND CT VAL)
```

can be rewritten as

```
(DOCOMMAND (COMTYP BUF) (GETVAL BUF))
```

if the variables VAL and CT are not used subsequently in the program (or by DOCOMMAND) and the expressions (COMTYP BUF) and (GETVAL BUF) can be exchanged.

Type information

Yet another kind of information which SCOPE is able to provide concerns data types. In LISP, variables do not have data type declarations associated with them; rather, the *objects* that are passed as the values of variables, stored in fields of records or returned from procedures may have data types associated with them. Even though LISP (usually) has no type declarations, it is often possible to infer from the code some restrictions on the possible ranges

of variables. If a "data type" is construed to be a range of possible values (one of the many possible interpretations of "data type"), then SCOPE can be said to perform data type inference. For example, SCOPE can infer that the procedure:

```
(PUTLIN
  [LAMBDA (BUF OUT)
    (for X in BUF do (PUTCH X OUT])
```

expects BUF and OUT to be a list of characters and file name respectively, and that PUTLIN returns NIL. The type declarations which are so inferred are useful both as information to the programmer and as possible additional information to the LISP compiler.

1.4 DESIGN PHILOSOPHY

The most important constraint on SCOPE's design was that it should be a practical tool of general utility for use with almost all INTERLISP programs. In the course of designing SCOPE, several issues have arisen which have critically affected the way in which the system works. This section lays out some of those design constraints.

Non-intrusive

A tool should not get in the way when it is not needed. Program analysis tools which require the programmer to input a large body of assertions about the program in addition to the program itself will not have much success as practical programming tools, because the assertions play no part other than error checking in the program development process. This claim has been partially refuted by the increasing popularity and success of programming languages which enforce strict type checking such as PASCAL, ALGOL 68, and MESA [Geschke, Morris & Satterthwaite 1976]. However, declarations in those languages contribute to program efficiency and aid in storage management as well as providing for static checking.

In adding program inference capabilities to an existing language, it is important not to add to the burden of programming. A large program is in fact a mine of information—information which any competent programmer might be able to infer, given sufficient time. The goal of this work has been to embody that capability within the programmer's mechanical assistant. It is possible to build an assistant which can infer relationships from the program as written without requiring the user to make additional assertions.

Correct, but imprecise

It is possible to take an intractable problem (automatic program creation and modification) and turn it into a tractable one (a programmer's assistant) by building an *aid* rather than an automatic device. The "low road" to automatic programming has had high payoff to real programmers today.

A specialization of this rule is as follows. It is now recognized that proving simple properties of even small programs is often either not decidable or else computationally infeasible [Jones & Muchnick 1977]. It is necessary to take a heuristic approach to understanding in order to make headway; thus, program analysis almost always results in approximate assertions. For example, in computing flow information, it is impossible to tell if a particular path through a program will actually be taken; it might be that the test in a conditional is always false.

Benefit for cost

To achieve acceptance of any programming tool, the benefit of using the tool must exceed the cost. However, cost should not be measured in computer cycles. It has generally been the trend that manpower costs have increased, while the cost of machine cycles has decreased. With the advent of personal computers, the notion of computer time as a limited resource may well become obsolete—imagine being accused of wasting cycles on a hand-held calculator. In designing programmer tools, it is important to minimize the time that the *user* needs to spend to perform a given task; when the task is performed with computer assistance, then the time the user must wait for a response remains critical. Because SCOPE only performs analysis as a direct result of a user's request, the user always has the choice of waiting for SCOPE's response or aborting the computation.

Uniform interface to multiple sources of information

SCOPE provides a uniform way in which diverse kinds of program information can be used together. The synergistic effect of multiple sources of knowledge within a single framework has become evident with the use of SCOPE. For example, in the command `WHO ON FORMAT IS CALLED BY WHO THAT BINDS BLANKS`, flow information (`BINDS`) is used in conjunction with filing information (`ON FORMAT`) and cross reference information (`CALLED`).

1.5 THE SETTING

> . . . [LISP's] core occupies some kind of local optimum in the space of programming languages given that static friction discourages purely notational changes. . . . LISP still has operational features unmatched by other languages that make it a convenient vehicle for higher level systems for symbolic computation and for artificial intelligence. [McCarthy 1978]

LISP systems have been used for highly interactive programming for more than a decade. During that time, special properties of the LISP language have enabled a certain style of interactive programming to develop. Sandewall [1978] has written an excellent survey article describing this style of program development.

In particular, INTERLISP is a programming environment in wide use within the artificial intelligence community for a variety of application programs. It is a complete programming environment with sophisticated debugging tools, multiple extensions to the basic LISP language, a large subroutine library, and various tools for improving efficiency of user's programs.

While a SCOPE-like facility could be of great utility in environments other than LISP, several characteristics of the LISP style of programming had particular impact on the ease of implementation and the utility of the result for SCOPE.

Impact of environment on utility

First, INTERLISP is an *interactive* environment. The class of programmer assistance and interactive retrieval tools of which SCOPE is a representative does not make much sense in a batch environment. It is only in the context of using the computer as an active tool with which to build programs that an interactive assistant can be of use. A system for answering questions about program organization makes an effective tool only if the question-answering process is easier and faster than performing the same task without assistance.

Secondly, SCOPE is intended for use in the development of *medium- to large-scale* programs. It is unnecessary to provide information retrieval capabilities for short programs which can be understood by simple examination. SCOPE is most useful when the program has grown so large that the programmer cannot grasp it as a whole. INTERLISP, through its incremental style, allows the development of programs which can easily exceed the grasp of a single programmer; in that sense, SCOPE fills a real need.

Finally, the power of SCOPE is amplified greatly by being embedded in an *integrated* environment such as INTERLISP. It is important that facts about the program are available within the debugger and editor, so that the information is always at the fingertips of the programmer. Without this integration, the question-answering process might fail to be easier or quicker than obtaining the same information without assistance. For example, a cross reference

listing on the desk might at times be more convenient than interrupting an editing session to invoke a special purpose question-answering program.

Impact of environment on ease of implementation

Several other qualities of LISP and INTERLISP made the development of SCOPE easier. LISP has a simple representation of programs which is easy to analyze. The extensions to the syntax of LISP contained in INTERLISP did not pose major additional problems in providing accurate analysis routines.

Why not simplify?

> Well chosen and well designed programs of modest size can be used to create a comfortable and effective interface to those that are bigger and less well done. [Kernighan & Plauger 1976]

INTERLISP is a large and complicated system. In the course of answering questions about INTERLISP programs, features of the language which make analysis difficult are often found—non-uniform interface to language features, obscure or ambiguous semantics, and features which violate common intuitive assumptions about program execution. For the most part, the choice has been to deal with the language as it is rather than to attempt to fix it; elegant solutions are elusive. There is a great temptation to dismiss the complexity of INTERLISP as the result of bad design, lack of design, or, as is actually the case, too many designers, and to choose instead an artificial language with cleaner semantics as the target of analysis. There are continuing efforts to develop real programming languages with cleaner semantics (e.g., CLU [Liskov, et al. 1977], ALPHARD [Wulf 1974], SCHEME [Sussman & Steele 1975], and improvements to INTERLISP [Bobrow & Deutsch 1979]). These efforts are laudable and have made some progress in recent years. However, a certain amount of complexity is inherent in any programming system of maturity, and tools are needed for dealing with the complexity. Simple languages are not realistic. No programming language will be a panacea which will simplify the semantics of all programs—programs are inherently complex. In addition, the universe of design objectives for programming languages is somewhat self-contradictory; there are always compromises [Hoare 1973, Wasserman 1975]. Of the alternatives for dealing with complexity, powerful tools are often more effective and practical than attempts at simplification of the environment.

The INTERLISP system, though large and complicated, is written in LISP using only the primitives in the INTERLISP Virtual Machine definition [Moore 1976] (referred to as "the VM"), which is not nearly so large and complicated. The VM is the environment in which the INTERLISP system is implemented. It defines a basic set of abstract objects and LISP functions for manipulating them; the rest of the INTERLISP system is defined in terms of the primitives

supplied in the VM. Some of the "built-in" properties which Scope contains about INTERLISP primitives (e.g., side effect information) were derived by using SCOPE to analyze the INTERLISP system above the VM.

1.6 SOME ASSUMPTIONS AND LIMITATIONS

> ... standard halting-problem arguments show that no such system can be complete: execution time is not a decidable property of current programming languages. [In addition] the analysis of many algorithms requires considerable mathematical expertise: an expert system would necessarily include all the techniques in the monumental work of Knuth. The former is an absolute limitation: the latter establishes a boundary beyond which interactive assistance from a programmer or analyst is required. [Wegbreit 1975a]

Any attempt to design a program which knows something about other programs must necessarily carefully skirt the computability problem: given almost any interesting property of a program, it is almost always possible to come up with an example where the value of that property is not computable for a large class of programs; almost every such property is reducible to a halting problem. For example, the values which a variable might assume can depend upon the result of a predicate which is possibly (but not decidably) always false; thus, exact determination of the range of values for the variable is not possible.

It is therefore necessary to limit the domain of inferences about programs in a way which preserves many interesting properties. Along the path to program understanding, there are several obstacles. It is as if a fanciful landscape were being explored (Figure 1-2), inspired by Alan Perlis:

> There's a second [obstacle] that has recently come into being, and this is recognition that we are surrounded by mountains that are really unbelievably difficult or even impossible to scale. Indeed, the mountains are so bad that at the moment one of the greatest games around is to show that they are impossible to scale. All we're able to show is that one mountain is as bad as another.... This bothers people—it bothers me I know—until I find out that the vast majority of the tasks that we do not yet know how to do are not in those categories. [Perlis 1977]

The first obstacles travelers along the path to program understanding encounter are the Mountains of Complexity. For example, many papers in the literature show either the computational complexity or infeasibility of various flow analysis tasks [Hecht & Ullman 1973, Schaeffer 1973, Graham & Wegman 1976, Aho & Johnson 1976]. One of the reasons SCOPE is able to skirt the Mountains of Complexity is by its choice of incrementally updating the data base. An order n^2 algorithm might be computationally infeasible in a batch environment; however, incremental update can often reduce the complexity of the computation when a piece of program is changed to a manageable size. The next major obstacle, labelled the Cliffs of Non-Decidability, has a small Heuristic Gap winding through it. By carefully choosing the kind of information reported back from the analysis routines and not attempting to be too ambitious, it has been possible to provide useful results.

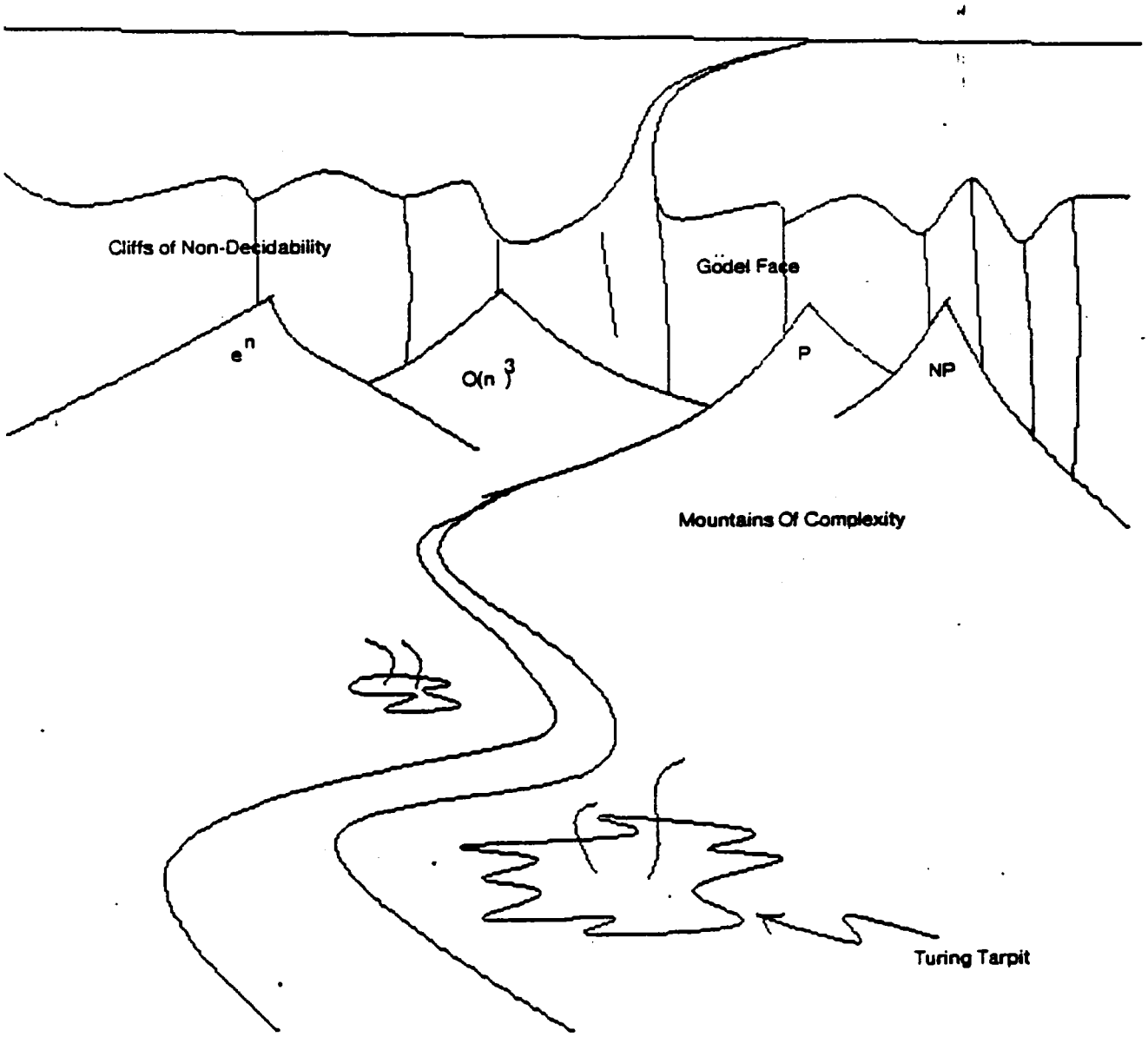

Figure 1-2—Perlis' Perils

Specific limitations of analysis

> ... In the rest of this thesis we will assume that the only way to change the state of a program is by transfer of control or modification of a variable and that no programs access any asynchronously modified data. [Banning 1978]

For the reasons outlined above, it has been necessary to limit the kinds of analysis with which SCOPE deals. One place where this limitation has become quite evident has been in dealing with the unconventional control structures which are available to INTERLISP users (in particular, EVAL, APPLY, and the spaghetti stack features). Flow analysis or even simple cross reference information is difficult to compute in the presence of those primitives.

INTERLISP contains a complex interrupt system which can, in practice, cause user-defined computations to occur at arbitrary places in the computation. Some uses of the interrupt system can violate the intuitive interpretation of the program to the extent where almost no analysis would be possible: for example, following an assignment X←Y, if an interrupt occurs which reassigns Y, then X=Y would be false. Thus, most of the program properties inferred by SCOPE are assumed to hold "unless an interrupt with interfering side effects happens".

INTERLISP also contains a sophisticated error recovery mechanism whereby the programmer can specify an arbitrary computation to be executed when an error of any given type occurs. For example, the programmer can specify that the addition of two strings should *not* cause an error, but rather that the result should merely be the concatenation of the two strings. SCOPE's type analysis, however, assumes that the INTERLISP error mechanism is not used to continue computations which would normally be in error.

Finally, INTERLISP is an interactive system, and one can write INTERLISP programs which define new procedures or data structure types, or modify old ones. The ability to do so is quite powerful, and makes it possible to write programming tools within INTERLISP itself (most of the INTERLISP environment, including the editor and debugger, is written in INTERLISP). Analysis of programs which modify their own code is beyond the capability of SCOPE. SCOPE assumes, for example, that procedure definitions do not change during program execution. SCOPE *is* able to note when the capability to redefine or modify existing programs is used and is able to warn the user when its analysis may be incomplete.

Anyone who attempts automatic program analysis, whether that analysis be verification, performance analysis, or measurement of complexity, faces many problems. Only a few of those problems have been solved here; what is provided are fundamental mechanisms for embedding the solutions to those problems (insofar as Scope can use a variety of program analysis techniques), and, for gracefully *not* solving them (by relying on correct but imprecise information, and by knowing when its analysis is possibly incorrect).

Only INTERLISP

While a SCOPE-like facility could be built for other languages, SCOPE currently only works for INTERLISP programs. The fundamental idea of an interactive assistant which is able to answer questions about a program is clearly relevant to any programming language. The particular representation scheme used by SCOPE to represent properties of pieces depends only on the ability to split up the user's program into separately produced parts which have separable semantics. However, while many of the individual relations known to SCOPE are applicable to most conventional programming languages (e.g., cross reference), some of them relate to features which are rarely found in non-LISP systems. For example, SCOPE's check for misuse of free variable analysis is useful only in systems with dynamic binding and SCOPE's type analysis is applicable only to languages with a run-time type system. To implement a SCOPE-like system for other languages would require studying the real information needs of the programmers of those languages to determine which kinds of analysis are most appropriate.

1.7 RELATED WORK

Work related to SCOPE and described within the literature falls into two rough categories: that which attacks a similar problem, and that which uses related techniques. In the former category are other efforts along the spectrum of "automatic programming", from interactive programming environments to automatic programming systems. The category of related techniques includes work on verification, flow analysis and type inference.

Although the idea of interactive tools to aid programmers is not new, complete, integrated programming environments have only recently become popular. Teitelman [1969, 1972] was an early proponent of a complete programming environment. Mitchell [1970] and Swinehart [1974] proposed interactive programming environments for Algol-like languages. Several tools available under the UNIX operating system [Dolatta, et al. 1978, Feldman 1979] are directed toward the programming language C. Model [1979] has described interactive tools aimed at monitoring more complex processing systems.

There have been several calls for "smarter" assistance to the expert programmer, which go a step beyond interactive editing and debugging tools. Winograd [1975] proposed a unified programming environment in which automatic program synthesis and analysis are mixed. Rich, Shrobe and Waters [Rich & Shrobe 1978; Rich, Shrobe & Waters 1979; Waters 1979] are attempting to build a much more ambitious programmer's apprentice which can model a programmer's goal structure and relate the structure of the program to the semantics of the domain in which the program is operating. Shrobe [Shrobe 1979] describes a system called

REASON for providing "common sense" side effect analysis to aid programmers in understanding their programs. The most significant difference between these works and SCOPE is that they deal with informal reasoning about programs, and the intentions of the programmers. SCOPE, on the other hand, concentrates on formally definable properties of programs. Clearly, a true "programmer's apprentice" would have the ability to mix both kinds of knowledge.

Researchers in program verification attempt to provide mechanical assistance for proving that programs are correct. Verification shares with program assistance the character of extracting information from programs. Interactive, incremental verification systems such as the one described by Moriconi [1978] share with SCOPE the mechanisms of change propagation, although the class of assertions that they deal with is, on the one hand, more complex, and on the other, not as broad.

Finally, global flow analysis is a fruitful approach to program optimization (e.g., [Banning 1979; Barth 1977, 1978]). SCOPE provides a framework in which flow analysis can be placed. Many researchers are actively investigating flow analysis techniques and applications in many different forms and in particular interprocedural flow analysis [Rosen 1979]. Fosdick and Osterweil [1976] have applied data flow analysis to detecting errors in programs.

1.8 CONCLUSION

> The proper study of those who are concerned with the artificial is the way in which that adaptation of means to environments is brought about—and central to that is the process of design itself. [Simon 1969]

The major contribution of this work is that it shows a pragmatic approach to the construction of a programmer's assistant. In order to delineate an approach to designing programmer assistants, a set of *design criteria* is first outlined—important criteria which a useful programmer assistant tool should satisfy. Second, a design which meets the criteria is described. Finally, an implementation of the design provides some validation that (a) the design criteria are in fact desirable, and (b) the design satisfies the design criteria.

It seems to be common within the computer science literature that an author will introduce the reader to a particular problem and then proceed to present a solution which is simply asserted to solve the problem at hand without further evidence [Kling & Scacchi 1979]. While the programming tools described in this dissertation have not undergone rigorous tests to determine whether they improve productivity, a subset of these facilities have been in use for several years within the INTERLISP user community and, as indicated by the results of an informal survey, many INTERLISP users have found them invaluable. While no controlled study

has been performed by which one could make strong claims of increased programmer productivity, there is good evidence that at least the programmers themselves have found some benefit.

Chapter 2—Uses of SCOPE

SCOPE's range of applications is broad and it can help programmers in many different ways. In this chapter, a few typical applications of SCOPE are discussed. An analogy can be drawn to programming languages; if SCOPE were a programming language, this chapter would contain some examples of the kinds of programs one could write in it.

The examples in this chapter are based on the **FORMAT** program given in Appendix V; the program is a translation (from RATFOR into INTERLISP) of a text formatter which appears in Kernighan and Plauger, *Software Tools* [1976]. It was chosen because it is well-written (being an example in a text on writing good programs) and more than a few lines long. The code is relatively well documented (every routine has a comment which indictates what it does) and there is a fairly lengthy documentation on the operation of the program. The program accepts text to be formatted, interspersed with formatting commands telling it what the output is to look like. The reader must employ a little imagination; as programs go, **FORMAT** is still quite short and some of the problems illustrated in this chapter could be performed as simply either by hand or with a simple text editor. SCOPE is intended to help programmers who must deal with systems which are an order of magnitude more complicated.

This chapter discusses three areas where SCOPE is of general utility; within those areas, particular applications to INTERLISP will also be described: (1) helping the programmer to understand or modify a large program that was written by somebody else or written a long time ago, (2) checking for programming errors, and (3) improving the quality of compiled code.

2.1 AID TO PROGRAM UNDERSTANDING AND MODIFICATION

SCOPE can help programmers who are trying to understand or change a large or unfamiliar program in several ways explained in detail in sections 2.1.1 through 2.1.5 below. SCOPE can (1) present summaries of the graph of interrelations of pieces of program, (2) show how a piece of program is used, (3) answer questions about program flow, (4) answer questions about side effects, and (5) answer questions about data types.

The set of information which SCOPE can provide to the user is completely directed by the queries asked; there is no fixed table of information which is displayed. Because SCOPE is interactive and its command language simple, SCOPE is responsive to the needs of the user. The difference between trying to understand a program with a summary of possibly useful information and using the kind of interactive response that SCOPE provides is like the

difference between debugging with a core dump and debugging with an interactive debugger. Just as the interactive debugger is a much more finely tuned efficient tool, SCOPE is much more useful than any one static display of information would be. While carefully written documentation would provide guidelines for the program reader, program documentation is often out of date, inconsistent, or incomplete. In the absence of careful documentation, SCOPE can be of great assistance in understanding a program and in fact, having access to a massive set of documentation, no matter how complete, is not as convenient as being able to interactively ask and receive immediate answers to specific questions.

SCOPE can also be used to *generate* program documentation. For example, SCOPE can be used to generate cross reference listings or lengthy summaries of program properties, or to annotate programs with type declarations. (Some INTERLISP users have used MASTERSCOPE to produce cross-reference documentation in lieu of flow charts required by their funding agencies.) However, such documentation is static—it does not change when the program changes—and inconvenient—one must search the documentation to find an answer rather than asking directly. SCOPE-added type declarations often obscure the simplicity of the code and are also not as useful as interactively available information.

Display of program structure

> A suitably printed version of the call graph provides a useful documentation and debugging aid. [Ryder 1979]

In working with large programs, a programmer may lose track of the hierarchy which defines his program structure. SCOPE can aid the user by displaying a tree structure which concisely shows the interrelations of the pieces of a program. For example, the command `SHOW PATHS FROM FORMAT` would print the following tree structure of calls shown in Figure 2-1. (A similar figure appears in Kernigan & Plauger, p. 245.)

This display shows, for example, that the function `FORMAT` calls the functions `FORMATINIT`, `COMMAND`, `TEXT`, `SPACE`, `CHARBUFFER` and `GETLIN`; `FORMATINIT` calls `CHARBUFFER`, and `COMMAND` calls `BRK`, `GETTL`, `SPACE`, `COMTYP`, `GETVAL`, and `FORMATSET`. Only calls to the user's own functions are included—system functions are not traced or displayed. For example, `FORMAT` also calls the system functions `CAR` and `IGREATERP`, but SCOPE knows that `CAR` and `IGREATERP` are INTERLISP system functions and does not display them in response to the `SHOW PATHS` command.

