A Programming Environment for Fortran[1]

Robert T. Hood
Ken Kennedy

Rice MASC TR83-22

Department of Mathematical Sciences
Rice University
P.O. Box 1892
Houston, TX 77251

[1]Support for this work was provided by the National Science Foundation under grants MCS-8104006 and MCS-8121884.

A Programming Environment for Fortran

Robert T. Hood
Ken Kennedy

Abstract

Although it has many flaws by comparison with more modern programming languages, Fortran remains the standard language for numerical programming—partly because there is a large body of important software coded in it, partly because good optimizing compilers can be written for it, and partly because it is reasonably easy to transport Fortran programs from one machine to another. The numerical analysis community has developed reliable packages of subroutines to solve various kinds of problems (e.g., LINPACK, EISPACK and MINPACK) and these packages are widely distributed.

Yet, in spite of its popularity, most of the sophisticated new tools for program development have not been extended to support it. As a part of the $\mathbb{R}^n$ project at Rice University, we are building a retargetable interactive programming environment for Fortran. At the heart of the environment is project management software that maintains a data base on all the program modules. The data base contains not only the source for each module in the system but also related information such as the composition of programs, cross-module procedure call information, interprocedural data flow information, and specifications for calls to existing procedures. The data base is used by all the other tools in the environment: an intelligent editor that knows enough Fortran to assist the user in preparing programs, an interactive source-level debugging system, and an optimizing compiler with extensive interprocedural analysis.

Of particular interest, because of our previous work on compiler optimization, is the assistance such a system can give to interprocedural data flow analysis and optimization. Because all the modules of a program are saved in the data base, the information needed to analyze data flow effects and improve code across procedure boundaries is conveniently available to the optimizing compiler. Thus, the environment will permit us to build compilation systems which optimize the program as a whole rather than module by module—something which has not been possible with conventional compilation techniques.

A Programming Environment for Fortran[1]

Robert T. Hood
Ken Kennedy

Department of Mathematical Sciences
Rice University
Houston, Texas 77251

1. Introduction

It is well known that progress in software has not kept pace with the dramatic advances in hardware technology. In a period when the power of machines has advanced by a factor of more than 100, the average productivity of the programmer has increased by less than a factor of 2. We must find ways to increase programmer effectiveness if we are to have sufficient manpower to realize the promise of the computer age.

To attack this problem, we must first understand why programming is so difficult, especially in large programming projects. The observations that follow are derived from our experiences in managing the development of a large language-translation system produced by a team of approximately five students and faculty members over a period of three years [Alle82a].

First, there seem to be natural limits to the complexity that a single programmer can deal with. The implication is that if programmers can work at a higher level of abstraction they will be more productive. Certainly programmer effectiveness has improved through the use of high level languages. We could see even more progress if very high level languages were used. The problem with this approach is that current implementations of very high level languages are too inefficient to come into widespread practical use. Dramatic advances in techniques for compiler optimization would help overcome this problem.

In large systems a second problem arises because important information needed by the programmer is not conveniently available. How many times, while programming, have you found yourself knowing exactly what you want to do but not quite sure of the syntax required or how a key variable is declared? The ensuing search for information can break an important train of thought and significantly interfere with your productivity. Programmers new to a project face

---

[1]Support for this work was provided by the National Science Foundation under grants MCS-8104006 and MCS-8121884.

this problem compounded several times. They usually suffer a long learning period before making contributions. Automatic aids to provide programmers with information such as variable declarations and subprogram specifications would ease this difficulty.

There are also a number of problems that arise from the sheer size of a system. In a program of tens or hundreds of thousand lines of code, implemented by several programmers, no one person is completely familiar with the whole program. Each must rely on others for information, which means that a good deal of effort must be spent on communication. Interfaces must be carefully designed and documented if the final system is to work well and be available on time.

Testing a large system also poses problems. Although we teach the doctrine of independent testing in our programming classes, it is sometimes difficult to follow in practice. This happens because a module that we might wish to test operates on a complex data structure that is very difficult to build. In these cases, running the program in which the module is to be incorporated can be the simplest way to build the data structure and hence to test the module. But what if the module to be debugged is executed only after a long execution time in other parts of the program? The programmer must then choose between two unsatisfactory options: either run the whole program using the slow, high level debugger or insert debugging statements in line and run a compiled version. Some system that permits the testing of modules in the context of a compiled program must be found.

The management of a project requiring more than one programmer poses a number of problems. One of the most important is making sure that programmers don't "shoot each other in the foot". It is common for one programmer to make a small change to a working system that "breaks" the whole program. The rest of the team arrive the next morning to discover that things which worked the day before no longer do. Some method to manage system modification is needed so that changed modules can be debugged in a test environment before being integrated into the working system.

The same method should also solve a related problem, that of making sure that the whole system is carefully remade after a change invalidates a portion of the program. When a data structure is changed, every module that uses that data structure (but *not* every module that includes its declarations) needs to be recompiled. We shall refer to this as the problem of maintaining *consistency* in a system.

Any system for managing a large project should also provide aids for both testing and documentation. During the maintenance phase, small changes intended to fix a particular bug often inadvertently create new unforseen bugs. The system should provide semiautomatic assistance for detecting such bugs. The production of clear consistent documentation can be encouraged by the programming system—it should make the entering of such documentation convenient and its dissemination automatic. A hierarchical approach might be very helpful here.

Habermann has identified three approaches to the solution of these problems[Habe82a]. In the *disciplinary* approach, programmers would exercise enormous self-control in following manual procedures designed to prevent the insertion of bad code. While this method, often referred to as *structured programming* has been quite helpful over the past ten years, it can suffer from the programmer's natural humanity. Under pressure, a programmer will tend to take shortcuts, some of which bypass the formal procedures designed to inhibit errors. Some mechanism to automatically assist the programmer in maintaining this discipline is needed.

The *toolkit* approach would provide a collection of simple but powerful tools that can be interconnected in various ways to suit the programmer's needs. Unix[2] and Programmers Work Bench (PWB) are examples of this strategy. The main difficulty with this approach is that the tools are usually not fashioned to the task at hand so each programmer will adapt the tools to his or her own needs. Often the programmers on a single project will each have a different collection of favorite tools—this hardly enhances the level of communication on the project. The toolkit method does not take advantage of any knowledge of the language being used or the problem being addressed.

The *environment* approach, is characterized by a collection of tools explicitly tailored to the task at hand. Typically, these tools include a language-oriented editor, facilities for management of the program source, debugging tools tailored to the programming language being used, and facilities for making consistent executable modules. Examples of environments abound in the literature. The *Cornell Program Synthesizer* is an environment to support development of small PL/I programs by introductory programming students [Teit81a]. *Mentor*, one of the earliest environment projects, supports programming in Pascal [Donz75a]. The *Interlisp* system contains many Lisp-specific tools to assist the programmer [Teit77a]. Systems of macros, tailored to editing a particular language can make

[2]Unix is a trademark of Western Electric.

the extendible editor *Emacs* into a programming environment for that language [Gosl83a]. The *Gandalf* project at CMU is building an environment-generating system that can be used for any of a number of languages [Habe82b]. Finally, a group at GTE is working on a programming environment for the *Chill* language.

