

# 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 fulfillment 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

1.	Introduction	1
1.1	Motivation	1
1.2	Overview	2
1.3	What SCOPE can do	4
1.4	Design philosophy	7
1.5	The setting	9
1.6	Some assumptions and limitations	11
1.7	Related work	14
1.8	Conclusions	15
2.	Uses of SCOPE	17
2.1	Aid to program understanding and modification	17
2.2	Checking for errors	25
2.3	Code improvements	28
3.	Characteristics of SCOPE's Representation System	31
3.1	Units and relations	32
3.2	Exhaustiveness	33
3.3	Operational correspondence	33
3.4	Inference	35
3.5	Access	37
3.6	Self awareness	37
3.7	Conclusions	38
4.	What SCOPE Knows About Programs	39
4.1	Cross reference	39
4.2	Flow information	40
4.3	Type information	44
4.4	Filing properties	46
4.5	Conclusions	47
5.	Program Analysis Techniques	48
5.1	Cross reference analysis	48
5.2	Flow analysis	49
5.3	Type inference	51
5.4	Conclusions	53
6.	Implementation Notes	54
6.1	Parser	54
6.2	Interpreter	55
6.3	Answering questions	56
6.4	Data base	59
6.5	Conclusions	61
7.	Future Directions	62
7.1	Improving the current implementation	62
7.2	Added capabilities	64
7.3	Beyond SCOPE	65
Appendices		
I	Relations Used in SCOPE	67
II	The SCOPE Command Language	75
III	The SCOPE Intermediate Query Language	82
IV	Templates for Computing Cross Reference	84
V	A Sample Program	86
Bibliography		98

## List of Figures

1-1	Overview of SCOPE	3
1-2	Perlis' Perils	12
2-1	Tree structure of function calls	19
3-1	Mapping between world and knowledge states	31
3-2	Mapping between program and SCOPE's data base	31
6-1	Implementation of SCOPE	54
6-2	What SCOPE knows about a relation	56



## 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**:  $\text{Call}[\text{FN}_1, \text{FN}_2]$  holds if the definition whose name is  $\text{FN}_1$  contains a call to a procedure named  $\text{FN}_2$ . 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.

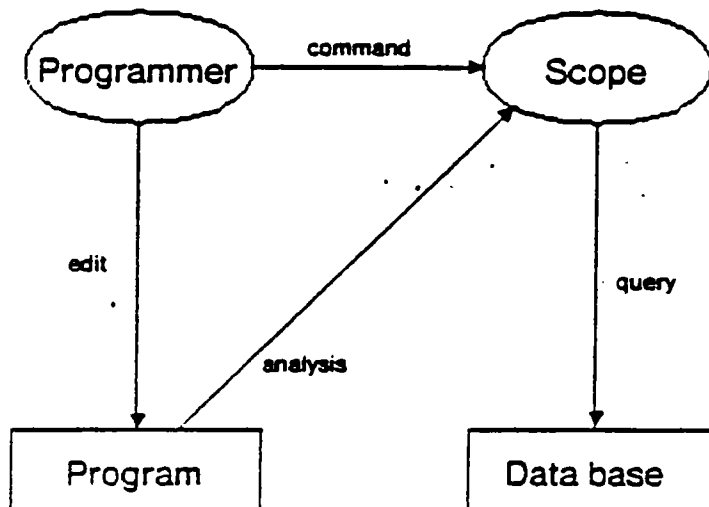


Figure 1-1—Overview of SCOPE

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 ← ARGTYPE 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 C, 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.