```
1.FORMAT FORMATINIT CHARBUFFER
2.       COMMAND BRK FORMATPUT PHEAD SKIP
3.       |       |             |     PUTTL PUTDEC
4.       |       |             |           PUTC
5.       |       |             PUTC
6.       |       |             PUTLIN PUTCH
7.       |       |             SKIP
8.       |       |             PFOOT PUTTL {3}
9.       |       |                   SKIP
10.      |       GETTL SKIPBL SKIPBL {10}
11.      |       |     SCOPY
12.      |       SPACE BRK {2}
13.      |       |     PHEAD {2}
14.      |       |     SKIP
15.      |       |     PFOOT {8}
16.      |       COMTYP
17.      |       GETVAL SKIPBL {10}
18.      |       |      CTOI
19.      |       FORMATSET
20.      TEXT LEADBL BRK {2}
21.      |    UNDERL SCOPY
22.      |    CENTER WIDTH
23.      |    FORMATPUT {2}
24.      |    PUTWRD SPREAD
25.      |    |      BRK {2}
26.      |    |      SCOPY
27.      |    |      WIDTH
28.      |    |      FORMATLENGTH
29.      |    GETWRD
30.      SPACE {12}
31.      CHARBUFFER
32.      GETLIN
```

Figure 2-1—Tree structure of function calls

The numbers in braces { } after a name are backward references indicating that the tree for that function was expanded on a previous line. For example, the call tree of BRK is not expanded on line 12 because BRK's tree (which shows that it calls FORMATPUT) had been displayed on line 2. Backward references are necessary because the call *tree* is actually a call *graph*: the structure of function calls can form an arbitrary directed graph.

In addition to displaying a tree structure of function calls, SCOPE can display an *inverted* tree; that is, one which shows how given functions can be reached. For example, the command SHOW PATHS TO BRK would display the following structure:

```
1.BRK COMMAND FORMAT
2.    SPACE FORMAT
3.    |     COMMAND {1}
4.    LEADBL TEXT FORMAT
5.    PUTWRD TEXT {4}
```

This display shows, for example, that BRK is called by COMMAND, SPACE, LEADBL, and PUTWRD, while COMMAND is called by FORMAT.

SCOPE's SHOW PATHS command has several other options which allow the programmer flexibility in controlling the nature of the structure displayed. The various options of SCOPE's SHOW PATHS command are listed in Appendix II.

The SHOW PATHS command allows the user to obtain a concise overview of the structure of a program from a number of different viewpoints. Often, the call structure of a large system is too complex to be clear from any one display of its structure; that the perspective from which the program is viewed is not *fixed* but rather determined by the programmer is critically important in the usefulness of the display. When a single structure is too complex to be displayed in any one orientation, an interactive display which allows multiple perspectives can help in understanding of the structure. An analogy can be drawn to computer aided design facilities, in which an interactive display of a structure being designed (a circuit, a building, or a ship) is much more useful in understanding the structure than any one drawing.

Cross reference

SCOPE can be used as an interactive index when browsing through a large program. Often, the key to understanding a piece of program (e.g., a subroutine or a data structure) is to see how that piece is used. Program documentation generally includes descriptions of how a program works and what various pieces do but rarely includes descriptions of the ways in which those pieces are employed. In reading an unfamiliar program, the reader may come across a short procedure which performs a simple action. It may be perfectly clear how that procedure works, but the reader still has no clear idea of why the procedure is there at all. Only by examining some of the uses of the procedure will its applications become evident. SCOPE can help by allowing the program browser immediate access to the places which refer to a given piece of program, through SCOPE's SHOW or EDIT commands.

For example, when browsing the FORMAT program, the program reader encounters a procedure GETTL:

```
(GETTL
  [LAMBDA (BUF TTL)                                    (* copy title from buf to ttl)
    (while BUF:1~=BLANK and BUF:1~=TAB and BUF:1~=NEWLINE
        do BUF←BUF::1)
    BUF←(SKIPBL BUF)
    (if BUF:1=SQUOTE or BUF:1=DQUOTE
        then BUF←BUF::1)
    (SCOPY BUF TTL])
```

Apparently, GETTL accepts a list BUF, removes any initial blanks, tabs, or new-line elements, calls SKIPBL, removes any quote characters, and then calls SCOPY. While it may be evident what GETTL does (it copies a "title" from one place to another) it is not evident anywhere nearby *why* it does it. Only by seeing how GETTL is used is the reason for it made clear. The SCOPE command

```
←. SHOW WHERE GETTL IS CALLED
```

will display the context of the calls to GETTL:

```
{in COMMAND :}
  (GETTL BUF HEADER)
  (GETTL BUF FOOTER)
```

The resulting display shows that GETTL is called twice, once with TTL the HEADER and once with TTL the FOOTER. It also shows that GETTL is used as a subprocedure to only one other routine (COMMAND).

While SCOPE's SHOW command prints out the immediate context of the use of the symbols asked about, sometimes a more thorough examination of the context of use of a piece of program is necessary. In such cases the programmer might employ the SCOPE EDIT command in order to use the interactive editor to more thoroughly explore the surroundings of the reference. Thus, after the command

```
←. EDIT WHERE GETTL IS CALLED
```

the user is placed in the editor pointing at the first occurrence of GETTL in the COMMAND routine. COMMAND has a fairly lengthy set of cases in a clause

```
(SELECTQ (COMTYP BUF) --).
```

The two occurrences of GETTL appear in distinct clauses:

```
(SELECTQ --
  --
  (HE (GETTL BUF HEADER))
  (FO (GETTL BUF FOOTER))
  --).
```

The documentation of the program operation indicates that the formatter has two commands, **he** and **fo**, which respectively set the title line on the top of the printed page and on the bottom. Putting these clues together, it is possible to infer that HEADER and FOOTER are two buffers which are used to store the current lines for the page head and foot, and that GETTL is a routine which extracts a title from a command line.

Through the use of the reference information which SCOPE provides, the programmer can

answer questions which arise when examining a program: *Why is this here? What does it do?*

Cross reference when making changes

> Suppose you have to convert a 5000-line Fortran program from one computer to another, and you need to find all the FORMAT statements, to make sure they are suitable for the new machine. How would you do it?
>
> One possibility is to get a listing and mark it up with a red pencil. But it doesn't take much imagination to see what's wrong with red-penciling a hundred pages of computer paper. It's mindless and boring busy-work, with lots of opportunities for error. And even after you've found all the FORMAT statements, you still can't do much, because the red marks aren't machine readable. [Kernighan & Plauger 1976, p. 1]

Many program changes involve widely scattered portions of the program text; making those changes is difficult because the programmer is uncertain that he has caught all of the places which need to be changed.

For example, the **FORMAT** program currently maintains two separate variables while formatting: CURPAG is the current output page number, while NEWPAG is the number of the next page. Suppose that the programmer wants to change the program using only one variable rather than two (this is Exercise 7-5 in Kernighan & Plauger). The programmer would like to find and edit the places where CURPAG and NEWPAG might be used. First, the programmer might use SCOPE's SHOW command to get an overview of the ways in which the variables are used:

```
←.  SHOW WHERE ANY USE NEWPAG OR CURPAG
FORMATINIT :
   (SETQ CURPAG 0)
   (SETQ NEWPAG 1)
COMMAND :
   (SETQ CURPAG (FORMATSET CURPAG VAL (ADD1 CURPAG)
(IMINUS HUGE) HUGE))
   (SETQ NEWPAG CURPAG)
PHEAD :
   (SETQ CURPAG NEWPAG)
   (SETQ NEWPAG (ADD1 NEWPAG))
   (PUTTL HEADER CURPAG)
PFOOT :
   (PUTTL FOOTER CURPAG)
done
```

The user decides that the proper change is to eliminate CURPAG and to replace occurrences of CURPAG with NEWPAG-1. The command to SCOPE to EDIT WHERE ANY USE CURPAG makes the change simple. Because SCOPE does the bookkeeping, making sure that *every* reference is examined, and presenting them one by one to the programmer, the change is much less risky.

SCOPE's ability to interactively locate references to a symbol is particularly useful when changing a data structure; in a conventional programming system, a change to a data structure often can ripple through the system, taking large amounts of time and leaving residual bugs. SCOPE's EDIT WHERE command can be used when changing a data structure to find all of the places which refer to the data structure or any of its parts.

Changing a name

One specific application of cross reference information is in efficiently making global program manipulations. For example, programmers occasionally want to be able to change the name of a symbol in a program, perhaps because the old name is not as intuitive, or it conflicts with a new name. Performing this operation using a program editor is difficult for several reasons. First, finding the places where the symbol is referenced may be inefficient, involving scanning of many source files of text. Cross reference information can be of use in reducing the number of places that need to be examined. Secondly, in a language where the same names can be used for different purposes, it is necessary to look for the use of symbols in a particular semantic context, rather than employing a simple text substitution rule. To aid the programmer in this task, SCOPE includes a RENAME facility which automatically performs all of the necessary actions to rename a symbol; the SCOPE command

```
←. RENAME THE FUNCTION COMMAND TO BE ProcessCommand
```

will (a) give COMMAND's definition as a function to ProcessCommand, and (b) invoke the INTERLISP editor on all locations which reference COMMAND as a procedure, changing them to ProcessCommand but leaving alone the occurrences of the symbol COMMAND used as a variable.

Flow information

SCOPE also provides summary information about program *flow* which is useful when trying to understand a large or unfamiliar program. Flow information includes facts about the *way* in which one procedure calls another, as well as information about the use of variables.

In LISP programs, it is often useful to be able to determine if the value of a function is used or if it is called only for its effects (e.g., as with an output routine). LISP, unlike many languages, does not enforce any distinction between procedures and functions. SCOPE, during flow analysis, distinguishes between a CALL FOR EFFECT and a CALL FOR VALUE, so that a programmer can easily ask about the ways in which a procedure is used. For example, SCOPE will tell the user whether the value of the procedure GETTL is ever used in response to the query IS GETTL CALLED FOR VALUE; the NO response indicates that the value returned by

GETTL is not used but is merely an accidental byproduct.

The reader of the procedure GETTL is left with some questions. One might reasonably guess that the values BLANK, TAB, NEWLINE, SQUOTE, and DQUOTE are constants. However, the command:

```
←. SHOW WHERE THE VARIABLES USED FREELY BY GETTL ARE SET
```

will quickly confirm the guess:

```
{in FORMATINIT :}
  (SETQ BLANK (QUOTE % ))
  (SETQ TAB (QUOTE %        ))
  (SETQ NEWLINE (QUOTE %
))
  (SETQ SQUOTE (QUOTE %'))
  (SETQ DQUOTE (QUOTE %"))
```

Apparently, the variables are all set initially in the routine FORMATINIT and never changed elsewhere.

Side effects

SCOPE analyzes the side effects of procedures during flow analysis, in addition to the information it collects about procedure calls and the use of variables. SCOPE's side effect information is also useful to programmers dealing with unfamiliar code because the code's meaning can be understood much more readily if it is possible to be sure that procedures mentioned have no invisible side-effects. If the programmer is examining a function GETTL and wants to understand how GETTL's subfunctions might affect the execution state of the program, the query WHAT CAN BE CHANGED BY WHICH FUNCTIONS THAT ARE CALLED BY GETTL can be used. This will cause SCOPE to print out a summary of the side effects of any of the procedures referred to inside GETTL:

```
SCOPY - CAR
```

This display indicates that the SKIPBL routine, which is also called by GETTL, has no side effects but that the SCOPY routine has possible side effects; namely, SCOPY can modify the CAR of some list, i.e., perform RPLACA's.

Type information

The results of type analysis can also help in the understanding of a poorly documented procedure; knowing the types of variables provides important clues as to how a program will execute. In strongly typed languages type declarations are an important kind of documentation,

but in languages in which type declarations are not often used, it is difficult to determine the types of variables or the types of the values returned by procedures. Knowing that a value is a number rather than a list can be a critical piece of information in understanding an unfamiliar routine. This is especially important when the same language primitive is used to mean different operations, depending on the type of its arguments, e.g., many programming languages use the symbol "+" to denote both integer addition and floating point addition while some even use the same symbol for string concatenation and addition of non-scalar (array) quantities.

SCOPE provides type information in several different ways. First, the user can ask SCOPE about the types of the arguments of a procedure and the value the procedure returns, e.g., `GETTL EXPECTS WHICH ARGUMENTS TO BE WHAT` will display the types of GETTL's arguments:

```
BUF  -- LIST
TTL  -- LIST
```

while `WHAT TYPE DOES SKIPBL RETURN` will show that SKIPBL returns a `LIST`.

Second, SCOPE's type information can be used to find the type of a given form. The user can point in the INTERLISP editor to an expression and ask SCOPE to show the type which that expression is expected to be from its context, as well as the type which it is inferred to be from its internal structure. For example, the expression `(BUF:1=BLANK)` is known to be of type `BOOLEAN`. (The type `BOOLEAN` means an value which is either `NIL` or `T`.)

2.2 CHECKING FOR ERRORS

Errors which involve examination of the whole of a large program are difficult for people to find, but easy for SCOPE. Examples of errors which SCOPE can detect include the misuse of free variables and violation of modularity constraints; other errors which SCOPE could easily be programmed to detect include uninitialized variables and type violations.

Misuse of dynamic variables

SCOPE can help with the misuse of dynamic variables, a common error in LISP programs. Many LISP systems allow dynamic binding of variables; that is, a procedure can reference a variable, and the identity of the referenced variable is determined by the run-time context in which the procedure is called. While dynamic binding is quite powerful, it often leads to programming errors. Indeed, incorrect use of free variables is one of the most common errors in INTERLISP programs. Errors occur because the interface to a routine (its interactions with

the "outside world") includes not only its arguments but also any other data structures to which it might refer (e.g., properties of atoms or values of free variables). The global program analysis performed by SCOPE can detect possible misuse of dynamic variables.

For example, suppose that the programmer decides that the set of "blank" characters to be ignored in a command line should be a parameter of the FORMAT routine. Then FORMAT, SKIPBL, and GETTL might be coded as follows:

```
(FORMAT
  [LAMBDA (STDIN STDOUT BLANKS)                    (* text formatter main program)
    ....])

(GETTL
  [LAMBDA (BUF TTL)                                (* copy title from buf to ttl)
    (while BUF:1 memb BLANKS do BUF←BUF::1)
    ....])

(SKIPBL
  [LAMBDA (BUF)                                    (* skip blanks)
    (if BUF:1 memb BLANKS
        then (SKIPBL BUF::1)
      else BUF])
```

This change is fine, unless there is a calling path to SKIPBL along which the variable BLANKS is *not* bound. In that case, BLANKS will be uninitialized. SCOPE's "local-free" error check would detect such a bug.

One problem with SCOPE's error checking is that the error check is only an approximation. For example, consider the following two functions:

```
(CALLER
  [LAMBDA NIL
    (if condition1 then (PROG(FREEVAR) (CALLEE))
     else (CALLEE])
```

and

```
(CALLEE
  [LAMBDA NIL
    (if condition2 then FREEVAR
     else NIL])
```

Suppose that CALLER is an entry (it is externally accessible), and CALLEE is not. In this situation, SCOPE will report that there is a possible program error because there is a way for CALLER to reach CALLEE without FREEVAR being bound. However, if $condition_2$ implies $condition_1$, no actual program execution would actually use FREEVAR incorrectly. (Although this example may seem contrived, analogous but more complex examples have occurred in real INTERLISP programs.) In this example, it is evident that any further refinement of the error

check would require mechanical analysis that $condition_2$ implies $condition_1$—in general, not a decidable question.

Experience has shown, however, that SCOPE's error signal, even when not accurate, is an important warning for the programmer. Good programming practice dictates that programmers should avoid constructs which are error prone, and thus, even though the above situation is not technically an error, it is wise for the programmer to fix it anyway—if SCOPE can't understand that it is not in error, very often, neither can programmers.

While dynamic variables are peculiar to LISP systems, a similar kind of analysis is often useful in other programming languages. For example, the signal mechanism in the MESA language [Mitchell, et al. 1978] shares many of the characteristics of dynamic variables in that the execution context of the procedure which raises a signal determines which catch phrase will be invoked. Errors involving signals which might not be caught are similar in nature to the errors involving dynamic variables, and an essentially identical error mechanism can be employed to discover them.

Checking imports and exports

There has been growing recognition that modularization, the constraining of interactions between separately developed pieces of program, is an effective mechanism for improving the flexibility and comprehensibility of a system while allowing the shortening of its development time [Parnas 1972a,b], [Morris 1973]. While many recent languages provide mechanisms for enforcing programmer-declared constraints on the interactions between programs, many older languages, including LISP, do not. SCOPE, however, can be used to find violations of user-declared constraints on the cross reference; that is, the user can declare for a given package that it *exports* a set of symbols (they are available externally) and that it *imports* another set of symbols. The export of a package is the set of symbols defined internally which are used outside, the import the set of symbols defined outside which are used inside. The export and import of a package can be computed directly from the cross reference information that SCOPE stores. If the user has given SCOPE a declaration of exports and imports, SCOPE can check if any additional cross-package references have been made, and warn the programmer; alternatively, SCOPE can be used to create the initial export and import declarations from the actual cross-references which exist. The programmer can decide if the formal separation of components is worth the effort. In some cases, e.g., when doing small "throw away" programs, extra mechanisms for enforcing modularity are cumbersome. When an interactive mechanism such as SCOPE is available, the enforcement of such constraints becomes an administrative decision rather than a technical one. This flexibility to either enforce constraints or not is greater than that found in programming systems which automatically enforce modularity

constraints.

Type violations

While SCOPE's type inference mechanism is designed to infer the types of variables and procedures, it can also be used for type *checking*. Type checking provides assurance that the programmer has not supplied parameters of an incorrect type to a procedure. For example, given the procedure:

```
(LAMBDA (X)
  ... (IPLUS X 3) ...
  ... (CAR X) ...]
```

SCOPE will infer from the IPLUS that X must be of type NUMBER. When SCOPE sees the expression (CAR X), it will infer that X must be of type LIST. If the value of X cannot be changed between the CAR and the IPLUS, then SCOPE will deduce that X must be both a NUMBER and a LIST; however, there is no value which is in the intersection of those two types. Thus, SCOPE will deduce that X is of type NONE. Type-checking thus can be performed by checking the results of type inference for items whose type is NONE.

2.3 CODE IMPROVEMENTS

One application of the kind of information which SCOPE can provide is code optimization performed by a compiler. In this application, SCOPE provides information directly to the compiler rather than to a programmer. There are three ways in which SCOPE can help improve the quality of compiled code. First, SCOPE's flow and side effect information can be used when performing code improvements; many common program optimizations have preconditions which are expressed in terms of the effects and uses of the expressions involved. Second, the type information which SCOPE derives can be used when compiling. Third, SCOPE can help the programmer organize his program into blocks and to check for errors in compiler declarations. These applications are explained below.

Code transformations

There are many different aspects to code optimization. Tradeoffs exist for any given optimization between how much it costs to implement and the amount of improvement possible; in some situations, code optimization is not worth while. Many transformations for code improvement (in optimizing compilers and elsewhere) can be expressed in terms of code movement; that is, given two pieces of program, certain optimizations can be performed if it does not matter in which order the pieces are evaluated ([Allen & Cocke 1971], [Aho &

Ullman 1977]). While it is difficult to accurately characterize when the order of evaluation of two expressions can be exchanged, there is an approximation which works quite well: it is permissible to switch the order of evaluation of two forms if the effects of one are independent of the usage of the other. SCOPE's effect and usage analysis computes information which could be used by an optimizing compiler when performing code transformations. Given two expressions e_1 and e_2, e_1 followed by e_2 is equivalent to e_2 followed by e_1 when the effects of e_1 are disjoint from the usage of e_2 and vice versa; i.e., neither can change something the other uses.

For example, the program fragment:

```
(VAL←(GETVAL BUF))
(CT←(COMTYP BUF))
(DOCOMMAND CT VAL)
```

can be rewritten as

```
(DOCOMMAND (COMTYP BUF) (GETVAL BUF))
```

if the variables VAL and CT are not used subsequently in the program (or by DOCOMMAND) and the expressions (COMTYP BUF) and (GETVAL BUF) can be exchanged. SCOPE's information that COMTYP has no effects, that the GETVAL cannot change anything used by COMTYP, and that DOCOMMAND does not use VAL or CT freely means that the transformation can be made.

Using type declarations

Type declarations are used in many programming languages to aid the compiler in generating efficient code for various constructs in the programming language. This is especially true for constructs which involve operators which are "overloaded", i.e., which have different meanings depending on the types of the operands. For example, in many programming languages, the operator "+" is used for addition of a variety of data types.

If the type of the operands is known at compile time, the compiler can generate more efficient code, e.g., linking the code directly to the specific operation which is requested, rather than a more complicated routine which must test the nature of its arguments before proceeding.

Block compilation

SCOPE's information can be used to improve program performance by aiding the programmer in separating a program into tightly coupled components. INTERLISP, as well as

many other systems, provides facilities for grouping together a set of code. Generally, calls within a block are much less expensive than calls across block boundaries, and often blocks are allocated in contiguous memory spaces and automatic memory management treats them as a unit. Given a large program, SCOPE is quite useful when attempting to separate programs into blocks because of its capabilities of quickly showing the inter-block flow structures. From SCOPE's flow properties, it is possible to find connected components of a program—subsets of the procedures which have no interactions with any other procedure.

In INTERLISP, SCOPE can provide additional assistance. The INTERLISP block compiler makes certain restrictions on constructs which are legal inside blocks; SCOPE's CHECK command will check for violation of those constraints.

Chapter 3—Characteristics of SCOPE's Representation System

SCOPE remembers facts about the programs that it analyzes. A system for remembering facts is often called a "representation framework", a scaffolding into which facts can be placed. In general, a "representation" is a distillation of aspects of the world. Suppose a "snapshot" of the world in a particular state is taken at some instant in time. Call this state **world-state**. Through some mapping **M**, a representation (call it **knowledge-state**) is created which corresponds to **world-state**. **Knowledge-state** corresponds with **world-state** in the sense that questions about **world-state** may be answered by direct observation of the world state or by questioning of the corresponding **knowledge-state** (Fig. 3-1).

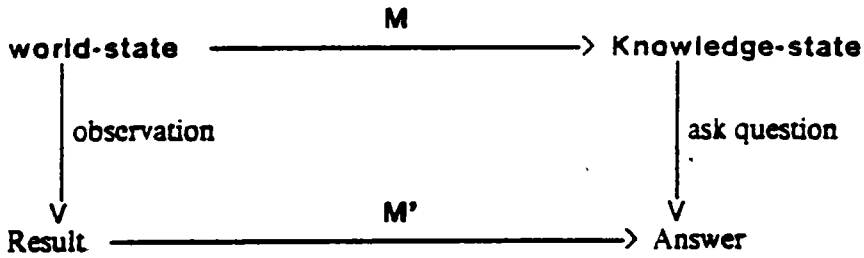

Figure 3-1: Mapping between world and knowledge states (from [Bobrow 1975])

In SCOPE, the world which is being modeled is the program under development; its state is the current version of the program as it is being modified by the programmer (Fig. 3-2).

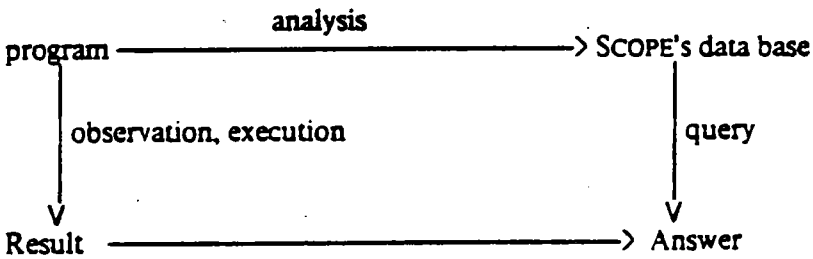

Figure 3-2: Mapping between program and SCOPE's data base

Much recent work in artificial intelligence has focused on developing representation systems for encoding knowledge about the real world within a computer system. Several lines of research have concentrated on developing general purpose representation systems which provide a ready-made framework for representing facts, performing inferences based on those facts, and changing the memory of the system based on the changes to the real world; general purpose representation systems include KRL [Bobrow & Winograd 1976], KLONE [Brachman

1978], and FRL [Roberts & Goldstein 1977].

Real world representation systems encounter many difficult representation problems. The small domain of simple predicate facts about programs with which SCOPE deals is much more specialized and thus it was possible to build a special representation system to hold SCOPE's knowledge about its world.

Bobrow [1975] characterizes representation systems along several different dimensions; SCOPE's representation system will be described in terms of the dimensions outlined below:

- *Units* and *relations:* What is being represented? How do objects and relationships in the world correspond to units and relations in the model?
- *Exhaustiveness:* Does the model represent not only the truth, but the whole truth?
- *Operational correspondence:* In what ways do the operations in the representation correspond to actions in the world?
- *Inference:* How can facts be added to the knowledge state without further input from the world?
- *Access:* How are units and structures linked to provide access to appropriate facts?
- *Self-awareness:* What knowledge does a system have explicitly about its own structure and operation?

3.1 UNITS AND RELATIONS

The objects with which SCOPE deals are pieces of programs, and in particular, definitions of symbols. The "grain" of SCOPE's knowledge is deliberately coarse; SCOPE does not model properties of individual statements in the program, the micro-syntax of symbols, the presence of formatting information, etc. SCOPE knows individual facts about procedures, variables, data structures, and other pieces of a program which can be assigned as the definition of symbols.

The knowledge SCOPE has about definitions of symbols is a set of properties—simple relations derived from analysis of the pieces. The relations form a class of "approximate assertions" [Cousot & Cousot 1979, Wegbreit 1975b] which hold about individual pieces of programs. Relations are of the form $R[sym_1,...,sym_n]$, i.e., a predicate defined on a set of symbols. For example, a simple flow relation is **MayCall**; the assertion **MayCall[A,B]** holds if the procedure **B** can be called by the procedure **A**. Another relation derived from type-analysis is **Returns**; **Returns[F,T]** holds if the value of the function **F** is of type **T**. Different relations deal with different kinds of information. For the most part, relations in

SCOPE are 1, 2 or 3 place predicates. For example, a one-place predicate might be **Recursive[F]**, meaning that the function **F** can call itself. The predicate **ExpectsArg[F,A,T]** is a three-place predicate which means that the function **F** expects its argument **A** to be of type **T**. (Appendix I contains a complete list of SCOPE's relations.)

The summary information which the analysis routines provide is of a highly structured and restricted nature, e.g., arbitrary verification conditions could not be represented in this formalism. This is an important difference between SCOPE and verification systems for proving programs correct; most verification systems can represent, for example, arbitrary quantified predicate calculus equations in their internal representation of program properties.

3.2 EXHAUSTIVENESS

A representation system is exhaustive with respect to a property if, for any object, the property is always stored explicitly with the object whenever the object has the property. In a non-exhaustive representation system, the absence of a fact in the knowledge base does not signify that the fact is not true. The important distinction is whether it is possible to make inferences about the absence of information.

A related characteristic of representation systems is its currency; that is, whether the knowledge base is always "up to date". A system can be exhaustive even if it only will contain the "whole truth" after some finite amount of computation.

SCOPE's representation system is exhaustive. It is possible to make inferences based on the absence of a property in SCOPE's data base and, in fact, for most applications those inferences are the most common. For example, the relation **MayCall[A,B]** denotes that a call to **A** may result in a call to **B**; however, in many applications it is most useful to know what things **A** does *not* call.

However, SCOPE's knowledge is hardly ever "up to date". Thus, until some query is made which requires knowing about calling relations which might involve **A**, no analysis of **A** will be performed; if **A** changes, then re-analysis will be postponed until necessary. In this way, SCOPE avoids the computational problems normally found in exhaustive representation systems which require all information to be up to date.

3.3 OPERATIONAL CORRESPONDENCE

In a general representation system, operations on the model-state are made to reflect

changes in the world. A major design problem in modeling actions is updating the representation with respect to a chain of changes caused by a single action—situations where one change causes another change which in turn causes a third.

In the world of programs, actions are program changes. SCOPE modifes its state of knowledge about a program to reflect changes to the program by propagation of changes from one SCOPE relation to another. When remembered information is the result of computations using values which may change, it is necessary to propagate the changes. One of the most important features of SCOPE is its ability to maintain the illusion that what it knows always reflects the current state of a program, even during an interactive editing and debugging session. In this case, "illusion" is not being used in a pejorative sense; rather, it is an essential property. To maintain the illusion of being up to date, SCOPE detects when the definition of a symbol has changed, and marks the data base appropriately. When a question is asked which requires a particular fact, SCOPE first checks to see if the information in its data base is no longer valid, and performs whatever reanalysis is necessary before answering the question.

The concern for change propagation is actually one of computational complexity. As Moriconi [1977] points out when describing his incremental verification system, incremental systems respond to changes by ensuring that the final problem solution is consistent and by keeping intact as much still-valid work as practical. Both the user and system perspective on how this happens is important. From the user's viewpoint, the system keeps intact still-valid work without redoing previous work. In actuality, however, a limited amount of reprocessing may be desirable.

There is a spectrum of ways in which program analysis systems can keep intact still-valid work. At one end of the spectrum is the most straightforward way to respond to changes—simply redo everything. Although this "batch" approach conveys an incremental view to the user, it is often too inefficient. For example, answering a simple cross reference question `WHO CALLS FOO` after a change would require re-scanning the entire program—much too costly for a very large program. The approach at the other end of the spectrum would be to isolate the exact impact of changes and not redo any still-valid previous work. This too can be highly inefficient since it may require as much work (or more) to figure out how to modify the data that it would to recompute it.

The strategy employed by SCOPE lies somewhere between these endpoints. Programs in INTERLISP fall naturally into "chunks"—namely, definitions of symbols, e.g., the definitions of procedures, data structure types, macros. When any part of a definition changes, SCOPE recomputes any information which depends on that definition; SCOPE does not note *what* the change was.

When a changed definition *is* reanalyzed, SCOPE notices whether the information computed has changed (by comparing the result of the analysis against the previous information in the data base). If the computed information has changed, the changes are propagated to any relations which might depend on the changed relations.

Re-analysis and change propagation is triggered by queries to SCOPE which require the information to be up-to-date. SCOPE limits the kinds of analysis it performs in response to a query to those that are necessary to answer the query.

For example, suppose the user originally had the following set of procedures with side effects:

```
(F1 [LAMBDA (X Y) (X:FIELD1 ← Y)])
(F2 [LAMBDA (X Y) (X:FIELD2 ← Y) (Y:FIELD1 ← 3) (F1 X 3)])
(F3 [LAMBDA (A B) ... (F2 A B) ...]).
```

SCOPE's side effect summary information indicates that procedure F1 can change a FIELD1 element, procedure F2 can change FIELD1 and FIELD2, and F3 the same.

If the user edits F1, it is necessary to recompute all of the properties which depend on the definition of F1. Suppose the programmer changed F1's definition to be

```
(F1 [LAMBDA (X Y) (X:FIELD2 ← Y)]).
```

When asked any question about side effects, SCOPE first checks to see if its knowledge is up to date. In this case, it knows that F1's side effect information is possibly wrong, since F1 has been edited. When SCOPE reanalyzes F1, it notices that the set of side effects of F1 have been changed too. Because side effect information is propagated during analysis, SCOPE knows that its side effect information for F2 (and for any other procedure which calls F1) is possibly incorrect as well, and must be reanalyzed. However, when SCOPE reanalyzes F2, it notes that the description of the possible side effects of F2 have not changed, and thus the change need not be propagated; there is no reason to reanalyze F3.

3.4 INFERENCE

The relations defined by SCOPE's analysis routines are, for the most part, independent of each other, in the sense that one does not determine any other. For example, the **Call** cross reference relation, which says that one procedure mentions another, cannot be deduced from any combination of other relations. There are some cases, however, where SCOPE employs translation rules which define one relation in terms of others. There are two reasons why the capability to define one relation in terms of others is important. First, some of SCOPE's

relations which are of interest to the user are not output directly by the analysis routines. Second, this capability allows for a reduction in the size of SCOPE's data base.

The particular facts that a user might wish to know are not necessarily the facts which SCOPE's analysis routines represent. For example, the SCOPE relation **NotLocalFree** is used for detecting errors involving dynamic variables. (**NotLocalFree[VAR]** means that **VAR** is used in a situation where it is possibly not dynamically bound.) **NotLocalFree** is not computed directly from the analysis routines, but rather is computed from the relation **Entry** and the flow analysis relation **RefFree**. (**Entry[FN]** means that the user has declared **FN** to be an externally available procedure; **RefFree[FN,VAR]** means that a call to **FN** can result in a free use of **VAR**.)

A relation which is defined in terms of other relations is referred to in the data base literature as a *view*; the important characteristic of views is that the user of the data base does not know which information is stored directly, and which is derived from the stored information. When applications are independent of the format of the stored information, changing the format of the stored information or adding new information is simplified.

SCOPE implements views by providing the ability to define new relationships in terms of the old. For example, the relation **MightSetUnkown[FN]** (which means that **FN** can perform an assignment to a variable whose identity is only known at run-time) is defined as the disjunction of several **MayCall** relations. (There are nine primitive functions in INTERLISP which perform the assignment of variables whose identity is only determined at run-time.) The relation **!MayCall** (which means that one procedure can reach another) is defined as the transitive closure of the flow relation **MayCall**.

One parameter of a view is an indication whether the new relation should be recomputed every time a request is made or if it should be remembered (and forgotten if the information from which it was computed changes). For example, the **NotLocalFree** relation is not stored but is computed when necessary. On the other hand, SCOPE's **!MayCall** relation, once computed, is remembered between SCOPE queries (until a change occurs which invalidates it).

The choice between storing information and recomputing it affects the efficiency and storage space of the system. Several factors enter in making the decision. The major considerations are the relative frequency of a relation changing with respect to the frequency that it is interrogated, and the simplicity of computing the relation. **NotLocalFree** is simple to compute from **RefFree** and **Entry** and so is not stored directly. **RefFree** is not as simple to compute and is interrogated frequently with respect to how often it changes, and so is stored. **!MayCall** is difficult to compute but is also asked about infrequently; it is stored, but

if a change occurs which might affect it, the entire relation is discarded.

Facts are added to SCOPE's data base without further program analysis by deducing relations according to its relation definition tables. SCOPE does not employ a general-purpose inference mechanism, but rather simplified inference techniques, because the assertions have a particularly simple form. The procedures for computing defined relations are attached to the relations which are so defined. In addition, when a new relation is defined, the dependencies are propagated so that each relation has an associated table of other relations which must be checked after changes.

3.5 ACCESS

An important feature of a representation system is the mechanism by which information can be retrieved. General representation systems often contain mechanisms which help in improving the efficiency of question answering by attachment of links in critical areas.

In SCOPE, the domain is simple enough that the straightforward model of relational data base retrieval could be used as the way in which questions are answered. There is little need for matching (which might be needed to handle queries now outside of SCOPE's capabilities, like "which functions use successive approximation" [cf. Rich & Shrobe 1976]) or for many of the other complications found in general-purpose representation systems.

The mechanism by which information is retrieved from SCOPE's data base is a query. SCOPE's intermediate query language is described in Appendix III. Generally, a query consists of a predicate calculus equation which the data base will instantiate or reject.

3.6 SELF AWARENESS

The final relevant dimension along which representation systems are often placed is the dimension of self-awareness—how much does the representation system know about its own operations. SCOPE is not self-aware, in the sense that there are any relations which describe the state of its own data base (of course, SCOPE has been used to analyze SCOPE, but it was not aware that it was operating on itself). There are two places, however, where SCOPE contains information about its own state of knowledge and processing. First, in order to improve the efficiency of query handling, SCOPE has a rough idea of the efficiency of different ways of processing queries. (Query optimization in SCOPE is discussed in Chapter 6.) Second, SCOPE "knows what it doesn't know", i.e., it is able to decide, when answering a question, that the information stored is out of date or non-existent—a minor form of self awareness.

3.7 CONCLUSIONS

A system such as SCOPE which attempts to encode knowledge about another system must naturally take a simplistic and incomplete view of the world it is attempting to model. No description can completely encapsulate all that is knowable nor is this a reasonable goal, for the reduction of information content within the representation is both a strength and a weakness. The distillation of a few interesting facts from all possible facts can reduce an information management problem to a tractable size. So it is with the view of programs which SCOPE embodies: the view of programs and knowledge about them is intentionally and necessarily limited.

Nevertheless, SCOPE's representation framework is adequate for handling the jobs to which it has been applied. The notion of simple relations which involve the definition of symbols has been adequate to summarize the results of cross-reference, type analysis, and flow analysis (although not the general flow-analysis symbolic expressions of Rosen [1979]). The inference and access mechanisms have been powerful enough to provide useful information to programmers about their programs.

Chapter 4—What SCOPE Knows About Programs

SCOPE collects information about programs while analyzing them. This chapter discusses in detail the different kinds of information SCOPE collects and the meaning of the information. The program properties known to SCOPE fall roughly into four categories—cross reference, data flow, type, and filing—each of which is discussed below. A comprehensive list of the program properties is in Appendix I.

4.1 CROSS REFERENCE

Cross reference information is information about the location of references to symbols. A "reference" is a mention of a symbol in a way which indicates some dependence on the meaning of the symbol. Reference information is often computed using simple text processing techniques. However, reference information computed this way is not always accurate because cross reference information is not just a textual property of the program—a reference may not be directly evident from the program's text. For example, in INTERLISP (and in other languages which allow macros) "hidden" references can occur because of macro expansions; it is necessary to parse the program and expand macros before the references become evident.

There are two applications of cross reference which may help clarify its meaning. First, reference information can be used to eliminate obsolete pieces of a program. If X has a definition, no other definition refers to X, and X is not externally available, then X's definition has no effect on the meaning of the program and can be removed. Second, reference information can be used to find the repercussions of a change to a piece of a program. In order for the programmer to be aware of the effect of a change, those parts of the program which refer to the changed parts must be checked for interactions (a more rigorous check for interactions is usually not possible). If a definition changes, only definitions which refer to the changed definition will be affected. Of course, second order effects must also be considered; if one part of the program changes, it may cause a chain of effects which reaches many other parts. Examination of the first order references may reveal to the programmer which references require further tracing.

Cross reference in INTERLISP

INTERLISP has many different types of symbol definitions (e.g., functions, record types, macros, edit commands) and each possible interaction between one type of symbol and another must be represented by a separate relation. SCOPE maintains several relations which denote different kinds of references. A reference to a procedure is denoted by the **Call** relation. A

reference to a variable is denoted by the **Use** relation. References to records (data structures) and their fields are denoted by the **UseAsRecord** and **UseAsField** relations respectively. The relation **FieldOf** between a record and a field means that the field is defined as part of the record. Other cross reference relations analyzed by SCOPE are **EditInvoke**, **EditCall**, **TopLevelCall**, **MacroCall**, and **FileCall**.

Obstructions to exact information

Cross reference in INTERLISP is not as simple as it might seem at first glance because some INTERLISP features interfere with the gathering of exact cross reference information; when the features are used the dependence of one piece of program on another is not evident from the text of the code. There are two categories of features in INTERLISP which obstruct exact information.

First, there are those features which, when misused, will cause programs not to work in the way one normally expects: the INTERLISP interrupt system, which can cause arbitrary user-defined computations to occur at arbitrary places in the computation, (2) the INTERLISP error-handling mechanism, which can be modified by the programmer to redefine what happens at what would normally be an error, and (3) those facilities in INTERLISP which allow programmers to dynamically modify their programs (e.g., `PUTD`, `DEFINE`, or `EDITF`). The answers SCOPE gives are made with the implicit assumption that these features are not used to change existing pieces of program.

The second type of obstruction to exact information involves the ability in LISP to invoke arbitrary procedures or assign arbitrary variables. For example, given a call to the function `EVAL`, determining what definitions might be referred to in the execution of the call is difficult. Fortunately, only a few of the primitives in the foundations of INTERLISP contribute to inaccurate cross reference information, and these primitives are used infrequently. SCOPE notes when these primitives are used, and is able to warn the user when its analysis may be incomplete.

4.2 FLOW INFORMATION

Data flow analysis has been studied extensively in the literature of global program optimization. Execution of a program normally implies the input of data, operations on it, and the output of the results of these operations in a sequence determined by the program and the data. The sequence of events is a flow of data from input to output. Data flow information is derived not from an execution of the program being analyzed but rather from a static analysis of the program. It is a summary of information available at specific program statements

derived by propagating the semantics of individual statements through the program in a manner which reflects the control structure.

In procedural languages, it is necessary to summarize the flow properties of procedure bodies so that the summary information can be used at the point of call. There actually are many different summary properties of programs which can be derived using flow analysis techniques. Flow properties include possible procedure calling sequences, use of variables, and side effects. The general techniques for computing flow information remain the same independent of the kind of information being computed.

Side effects

A particular data flow property relates to side effects. The execution of a procedure can cause changes to the run-time state of the program in addition to any values which might be directly reported as a result. These changes are called side effects (presumably to distinguish them from the main effects of a program). Side effects present serious problems in program understanding since modifications of shared data structures allow global interactions which are difficult to understand, even for experienced programmers. In addition, inference of the types of complex structures requires knowledge of the possible side effects of a procedure call. For example, in the program