For the most part, the modern numerical programmer is unable to benefit from the current work on programming environments. The reason for this is twofold. First, Fortran is the common language of numerical discourse—an enormous inventory of software and many useful packages (such as LINPACK, EISPACK, and the IMSL library) have been written in it. Second, Fortran is not popular among non-numeric computer scientists. Hence most of the major environment projects have ignored it. (An exception is the Toolpack project [Oste81a].) Nevertheless, the numerical programmer has begun to use a few modern tools such as screen editors and interactive debuggers. These have provided them with an enormous increase in productivity and a similarly large appetite for more advanced tools.

In this paper we discuss a project underway at Rice University to provide a system of tools for developing, testing, and maintaining Fortran programs that makes effective use of the programmer's time without sacrificing run-time efficiency. This system represents a natural evolution of our previous work on automatic program analysis.

2. Overview of the Environment

The Fortran Programming Environment is a major component of the $\mathbb{R}^n$ project at Rice University, which is building a network of high performance workstations designed to provide the modern scientist, engineer, or numerical analyst with a computational resource tailored to his needs. The specifications for such a workstation are summarized in Table 1. Thus the environment is designed from

- 1-2 mip CPU
- high speed floating point
- large virtual address space
- 1-2 megabytes real memory
- 800 × 1000 pixel bit-mapped display
- graphic input device (mouse or tablet)
- network interface
  - file servers
  - print servers
  - compute servers
  - gateways to other networks
- Unix software base
- Reasonable cost ($5K-15K)

Table 1. Workstation Specifications.

the outset to run on such a workstation and to take advantage of the high resolution graphics, the graphical input device, and the local computational power. The environment may also run on simpler graphics devices, such as dumb terminals, but we have not compromised the design to accommodate such devices. We envision that the network will have other resources connected to it such as a computation server, a long-haul network gateway, and a file server.

The programming environment will be partitioned between the file server and the workstation as depicted in Figure 1. At the heart of the environment is the project data base which resides on the file server. It records all information about the programs and modules in a project, including source, specifications, test data, documentation, interprocedural information, and much more. Residing on the workstation itself is a command processing program that will provide a uniform interface for all the user-invoked operations. This program integrates the major environment tools:

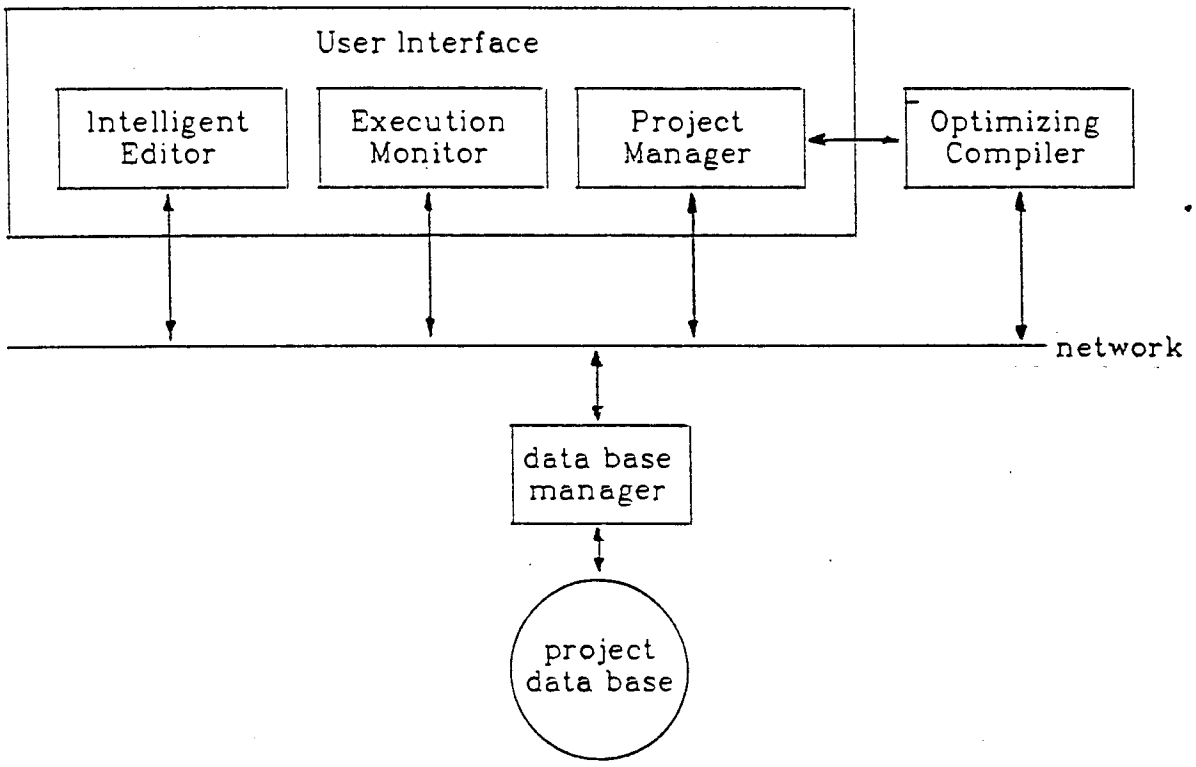

Figure 1. The $\mathbb{R}^n$ Programming Environment for Fortran.

(1) An *intelligent Fortran editor* which not only helps the programmer build syntactically correct programs, but also warns of possible run-time anomalies that can be detected at compile time. In addition, the editor will be able to query the data base, thus providing a mechanism for the user to call up documentation, declarations, or specifications while editing or browsing program text.

(2) An *execution monitor* which can step through parts of the program allowing the programmer to interrupt and perform debugging operations. With the help of the project manager, the monitor would be able to handle a hybrid program consisting of some compiled and some interpreted modules. Thus control could be made to pass quickly through most of the program to the module under development which could then be interpreted.

(3) A *project manager* which provides the sole data base access mechanism for all other environment utilities. It stores and retrieves the source files in the data base, answers questions about the programs and modules, insures that project rules are obeyed, and makes consistent versions of the programs for execution.

(4) An *optimizing compiler* that converts the partially compiled version of the program maintained by the editor to an optimized form suitable for integration with the rest of the system. With the help of the project manager, it uses the system data base to do a thorough job of interprocedural analysis and optimization. It should be noted that the compiler will not be directly invoked by the user. Instead, it will be invoked by the project manager when recompilation of a module is necessary.

