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.

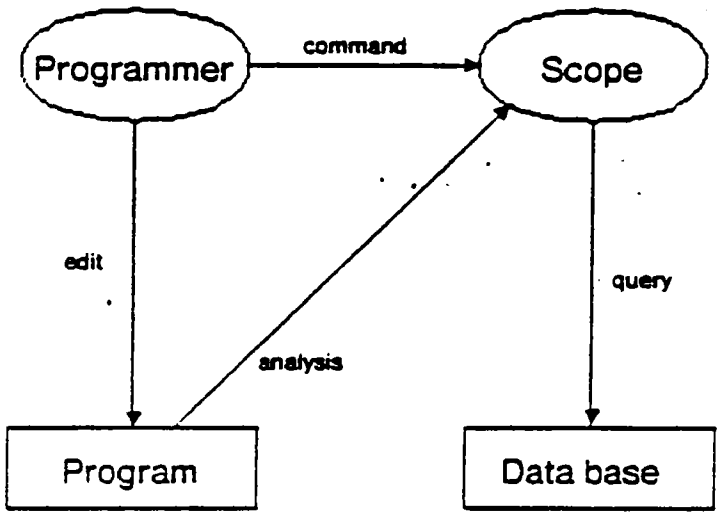

Figure 1-1—Overview of SCOPE

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.

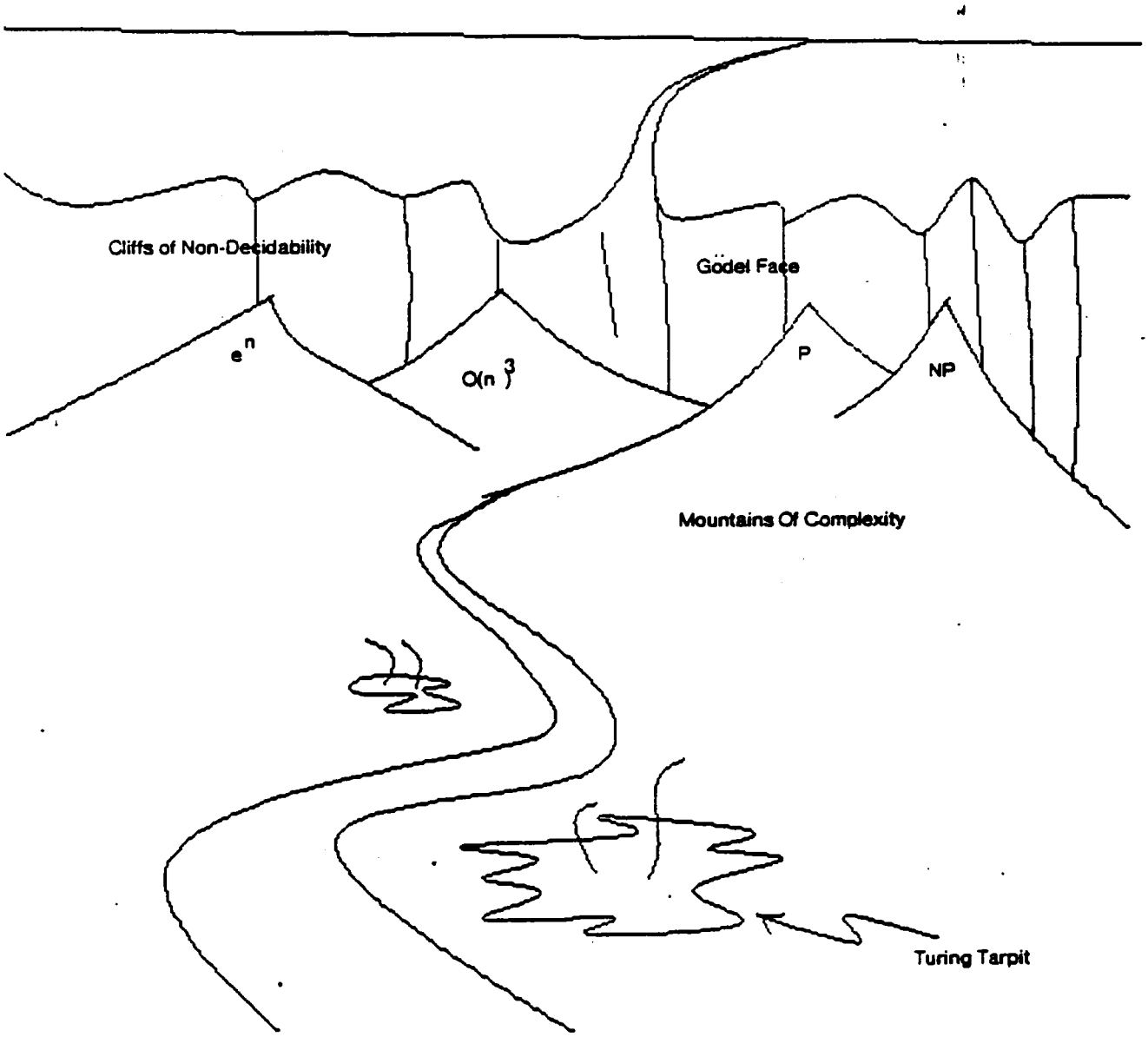

Figure 1-2—Perlis' Perils

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.