```
...
LST←<A P+Q>
BUF←(SKIPUNTIL BUF LST)
Y←LST:2
...
```

it is only possible to infer that `Y` is an integer if `SKIPUNTIL` can be guaranteed not to modify the `CDR` field of `LST`.

Difference between flow and cross reference properties

Flow properties may resemble cross reference properties even though their interpretation rests on different foundations. For example, the mention of one procedure by another is an important cross reference property; the possibility that one procedure might invoke another directly during execution is a flow property. Cross reference is more inclusive that flow; there are more ways in which a procedure can be mentioned than in a direct call (e.g., the situation of one procedure passing another as an argument). Flow properties thus reflect characteristics of the set of possible executions of a piece of program rather than the dependency of the pieces. However, in many large programs, there is an exact correspondence of the cross reference and flow relations between procedures.

Flow information for INTERLISP

The INTERLISP flow properties of which SCOPE keeps track include procedure calls (**MayCall**), variable use (**Bind**, **Ref**, and **Set**), use of records and fields (**Fetch**, **Replace**, and **Create**), and general information about data structure use and effects (**Uses**, **Affects**). This set of flow properties, while not exhaustive, is sufficient for the current applications of SCOPE.

Because LISP is primarily an applicative language which encourages short procedures, the emphasis in data flow analysis techniques differs from that of more sequentially organized languages such as FORTRAN and ALGOL. Because procedure calls and recursive procedures are common, it is important to deal correctly with those constructs; because long sequences of intertwined GO's are uncommon, it is possible to allow a great deal of imprecision when those constructs are encountered.

Variable use

SCOPE's flow information includes relations which describe the way in which procedures use variables. Variables in INTERLISP can be bound in a PROG or LAMBDA, either internally to a function or as an argument. For flow analysis, it is necessary to split functions up into "frames" which correspond to PROG or LAMBDA bindings. For example, the program

```
(COM2
  [LAMBDA (BUF)
    (PROG ((W (WIDTH BUF)) (BLANKS (LIST TAB SPACE)))
          ... (COM3 W 3) ...])
```

binds the variable BUF in its top level frame, and the variables W and BLANKS in an interior frame.

SCOPE generates names for the interior frames so that they can be identified; in this case, the top frame is the name of the function COM2, while the first frame is called COM2:1. The **SubFrame** relation holds between a frame and its subframes, e.g., **SubFrame**[COM2,COM2:1]. The **SubFrame** relation is built-in in SCOPE, i.e., SCOPE does not need to store any data to represent it.

SCOPE separates flow information relating to procedure calls (**MayCall**) and variable use (**Bind**, **Ref**, and **Set**) for each frame. Thus, SCOPE knows that the $\mathbf{Bind}_f$ relation holds between COM2 and BUF, and also between COM2:1 and W and BLANKS. SCOPE is able to tell that the call to COM3 in COM2 will occur under the COM2:1 frame, i.e., at a time when W and BLANKS are bound. This information is necessary to detect possible misuse of free variables; if the call to COM3 occurred outside of the binding of BLANKS and if COM3 uses BLANKS freely,

there would be a possible free variable error.

Although flow information is computed separately for each frame, most applications only require information on how the entire procedure behaves. In general, the summary flow behavior for a procedure can be computed from the specific information for its frames. For example, the **MayCall** relation can be formally defined as $\mathbf{SubFrame}^* \circ \mathbf{MayCall}_f$, the composition of the closure of **SubFrame** with $\mathbf{MayCall}_f$. Thus, **MayCall**[COM2, COM3] holds because **SubFrame**[COM2, COM2:1] and $\mathbf{MayCall}_f$[COM2:1, COM3] hold.

Side effects in INTERLISP

Another kind of flow information SCOPE maintains is a characterization of the effects of a procedure as well as the external state upon which the procedure relies. This characterization consists of a finite set of elements; each element describes some part of the state of the program execution which can be changed or used. The INTERLISP Virtual Machine definition [Moore 1976] defines "fields" which are elements of data structures in INTERLISP. Some of these fields are directly accessible to the programmer, e.g., CAR and CDR fields of a CONS cell or the PROPLIST field of an atom, while some are only accessible indirectly, e.g., the END-OF-FILE field of an open file or the CONSCOUNT field for the system. For each primitive operation in INTERLISP, SCOPE has a description of the fields the primitive might use and the fields it might modify. For example, RPLACA modifies the CAR field while CAR uses the CAR field of a CONS-cell; GETPROP uses PROPLIST, CAR, and CDR, SETFILEPTR modifies a file's POSITION and FILEPOINTER, while PRINT both uses and modifies POSITION and FILEPOINTER as well as modifying FILECONTENTS. (To reduce the burden of creating the initial data base of uses and effects, some distinct fields are grouped together in SCOPE's initial data base. For example, SCOPE does not distinguish between any of the different fields in a readtable. In addition, a number of fields are grouped together under OTHER. Finally, there is a use/effect category called ANY which subsumes every other field. This is used for primitives which can have unknown side effects on the program, e.g., the INTERLISP-10 function CLOSER can modify arbitrary memory locations.)

SCOPE's characterization of side effects is very rough. For example, SCOPE makes no distinction between one CAR field and another, because there is no analysis of possible structure sharing. SCOPE assumes that *any* two structures can be shared. However, this characterization is sufficient for enabling many compiler optimizations [Steele 1978], and also presents useful program documentation.

Unusual control structures

SCOPE's analysis of program flow does not deal with some of the more unusual control

structures allowed in INTERLISP. For example, SCOPE's flow analysis routines do not currently deal with ERRORSET used in tandem with intentional errors or with RETFROM. During analysis, SCOPE keeps a relation **UsesUnusualControlStructures**, so that, when responding to a request to check for programming errors, SCOPE can warn when its analysis might be incorrect. SCOPE also maintains the relations **MightSetUnknown** and **MightCallUnknown** so that it is always known when an arbitrary variable might be set or an arbitrary form evaluated; these relations are used to avoid making compiler transformations which might be incorrect and to temper error messages. For example, if a procedure binds a variable which is apparently not used by any routine called beneath it, but there is some path to a procedure which **MightCallUnknown**, then SCOPE can add a caution to its warning message which indicates that the unknown procedure might be the one which uses the given variable. Some flow analysis algorithms in the literature are able to analyze flow relations more accurately in the presence of procedure valued variables. In INTERLISP, however, the cases where procedure variables are used and where the range of values can be determined for those procedures is actually quite small.

4.3 TYPE INFORMATION

The definition of "type inference" as it is used in the literature depends on whether the language for which the inference is being performed is strongly typed or allows dynamic types. For strongly typed languages, a variable or function value can be of one and only one type; further, there is a predefined finite set of existing types. Type inference in strongly typed languages is the process of choosing *the* correct type for each variable or procedure. Languages like LISP, SETL, and APL, however, allow dynamic types. Such languages do not *a priori* have typed variables, but allow any variable to assume any value. For example, in LISP the same variable can be used both to store a list at one time and a number at another. The "type" of a variable is thus similar to a range specification. An analogy to type inference in dynamically typed languages would be the specification, in a FORTRAN program, of the numeric ranges of values for all of the variables. For some variables, it is not possible to draw any inferences about the range of values (the values can assume any value allowed by the implementation) while other variables *can* be restricted to a small range. Type inference of this nature can be performed in a heuristic manner: Since "any value" is a legitimate range specification, a legitimate (but useless) result of type inference might be that every value is within the range "any value". The goal of type inference is to improve the restrictions assigned to values as much as is computationally feasible.

Type information in INTERLISP

When SCOPE analyzes a program, it determines sub-ranges within which values must remain in order for the program to execute correctly. Summary type information for a function includes the range of values expected for each argument (**ExpectsArg**), the range of values which might be returned (**Returns**), the range of values assigned to free variables during its execution (**SetFreeType**), and the range of values expected of any free variables (**ExpectsFree**).

In INTERLISP, the structure of the set of values that a variable can assume is much more complex than a simple number line and the descriptions for sets of values are thus necessarily more complex. As with side effects, the characterization of the ranges of values for variables and procedures can be described in a multitude of ways; while any description scheme can be refined, the undecidable property of range analysis prevents any description from being completely accurate. The type descriptions are expressed using an extension of INTERLISP's DECL package [Teitelman et al. 1978, p. 24.53].

Briefly, the type description language allows range descriptions to be a specific list of constants; for example, a value which is either NIL or T can be described as (MEMQ NIL T). The names of INTERLISP's basic data types can be used as type descriptions; for example, floating point numbers are described by FLOATP and literal atoms by LITATOM. A type description can provide a predicate which elements of the type must satisfy; for example, (LITATOM SATISFIES (NTHCHAR X 1)='A) denotes the set of literal atoms whose first character is the letter "A". Type descriptions can be a set of alternative types; for example, (ONEOF LITATOM FLOATP) denotes the set of values which are either literal atoms or floating point numbers. A complex type can be "named" and the name used interchangably with its definition; for example, NUMBER is defined as (ONEOF FLOATP FIXP)—either a floating or a fixed point number.) Type descriptions for composite data structures (e.g., records and CONS cells) can include restrictions on the types of the fields; for example, (LISTP WITH CAR FLOATP) describes list cells whose CAR field contains a floating point number. Type declarations can be recursive; for example LIST, the type of "proper" lists, is declared as (LISTP WITH CDR LIST). The type mechanism allows for abstract type declarations via the SUBTYPE mechanism; for example, the declaration of FILENAME is merely (SUBTYPE LITATOM). This declaration says that a FILENAME is a kind of a literal atom, without giving any other information about the restriction. The type mechanism can deduce that if X is a file name, it is also a literal atom. However, the importance of a FILENAME is that certain INTERLISP primitives expect to be passed a filename (specifically the input-output functions). Finally, the type ANY subsumes all other types, and NONE denotes the empty set.

4.4 FILING PROPERTIES

The word "filing" is used here to denote those organizational properties of the program which are independent of the meaning of the program. e.g., which piece was edited by whom at what time. These properties have little to do with the meaning of the program, but rather about the process with which the program is created and maintained.

Filing in INTERLISP

Filing properties in INTERLISP include those which the progammer has communicated to SCOPE or the programming system; these include properties about which pieces of the program belong in which file (the **Contain** relation) as well as information about which piece was edited by whom at what time (the **Edited** relation). In addition, the **Entry** relation, which describes a declaration the user has made about his program, is classed under filing.

Proper interpretation of filing information also has subtle complications. For example, the SCOPE relation **Contain** denotes the relation between a file and the names which it defines. If **Contain[FILE,NAME,DEFTYPE]** is the relation which denotes that definition of **NAME** as a **DEFTYPE** is stored on the file named **FILE**, there are several different principles which might be implied:

(1) Loading **FILE** will modify the definition of **NAME** as a **DEFTYPE**.

(2) Loading **FILE** will cause **NAME**'s definition to be completely specified. (No part of **NAME**'s definition is not on **FILE**.)

(3) Changing the definition of **NAME** as a **DEFTYPE** means that previous external representations of **FILE** are obsolete.

(4) **FILE** contains the entire current definition of **NAME**.

Relation (4) is the most common instance of **Contain** in the INTERLISP file package and also is the most straightforward; (4) implies that (1), (2), and (3) hold.

These different meanings for **Contain** are mutually independent; all combinations of (1), (2) and (3) can be achieved in the INTERLISP file package. Representing these different relations complicates SCOPE's interface to the file package.

4.5 CONCLUSIONS

The program properties with which SCOPE deals are by no means all-inclusive; the possible summary properties of programs are without number. The choices for SCOPE were made because they met particular needs and showed the variety of program properties which could be dealt with in a single representation scheme. In each instance, the information SCOPE deals with has the characteristic that correct (but possibly inaccurate) information can be gathered for *any* program; increasing accuracy is available at additional computational expense.

Chapter 5—Program Analysis Techniques

SCOPE employs several different kinds of analysis; each kind of analysis is performed by a separate analysis module. This chapter discusses the methods of program analysis (in general and as performed by SCOPE) and some of the problems involved.

In general, program analysis takes a piece of program, and computes a set of relations which hold for that piece. Analysis begins with a parse tree of the piece of program being analyzed. Each analysis routine "walks down" the parse tree, generating new assertions at each node in the tree. It is sometimes necessary to do some global searching (e.g., to find all of the labels in a block) and often necessary to keep track of "state". In many respects, analysis resembles symbolic execution; the analysis routines trace through the code as an interpreter might, although all alternative branches are explored.

In LISP, the parse trees for programs are easily obtained as the S-expression representation of the program. LISP is one of the few programming languages which has a natural representation of programs as data. For other languages, more complicated analysis is necessary to obtain a parse tree. Furthermore, the basic syntax for LISP is very simple; LISP programs consist of nested "forms" where each form is a list and the head of the list is either a special token or a procedure name.

5.1 CROSS REFERENCE ANALYSIS

For cross reference analysis, the parse tree gives indications of the use and identity of symbols. Even in languages like ALGOL it is simple to distinguish a procedure reference, although in some languages there might be ambiguity, e.g., in FORTRAN it is not possible to immediately tell the difference between a reference to a global array and a function call.

Cross reference analysis in SCOPE

While SCOPE uses the basic method of recursive descent analysis of the parse trees of LISP programs, special problems are encountered. Through the years, LISP systems and INTERLISP in particular have acquired many syntactic extensions. There are several ways that syntactic extensions have been implemented; the way in which they were implemented affects the complexity of analyzing them. Some syntactic extensions were implemented as "macros", e.g., one syntactic construct would have a description of how it was to be translated in terms of regular LISP expressions. In such a case, SCOPE's analysis can merely translate and analyze the translation. On the other hand, some of the extensions, e.g., the LISP constructs `PROG`, `AND`, `OR`, are implemented using "`FEXPR`s" (special forms for which the arguments are passed unevaluated), with the LISP compiler extended to handle them as special cases. In general,

three choices are available for handling syntactic additions: (1) extend the analysis routines to include knowledge of the additions; (2) treat the extensions as macros; or (3) provide a way to associate with each form a data structure which describes features of the syntactic form. Choice 1 makes analysis routines more complex. Choice 2 is not always possible; for example, while is always possible to translate an AND expression into an equivalent COND, a COND cannot in general be translated into more primitive operations. Choice 3 requires the user to specify new templates for each new syntactic extension and thus is burdensome.

SCOPE uses templates when performing cross reference analysis (choice 3), but expands macros (choice 2) when performing all other types of program analysis. In a few instances (for simplicity or because macro expansion is not possible), SCOPE's analysis routines contain special purpose code for LISP syntactic additions (choice 1). The structure of the templates in SCOPE is described in Appendix IV.

Even without templates, simple cross reference analysis on LISP forms requires understanding of the "argument type" of various functions, because the text of the function does not determine whether the arguments to called functions are LAMBDA or NLAMBDA. Thus, even if a function has no template or simplification, it is necessary to remember if it expects its arguments to be evaluated. For this reason, SCOPE maintains a relation **NLambda[FN]** which denotes that **FN** expects unevaluated arguments.

Multiple analysis routines

INTERLISP has many different kinds of symbol definitions, each with a separate syntax. Different analysis routines are required for each of them. For example, record declarations must be analyzed separately to find their cross reference relations. Editor commands, "top level" commands, compiler macros, and templates each require a separate analysis routine to determine the cross reference relations in which they are involved. In some cases, the analysis of cross reference in definitions of symbols was available as part of the INTERLISP environment; for example, the INTERLISP record package itself provides sufficient information to compute the **FieldOf** cross reference relation. Some of the INTERLISP "commands", e.g., top level commands (LISPXMACROS) and debugger commands (BREAKMACROS) require use of SCOPE's function-cross reference routines.

5.2 FLOW ANALYSIS

The literature contains several algorithms for computing interprocedural flow information, and work is continuing on development of efficient techniques [Rosen 1979, Barth 1977, Banning 1979]. Each of the algorithms for flow analysis uses a different computational strategy, and computes somewhat different information. The techniques for flow analysis represent

tradeoffs between computational expense and expected quality of computed information. The most appropriate algorithm for a particular application will depend on the value placed on the quality of information computed.

The differences in interprocedural flow analysis algorithms revolve around the handling of the summary of properties of sub-procedures. One important characteristic of interprocedural flow analysis is the nature of the summary information computed for each procedure. In particular, the flow algorithms described by Barth [1977] and Banning [1979] use simple binary relations to represent the summary flow information for a procedure. The flow algorithms described by Rosen [1979], however, use a more complicated representation.

In computing flow information, it is important to distinguish between two categories; those properties which "may" hold and those which "must" hold. The distinction becomes important when tracing flow through conditionally executed code. For example, a variable is extraneous if there is no way in which it might be used. A use which only happens in some executions will still imply that the variable is not useless. On the other hand, an assignment to a variable is extraneous if, during any subsequent path through the program the variable will be reassigned. In this case, the assignment must occur in every possible subsequent path.

Flow analysis in SCOPE

In the range of flow analysis techniques, SCOPE's algorithm results in relatively poor quality flow information, but with low computational overhead. SCOPE uses its relational representation scheme for handling the intermediate summary properties of procedures. SCOPE's analysis is simplified because all of the flow properties SCOPE computes are "may" properties. For example, the **Ref** relation denotes only that a variable *may* be used within a function. In order to compute **Ref**, SCOPE needs only to accumulate the variable references it sees during the recursive examination of the program.

SCOPE's flow analysis notices procedure calls (for **MayCall**), binding and use of variables (for **Bind**, **Ref**, and **Set**), use of records and fields (for **Fetch**, **Replace**, and **Create**), and use of data structures (for **Uses** and **Effects**). Analysis starts with the LAMBDA-expression definition of the procedure, and notices the variables bound as arguments. Then analysis examines the body of the procedure. Built-in INTERLISP control primitives (e.g., COND and AND) are traversed recursively; at procedure calls, SCOPE notices the procedure (for **MayCall**) and then analyzes the arguments (unless the procedure is known to be a FEXPR by the **NLambda** relation). Variable assignments are detected at SETQ's. SCOPE also recognizes the appropriate context for record package accesses. When a new set of bindings is encountered (as an internal PROG or LAMBDA), SCOPE generates an appropriate frame name to associate with subsequent properties.

Side effects are accumulated in a separate application of the flow analysis routines; no notice is taken of frames or frame names when accumulating information for **Uses** and **Affects**. Because SCOPE is not currently able to analyze structure sharing, side effect analysis consists merely of accumulating the uses and effects of any INTERLISP primitives which occur in the definition of the function being analyzed.

Applying flow information

Most applications of flow information are based on the accumulated effects of a set of functions reachable via a chain of calls (**MayCall**): this is expressed using the corresponding augmented flow relations. For example, the preconditions for most code transformations are generally expressed in terms of the **!Uses** relation (= $\mathbf{MayCall}^{*} \circ \mathbf{Uses}$) rather than what any individual function might use. In addition, when applying SCOPE's flow information, it is necessary to take into account the possible use of unusual control structures. For example, the correct test to determine if two procedure applications can be interchanged is:

- There is no data structure which one uses (**!Uses**) and the other modifies (**!Affects**).
- If one procedure has unknown references (i.e., it **!Uses ANY**), the other must have no effects at all (i.e., there is no value the other **!Affects**); similarly, if one of them has unknown side effects (i.e., it **!Affects ANY**), the other uses no data structures.
- There is no field which one can reference (**!Fetch**) and the other modify (**!Replace**).
- There is no variable which one function can set (**SetFree**) which the other uses (**RefFree**).
- If one procedure can modify an arbitrary variable (**!MightSetUnknown**), the other cannot reference (**RefFree**, **!MightUseUnknown**) any variable.
- Neither one uses unusual control structures (**!MightCallUnknown** + **!UsesUnusualControlStructures**).

5.3 TYPE INFERENCE

For strongly typed languages, a variable or function value can be of one and only one type. Type analysis can be performed by application of a small set of rules. (Gordon et al. [1977, pp. 36-40] describe a system which performs type inference in a strongly typed language.) Algorithms for determination of type in typeless language have been described by Tenenbaum [1974] and Kaplan and Ullman [1978]. Jones and Muchnick [1979] describe a method for type inference in a simplified LISP-like language which allows side effects. Generally, type inference algorithms employ two techniques. Both start with the assumption

that the program is to operate correctly, and attempt to infer the ranges of variables and procedures starting from that assumption. The two techniques used are *forward inference* and *backward inference.*

Forward inference across a single statement takes the *a priori* assumptions of variable types, and modifies them by the possible effects that the statement might have. For example, after the statement `X←SIN(THETA)`, it can be assumed that `X` is a floating point number in the range [0,1].

Backward inference, on the other hand, takes the *a posteriori* assumptions of variable types, and strengthens them according to any additional requirements that the statement might imply. For example, across the same statement `X←SIN(THETA)` it can be inferred that `THETA` must be a number.

Sequences of statements are analyzed by chaining the information propagation across them. For example, given the sequence `X←SIN(THETA); Y←X+1`, forward inference results in the additional assertion that `Y` is a floating point number in the range [1,2].

Type inference in SCOPE

SCOPE's type inference starts with a simplification of the definition of the function to be analyzed. Mapping functions and `PROG`'s are transformed into (possibly recursive) internal functions [Moore 1975], macros are expanded, and whenever possible, complex expressions are put into a canonical form. For example, `SELECTQ`'s are transformed into an equivalent `COND`. The simplification is performed so that the rest of the type analysis is not filled with special cases for each syntactic extension to INTERLISP.

SCOPE's type inference routine is given a set of pre-conditions, a set of post-conditions, and a form. Its job is to propagate the pre-conditions forward and the post-conditions backwards in a symbolic execution of the form, generating new conditions for proper execution. SCOPE uses the type information for called functions (**ExpectsArg** and **ReturnsType**) when a procedure call is found, and **Affects** to possibly modify the types in the presence of side effects. For example, in analyzing the form