These tools work together to assist the programmer in preparing, documenting, and testing the program. They also cooperate to make the final programs as efficient as possible. We will discuss each of them in more detail, but first we describe the underlying data base.

3. The Data Base

The project data base organizes all information known to the programming environment. Its basic framework is depicted in Figure 2. In this scheme, the largest entity is the *project*, which may be thought of as a collection of programs that are being worked on by a common pool of programmers and that may make use of a common group of subprogram modules. A project might contain one program that is the central focus of the work, along with a collection of test versions, or it might include several central programs.

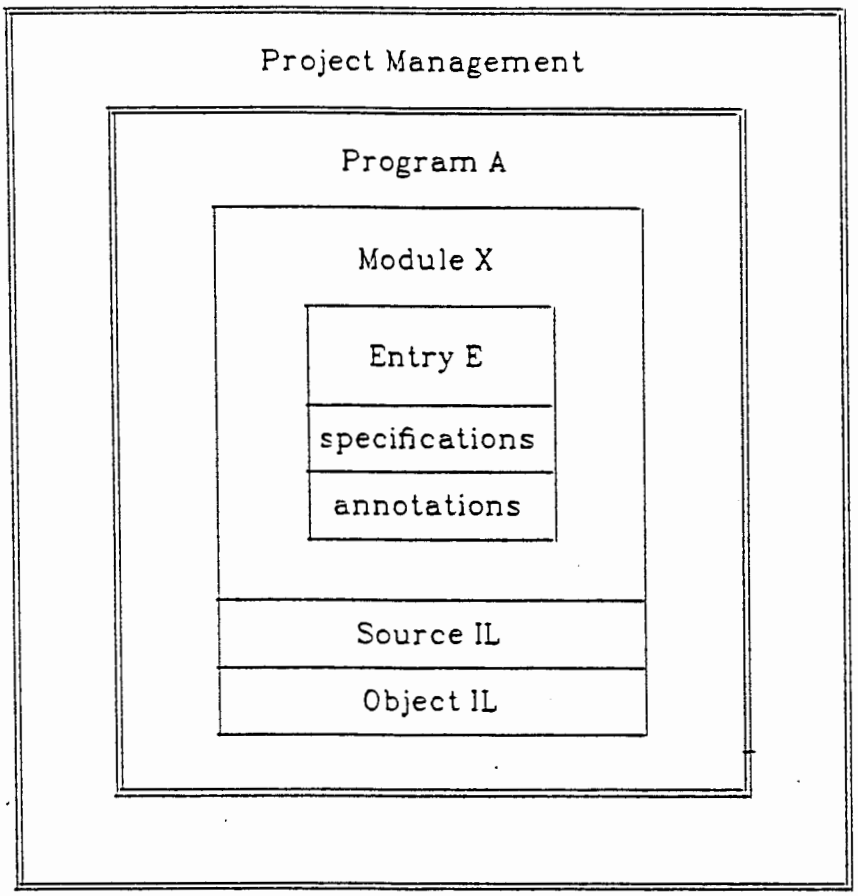

Figure 2. A conceptual framework for project management.

A *program* is simply a collection of modules that, when integrated into a whole, may be executed. Several programs in a project may share the same modules, so the system must support some mechanism for sharing. A program may have several named *versions*, each using a different set of modules. A program may be viewed as nothing more than a recipe indicating how to *compose* the modules (with versions specified) that it incorporates. Indeed, this is how programs are implemented in our preliminary system.

A *module* is a collection of entry points that is always edited and compiled as a whole. Modules may also have named *versions*. For example, there will usually be both an "official" and a "test" version of any module that is being modified. Presumably, the programmer will be working on the test version.

Associated with modules are various kinds of information, including source and compiled intermediate language. Each module also contains some number of *entry points*, which are the externally known names by which the module is accessed. These will include all the callable entry point names.

Associated with each entry point will be two kinds of information. *Specifications* are properties of the entry point that are entered by the programmer. In the first system, the specifications will consist only of the number and types of parameters. Later this may include other information about the intended behavior of the entry. *Annotations* are facts about the behavior of the entry that are gleaned by any of the tools in the environment. For example, the editor might add an annotation that indicates which other entries might be called as a result of a call to an entry point.

To conveniently provide the facilities of standard groups of modules, such as LINPACK or IMSL, the system will need to understand the concept of a *library*, which is an external project in which some modules have been declared to be "public." Single modules may be incorporated into a project from any of a number of specified libraries for the project.

Finally, any system that supports versions of its basic components must also support defaults. The environment we are developing would always have a *standard* version of every object for which versions are supported.

4. User Interface Program

The *user interface program* will serve as the programmer's home environment from which all activities are invoked. It will permit the programmer to navigate through projects and programs; it will acquire modules and other information from the data base; and it will make extensive use of a sophisticated graphics interface featuring multiple windows, highlighting, and the use of a pointing device such as a mouse.

The user interface program is really an operating system that provides utility services, such as graphics and query processing, to a variety of command processors, including a family of similar editors. These editors are designed to make the various functions of the environment look similar to the user. Hence, users can browse through various displays and, with the appropriate permissions, modify them in an editor-like setting. Such diverse operations as correcting a bug in a program and releasing a module to "the public" are both editing operations. The first is a traditional function for an editor. The second can be seen as editing when viewed from the perspective of modifying a list of properties associated with a module.

The existence of high resolution graphics and a mouse-like device allow for a menu-oriented milieu that is very easy to use. The display always has a list of legal operations. The programmer selects one of the actions by positioning the pointing device and then "clicking" one of its buttons. In the process of performing various actions, the menu of legal operations may, of course, change. In addition, when an operation requiring an object is invoked, a window of the possible objects will pop up. For example, if the user selects the operation "run", the system will respond with a list of programs.

In addition to using the mouse to select menu items, it can be used to scroll around a display or to select an element of the display for further examination. The latter action allows an object to be specified in advance of an operation. For example, the user can run a program by first selecting the program name, if it is on the screen, and then selecting the action "run". An example of an environment session follows.