```
(SETQ W (WIDTH BUF))
```

the precondition that `BUF` is a `LIST` is inferred because `WIDTH` expects its first argument to be a `LIST`; the postcondition that `W` is a `LIST` is inferred because `WIDTH` returns a `LIST`. Any preconditions which describe the type of `W` are removed from the derived postconditions. If `WIDTH` had side effects, e.g., if `WIDTH` could change the `CDR` field of some list, then the

preconditions would be "weakened" by removing any assertions about CDR's of any list structures.

Recursive procedures are more difficult to analyze. To analyze a system of mutually recursive procedures, SCOPE first invents new "abstract" data types which correspond to the **ExpectsArg** and **ReturnsType** of each of the procedures. SCOPE then performs type analysis to compute new types. If the new types do not mention the old types, they are replaced as the values of **ExpectsArg** and **ReturnsType**. If the new types do mention themselves, they are used as the definitions of recursive data structures [cf. Jones & Muchnick 1979].

Complications

Although it is possible to convert an arbitrary PROG into a collection of recursive functions, not all of the transformations are made. In particular, SCOPE's type analysis does not perform well when faced with GO's which extend across PROG boundaries or RETURN's which occur inside computations. In such situations, the only pre-conditions generated are those which can be inferred by the program up to the first label or GO, while the only post-conditions inferred are those derived by the disjunction of post-conditions of all of the embedded RETURN expressions. The types of the values manipulated by a program is a more precise description of the operation of the program than SCOPE's simple flow analysis, and for that reason type analysis is more strongly affected by the same obstacles to exact information. In particular, type inference inside PROG's is difficult, because of the possibilities of a GO or RETURN. A RETURN or GO embedded within an expression indicates that any subsequent execution may conditionally not be executed. It is easier to derive information from a mapping function than from the equivalent PROG.

5.4 CONCLUSIONS

A variety of program analysis techniques are employed in SCOPE. Among the possible techniques reported in the literature, SCOPE's analysis methods tend to be those which provide sufficient quality of information without enormous computational expense. Thus, for example, SCOPE does not analyze structure sharing when characterizing side effects because (a) SCOPE's weak characterization of side effects is sufficient for many applications (e.g., compiler improvement), and (b) the quality of information derived from state-of-the-art techniques is not sufficiently better to warrant the extra effort.

Chapter 6—Implementation Notes

SCOPE is implemented in several parts (Figure 6-1). Users communicate to SCOPE via the SCOPE command language. The **Parser** parses the command language and produces queries; the **Interpreter** interprets the parsed structure and produces queries. The **QueryProcessing** module uses SCOPE's relation-description tables to cause any required program analysis and to translate the input queries into data base accesses. The **DataBase** module accesses and maintains the data base of facts. Each of these parts is discussed below.

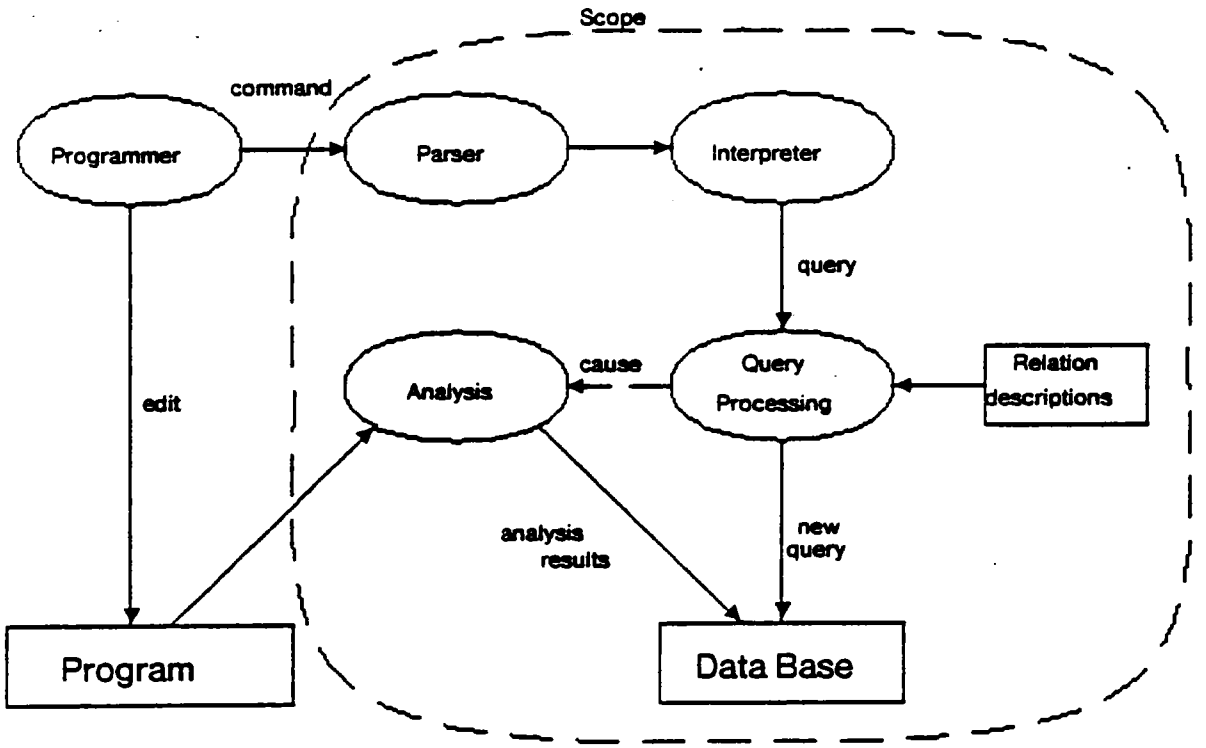

Figure 6-1—Implementation of SCOPE

6.1 PARSER

> You can't state something simple to an unknowledgeable mechanical recipient and expect it to alter its behavior in major ways. [Standish 1971]

The interface to SCOPE is in two parts. First, an English-like command language is available for casual interactive use. (SCOPE's command language is described in Appendix II.) Second, a formal query language is available for more efficient access to primitive operations. (The formal query language is described in Appendix III.) SCOPE's command language

processor translates queries in the SCOPE command language into the formal query language.

SCOPE's command language processor employs a top-down recursive descent algorithm with lookahead and partial backtracking. In parallel to the top-down parse, a bottom-up scanner looks for adverbs and prepositional phrases which are attached to the components which they most likely to modify. Language tables describe the classes of nouns, verbs, modifiers, etc., along with attachments which indicate how the words are to be interpreted. The command language is simple enough so that a more sophisticated parser is not necessary. SCOPE's language was designed so that, while it is a very limited subset of English, any command which *is* accepted is interpreted with a meaning which corresponds to the intuitive meaning of the English sentence. SCOPE's command language language was designed to be comprehensive and flexible, with frequently used commands kept as simple as possible.

Spelling correction

If a command cannot be parsed, SCOPE attempts to correct spelling by retrying the parse with a "fuzzy match" of terminal symbols in the parse tree. (The standard INTERLISP spelling correction routines are used to select probable respellings among the candidates generated by SCOPE.) Only one misspelling per phrase is allowed as a compromise between the extreme of looking for all possible misspellings and merely giving up and saying that the command was not parsable. Because of the combinatorial explosion of possibilities, looking for multiple misspellings would probably take much longer than the time it would take the user to recognize the error, correct it, and re-type the command.

6.2 INTERPRETER

Once a command has been parsed and an internal representation of its structure generated, SCOPE translates the parsed structure into a retrieval request. In general, a noun-phrase corresponds to a quantified referent, while a verb corresponds to a relation. Quantification is assumed to take a left-right precedence.

SCOPE's command processor translates the command language input of the user into a formal query of the data base. User programs can directly interface with the data base using the same formal query language. The formal query language consists of first order predicate calculus equations. For example, `WHICH FUNCTIONS THAT BIND X CALL Y` translates into the formal query `(find Z suchthat (AND (Bind Z 'X) (Bind Z 'Y)))`.

6.3 ANSWERING QUESTIONS

Once SCOPE has a query expressed in its formal query language, it must first update its knowledge about the user's program as necessary to correctly answer the question, and secondly, direct the data base module to retrieve the required information. In order to do this processing, SCOPE needs to consult its general knowledge about the relations it maintains.

What SCOPE knows about relations

There are two kinds of facts SCOPE knows: facts about a particular program and facts about programs in general. The facts SCOPE knows about a particular program are instances of relations between the pieces of that program; the facts SCOPE knows about programs in general are the rules it has which tell it how to process those relations. For example, SCOPE may know that the procedure COMMAND can cause the procedure PHEAD to be called—a fact about a particular program. SCOPE knows that the **!MayCall** relation (used to express facts about one procedure eventually calling another) is the transitive closure of **MayCall**—a general piece of information which is independent of any specific program.

SCOPE knows several general properties about each of the relations with which it deals (Figure 6-2). For each relation, it knows the types of each element which can be involved in the relation, the manner of derivation, whether the relation is stored in the data base, and information about propagating changes. These elements are discussed in detail below.