When the user first enters the programming environment he will be at the root of the data base. The display will have a list of projects as shown in Figure 3. Using the mouse, the programmer can select a project, say the "Fortran Programming Environment" and be shown the list of programs that it comprises. At that point he could select one of the programs, say the "user interface", and do any of a number of things with it. For example, he could debug it by clicking the mouse while positioned on "run". At that point a window would pop up with a list

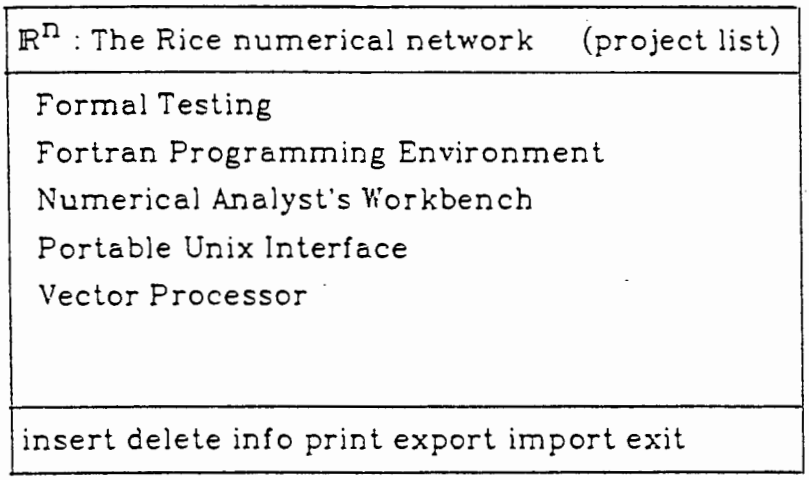

Figure 3. The root of the data base

of possible data sets (Figure 4). Suppose that he did not know the contents of one of them; he could, after selecting it with the mouse, get a brief description by clicking "info", or see it displayed in another window by clicking "show". He could also enter a new data set by clicking "insert".

5. Project Management

It is the role of the *project manager* to control access to all programs and modules within the project and maintain a data base of semantic information about them that can be used by other tools in the project. In addition, it must maintain the consistency of programs within the project, and provide information about the project or any of the programs and modules within it. For example, the manager must keep track of which programmers are working on which versions of a given module. Also, the manager will provide tools by which new programs can be constructed from modules in the data base. Finally, the project manager will provide the interface through which all queries about the project must pass.

To understand the role of the project manager, it is helpful to consider a selection of the functions we envision it performing. There are essentially three main functions performed by project management.

(1) *Query Answering*—in this category, we include any operation that provides information about the project in a non-destructive fashion. In other words any operation that does not cause a change in the current project state.

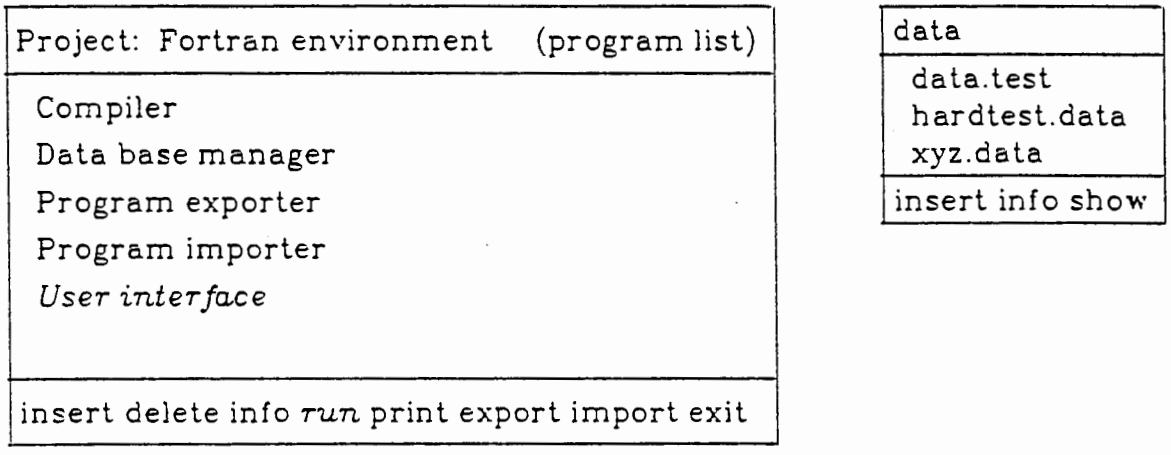

Figure 4. Choosing a data set

Examples are requests to browse source modules, questions about specifications or annotations for a given entry point, and questions about the structure of a given program, such as a request to display the call graph.

(2) *Module Creation and Modification*—in this category are all operations on modules in the project that lead to new or changed modules being stored in the data base. Examples are requests to edit a given module or to create a new one.

(3) *Program Creation*—in this category we find the function of program composition. In the system, programs are created by specifying a collection of modules to be incorporated in the composition. The project manager then adds enough modules to make a complete program or until it must report that the program is incomplete.

An issue related to program composition is the *current context program*. In the process of working on a project, we envision that the programmer will establish a program as the one in which he or she is currently working. The current context program establishes the default for many operations of the project manager. For example, when a query asks for information about a given entry point name, the project manager will assume that the query refers to the version of that entry point in the current context program. Similarly, in performing a composition, the program is completed by adding modules from the current context program.

The project manager will also be responsible for maintaining the project privilege rules. We use a very simple mechanism for deciding the authority of project programmers to perform certain functions. Each module and program has a *creator*, a *status* (public or private) and a *reference count*. A private program or module belongs to its creator and may be modified or released by that programmer. A program may be made public by its creator. By doing so, the creator relinquishes his or her authority over the program and every module contained in it. The creator may not modify a public module or program. He or she must create a new version of the module and build a whole new program composition in order to make such a change. Presumably, this composition will be private.

This mechanism insures the stability of public programs. Only the chief project programmer may release a public program; it then reverts to private status. Modules released by the chief project programmer revert to private status only when all references to it are by programs owned by the creator of the module.

6. Intelligent Editor

As we have seen, the user interface program provides a powerful browsing and editing environment. The *intelligent Fortran editor* is a subset of that interface that combines a knowledge of Fortran together with access to the data base in order to simplify the programming process.

The editor will assist the programmer in entering Fortran by providing commands that generate templates for the major language constructs. For example, to insert a do-loop, the programmer need only invoke the do-loop command and the cursor will be replaced by a do-loop template with *place markers* in the positions where further text should be entered.

```
do <iterator> {
  <body>
}
```

The syntax displayed above is taken from Ratfor [Kern78a] which illustrates another strength of the environment paradigm—the ability to tailor the display format to the tastes of the user.

Not only does the editor help a programmer enter syntactically correct programs, it also obviates the need for a parser by directly constructing the abstract syntax tree for the program. All components of the environment can then use the abstract syntax tree as the standard program representation. The display is constructed by unparsing the abstract syntax tree. The existence of a map from screen position into the tree facilitates the use of a pointing device to move the cursor around the program.

The high-resolution display on the workstation permits a particularly convenient view of the program to be presented. Typically, the display will have two windows. In addition, there will be a menu showing the legal operations at the current cursor position (Figure 5). The main window will display the current region on the screen, with region hiding as appropriate. A parallel window will always display the current declarations for each variable used in the region of program on the screen. For example, in Figure 5, the cursor, whose position is indicated by Italics, is at a statement location, so any statement may be inserted. An option may be selected using the mouse or by explicitly entering the command.

In our system the editor will also be able to detect and report many subtle semantic errors such as uninitialized variables. It will make use of information stored in the project data base to help construct subprograms that are consistent with the program being developed. For example, when a programmer wishes to insert a call to an external subroutine, the editor will query the data

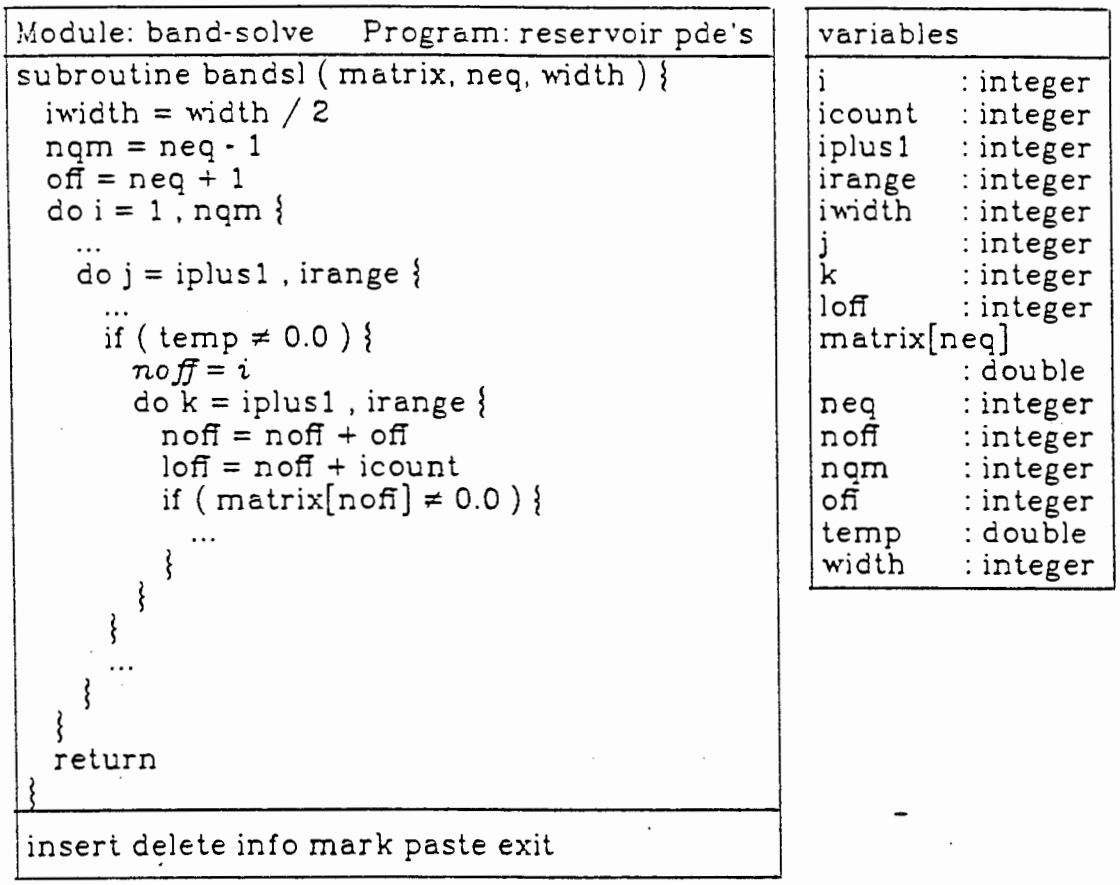

Figure 5. A typical editor display.

base to provide a template for the parameters that are required.

CALL bandsl(<matrix[neq] : double>, <neq : integer>, <width : integer>)

In this statement, the programmer inserted "CALL bandsl" and the system provided the parameter template.

There are several documentation functions that the editor will perform, including prompting the programmer for certain kinds of specifications, and maintaining a modification history. The editor will also compute and record summary data flow information for each module that it creates or modifies; such information can be used in both optimization and error detection.

In advanced versions of the editor, we will experiment with incremental data flow analysis. New results by Reps [Reps82a], Wegman [Wegm82a], and Zadeck [Zade83a], lead us to believe that use-definition chains (pointers from statements that use variables to the statements that might create the value used) can be

efficiently created by the editor in an incremental fashion. If this is true it will be possible to provide some powerful diagnostic features.

For example, it will be possible to have a function that scrolls back from a usage point to successive points of definition for the value used (see Figure 6). This facility would be extremely useful in debugging because most errors are detected when a bad value causes some fault to occur. The point of fault is easily located. However, the real error probably occurred where the bad value was *created*. Use definition chains can help us quickly find all possible creation points.

7. Execution Monitor

The *execution monitor* will enable the programmer to step through parts of a given program allowing him to interrupt and observe the progress of his program during its execution. Like the editor, it will also make effective use of the high resolution graphics. As we envision it, the programmer will be able to monitor execution using a display similar to the one depicted in Figure 7. While highlighting the statement being executed in one window, the debugger will simultaneously display the changed values of variables in a second and the program output in a third window.

An important design goal is to support *hybrid execution*, in which compiled and interpreted modules are intermixed. This will permit interpretive testing of a

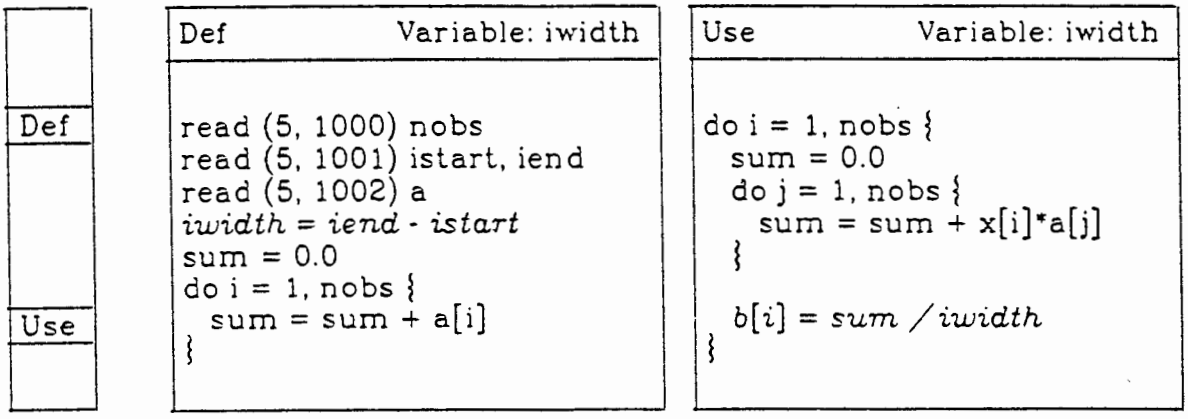

Figure 6. Scrolling back to definition points.