```
RelationName[arguments]
  test and generate procedures (if built in)
  translation rule (if computed)
  analysis routine name  (if direct result of analysis)
  data base information (if not always recomputed)
  dependency information
  change propagation information
```

Figure 6-2—What SCOPE knows about a relation

Relation arguments

SCOPE relations are facts which relate one piece of program to another. Each relation has a number of arguments. With each relation, SCOPE stores a description of the kind of elements which can be in each argument position. For example, the **Edited** relation relates a name, a type, a person, and a date.

Built-in relations

Some relations are "built-in"—no information is stored in SCOPE's data base but rather, the relation "knows" how to solve itself. For example, the **SubFrame** relation is built-in; attached to the **SubFrame** relation is a pointer to a routine which can test if one frame is a direct descendent of another, and another routine which, given the name of a frame, returns the name of the frame's parent.

Translation rules

Often, one relation is defined in terms of other relations. There are two reasons why this is desirable.

First, the relations which occur in different contexts are not necessarily the same. There are three different situations in which relations occur: (1) relations are used to express queries; for example, the question `WHO CAN CALL WIDTH` in the SCOPE command language is expressed as the query `(find X suchthat (MayCall X 'WIDTH))`; (2) relations are derived as the result of of program analysis; the flow analysis routine determines, for any procedure, the set of other procedures that it may call; and (3) information is stored in SCOPE's data base in terms of relations. The relations used in these three different ways are not necessarily the same. For example, the **!MayCall** relation can be used in a query, but is not derived as a direct result of analysis; rather, SCOPE must perform some processing to derive **!MayCall** from **MayCall**.

The second reason it is desirable to be able to define one relation in terms of other relations is because the relations in SCOPE's data base may actually be quite different from both the relations available for query and the relations derived from analysis; again, a translation is necessary.

To perform the transformation from one relation to another, SCOPE employs *translation rules*, which tell how one relation can be derived from another. Each of the relations which can be used in a query have a translation rule which says how that relation can be derived from either other query relations or from the relations stored in the data base. For example, **!MayCall**'s translation rule says that **!MayCall = MayCall***.

Relations derived from analysis

Some relations are *not* computed in terms of others, but are the direct result of analysis. For these relations, SCOPE stores the name of the anlysis routine and the proper way of invoking the analysis program.

Data base information

Some relations are not stored at all; rather SCOPE merely recomputes the information whenever a question is asked. Other relations *are* stored, and must be updated if the program has changed since the last time a question was asked. For relations which are stored, SCOPE must know where the information is stored, i.e., the name of the data base in which it resides.

Dependency information

A second kind of general knowledge SCOPE has tells it how one relation *depends on* other information. SCOPE's dependency rules are used when a part of the program has changed. Changes to the program require SCOPE to update any incorrect facts in its data base. To help SCOPE to do so, each relation in the data base has a *dependency rule* which indicates other relations or definitions upon which this relation depends. In order to simplify change propagation, the dependency rules are actually *inverted*—stored along with the relations which are depended upon.

Change propagation information

Because SCOPE propagates changes only when necessary, SCOPE must remember the specific information about what is out of date in its data base. For this purpose, SCOPE maintains a set of *invalid tables*. Each relation in the data base has a pointer to one or more of the invalid tables. Before performing a retrieval, SCOPE first propagates the invalid tables based on the dependency rules, and then updates any relations which are invalid according to the tables.

How SCOPE processes queries

SCOPE receives a query as the result of processing a user's command or directly from the program interface. Often the information required is not directly stored in SCOPE's data base but must be computed. The computations SCOPE performs are pure deduction; given a query, and the data base of facts, SCOPE uses a set of deductive rules (the relation definitions) to produce an answer to the query. Transformation of the input query into terms which the data base stores is straightforward. Each relation is either primitive (stored in the data base and computed by analysis routines) or computed. For direct computed relations, there is a simple transformation which will turn the high-level query into a low-level query via substitution. Each complex computed relation has an attached procedure for computing it, as well as information about its dependency.

When a piece of program changes, it is necessary to mark any information which depends on the definition of that piece as invalid, and then to subsequently mark any information which might depend on the invalid information as invalid in turn. Each relation has markers which indicate which of its arguments are the names of definitions upon which the relation depends. In addition, direct computed relations clearly depend on the relations in their definitions. Complex computed relations have separate tables which list the other relations upon which they depend. When SCOPE processes a query, it first checks to see if there is any piece of program described in the data base which must be reanalyzed or if any of the computed and stored relations must be recomputed. For each definition type and type of analysis, there is a table of names of pieces which have changed and for which the analysis has not been performed.

6.4 DATA BASE

Once a query has been translated and processed, the **DataBase** module retrieves the required information from the data base. While the way in which information is retrieved is best described using the relational model of data bases [Codd 1970], the implementation of the SCOPE data base differs from the implementation of large relational data bases, for several reasons. First, SCOPE's data base is usually small and does not require mechanisms which very large data bases require. Although resident memory space is at a premium for INTERLISP users, the amount of disk space taken by a SCOPE data base is small enough that minimizing the size of the data base was not important. Second, many of SCOPE's relations are binary, and a special format for binary relations could be employed to advantage. Third, the kind of retrieval requests involved are often significantly more complex than that normally found in data base systems.

SCOPE stores relations indexed by each element which can be used as an access key. For example, the **Call** relation can be used for queries "WHO DOES X CALL", which requires an index on the first argument to **Call**, and also for queries "WHO CALLS X", which requires an index on the second argument. For binary relations, the index can actually contain the relation information itself; i.e., accessing the **Call** relation on the first argument FOO will retrieve the list of functions which FOO can call. For relations with more than two arguments, the index can either contain pointers to a common "tuple" table which contains the elements in each instance of a relation or can contain the tuples themselves. Only one of the indices needs be precise; the others can be used as "hints" which must be checked. For example, the Edited[NAME,DEFTYPE,PERSON,DATE] relation can be accessed by either NAME or PERSON. The NAME index contains a list of instances of (DEFTYPE.PERSON.DATE), while the PERSON index contains only a list of NAMEs. Some indices can be shared between

similar relations because of the capability of using "hints"; for example, the **VAR** index for **SetFree** and **RefFree** is shared—the index contains a list of functions for each variable which might either set or reference them freely. In order to access **SetFree** or **RefFree** relations by their **VAR** argument it is necessary to prune the elements in the index to be the functions which are in the forward index.

In the process of developing SCOPE, three separate versions of SCOPE's indexing scheme were implemented, each of which maintains the index in a different place. One implementation stores the index in the INTERLISP address space (using the INTERLISP function GETHASH). The second stores indices in a separate address space (using the "swapped arrays" of the PDP-10 implementation of INTERLISP). A third implementation uses a hash table stored in an external file as the index. These three implementations span a continuum of speed and memory requirements. The first can be accessed quickly, but consumes memory—a resource which is scarce for INTERLISP users. The last is slower but requires only a small amount of resident space. The second implementation lies between the others in both access time and storage space.

Because of the variations in storage methods, some relations are easier to "solve" than others; for example, it is easier to retrieve the set of functions which **Call** another function than it is to find the set which **Bind** a given variable (because the **VAR** index of **Bind** is stored with a hint). Associated with each retrieval method for a relation is a "difficulty rating". When SCOPE receives a retrieval request, SCOPE rewrites the query to improve the efficiency of retrieval. Given a request which requires the mutual satisfaction of several factors, SCOPE first ranks the clauses by their restrictive power, and evaluates the most restrictive clause first. For example, SCOPE responds to a request to find the set of functions which **Bind** BLANKS and also **Call** SKIPBL by retrieving the callers of SKIPBL first and testing each one for the **Bind** relation with BLANKS rather than the other way around.

SCOPE's query processing is simple compared to query processing in conventional data bases because SCOPE's relations generally have the property that for any relation, the number of instances of the relation involving a given item is quite small (average over all relations is less than 10), and thus the result of a relational query can be assumed to fit within memory. Furthermore, no complex mechanism is needed to minimize file accesses to intermediate data structures; SCOPE can use simple linked lists in LISP for intermediate results during data base retrieval.

6.5 CONCLUSIONS

For the most part, the implementation of SCOPE is straightfoward. Perhaps the most interesting and novel part of SCOPE's implementation is its mechanism for change propagation. SCOPE is a fairly large program—on the order of 200 pages of INTERLISP code. Approximately 20% of the code is the command language parser and interpreter; another 20% makes up query processing and the data base. The analysis routines comprise another 30%. The rest of the code consists of interface to INTERLISP, the program structure printer, error checking routines, and other utilities.

Chapter 7—Future Directions

SCOPE is by no means the end of the road. There are many directions in which it could be extended and improved. This chapter describes some of the directions for research for which SCOPE can provide a point of departure. These fall into three categories: (1) improvements to the current implementation of SCOPE; (2) additional analysis techniques and applications; and (3) ways of improving programmer productivity which are beyond SCOPE-like facilities.

7.1 IMPROVING THE CURRENT IMPLEMENTATION

While there are several areas where the current implementation of SCOPE could be improved, the most important areas which need improvement are the user interface and the data base.

Improving the user interface

> Since this work has not focused on natural language understanding, a range of simple techniques has been used to provide the level of performance required.... This approach appears to be viable where unrestricted dialog is not the goal, and in domains where there is available a semiformal technical language with a low degree of ambiguity. [Davis 1978]

The user interface to SCOPE (and MASTERSCOPE) has been successfully used by INTERLISP programmers. However, the general problem of natural language understanding is far from being solved. Even within SCOPE's limited domain, problems have been encountered. For example, commands which use the conjunction "and" may be misinterpreted due to "and"'s inherent ambiguity.

The problem of presenting a consistent, easy-to-use interface to a casual user of SCOPE is quite similar to the problem which confronts the designer of an interface to a conventional data base system. The present interface to SCOPE could be extended in several ways. One way to extend SCOPE would be to provide a more extensive grammar, thus allowing greater freedom in how queries are expressed.

Natural language interfaces for question-answering systems have been explored by many researchers (e.g., Hendrix [1977], Burton [1976], Woods [1977], S.Kaplan [1979]); many difficult problems have been uncovered, some of them solved. SCOPE could benefit from the improvements which these works represent.

However, two important problems exist with the use of *any* natural language interface. First, no current natural language interface "understands" the user very well, even in domains

as limited as SCOPE's. In many respects, SCOPE's domain resembles that of the LUNAR system [Woods 1977]; in both instances, the user is manipulating a data base of facts about objects (in SCOPE, programs, and in LUNAR, moon rocks) and many of the same problems arise, e.g., difficulties with conjunctions, elipses, and resolving the scope of quantifiers [Vanlehn 1978]. Second, a natural language interface tends to lure the user into attributing greater powers to the system than warranted. Transcripts of MASTERSCOPE sessions reveal many users stumbling into this pitfall. As with other natural-language systems, some users would infer processing capabilities which were not present, e.g., they would expect MASTERSCOPE to answer such questions as "Why doesn't my program work?" and, more seriously, "Who depends on the format of X?". The natural language approach does allow users to form a simple model of the capabilities of the system.

Alternative interfaces which would not require the ability to understand natural language include query by example and graphical interfaces. Unfortunately, such interfaces lack the simplicity of use and understanding and the conciseness of expression provided by a natural language interface.

Improving the data base

While MASTERSCOPE was carefully designed to be responsive to INTERLISP users, some of the efficiency was sacrificed for generality in the implementation of SCOPE, and in particular, in the implementation of the SCOPE data base and inference modules. Fortunately, efficiency and generality in data base design is an area which is being thoroughly explored by other researchers (e.g., [System R 1976], [Milman 1977]); SCOPE's data base modules could profit by many of the techniques now being investigated.

Background computation

One direction which was briefly explored in SCOPE was the use of background computation to update SCOPE's data base. As personal computers become more commonplace, it is possible to employ the unused time ("between the keystrokes") to perform computation which might be of use in the future. In SCOPE, for example, it is possible to reanalyze changed parts of the program "in the background". Such analysis may never be used, for example, if the program changes again before any question is asked. However, if the computer has no other task to perform, background analysis can speed SCOPE's responses even when the program does undergo major changes betweeen SCOPE queries.

7.2 ADDED CAPABILITIES

The most important additional capabilities from which SCOPE would benefit are the ability to deal effectively with many different programmers on the same project, and additional analysis tools.

Multiple users

A serious deficiency with the current design for SCOPE as a practical programmer's assistant is that it is designed for use by only a single programmer. In large-scale software development, *teams* of programmers frequently work together on a single, very large program—just the kind of program where SCOPE is most useful. Extending SCOPE to be a useful tool in a multi-person project involves solving some difficult problems. One problem is that programmers would like to have a view of the system which included summaries of the publicly available parts of his colleague's programs, while seeing all of their own. Mechanisms which allow multiple users to share a set of data, but which limit access for individual users, would be required to allow SCOPE to be used easily by a team.

Other kinds of analysis and applications

Many program analysis tools exist which could be integrated with SCOPE. In particular, analysis of program performance [Wegbreit 1975a] and characterization of structure sharing [Jones & Muchnick 1979] are attractive possibilities.

Some researchers in automatic programming have pursued automatic data structure selection [Low 1976] based on a characterization of how the data structure is used. Such data structure selection could be made based on global analysis provided by SCOPE, where the data structures could change incrementally as the program was developed.

Other tools which can profit from the availability of global analysis include program testing aids (e.g., path constraint generators and constraint solvers), maintenance tools which validate modifications, tools for performance and software quality assurance (e.g., tools for estimating execution time and for simple metrics of software quality), program standards enforcers (which enforce project naming conventions, commenting conventions and detect error-prone constructs), and finally program restructuring tools [Ramamoorthy & Ho, 1977].

7.3 BEYOND SCOPE

There are several directions which deserve investigation but are beyond simple improvements to the capabilities of SCOPE. These include programmer assistants which provide more knowledgeable aid, and new directions for programming languages.

Understanding programmer intentions

While the kinds of analysis which SCOPE performs relate to the semantics of the program insofar as SCOPE understands dependency-of-computation for cross reference and the execution properties of the program for flow analysis, a more ambitious project would be to analyze programs to determine the programmer's concept of what the program does. It is likely that such understanding would require understanding of the more formal properties of programs such as SCOPE provides.

Implications for programming language design

One property of an easily understood and modified program is *locality* [Goodenough & Shafer 1976]. A program posseses locality when it is unnecessary to look very far away to understand any particular piece of code. Many of the efforts in programming language design have focused on providing programming languages which encourage locality. Computer languages can never succeed completely in the goal of providing for locality of information, however, as the one-dimensional nature of computer program listings almost always leads to the placing of related parts of a program in widely separated locations. Such separation leads to errors, as the programmer may make assumptions about the execution of a piece of program which are no longer true.

SCOPE provides a way around the one dimensional nature of programming languages, by providing an interactive index to the interrelationships of pieces of programs. The "proximity" of one piece of program to another is no longer related to their physical separation within a listing, but rather to the amount of difficulty in finding the related texts. Because SCOPE makes finding related pieces of program easy, it effectively brings different pieces of the program closer together, and improves locality by increasing the connectivity of the program.

Rule-based or production systems

> The ultimate impediment to further discovery was the lack of rules that could reason about, modify, delete, and synthesize other rules. [Lenat & Harris 1978]

Applications of SCOPE-like tools to rule-based programming languages holds much promise: the requirement for locality in the programming language weakens when the

programmer has good global analysis tools available, and global program information improves the attractiveness of languages with pattern-directed invocation [Waterman & Hayes-Roth 1978, Davis & King 1975]. One of the major disadvantages of production systems up to now has been the difficulty of debugging them, because of unforeseen interactions between old and new rules.

Beyond programming languages

Even in a programming environment enhanced by SCOPE, it remains the case that the program itself is the "truth" about the computation, and the information with which SCOPE deals is merely derived summary information; that is, the program is captured completely in its listing. A programming environment could be imagined in which no single canonical representation of the program exists. The programmer creates various descriptions of the processes to be invoked and examines the program from various viewpoints, but does not rely on a single distinguished external representation. In such a system, *a program is not its listing*; the listing is merely one of many possible external forms.

Appendix I—Relations Used in SCOPE

In SCOPE, assertions about programs are expressed in terms of relations. This section lists those relations, and briefly describes each one.

I.1 ABBREVIATIONS

The description of a relation may contain some of the following abbreviations:

Abbreviation	Meaning
cross-ref	cross-reference property
flow	data flow property.
type	type inference property.
filing	organizational property relating to file system.
built-in	Built-in relation. (Built-in relations are not derived via analysis or stored, but rather are handled by procedures inside SCOPE.)
=query	Derived via the query given.
1-1, 1-n, n-1	Restrictions on number of solutions.

I.2 NOTATION

When dealing with binary relations, it is possible to do some symbolic manipulations on the relations themselves.

Given two binary relations R and P, R∘P, the composition of R with P, is the relation defined by

$$(R \circ P)[X,Y] = \exists\ Z \text{ s.t. } R[X,Z] \wedge P[Z,Y]$$

Composition is called "join" in the data base literature. Relations can be added:

$$(R+P)[X,Y] = R[X,Y] \vee P[X,Y]$$

or subtracted:

$$(R-P)[X,Y] = R[X,Y] \wedge \neg P[X,Y]$$

or combined (intersected):

$$(R \cdot P)[X,Y] = R[X,Y] \wedge P[X,Y].$$

Given a binary relation R, R^*, the transitive closure of R, is a new binary relation given by:

$$R^* = I + R + R \circ R + R \circ R \circ R + \ldots$$

where I is the identity.

Flow relations which deal with individual frames are distinguished by a subscript f. Flow relations written R! are defined as $\mathbf{MayCall}^* \circ R$, i.e., the composition of the transitive closure of **MayCall** with the relation R.

I.3 Relation arguments

Each argument of a n-place relation has a class which describes the argument. For example, the relation **Use** relates functions and variables, and is described as **Use[FN,VAR]**. The classes of relational arguments are:

FN	a function.
VAR	a variable.
FRAME	a frame—i.e., a set of bindings generated by a PROG or LAMBDA expression.
RECORD	a record—an Interlisp declared data structure.
FIELD	a field of a record.
EC	an editor command.
TC	an Interlisp top level command (also known as a LISPXMACRO).
MACRO	A compiler macro.
N	a number—any integer.
TYPE	a description of a Lisp data type (a range of values).
NAME	the name of an arbitrary item. This is used in relations which have a separate "argument" which is the kind of definition **NAME** is supposed to be.
DEFTYPE	the type of **NAME**.
FILE	a collection of definitions (which corresponds to an external file in the filing system).

USAGE	a VM "field" name.
PERSON	the login name of a programmer.
DATE	a date specification.

I.4 SCOPE's RELATIONS

Below is a list of the relations SCOPE knows. Along with each relation are the elements it relates and a brief definition.

*** Cross reference relations**

Call[FN_1,FN_2] *cross-ref*
: **FN_1** mentions **FN_2** as a function.

Use[FN,VAR] *cross-ref*
: **FN** mentions **VAR** as a variable.

NLambda[FN]
: **FN** is a procedure which takes its arguments unevaluated. When changed, it is necessary to mark invalid *any* property of any caller (**Call**) of **FN**.

UseAsField[FN,FIELD] *cross-ref*
: The body of **FN** mentions **FIELD** as a record field.

UseAsRecord[FN,RECORD] *cross-ref*
: The body of **FN** mentions **RECORD** as a record name.

FieldOf[FIELD,RECORD] *cross-ref*
: **FIELD** is a field of **RECORD**.

Accessfn[RECORD,FN] *cross-ref*
: **RECORD** uses **FN** as an access function.

EditInvoke[EC_1,EC_2] *cross-ref*
: Denotes when one editor command invokes another.

EditCall[EC,FN] *cross-ref*
: The edit command **EC** invokes the function **FN**.

TopLevelCall[TC,FN] *cross-ref*
: The top-level command (LISPXMACRO) **TC** invokes the function **FN**.

MacroCall[MACRO,FN] *cross-ref*

The compiler macro **MACRO** invokes the function **FN.**

FileCall[FILE,FN] *cross-ref*

The initialization of the file **FILE** invokes the function **FN.**

* Flow relations

SubFrame[$FRAME_1$,$FRAME_2$] *built-in 1-n*

The top level frame of a given function is the function itself; the set of frames of a function **FN** is {**FR : SubFrame*[FN,FR]**}. The **SubFrame** relation is 1-n; given a frame, it can have multiple subframes, but any frame has at most one parent. The "top level" of a function is also a frame which binds arguments of the function. Interior frames correspond to internal PROG and LAMBDA bindings. When performing flow analysis, SCOPE makes up "frame names", one for each internal PROG or LAMBDA which binds variables so that flow relations of **MayCall**, **Ref**, and **Bind** can be identified separately for each frame. The value of a **FRAME** is a frame name; frame names have the format `FN:n:m:...` where `FN` is the name of the top function and each level of binding adds another integer onto the tail. This format is chosen so that testing **Subframe[X,Y]** is possible without any data base retrieval, as is finding the "parent" of a given frame.

$Bind_f$[FRAME,VAR] *flow*

Denotes that the variable **VAR** is bound in the frame **FRAME.** This relation associates with a frame the names of the variables bound within it.

Bind[FN,VAR] *flow*

FN contains a binding of the variable **VAR.** Note that **Bind = SubFrame* ∘ $Bind_f$**, i.e., that **Bind** is the composition of the transitive closure of **SubFrame** composed with **$Bind_f$.**

$MayCall_f$[FRAME,FN] *flow*

A call to **FN** may occur under frame **FRAME.**

MayCall[FN_1,FN_2] *flow*

MayCall = SubFrame* ∘ $MayCall_f$. Again, the relation normally used for retrieval relates one function with another; however, the calls are actually separated by the frames in which they occur.

!MayCall[FN_1,FN_2] *flow*

!MayCall = MayCall*. This means that there is some calling path from **FN_1** to **FN_2.**

Ref_f[FRAME,VAR] *flow*

The variable **VAR** is referenced in frame **FRAME**.

Set_f[FRAME,VAR] *flow*

An assignment (SETQ) of the variable **VAR** is made within frame **FRAME**.

$RefFree_f$[FRAME,VAR]. RefFree[FN,VAR] *flow*

Means that the variable **VAR** is used freely below **FRAME**. $RefFree_f$ and the relation **RefFree[FN,VAR]** can be computed using **Bind** and **Reference** with the recursive equations

$$RefFree[FN,VAR] = RefFree_f[FN,VAR]$$

and

$$RefFree_f = (Ref_f + (SubFrame + MayCall_f) \circ RefFree_f) - Bind_f.$$

That is, a frame uses a variable freely if it doesn't bind that variable, and either the frame references the variable, or some subframe or called function uses the variable freely.

$SetFree_f$[FN,VAR], SetFree[FN,VAR] *flow*

Similar to **RefFree**, with **Set** used rather than **Ref**.

Ref[FN,VAR] *flow*

The function **FN** references somewhere the value of the variable **VAR**. $Ref = SubFrame^* \circ Ref_f$.

Set[FN,VAR] *flow*

The function **FN** references somewhere the value of the variable **VAR**. $Set = SubFrame^* \circ Set_f$.

Recursive[FN] *flow*

FN can call itself, i.e., **!Call[FN,FN]**.

CallForValue[FN_1,FN_2] *flow, cross-ref*

The value returned by FN_2 might be used within FN_1. This relation includes a combination of flow and cross reference information; that is, it means that FN_1 refers to FN_2 (i.e., the cross reference relation **Call**), but in a context where it cannot be determined that the value of FN_2 will be discarded (a flow property).

Fetch[FN,FIELD] *flow*

The field **FIELD** is accessed by the function **FN**. There is no need to compute on a frame-by-frame basis.

Replace[FN,FIELD] *flow*

The field **FIELD** is assigned by the function **FN**.

Uses[FN,USAGE] *flow*

The function **FN** directly uses the value of the data structure field **USAGE**. This relation is built-in for system functions (i.e., not derived from analysis).

Affects[FN,USAGE] *flow*

The function **FN** can directly modify the data structure field **USAGE**. This relation is built-in for system functions.

CreateRecord[FN,RECORD] *flow*

An instance of the record type **RECORD** is created by the function **FN**.

* Unusual constructs:

MightSetUnknown$_f$[FRAME], MightSetUnknown[FN] *flow*

FRAME contains a call to one of the functions SET, SAVESET, SETSTKARG, SETTOPVAL, SETATOMVAL, /SET, /SAVESET, /SETTOPVAL, or /SETATOMVAL whose first argument is not quoted; this means that some variable will be assigned by the call, but SCOPE does not know the identity of the variable.

MightRefUnknown$_f$[FRAME], MightRefUnknown[FN] *flow*

FRAME contains a call to one of the functions EVALV, STKARG, GETATOMVAL, or GETTOPVAL whose first argument is not quoted; this means that some variable will be accessed by the call, but SCOPE does not know the identity of the variable.

UsesUnusualControlStructures[FN] *flow*

FN uses an ERRORSET, RETTO, RETFROM, RESUME, ENVEVAL, STKEVAL, RETEVAL, SETCLINK, or SETALINK. These are the "spaghetti stack" primitives in Interlisp, which allow flow of control regimes which differ radically from normal procedure call and return. Most of SCOPE's error analysis assumes that these primitives are not used; this relation is used to test if a warning message must be generated that an analysis might not be correct.

MightCallUnknown$_f$[FRAME], MightCallUnknown[FN] *flow*

FN contains a construct which will cause execution of code which is determined at run time; in particular, a call to EVAL, APPLY, APPLY*, ENVAPPLY, ENVEVAL, or a mapping function in which the functional argument is not quoted.

* Type inference relations

ExpectsArg[FN,N,TYPE] *type n-m-1*

The function **FN** expects its **N**th argument to be within **TYPE.**

ExpectsFree[FN,VAR,TYPE] *type*

The function **FN** expects the variable **VAR** to be of type **TYPE** to work correctly.

SetFreeType[FN,VAR,TYPE] *type*

At the termination of the procedure **FN**, the variable **VAR** has been set (freely) to **TYPE.**

Returns[FN,TYPE] *type*

The value of **FN** is of type **TYPE.**

* Filing relations

Contain[FILE,NAME,DEFTYPE] *filing*

This can be broken down into separate relations for each type, i.e., there is a relation $\textbf{Contain}_t\textbf{[FILE,NAME]} = \textbf{Contain[FILE,NAME,t].}$

Entry[FN] *filing*

FN is declared to be an entry function. **Entry** is an extrinsic property of functions: the declaration of whether a function is intended to be an entry is something which is an attribute of the file on which that function resides.

Edited[NAME,DEFTYPE,PERSON,DATE] *filing n-m-1-1*

Says that **NAME**'s definition as a **DEFTYPE** was last edited by **PERSON** on **DATE.** Extrinsic property, stored by watching editor. Given a **NAME** and a **DEFTYPE**, **PERSON** and **DATE** are unique.

UserFunction[FN] *filing*

SCOPE distinguishes between Interlisp system functions and user functions, by keeping track of which functions the user has noticed upon loading.

* Miscellaneous built-in relations

After[DATE_1,DATE_2] *built-in*

Basic predicate for comparing dates.

Covers[TYPE_1,TYPE_2] *built-in*

TYPE_2 is a subset of TYPE_1, e.g., **Covers[NUMBER,INTEGER].**

Disjoint[$TYPE_1$,$TYPE_2$] *built-in*

$TYPE_1$ and $TYPE_2$ are disjoint (no value is in both).

Meet[$TYPE_1$,$TYPE_2$,$TYPE_3$] *built-in*

$TYPE_3 = TYPE_1 \cap TYPE_2$.

Appendix II—The SCOPE Command Language

The user communicates with SCOPE via an English-like command language. Through the commands, the user can interrogate SCOPE's database and perform other operations. The basic building blocks of the language are specifications of sets—sets of functions, sets of variables, etc. A set can be specified in a variety of ways, either explicitly, e.g., `THE FUNCTION NAMED FOO`, or implicitly, e.g., `ANY FUNCTION THAT DOES NOT CALL FIE`. The specifications of sets correspond to English nouns and noun phrases.

The *relations* between sets are denoted in the SCOPE command language by verbs and prepositions. For example, the verb `CALL` is used to talk about the **Call** cross reference relation. Noun and verb phrases, plus a few additional words, form English-like sentence commands. For example, the command

```
←. WHICH FUNCTIONS ON FOO USES X FREELY
```

will print out the list of functions contained in the file `FOO` which use the variable `X` freely. The command

```
←. EDIT WHERE ANY FUNCTION CALLS ERROR
```

will direct the INTERLISP editor at those places in functions which mention the function `ERROR`, pointing at each successive expression where `ERROR` actually occurs.

II.1 NOUN PHRASES

Noun phrases are used in the SCOPE command language to denote an object or a set of objects, and consist of three parts: (1) a *determiner*, e.g., `A`, `THE`; (2) a *noun*, e.g., `FUNCTIONS`; and (3) a *restriction*, e.g., `CALLED BY FOO`. The first part of the noun phrase can also be a simple pronoun, e.g., `WHO`. The parts of the noun phrase in the SCOPE command language are explained below.

Determiners

The determiner in an English noun phrase is used for a variety of purposes; in SCOPE, it is used to mark a noun phrase as a question, or in a syntactic role to introduce the noun phrase. The determiners recognized by SCOPE include `A`, `THE`, `ANY`, `SOME`, `WHICH` and `WHAT`.

Nouns

The nouns available in the SCOPE command language are the names of the types of symbols—FUNCTION, VARIABLE, RECORD, FILE, MACRO, COMMAND and FIELD. (Of course, various abbreviations and alternate phrasings are allowed.) Noun phrases denote objects or collections of objects; these objects have a *type*, i.e., the type of symbol which is denoted. The major purpose of the noun within the noun phrase is to specify what type of symbol is being denoted; if there is no noun, the type is determined from context, e.g., in WHO IS CALLED BY X, the type of the noun phrase WHO is FUNCTION, since only functions can be called. Nouns may occur in either singular or plural form, although the SCOPE parser does not distinguish number except to resolve ambiguous parsings.

Restrictions

The last part of the noun phrase in the SCOPE command language is a specification of the range of values denoted by the phrase, and may take a variety of forms:

NAMED name — The simplest way to specify a set consisting of a single symbol is by the name of that symbol. For example, THE FUNCTION NAMED FOO. A name, used alone, takes the role of a "proper noun" in English; thus FOO is allowed for ANY NAMED FOO. Of course, there is some danger of ambiguity in the SCOPE command language, e.g., if the programmer has a function named ANY. The NAMED construct can be used to avoid the ambiguity.

IN expression — Because SCOPE is available at any time in the INTERLISP environment, one powerful feature is the ability to refer to the interactive environment from inside a SCOPE command. The IN phrase allows the user to give a LISP expression to be evaluated, and have the value treated as a list of the elements of a set. For example, THE FUNCTIONS IN (FILEFNSLST 'FOO) denotes the set of values returned by evaluating (FILEFNSLST 'FOO).

@ predicate — Another way in which the embedding of SCOPE inside LISP provides power is the ability to write arbitrary LISP predicates as a "test" for set membership in the specification of a SCOPE command language noun phrase. The specification represents all elements for which the value of "predicate" is non-NIL. For example, the phrase ANY @ GETD denotes elements which have a LISP definition.

Another way of specifying the range of denotation of a noun phrase is to give a relation which must hold for the set of objects denoted:

verbING np
verbED BY np — Refers to the set of all objects which have the relation denoted by verb with some element of the set denoted by np. The notation "verbING" is used generically to mean any of SCOPE's verbs phrases (described in Section II.2) in the present participle form. For example, USING ANY IN FOOVARS FREELY specifies the set of functions which uses freely any variable in the value of FOOVARS; CALLED BY X specifies the set of functions called by the function X.

EDITED [BY person] [AFTER date] [BEFORE date]
Restricts the set denoted to include only objects which were last edited by the person specified or during the interval specified. Uses the **Edited** relation.

Sets may also be specified with relative clauses introduced by the word THAT, e.g., THE FUNCTIONS THAT BIND X.

(FIELD) OF records — This denotes the field names of the records specified, the **FieldOf** relation.

ON A PATH pathoptions
Refers to the set of functions which would be printed by the command SHOW PATHS pathoptions. For example, IS FOO BOUND BY ANY ON A PATH TO 'PARSE tests if FOO is bound above the function PARSE. Pathoptions are explained in detail later.

RECURSIVE — Those functions which satisfy **Recursive**. The word RECURSIVE may actually occur between the determiner and the noun in a noun-phrase.

NOT restriction — This allows the restriction of a noun phrase to be complemented, e.g., THE FUNCTIONS NOT CALLED BY FOO or ANY NOT IN FOOVARS X.

Conjunctions

The role of conjunctions in English is complex; in SCOPE, only noun phrases may be joined by the conjunctions AND and OR to denote the corresponding intersection and union of the components. For example, CALLING X OR Y specifies the set of all functions which call the function X or the function Y.

Because of SCOPE's limited understanding of conjunctions, it often fails to follow the correct interpretation in English. For example, "CALLING X AND Y" would, in English, be interpreted as those things which both call X and call Y; however, SCOPE interprets the phrase as "CALLING (X AND Y)", where (X AND Y) is necessarily the empty set (in SCOPE's world, an object cannot have two different names).

II.2 VERB PHRASES

Within the SCOPE command language, verbs are used to denote the relations between sets. For example, the verb "CALL" corresponds to the SCOPE relation **Call**; the user query "WHO CALLS FOO" corresponds directly to finding the set of all **X** such that **Call[X,FOO]** holds. Sometimes, adverbs are used to modify the meaning of the verb to denote a different relation. For example, the verbs USE, SET, SMASH and REFERENCE all may be modified by the adverb FREELY. Verbs can occur in the present tense (e.g., USE, CALLS, BINDS, USES) or as present or past participles (e.g., CALLING, BOUND, TESTED). The verbs (with their modifiers) which are recognized by the SCOPE command processor are:

verb	interpretation
CALL	**Call**
MAY CALL	**MayCall**
CALL FOR VALUE	**CallForValue**
CALL FOR EFFECT	**MayCall - CallForValue**
CALL INDIRECTLY	**Call - MayCall**
CALL SOMEHOW	**!MayCall**
USE	**Ref + Set, UseAsField, UseAsRecord, Uses** (interpretation depends on context)
USE FREELY	**RefFree + SetFree**
SET	**Set**
SET FREELY	**SetFree**
CHANGE	**Affects**
REFERENCE	**Ref**
REFERENCE FREELY	**RefFree**
BIND	**Bind**
FETCH	**FetchField**
REPLACE	**ReplaceField**
USE AS FIELD	**UseAsField**
USE AS A RECORD	**UseAsRecord**

CREATE	**Create**
CONTAIN	**Contain**$_t$, where t is determined from context.
RETURN	**ReturnsType**
EXPECT	**ExpectsArg**

Unfortunately, it was not possible to choose intuitive names for all of SCOPE's relations; for example, the fine distinction between CALL (the cross reference relation) and MAY CALL (the flow relation) is not evident in the casual presentation of the SCOPE command language.

The SCOPE command language actually allows more complicated constructs when dealing with verbs which denote more than two place predicates. For example, the verb EXPECTS is used to denote the **Expects** type-inference relation. The verb EXPECTS has not only a subject and object, but also a modifier TO BE, e.g., WHO EXPECTS ITS FIRST ARGUMENT TO BE LISTP. In this example, "ITS FIRST ARGUMENT" is a special noun phrase which is taken as a representation for the value 1 in the relation **Expects**.

II.3 COMMANDS

Commands are sentences in the SCOPE command language which direct SCOPE to answer questions or perform various operations. Commands to SCOPE may take the form of a question or an imperative.

Questions

Questions in the SCOPE command language have the same format as an English sentence with a subject (a noun phrase), a verb phrase (one of SCOPE's verbs or IS or ARE), and an object (another noun phrase). Any of the noun phrases in the question can have a question-determiner, e.g., WHO or WHICH. For example, SCOPE will respond to the question WHICH FUNCTIONS CALL X with the list of functions that call X. The verb phrase in the question may be in the present tense (e.g., CALL, BIND, TEST, SMASH) or passive (e.g., in the command WHO IS CALLED BY WHO). (Other variants are also recognized, e.g., WHO DOES X CALL, IS FOO CALLED BY FIE, etc.)

The interpretation of the command depends on the number of question elements present. If there is no question element, the command is treated as an assertion and SCOPE responds either T or NIL, depending on whether the assertion holds. Thus, SCOPE will respond to the question DO ANY IN MYFNS CALL HELP with T if any function in MYFNS calls the function HELP, and NIL otherwise. If there is one question element, SCOPE will respond with the list of items for which the assertion would be true. For example MYFN BINDS WHO USED FREELY BY YOURFN results in the list of variables bound by MYFN which are also used freely by

YOURFN. If there is more than one question element SCOPE will display a table of possible results:

```
←. WHO BOUND BY WHOM IS USED FREELY BY WHO ON MYFILE
FLGX --- RECFN1   --- RESULTX
VAR3 --- REMTOP   --- GETRESLT
VARK --- LISTER   --- VARKUSER
```

This means that FLG is bound by RECFN1 and is used freely by RESULTX, that VAR3 is bound by REMTYP and is used freely by GETRSLT, etc.

Imperatives

EDIT WHERE np verb-phrase np

Invokes the INTERLISP editor on each expression where the relation specified by verb-phrase actually occurs, e.g., EDIT WHERE ANY CALLS ERROR.

SHOW WHERE np verb-phrase np

Similar to the EDIT command, but merely prints out the expressions without calling the editor. (Note that this is different from the SHOW PATHS command which displays a tree structure.)

DESCRIBE np

Prints out a summary of potentially useful information about the items denoted by np. For example, the command DESCRIBE THE FUNCTION PRINTARGS might print out:

```
PRINTARGS[N;FLG]
binds : TEM,LST,X
calls : MSRECORDFILE,SPACES,PRIN1
called by: PRINTSENTENCE,MSHELP,CHECKER
```

which shows that PRINTARGS has two arguments N and FLG, binds internally the variables TEM, LST and X, calls MSRECORDFILE, SPACES and PRIN1 and is called by PRINTSENTENCE, MSHELP, and CHECKER.

CHECK np

Tells SCOPE to check for various abnormal conditions in the functions or files specified by np.

RENAME np TO BE np

Instructs SCOPE to (a) copy the definition of the first symbol to the second, and (b) change any place that references the first symbol to instead reference the second.

SHOW PATHS pathoptions

Causes SCOPE to display a tree structure of the CALL relation, according to the pathoptions; pathoptions consists of any number of the following:

FROM np — Display CALLs which originate with elements of np.

TO np — Display the CALLs leading to elements of np. If TO is given before FROM (or no FROM is given), the tree is inverted. When both FROM and TO are given, the first one indicates a set which must be displayed while the second restricts the paths that will be traced, e.g., the command SHOW PATHS FROM X TO Y will trace the elements of the set CALLED SOMEHOW BY X AND CALLING Y SOMEHOW.

AVOIDING np — Do not display any element of np. For example, SHOW PATHS TO ERROR AVOIDING ON FILE2 will not display (or trace) any function on FILE2.

MARKING np — Adds an asterisk to the display of elements of np. For example, the command SHOW PATHS TO SCINTERPRET MARKING ANY THAT BIND X will identify, in the tree of calls which can reach SCINTERPRET, those functions which bind the variable X.

II.4 CONCLUSIONS

SCOPE's command processor, as it was first used in MASTERSCOPE, enabled MASTERSCOPE users to find a workable set of commands which gave them the results they needed; however, in order to make MASTERSCOPE even more responsive to queries of casual users, transcripts of several hundred MASTERSCOPE sessions were recorded (with permission). Scanning these typescripts often revealed reasonable sentence structures which MASTERSCOPE rejected. These sentence structures were added to the command language although they were not documented. The SCOPE command language is actually more extensive than this documentation indicates; however, a brief documentation of a working subset of the language seems better than a complex documentation of all possible commands.

Appendix III—The SCOPE Intermediate Query Language

While the natural language interface for SCOPE is convenient for casual use, SCOPE provides a more formal interface for programs which wish to query SCOPE's data base. Other programs which use SCOPE do not usually need to go through the SCOPE command parser every time a query is made. The intermediate query language resembles a first-order predicate calculus language with conjunctions, quantifiers, and base assertions corresponding to SCOPE's relations.

*** Queries**

`(FIND` bindings predicate`)`

Generally, a SCOPE query asks SCOPE to find a set of quantities. bindings is a list of variables to be filled in with the names of symbols, and predicate is a predicate which should be satisfied, e.g., `(FIND (X Y) (AND (MayCall X Y) (Bind X 'FOO) (UseFreely Y 'FOO))`.

*** Predicates**

(relationname vc vc ...)

The basic form of a predicate involving one of SCOPE's relations. vc is either a variable which has been "bound" in an enclosing `FIND` or else a constant. (A vc can also be a set specification as outlined below.)

`(AND` predicate predicate ...`)`
`(OR` predicate predicate ...`)`
`(NOT` predicate ...`)`

SCOPE allows predicates to be joined by the normal logical conjunctions to form new predicates.

`(FORALL` bindings predicate`)`
`(THEREIS` bindings predicate`)`

These have the same form as the `FIND` query, but are the corresponding existentially and universally quantified predicates.

(MEMBER var LISP-expression)
(SATISFIES var LISP-predicate)

These predicates allow a simple escape to the LISP environment, by specifying respectively that the variable is an element of the list given by the `MEMBER` expression, or that the LISP functional predicate given returns non-`NIL` when passed the value of the variable as an argument.

(IN var set) This specifies that the variable var is an element of the set set, where set is a set specification, as outlined below.

*** Set specifications**

In addition to predicates, the SCOPE query language allows specification of sets of elements. These are specified in several constructive ways:

(SETOF binding predicate)

For example, (SETOF X (Call X 'ERROR)). The binding is a variable which will be used within predicate; the SETOF specification denotes the set of values for which predicate will hold.

(UNION set set ...)
(INTERSECTION set set ...)
(COMPLEMENT set)

Sets can be joined with the normal set theory conjunctions to form new sets.

*** Using sets within queries**

A convenient abbreviation in the SCOPE intermediate query language is the ability to use set specifications as the arguments to predicates, to avoid making up unnecessary variable names. A standard transformation is performed to expand the abbreviation: a predicate (relation ..1.. set ..2..) is transformed into (THEREIS X (AND (IN X set) (relation ..1.. X ..2..))) where X is a "new" variable.

Appendix IV—Templates for Computing Cross Reference

When computing cross reference for procedures, SCOPE associates with each special LISP form a *template* which describes the pattern of a function's evaluation. In SCOPE, a template is a list structure containing any of the following atoms:

PPE	If an expression appears in this location, there is most likely a parenthesis error.
EVAL	The expression at this location is evaluated normally.
NIL	The expression occuring at this location is not evaluated. For example, the template for QUOTE is (NIL . PPE).
FUNCTION	The expression at this point is used as a functional argument. For example, the template for MAPC is (EVAL FUNCTION FUNCTION . PPE).
FIELD	An atom at this location is used as a record field.
RECORD	An atom at this location is used as a record name.
BIND	An atom at this location is a variable which is bound.
CALL	An atom at this location is used as a function.

In addition to the above atoms which occur in templates, there are some "special forms" which are lists keyed by their first element.

.. template	Any part of a template may be preceded by two periods which specifies that the template should be repeated an indefinite number ($n \geq 0$) of times to fill out the expression. For example, the template for COND might be (.. (.. EVAL)) while the template for SELECTQ is (EVAL .. (NIL .. EVAL) EVAL).
(BOTH template template)	Analyze the current expression twice, using each template in turn.
(IF test template template)	Apply test to the current expression at analysis time, and if the result is non-NIL, use the first template, otherwise the second. For example, (IF LISTP (RECORD FIELD) FIELD) specifies that if the current expression is a list, then the first element is a record name and the second element a field name, otherwise it is a field name.

Templates may be changed and new templates defined. Whenever the template for a function changes, SCOPE knows that it must reanalyze any procedure whose analysis might be affected by the template.

Appendix V—The FORMAT Program

The following program is used as an example in the text of this dissertation. The program and its documentation was taken fairly directly from Kernighan and Plauger, *Software Tools* [1976], and was chosen as an example of a particularly well-written program. In translating the program from RATFOR into INTERLISP, a few liberties were taken—since INTERLISP does not manipulate character strings or arrays of characters as efficiently as lists, lists of characters are used instead of the character arrays in the original source; this caused the internal structures of some of the routines to change. Since INTERLISP does not have call by reference, the calling structure of a few of the routines were modified to return their values rather than perform assignments to reference arguments. Outside of those changes, the program is taken intact, comments included.

This example is a text formatter—a program for neatly formatting a document on a suitable printer. It produces output for devices like terminals and line printers, with automatic right margin justification, pagination, page numbering and titling, centering, underlining, indenting, and multiple line spacing. The **FORMAT** program is quite conventional. It accepts the text to be formatted, interspersed with formatting commands telling **FORMAT** what the output is to look like. A command consists of a period, a two-letter name, and perhaps some optional information. Each command must appear at the beginning of a line, with nothing on the line but the command and its arguments. For instance,

```
.ce
```

centers the next line of output, and

```
.sp 3
```

generates three blank lines.

By default, **FORMAT** *fills* output lines, by packing as many input words as possible onto an output line before printing it. The lines are also *justified* (right margins made even) by inserting extra spaces into the filled line before output. Filling can be turned off by the *no-fill* command

```
.nf
```

and thereafter lines will be copied from input to output without any rearrangement. Filling can be turned back on with the *fill* command

```
.fi
```

The action of forcing out a partially collected line is called a *break*. The break concept pervades **FORMAT**; many commands implicitly cause a break. To force a break explicitly, for example, to separate two paragraphs, use

```
.br
```

Of course you may want to add an extra blank line between paragraphs. The *space* command

```
.sp
```

causes a break, then produces a blank line. To get *n* blank lines, use

```
.sp n
```

By default output will be single spaced, but the line spacing can be changed at any time:

```
.ls n
```

sets line spacing to *n*. (*n*=2 is double spacing.) The **.ls** command does not cause a break.

The *begin page* command **.bp** causes a skip to the top of a new page and also causes a break. If you use

```
.bp n
```

the next output page will be numbered *n*. The current page length can be changed (without a break) with

```
.pl n
```

To center the next line of output,

```
.ce
line of text to be centered
```

The **.ce** command causes a break. You can center *n* lines with

```
.ce n
```

and, if you don't like to count lines, say

```
.ce 1000
lots of lines
to be centered
.ce 0
```

The lines between the **.ce** commands will be centered.

Underlining is much the same as centering:

.ul *n*

causes the next *n* lines to be unerlined upon output. But **.ul** does not cause a break, so words in filled text may be underlined.

The *indent* command controls the left margin:

.in *n*

causes all subsequent output lines to be indented *n* positions. The command

.rm *n*

sets the *right margin* to *n*. The traditional paragraph indent is produced with *temporary indent* command:

.ti *n*

breaks and sets the indent to position *n* for one output line only.

To put running header and footer titles on every page, use **.he** and **.fo**:

```
.he this becomes the top of page (header) title
.fo this becomes the bottom of page (footer) title
```

The title begins with the first non-blank after the command, but a leading quote will be discarded if present, so titles that begin with blanks can be produced. If a title contains the character *#*, it will be replaced by the current page number.

Since absolute numbers are often awkward, **FORMAT** allows *relative* values as command arguments. All commands that allow a numeric argument *n* also allow *+n* or *-n* instead, to signify a *change* in the current value.

The FORMAT program

Here is the **FORMAT** program in its entirety. (Note that the program is presented in its "CLISP" form [Teitelman 1973], while examples in the text of this dissertation are in standard S-expression notation; SCOPE automatically invokes the CLISP processor before analyzing or presenting code to the programmer):

```
(DEFINEQ

(FORMATINIT
  [LAMBDA NIL                          (* initialize variables for FORMAT)

                (* * misc program constants * *)

    PAGEWIDTH←60                        (* width of page)
    PAGELEN←66                          (* length of page)

    PAGENUM←'#                          (* character which signals page numbers in footer and header)
    COMMAND←'%.                         (* character which signals beginning of command)

    MAXLINE←200                         (* maximum size of internal buffer)
    MAXOUT←200                          (* maximum size of output line)
    INSIZE←200                          (* maximum size of input line)

    HUGE←10000000                       (* a very large integer)

                (* * special characters * *)
    NEWLINE←'%
                                        (* end of line (carriage return))
    TAB←'%                              (* tab character)
    SQUOTE←'%'                          (* single quote)
    DQUOTE←'%"                          (* double quote)
    PLUS←'+                             (* plus sign)
    MINUS←'-                            (* minus sign)
    UNDERLINE←'%                        (* underline character)
    BLANK←'%                            (* a space character)
    BACKSPACE←'%                        (* backspace)

                (* * page formatting variables * *)

    FILL←T                              (* fill if T)
    LSVAL←1                             (* current line spacing)
    INVAL←0                             (* current indent; GE 0)
    RMVAL←PAGEWIDTH                     (* current right margin)
    TIVAL←0                             (* current temporary indent)
    CEVAL←0                             (* number of lines to center)
    ULVAL←0                             (* number of lines to underline)
    CURPAG←0                            (* current output page number)
    NEWPAG←1                            (* next output page number)
    LINENO←0                            (* next line to be printed)
    PLVAL←PAGELEN                       (* page length in lines)
```

```
    M1VAL←2                          (* margin before and including header)
    M2VAL←2                          (* margin after header)
    M3VAL←2                          (* margin after last text line)
    M4VAL←2                          (* bottom margin, including footer)
    BOTTOM←PLVAL-M3VAL-M4VAL         (* last live line on page)
    HEADER←(CHARBUFFER MAXLINE)      (* top of page title)
    FOOTER←(CHARBUFFER MAXLINE)      (* bottom of page title)

                  (* * output common variables * *)

    OUTP←0                           (* last char position in OUTBUF)
    OUTW←0                           (* width of text currently in OUTBUF)
    OUTWDS←0                         (* number of words in OUTBUF)
    OUTBUF←(CHARBUFFER MAXOUT)       (* output buffer)

                  (* * used only by SPREAD * *)

    DIRFLG←NIL

                  (* * used by TEXT * *)

    WRDBUF←(CHARBUFFER INSIZE)       (* buffer for words)

    ])


(FORMAT
  [LAMBDA (STDIN STDOUT)                          (* text formatter main program)
    (PROG ((INBUF (CHARBUFFER INSIZE)))
          (FORMATINIT)
          (while (GETLIN INBUF STDIN)~='EOF
              do (if INBUF:1=COMMAND
                     then (COMMAND INBUF)         (* it's a command)
                   else (TEXT INBUF)              (* it's text)))
          (if LINENO gt 0
              then (SPACE HUGE)                   (* flush last output)]))


(COMMAND
  [LAMBDA (BUF)                                   (* perform formatting command)
    (PROG (VAL CT)
          (CT←(COMTYP BUF))
          (VAL←(GETVAL BUF))
          (SELECTQ CT
            .   (FI (BRK) FILL←T)
                (NF (BRK) FILL←NIL)
                (BR (BRK))
                (LS LSVAL←(FORMATSET LSVAL VAL 1 1 HUGE))
                (HE (GETTL BUF HEADER))
                (FO (GETTL BUF FOOTER))
                (SP SPVAL←(FORMATSET SPVAL VAL 1 0 HUGE)
                    (SPACE SPVAL))
```

```
        (BP (if LINENO gt 0
               then (SPACE HUGE))
            CURPAG←(FORMATSET CURPAG VAL CURPAG+1
                              (-HUGE) HUGE)
            NEWPAG←CURPAG)
        (PL PLVAL←(FORMATSET PLVAL VAL PAGELEN
                             M1VAL+M2VAL+M3VAL+M4VAL+1 HUGE)
            BOTTOM←PLVAL-M3VAL-M4VAL)
        (IN INVAL←(FORMATSET INVAL VAL 0 0 RMVAL-1)
            TIVAL←INVAL)
        (RM RMVAL←(FORMATSET RMVAL VAL PAGEWIDTH TIVAL+1
                             HUGE))
        (TI (BRK)
            TIVAL←(FORMATSET TIVAL VAL 0 0 RMVAL))
        (CE (BRK)
            TEVAL←(FORMATSET CEVAL VAL 1 0 HUGE))
        (UL ULVAL←(FORMATSET ULVAL VAL 0 1 HUGE))
        (* unknown command])

(COMTYP
  [LAMBDA (BUF)                                    (* decode command type)
    (* Kernighan & Plaugher must convert from characters
       to codes for commands. It can be simpler in Lisp)
    (U-CASE (PACK* BUF:2 BUF:3])

(GETVAL
  [LAMBDA (BUF)                                    (* evaluate optional numeric argument)
    (PROG (ARGTYP)
          (while BUF:1~=BLANK and BUF:1~=TAB and BUF:1~=NEWLINE
             do BUF←BUF::1)
          (BUF←(SKIPBL BUF))
          (ARGTYP←BUF:1)
          (if ARGTYP=PLUS or ARGTYP= MINUS
             then BUF←BUF::1)
          (RETURN (create COMARG
                          TYPE ← ARGTYP
                          N ← (CTOI BUF)])

(FORMATSET
  [LAMBDA (OLDVAL VAL DEFVAL MINVAL MAXVAL)        (* return new value for parameter)
    VAL←(if VAL:TYPE=NEWLINE
            then DEFVAL                            (* default value)
         elseif VAL:TYPE='+
            then OLDVAL+VAL:N                      (* relative +)
         elseif VAL:TYPE='-
            then OLDVAL-VAL:N                      (* relative -)
         else                                      (* absolute)
            VAL:N)
    VAL←(IMIN MAXVAL VAL)
    VAL←(IMAX MAXVAL VAL])
```

```
(TEXT
  [LAMBDA (INBUF)                                    (* process text lines)
    (PROG (I)
          (if INBUF:1=BLANK or INBUF:1=NEWLINE
              then (LEADBL INBUF)                    (* move left, set TIVAL))
          (if ULVAL gt 0
              then (UNDERL INBUF WRDBUF)             (* underlining)
                   ULVAL←ULVAL-1)
          (if CEVAL gt 0
              then (CENTER INBUF)                    (* centering)
                   (FORMATPUT INBUF)
                   CEVAL←CEVAL-1
            elseif INBUF:1=NEWLINE
              then (FORMATPUT INBUF)                 (* all blank line)
            elseif FILL=NIL
              then (FORMATPUT INBUF)
            else (do (INBUF←(GETWRD INBUF WRDBUF))
                     (if WRDBUF:1='EOS
                         then (RETURN))
                     (PUTWRD WRDBUF])


(FORMATPUT
  [LAMBDA (BUF)                    (* put out line with proper spacing and indenting)
    (if LINENO=0 or LINENO gt BOTTOM
        then (PHEAD))
    (for I from 1 to TIVAL do (PUTC BLANK))
    TIVAL←INVAL
    (PUTLIN BUF STDOUT)
    (SKIP (IMIN LSVAL-1 BOTTOM-LINENO))
    LINENO←LINENO+LSVAL
    (if LINENO gt BOTTOM
        then (PFOOT])


(PHEAD
  [LAMBDA NIL                                        (* put out page header)
    CURPAG←NEWPAG
    NEWPAG←NEWPAG+1
    (if M1VAL gt 0
        then (SKIP M1VAL-1)
             (PUTTL HEADER CURPAG))
    (SKIP M2VAL)
    LINENO←M1VAL+M2VAL+1])


(PFOOT
  [LAMBDA NIL                                        (* put out page footer)
    (SKIP M3VAL)
    (if M4VAL gt 0
        then (PUTTL FOOTER CURPAG)
             (SKIP M4VAL-1])


(PUTTL
```

```
  [LAMBDA (BUF PAGENO)                          (* put out title line with optional page number)
    (on old BUF while BUF:1~='EOS do (if BUF:1=PAGENUM
                                         then (PUTDEC PAGENO 1)
                                       else (PUTC BUF:1])


(GETTL
  [LAMBDA (BUF TTL)                                    (* copy title from buf to ttl)
    (while BUF:1~=BLANK and BUF:1~=TAB and BUF:1~=NEWLINE do
BUF←BUF::1)
    BUF←(SKIPBL BUF)
    (if BUF:1=SQUOTE or BUF:1=DQUOTE
        then BUF←BUF::1)
    (SCOPY BUF TTL])


(SPACE
  [LAMBDA (N)                                   (* space n lines or to bottom of page)
    (PROG NIL
          (BRK)
          (if LINENO gt BOTTOM
              then (RETURN))
          (if LINENO=0
              then (PHEAD))
          (SKIP (IMIN N BOTTOM+1-LINENO))
          (LINENO←LINENO+N)
          (if LINENO gt BOTTOM
              then (PFOOT])


(LEADBL
  [LAMBDA (BUF)                                 (* delete leading blanks, set TIVAL)
    (BRK)
    (PROG ((CBUF BUF))
          (for I from 1 while BUF:1=BLANK
            do BUF←BUF::1 finally TIVAL←I-1)
          (do (CBUF:1←BUF:1)
              (if BUF:1='EOS
                  then (RETURN))
              (CBUF←CBUF::1)
              (BUF←BUF::1])
```