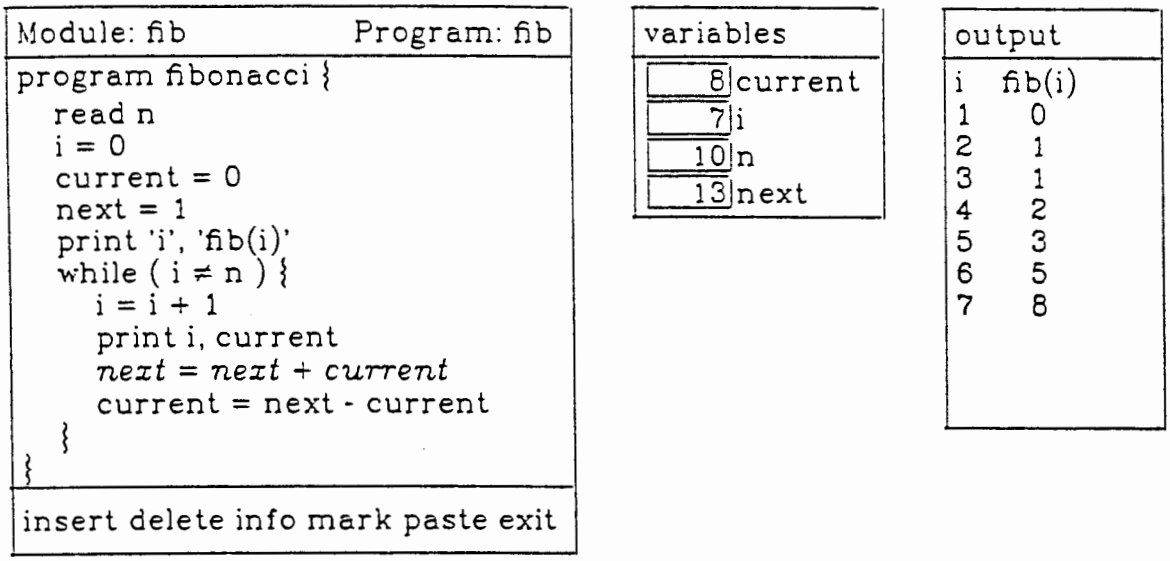

Figure 7. Execution monitoring.

module that may not be executed until the whole program has run for many minutes. To support this feature, it is absolutely critical that compiled and interpreted programs maintain a consistent layout of data in the program. This strategy will also make it more likely that the compiled and interpreted versions of the same module will behave identically.

Another important debugging feature we intend to support is reversible execution. There is no special difficulty to this. Since the abstract tree is doubly linked, we can easily move backward in it. A problem arises at three main points of ambiguity: assignments, gotos, and calls to compiled code. Traditionally, these are handled by saving on some file the value of the changed variable or the location from which control came. The main problem is presented by calls to compiled code. At these, the interpreter must save the values of every variable that might be changed before control returns. In the absence of better information, this means every variable in COMMON and every parameter.

In the $\mathbb{R}^n$ environment, the burden will be much smaller because the interprocedural analysis will provide the interpreter with a much more precise estimate of what might be changed by a call. Thus many fewer variable values will need to be saved.

Debugging is an extremely important programming activity that has received too little attention from Fortran implementors. Here the graphic capabilities of

the personal workstation will be especially valuable. One aspect of numerical debugging is common to non-numeric debugging: the elimination of semantic errors in the program, errors that cause the program to behave in an incorrect manner. But another type of debugging is also common in numerical programs—elimination of the errors of precision and accuracy that make the answers incorrect or the algorithm fail to converge rapidly enough. It is here that debugging truly takes on the flavor of experimentation and the ability to interactively follow execution while monitoring the output may permit an enormous saving of research time.

8. Optimizing Compiler

The *optimizing compiler*, which is really an optimizing code generator, will convert the partially compiled version of a module maintained by the editor to an optimized form suitable for integration into the program of which it is a part. The main advance in this tool over previous optimizing compilers for Fortran is its use of interprocedural analysis and optimization.

Compiler optimization researchers have long believed that the interprocedural effects are the last remaining major source of inefficiency in languages with optimizing compilers. Why then are there so few compilers with any interprocedural analysis and optimization? The answer is that the compiler would need access to all the code in a program in order to do a good job. It is unreasonable to expect to compile whole programs at once—the cost in computation time would be too great. It would be almost as impractical to perform data flow analysis on the whole program at each module compilation.

The solution is to save the interprocedural information needed for optimization between compilations in the project data base. This requires that the interprocedural information be updated each time a module is edited.

We intend to use the environment to attack two problems. First, we will investigate the use of interprocedural information to do *linkage tailoring*—the construction of efficient subroutine linkages tailored to the actual caller and callee. An example of linkage tailoring is inline substitution, but there are many less dramatic forms.

A second area is to compute the patterns of data usage and definitions as a result of procedure invocations. An example is the computation of *mod*(s), the set of variables that might be changed as a result of the procedure invocation at call site s. There are two components to this information.

(1) First there are the immediate effects of the procedure being invoked. These can be recorded in the data base by the editor. On putting a module away, the editor need only store the list of variables that are changed in some statement in the program.

(2) To this list must be added the secondary effects due to calls to other routines from within the called routine. These must be handled by solving a data flow problem on the call graph [Alle74a,Alle74b,Bann79a,Bart78a,Myer81a,Rose79a,Spil72a,Weih80a]. A recent dissertation by Cooper [Coop83a] describes fast algorithms to solve this problem in an incremental fashion. The basic idea is that whenever the editor puts away a module that is incorporated in a program, a demon is invoked to update the interprocedural information. This demon makes use of comparisons of old information with new information to keep from doing redundant work.

As a result of the actions of the demon described above, several modules may need to be recompiled in the light of new interprocedural information.

By using an integrated approach to interprocedural analysis involving the editor, the project manager, and the complier, we can use the $\mathbb{R}^n$ environment to mount a concerted attack on interprocedural optimization and analysis.

9. Conclusions

The four components of the $\mathbb{R}^n$ programming environment fit together nicely to provide a rich background for Fortran programming. Most of the ideas behind this system are not new, but until now we have not had the equipment and resources to explore them. With the advent of the personal numerical workstation, we will acquire the ability to make enormous advances in the quality of tools to aid the individual scientific or numerical researcher.

10. Implementation Status

We are currently working on a preliminary version of the environment that will include a simple structure editor and the basic project management software (including querying for call parameters). The structure of the preliminary implementation is depicted in Figure 8.

Within the project, there are three major directories:

(1) the *program directory*, which maps program names to the locations of descriptors for those programs.

(2) the *module directory* which maps module names and entry point names to the locations of descriptors for those modules and also indicates the default for each module.

(3) the *entry point directory* which maps entry point names to the modules that provide them and also contains the default mapping for entry points.

Thus, if a query asks about a particular program, the manager will use the program directory to find the information about that program. If a query asks about a particular entry point within a particular program, the manager will find the program descriptor to find out which version of the entry point in question is incorporated into the program and then use that information to find a module descriptor via the module directory.

On a standard Unix system, the project will occupy an entire subtree of the tree-structured directory. At the root of that subtree will be the privileges file, the entry point directory, and subdirectories *programs* and *modules*. Within the *programs* subdirectory, there will be a single subdirectory for each program. Within each program subdirectory there will be a subdirectory for each version. Each of these subdirectories will contain

(1) a program composition map *comp*

(2) a location map *location* which gives the module and version for each entry point in the program,

(3) an automatically-generated file which can serve as input to the Unix utility *make* [Feld77a], which will generate the sequence of operations needed to make a consistent object module for the program, and

(4) any other information that needs to be associated with the program.

In the *modules* subdirectory, one will find a separate subdirectory for each named module. Each of these will contain a separate subdirectory for each version of the module. The version subdirectory will contain several files:

(1) the source file,

(2) the compiled code,

(3) a list of entry point names,

(4) a list of programs into which this module is incorporated, and

(5) a subdirectory *entries*.

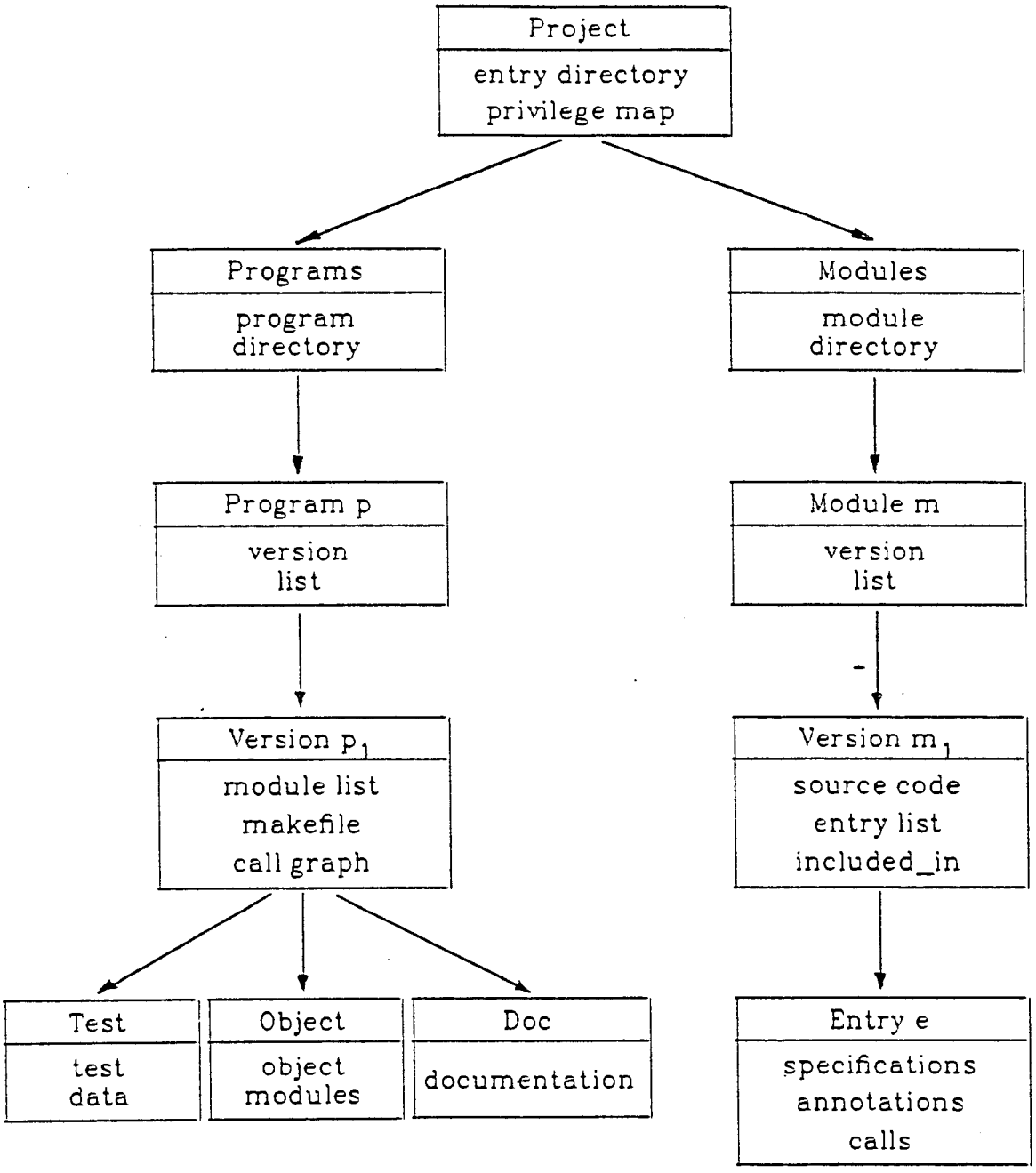

Figure 8. The Preliminary Implementation.

The *entries* subdirectory will contain a descriptor for each entry point in the module. Each entry point descriptor will contain

(1) specifications for the entry (e.g., parameter documentation),

(2) annotations (e.g., summary data flow information),

(3) a list of the entries called from within, and

(4) a list of entries and modules from which this entry is called.

In a program that is being optimized (a later implementation), we will need to recognize that the compiled code may be dependent on the context program in which the module is being compiled. We plan to place such program-dependent code modules in subdirectories of the program in which they are incorporated.

To see how this works, let us see how the system responds to a common query. Suppose the editor generates a query asking for specifications about entry point e in the current context program p. First the system manager looks at the composition map

programs/p/location

to find the module m/v that contains e within p. Next, it looks in the entry descriptor

modules/m/v/entries/e

to fetch the desired information.

We expect this to be finished by December 1983. In the year between December 1983 and December 1984, we will add the execution monitor and optimizing code generator, employing simple interprocedural analysis. Linkage tailoring and other sophisticated features will be put off until the basic functions are in place.

11. References

[ANSI81a] ANSI, Proposals approved for Fortran 8x. X3J3/S6.80 (preliminary document), American National Standards Institute Inc. (November 30, 1981).

[Albe81a] Alberga, C. N., , A. L. Brown, Leeman, G. B., Mikelsons, M., and Wegman, M. A program development tool. In *Conf. Record of the Eighth Annual ACM Symposium on Principles of Programming Languages*, pp. 92-104 (January, 1981).

[Alle74a] Allen, F. E. Interprocedural data flow analysis. In *Information Processing 74*, pp. 398-402 (1974).

[Alle74b] Allen, F. E. and Schwartz, J. T. Determining data relationships in a collection of programs. RC 4989, IBM T. J. Watson Research Center, Yorktown Heights, NY (August 1974).