```
(PUTWRD
  [LAMBDA (WRDBUF)                    (* put a word in OUTBUF: includes margin justification)
    (PROG (W LAST LLVAL NEXTRA)
          (W←(WIDTH WRDBUF))
          (LAST←(FORMATLENGTH WRDBUF)+OUTP+1)
          (LLVAL←RMVAL-TIVAL)
          (if OUTP gt 0 and (OUTW+W gt LLVAL or LAST ge MAXOUT)
              then                                          (* too big)
                   LAST←(LAST-OUTP)                       (* remember end of WRDBUF)
                   NEXTRA←LLVAL-OUTW+1
                   (SPREAD OUTBUF OUTP NEXTRA OUTWDS)
                   (if NEXTRA gt 0 and OUTWDS gt 1
                       then OUTP←OUTP+NEXTRA)
                   (BRK)                                  (* flush previous line))
          (SCOPY WRDBUF (NTH OUTBUF OUTP+1))
          (OUTP←LAST)
          ((NTH OUTBUF OUTP):1← BLANK)                    (* blank between words)
          (OUTW←OUTW+W+1)                                 (* 1 for blank)
          (OUTWDS←OUTWDS+1])


(WIDTH
  [LAMBDA (BUF)                       (* compute width of character string)
    (PROG ((WIDTH 0))
          (for X in BUF while X~='EOS
            do (if X=BACKSPACE
                   then WIDTH←(WIDTH-1)
                 elseif X~=NEWLINE
                    then WIDTH←(WIDTH+1)))
          (RETURN WIDTH])


(BRK
  [LAMBDA NIL                         (* end current filled line)
    (if OUTP gt 0
        then ((NTH OUTBUF OUTP):1←NEWLINE)
             ((NTH OUTBUF OUTP+1):1←'EOS)
             (FORMATPUT OUTBUF))
    OUTP←0
    OUTW←0
    OUTWDS←0])


(SPREAD
  [LAMBDA (BUF OUTP NEXTRA OUTWDS)    (* spread words to justify right margin)
    (PROG (NE NHOLES I J NB)
          (if NEXTRA le 0 or OUTWDS le 1
              then (RETURN))
          (DIRFLG←~DIRFLG)
          (NE←NEXTRA)
          (NHOLES←OUTWDS-1)
          (I←OUTP-1)
          (J←(IMIN MAXOUT-2 I+NE))                        (* leave room for NEWLINE, EOS)
```

```
        (while I lt J do (NTH BUF J):1←(NTH BUF I):1
                         (if (NTH BUF I):1=BLANK
                             then (if DIRFLG
                                      then NB←(NE-1)/NHOLES+1
                                    else NB←NE/NHOLES)
                                  NE←NE-NB
                                  NHOLES←NHOLES-1
                                  (while NB gt 0 do
                                        J←J-1
                                        (NTH BUF J):1←BLANK
                                        NB←NB-1))
                         I←I-1
                         J←J-1])


(CENTER
  [LAMBDA (BUF)                                          (* center a line by setting TIVAL)
    TIVAL←(IMAX (RMVAL+TIVAL+(-(WIDTH BUF)))/2 0])


(UNDERL
  [LAMBDA (BUF TBUF)                                     (* underline a line)
          (* * expand into TBUF, and then copy back into BUF * *)
    (PROG ((OBUF BUF)
           (OTBUF TBUF))
          (do (OTBUF:1←OBUF:1)
              (if OBUF:1=NEWLINE
                  then OTBUF:2←'EOS
                       (RETURN))
              (OBUF←OBUF::1)
              (if OTBUF:1~=BLANK and OTBUF:1~=TAB
                  and OTBUF:1~=BACKSPACE
                  then OTBUF:2←BACKSPACE
                       OTBUF:3←UNDERLINE
                       OTBUF←OTBUF::3
                else OTBUF←OTBUF::1)))
    (SCOPY TBUF BUF])


(PUTLIN
  [LAMBDA (BUF OUT)
    (for X in BUF while X~='EOS do (PUTCH X OUT])


(FORMATLENGTH
  [LAMBDA (STR)
    (for I from 0 while STR:1~='EOS do STR←STR::1
        finally (RETURN I])
```

```
(SKIPBL
  [LAMBDA (BUF)                                          (* skip blanks)
    (if BUF:1=BLANK or BUF:1=TAB
        then (SKIPBL BUF::1)
      else BUF])


(CTOI
  [LAMBDA (BUF)                                          (* convert string to integer)
    (bind I←0 do (SELECTQ BUF:1
                          ((0 1 2 3 4 5 6 7 8 9)
                            I←I*10+BUF:1 BUF←BUF::1)
                          (RETURN I])


(SKIP
  [LAMBDA (N)                                            (* output N blank lines)
    (RPTQ N (PUTC NEWLINE])


)

(* used to return values from GETVAL)

(RECORD COMTYPE (TYPE . N))

(* Utilities from earlier chapters)

(DEFINEQ

(SCOPY
  [LAMBDA (FROM TO)                                      (* copy characters from FROM to
TO)
    (do (TO:1←FROM:1)
        (if FROM:1='EOS
            then (RETURN))
        (TO←TO::1)
        (FROM←FROM::1])


(PUTDEC
  [LAMBDA (N W)                                          (* put decimal integer N in field width ge W)
    (SPACES W-(NCHARS N) STDOUT)
    (PRIN1 N STDOUT])
```

```
(GETWRD
  [LAMBDA (IN OUT)                                    (* get non-blank word from IN into
                                                       OUT and return new IN)
    (while IN:1=BLANK or IN:1=TAB do (IN←IN::1))
    (while IN:1~='EOS and IN:1~=BLANK and IN:1~=TAB
           and IN:1~=NEWLINE
        do (OUT:1←IN:1)
           (OUT←OUT::1)
           (IN←IN::1))
    OUT:1←'EOS
    IN])


)


(* interface to Interlisp)


(DEFINEQ


(PUTC
  [LAMBDA (CHAR)                                      (* output a single character)
    (PUTCH CHAR STDOUT])


(PUTCH
  [LAMBDA (CHAR OUT)                                  (* put char out on OUT)
    (PRIN1 (the CHARACTER CHAR) OUT])


(CHARBUFFER
  [LAMBDA (N)                                         (* initiate value for a character buffer)
    (to N collect 'EOS])


(GETLIN
  [LAMBDA (BUF FILE)                                  (* read a line of characters into BUF)
    (do (if BUF::1=NIL
            then (BUF::1←<NIL>)                       (* expand buffer if necessary))
        (if ~(ERSETQ BUF:1←(READC FILE))
            then (RETURN 'EOF)
          elseif BUF:1=NEWLINE
            then BUF:2←'EOS
                 (RETURN))
        (BUF←BUF::1])
)

(DECLTYPE CHARACTER
 (ONEOF (LITATOM SATISFIES (EQ (NCHARS X) 1))
        (MEMQ EOF EOS))
```

Bibliography

Key to abbreviations:

ACM	Association for Computing Machinery
BBN	Bolt Beranek and Newman Inc., Cambridge, MA
CACM	Communications of the ACM
ISI/RR	Technical report, University of Southern California Information Sciences Institute, Marina del Rey, CA
JACM	Journal of the ACM
IJCAI	International Joint Conference on Artificial Intelligence
MIT-AI-TR	Technical report, Artificial Intellegence Laboratory, Massachusetts Institute of Technology, Cambridge, MA
Xerox PARC	Xerox Palo Alto Research Center, Palo Alto, CA
POPL	ACM Symposium on Principles of Programming Languages
SIGPLAN	ACM SIGPLAN Notices, Special Interest Group on Programming Languages
SRI	SRI International, Menlo Park, CA
STAN-CS	Technical report, Computer Science Department, Stanford University, Stanford, CA

[Aho & Johnson 1976]

A. V. Aho and S. C. Johnson. "Code Generation for Expressions with Common Subexpressions". *Second POPL*, Jan. 1976.

[Aho & Ullman 1977]

A. V. Aho and Jeffrey D. Ullman. *Principles of Compiler Design.* Addison-Wesley, 1977.

[Allen & Cocke 1971]

F. E. Allen and J. Cocke. "A Catalogue of Optimizing Transformations". in *Design and Optimization of Compilers*, R. Rusten, ed., Prentice-Hall, 1971.

[Allen 1975]

F. E. Allen. "Interprocedural Analysis and the Information Derived by It." in *Programming Methodology: Lecture Notes in Computer Science, Volume 23*, Springer-Verlag, Heidelberg, Germany, 1975.

[Balzer 1972]

R. M. Balzer. *Automatic Programming.* ISI/RR-73-1, Sept 1972.

[Balzer 1975]

R. M. Balzer. *Imprecise Program Specification.* ISI/RR-75-36, Dec. 1975.

[Banning 1978]

J. P. Banning. *A Method for Determining the Side Effects of Procedure Calls.* STAN-CS-78-676, Nov. 1978.

[Banning 1979]

J. P. Banning. "An Efficient Way to Find the Side Effects of Procedure Calls and the Aliases of Variables." *Sixth POPL*, Jan. 1979.

[Barth 1977]

J. M. Barth. *A practical interprocedural data flow analysis algorithm and its applications.* Ph.D. dissertation, Computer Science Tech. Report 770520, U.C. Berkeley, May 1977.

[Barth 1978]

J. M. Barth. "A Practical Interprocedural Data Flow Analysis Algorithm". *CACM* 21:9, Sept. 1978.

[Bobrow 1975]

D. G. Bobrow. "Dimensions of Representation". *Representation and Understanding*, D. G. Bobrow and A. Collins (eds.), Academic Press, 1975.

[Bobrow & Winograd 1976]

D. G. Bobrow and T. Winograd. *An Overview of KRL, a Knowledge Representation Language.* Xerox PARC CSL-76-4, July 1976. Also in *Cognitive Science* 1:1, Jan. 1977.

[Bobrow & Deutsch 1976]

D. G. Bobrow and L.P.Deutsch. "Extending Interlisp for Modularization and Efficiency". *Proc. EUROSAM '79, Lecture Notes in Computer Science* 72, Springer-Verlag 1979.

[Brachman 1978]

R. J. Brachman. *A Structural Paradigm for Representing Knowledge.* BBN Report No. 3605, May 1978.

[Burton 1976]

Richard C. Burton, Richard R. *Semantic Grammar: An Engineering Technique for Constructing Natural Language Understanding Systems.* BBN Report No. 3453, Dec. 1976.

[Codd 1970]
E. F. Codd. "A Relational Model of Data for Large Shared Data Banks". *CACM* 13:6, June 1970.

[Cousot & Cousot 1979]
P. Cousot and R. Cousot. "Systematic Design of Program Analysis Frameworks". *Sixth POPL*, Jan. 1979.

[Davis & King 1975]
R. Davis and J. King. *An Overview of Production Systems*. STAN-CS-75-524, Oct. 1975.

[Davis 1978]
R. Davis. "Knowledge Acquisition in Rule-Based Systems - Knowledge About Representations as a Basis for System Construction and Maintenance". *Pattern-Directed Inference Systems*, D. A. Waterman and F. Hayes-Roth (eds.), Academic Press, 1978.

[Deutsch 1973]
L. P. Deutsch. *An Interactive Program Verifier*. Xerox PARC CSL-73-1, May 1973.

[Dolatta et al. 1978]
T. A. Dolotta, R. C. Haight, and J. R. Mashey. "The Programmer's Workbench". *Bell System Tech. Journal* 57:6, Jul.-Aug. 1978.

[Feldman 1979]
Stuart I. Feldman. "MAKE--a program for maintaining computer programs". *Software Practice & Experience* 9:4, April 1979.

[Fosdick & Osterweil 1976]
L. D. Fosdick and L. J. Osterweil. "Data flow analysis in software reliability". *ACM Computing Surveys*, 8:3, Sept. 1976.

[Goodenough & Shafer 1976]
J. B. Goodenough and L. H. Shafer. *A Study of High Level Language Features*. Softech, Inc., ECOM-75-0373-F. Available as NTIS AD-A021 206 and 207, Feb. 1976.

[Gordon et al. 1977]
M. Gordon, R. Milner, and C. Wadsworth. *Edinburgh LCF.* Internal Report CSR-11-77 (Part I), Dept. Computer Science, University of Edinburgh, Sept. 1977.

[Graham & Wegman 1976]
S. L. Graham and M. Wegman. "A fast and usually linear algorithm for global flow analysis". *JACM* 23:1, Jan. 1976.

[Hecht & Ullman 1973]
M. S. Hecht and J. D. Ullman. "Analysis of a Simple Algorithm for Global Data Flow Problems." *(First) POPL*, Oct. 1973.

[Hendrix 1977]
Gary G. Hendrix. *The Lifer Manual.* Technical Note 138, AI Center, SRI International, Feb. 1977.

[Hoare 1973]
C. A. R. Hoare. *Hints on Programming Language Design.* STAN-CS-73-403, Dec. 1973.

[Jones & Muchnick 1977]
Niel D. Jones and Steven S. Muchnick. "Even simple programs are hard to analyze". *JACM* 24:2, April 1977.

[Jones & Muchnick 1979]
Niel D. Jones and Steven S. Muchnick. "Flow Analysis and Optimization of LISP-like Structures". *Sixth POPL*, Jan. 1979.

[S.Kaplan 1979]
Samuel J. Kaplan. *Cooperative Responses from a Portable Natural Language Data Base Query System.* PhD Dissertation, University of Pennsylvania, 1979.

[Kaplan & Ullman 1978]
M. A. Kaplan and J.D.Ullman. "A General Scheme for the Automatic Inference of Variable Types". *Fifth POPL*, 1978.

[Kernighan & Plauger 1976]
B. W. Kernighan and P. J. Plauger. *Software Tools.* Addison-Wesley, 1976.

[Kling & Scacchi 1979]
Rob King and Walter Scacchi. "The DoD Common High Order Programming Language Effort (DoD-1): What Will the Impacts Be?" *SIGPLAN* 14:2, Feb. 1979.

[Lenat & Harris 1978]
D. B. Lenat and G. Harris. "Designing a Rule System that Searches for Scientific Discoveries". *Pattern-Directed Inference Systems*, D. A. Waterman and F. Hayes-Roth (eds.), Academic Press, 1978.

[Lientz et al. 1978]
B. P. Lientz, E. B. Swanson, and G. E. Tompkins. "Characteristics of Application Software Maintenance". *CACM* 21:6, June 1978.

[Liskov et al. 1977]
B. Liskov et al. "Abstraction Mechanisms in CLU". *CACM* 20:8, Aug. 1977.

[Low 1976]
J. R. Low. *Automatic Data Structure Selection: An Example and Overview.* University of Rochester, Computer Science Department Technical Report 14, Sept. 1976.

[McCarthy 1978]
J. McCarthy. *History of Programming Languages Conference, SIGPLAN* 13:8, Aug. 1978.

[Milman 1977]
Y. Milman. "An Approach to Optimal Design of Storage Parameters in Databases". *CACM* 20:15, May 1977.

[Mitchell 1970]
J. G. Mitchell. *The Design and Construction of Flexible and Efficient Interactive Programming Systems.* PhD thesis, Computer Science Department, Carnegie-Mellon University, Pittsburgh, June 1970.

[Mitchell et al. 1978]
J. G. Mitchell, W. Maybury, and R. Sweet. *Mesa Language Manual.* Xerox PARC CSL-78-1, Feb. 1978.

[Model 1979]

M. L. Model. *Monitoring System Behavior in a Complex Computational Environment.* STAN-CS-79-701, Jan. 1979.

[Moore 1975]

J S. Moore. *Introducing Iteration into the Pure Lisp Theorem Prover.* Xerox PARC CSL-74-3, March 1975.

[Moore 1976]

J S. Moore. *The Interlisp Virtual Machine Specification.* Xerox PARC CSL-76-5, September 1976.

[Moriconi 1977]

Mark S. Moriconi. *A System for Incrementally Designing and Verifying Programs.* ISI/RR-77-65, Nov. 1977.

[Moriconi 1978]

M. S. Moriconi. *A Designer/Verifier's Assistant.* SRI Tech. Report CSL-80, Oct. 1978.

[Morris 1973]

J. Morris. "Protection in programming languages". *CACM* 16:1, Jan. 1973.

[Parnas 1972a]

D. L. Parnas. "A Technique for Software Module Specification with Examples". *CACM* 15:5, May 1972.

[Parnas 1972b]

D. L. Parnas. "On the Criteria to be Used in Decomposing Systems into Modules". *CACM* 15:12, Dec. 1972.

[Perlis 1977]

Alan Perlis. Keynote speech. *Perspectives on Computer Science.* Academic Press, 1977.

[Ramamoorthy & Ho 1977]

C. V. Ramamoorthy and S. F. Ho. "Testing Large Software with Automated Software Evaluation Systems". *Current Trends in Programming Methodology, Volume II: Program Validation.* R. T. Yeh, (ed.), Prentice-Hall, 1977

[Rich & Shrobe 1976]
Charles Rich and Howard E. Shrobe. *Initial Report on a Lisp Programmer's Apprentice.* MIT-AI-TR-354, Dec. 1976.

[Rich & Shrobe 1978]
Charles Rich and Howard E. Shrobe. "Initial Report on a Lisp Programmer's Apprentice". *IEEE Software Engineering* SE-4:6, Nov. 1978.

[Rich, Shrobe & Waters 1979]
Charles Rich, Howard E. Shrobe, and Richard C. Waters. "Overview of the Programmer's Apprentice". *Sixth IJCAI*, Aug. 1979.

[Roberts & Goldstein 1977]
R. B. Roberts and I. P. Goldstein. *The FRL Primer.* MIT-AI 408, July 1977.

[Rosen 1979]
Barry K. Rosen. "Data Flow Analysis for Procedural Langauges". *JACM* 26:2, Apr. 1979.

[Ryder 1979]
B. G. Ryder. "Constructing the Call Graph of a Program". *IEEE Transactions on Software Engineering* SE-5:3, May 1979.

[Sandewall 1978]
E. Sandewall. "Programming in the Interactive Environment: The LISP Experience". *ACM Computing Surveys* 10:1, March 1978.

[Satterthwaite 1975]
E. H. Satterthwaite. *Source Language Debugging Tools.* STAN-CS-75-494, May 1975.

[Schaeffer 1973]
M. A. Schaeffer. *Mathematical Theory of Global Program Optimization.* Prentice-Hall, 1973.

[Shrobe 1979]
H. E. Shrobe. "Dependency Directed Reasoning in the Analysis of Programs Which Modify Complex Data Structures". *Sixth IJCAI*, Aug. 1979.

[Simon 1969]
Herbert A. Simon. *The Sciences of the Artificial.* MIT Press, 1969.

[Standish 1971]
T. A. Standish. "PPL - an Extensible Language that Failed". *SIGPLAN* 6:12, Dec. 1971.

[Steele 1978]
G. L. Steele. *RABBIT - a Compiler for SCHEME.* MIT-AI-TR-474, May 1978.

[Sussman & Steele 1975]
G. J. Sussman and G. L. Steele. *SCHEME: An Interpreter for Extended Lambda Calculus.* MIT-AI 349, Dec. 1975.

[System R 1976]
M. M. Astrahan, M. W. Blasgen, D. D. Chamberlin, et al. "System R: Relational Approach to Database Management". *ACM Transactions on Database Systems* 1:2, June 1976.

[Swinehart 1974]
D. C. Swinehart. *COPILOT: A Multiple Process Approach to Interactive Programming Systems.* STAN-CS-74-412, July 1974.

[Teitelman 1969]
W. Teitelman. "Toward a Programming Laboratory". *IJCAI 69,* May 1969.

[Teitelman 1972]
W. Teitelman. "Automated Programmering - The Programmer's Assistant", *Proc. AFIPS Fall Joint Computer Conference,* Dec. 1972.

[Teitelman 1973]
W. Teitelman. "CLISP - Conversational LISP". *Third IJCAI,* Aug. 1973.

[Teitelman et al. 1978]
W. Teitelman et al. *Interlisp Reference Manual.* Xerox PARC, Dec. 1978.

[Tenenbaum 1974]
Aaron M. Tenenbaum. *Type Determination in Very High Level Languages.* Courant Computer Science Report #3, Report NSO-3, Courant Institute, New York University, Oct. 1974.

[Ullman 1975]
J. D. Ullman. "A Survey of Data Flow Analysis Techniques." *Proceedings Second USA-Japan Computer Conference*, Tokyo, Aug. 1975.

[Vanlehn 1978]
Kurt A. Vanlehn. *Determining the Scope of Engish Quantifiers.* MIT-AI-TR-483, June, 1978.

[Wasserman 1975]
A. I. Wasserman. "Issues in Programming Language Design--An Overview". *SIGPLAN*, July 1975.

[Waters 1979]
R. C. Waters. "A Method for Automatically Analyzing Programs". *Sixth IJCAI*, Aug. 1979.

[Waterman & Hayes-Roth 1978]
D. A. Waterman and F. Hayes-Roth (eds). *Pattern-Directed Inference Systems.* Academic Press, 1978.

[Wegbreit 1975a]
B. Wegbreit. "Mechanical Program Analysis". *CACM* 18:9, Sept. 1975.

[Wegbreit 1975b]
B. Wegbreit. "Property Extraction in Well-Founded Property Sets". *IEEE Transactions on Software Engineering*, SE-1:3, Sept. 1975.

[Wilczynski 1975]
D. Wilczynski. *A Process Elaboration Formalism for Writing and Analyzing Programs.* ISI/RR-75-35, Oct. 1975.

[Winograd 1975]
T. Winograd. "Breaking the Complexity Barrier Again". *SIGPLAN* 10:1, Jan. 1975.

[Wulf 1974]
W. A. Wulf. *Alphard: Towards a Language to Support Structured Programs.* Technical report, Computer Science Department, Carnegie-Mellon University, April 1974.

[Yonke 1975]

Martin D. Yonke. *A Knowledgeable, Language-Independent System for Program Construction and Modification.* ISI/RR-75-42, Oct. 1975.