[Alle82a] Allen, J. R. and Kennedy, K. PFC: a program to convert Fortran to parallel form. Report MASC TR 82-6, Dep. of Mathematical Sciences,

Rice Univ., Houston, TX (March 1982).

[Bann79a] Banning, J. P. An efficient way to find the side effects of procedure calls and the aliases of variables. In *Conf. Record of the Sixth Annual ACM Symposium on Principles of Programming Languages*, pp. 29-41 (January 1979).

[Bart78a] Barth, J. M. A practical interprocedural data flow algorithm. *Comm. ACM 21*(9) pp. 724-736 (September 1978).

[Cart82a] Cartwright, R. S., Dennis, J. E., Jump, J. R., and Kennedy, K. $\mathbb{R}^n$: an experimental computer network to support numerical computation. MASC TR 82-5, Dep. of Mathematical Sciences, Rice Univ., Houston, TX (1982).

[Coop83a] Cooper, K. D. Interprocedural Data Flow Analysis in a Programming Environment. Ph.D. Dissertation, Dep. of Mathematical Sciences, Rice Univ., Houston, TX (May 1983).

[Deme81a] Demers, A., Reps, T., and Teitelbaum, T. Incremental evaluation for attribute grammars with application to syntax-directed editors. In *Conf. Record of the Eighth Annual ACM Symposium on Principles of Programming Languages*, pp. 105-116 (1981).

[Donz75a] Donzeau-Gouge, V., Huet, G., Kahn, G., Lang, B., and Levy, J. J. A structure oriented program editor: a first step toward computer assisted programming. *International Computing Symposium 1975*, pp. 113-120 North-Holland Publishing Company, (1975).

[Feld77a] Feldman, S. I. Make—a program for maintaining computer programs. *Software—Practice and Experience 9* pp. 255-265 (1977).

[Fosd76a] Fosdick, L. D. and Osterweil, L. J. Data flow analysis in software reliability. *Computing Surveys 8*(3) pp. 305-330 (September 1976).

[Gosl83a] Gosling, J. Unix Emacs. User's Manual, Carnegie-Mellon Univ., Pittsburg, PA (1983).

[Habe82a] Habermann, A. N.. computer science colloquium, Rice Univ. April 1982.

[Habe82b] Habermann, A. N. and Notkin, D. S. The Gandalf software development environment. Research report, Computer Science Dep., Carnegie-Mellon Univ., Pittsburg PA (January 1982).

[Kenn80a] Kennedy, K. Automatic translation of Fortran programs to vector form. Rice Technical Report 476-029-4, Rice Univ. (October 1980).

[Kenn81a] Kennedy, K. A Survey of Data Flow Analysis Techniques. pp. 1-54 in *Program Flow Analysis: Theory and Applications*, ed. N. D. Jones,Prentice-Hall, New Jersey (1981).

[Kern78a] Kernighan, B. W. RATFOR—A preprocessor for a rational Fortran. in *UNIX Programmer's Manual*, Bell Laboratories (1978). Seventh Edition.

[Kuck80a] Kuck, D. J., Kuhn, R. H., Leasure, B., and Wolfe, M. The structure of an advanced vectorizer for pipelined processors. In *Proc. IEEE Computer Society Fourth International Computer Software and Applications Conf.*, (October 1980).

[Lome77a] Lomet, D. B. Data flow analysis in the presence of procedure calls. *IBM J. Research and Development 21*(6) pp. 559-571 (November 1977).

[Masi80a] Masinter, L. M. Global program analysis in an interactive environment. SSL-80-1, Xerox PARC, Palo Alto, CA (Jan. 1980).

[Mike80a] Mikelsons, M. and Wegman, M. N. PDE1L: the PL/1 program development environment principles of operation. RC 8513, IBM T. J. Watson Research Center, Yorktown Heights, NY (Nov. 1980).

[Myer81a] Myers, E. W. A precise inter-procedural data flow algorithm. In *Conf. Record of the Eighth Annual ACM Symposium on Principles of Programming Languages*, pp. 219-230 (1981).

[Oste81b] Osterweil, L. Using data flow tools in software engineering. pp. 237-263 in *Program Flow Analysis: Theory and Applications* , ed. N. D. Jones,Prentice-Hall, New Jersey (1981).

[Oste81a] Osterweil, L. TOOLPACK architectural design. Toolpack Ref. LO-10304, Univ. of Colorado at Boulder, Boulder, CO (March 1981).

[Pinc73a] Pinc, J. H. and Schweppe, E. J. A Fortran language anticipation and prompting system. In *Proc. ACM Nat. Conf.*, (1973).

[Reps82a] Reps, T. Optimal-time incremental semantic analysis for syntax-directed editors. In *Conference Record of the Ninth Annual ACM Symposium on Principles of Programming Languages*, pp. 169-176 (1982).

[Rose79a] Rosen, B. K. Data flow analysis for procedural languages. *J. ACM 26*(2) pp. 322-344 (April 1979).

[Rose81a] Rosen, B. K. Linear cost is sometimes quadratic. In *Conf. Record of the Eighth Annual ACM Symposium on Principles of Programming Languages*, pp. 117-124 (1981).

[Rudm81a] Rudmic, A. and Moore, B. The Chill compiling system: towards a Chill programming environment. In *Proceedings of the Fourth International Conference of Software Engineering for Telecommunications Software*, pp. 187-190 (1981).

[Russ78a] Russell, R. M. The CRAY-1 computer system. *Comm. ACM 21*(1) pp. 63-72 (January 1978).

[Shar81a] Sharir, M. and Pnueli, A. Two approaches to interprocedural data flow analysis. pp. 189-233 in *Program Flow Analysis: Theory and Applications* , ed. N. D. Jones,Prentice-Hall, New Jersey (1981).

[Site78a] Sites, R. L. An analysis of the Cray-1 computer. In *Proc. Fifth Annual Symposium on Computer Architecture*, pp. 101-106 (April 1978).

[Spil72a] Spillman, T. C. Exposing side effects in a PL/1 optimizing compiler. In *Information Processing 71* , pp. 376-381 (1972).

[Teit81a] Teitelbaum, R. T. and Reps, T. The Cornell program synthesizer: a syntax-directed programming environment. *Comm. ACM 24*(9) pp. 563-573 (September 1981).

[Teit77a] Teitelman, W. A display-oriented programmer's assistant. In *Proc. Fifth Int. Joint Conf. Artificial Intelligence*, pp. 905-915 (1977).

[Wegm80a] Wegman, M. N. Parsing for a structural editor. In *Proc. Twenty-first Annual Symposium on Foundations of Computer Science*, IEEE Computer Society, (1980).

[Wegm82a] Wegman, M. N. Summarizing graphs by regular expressions. RC 9364 (41252), IBM Research, Yorktown Heights, NY (April 1982).

[Weih80a] Weihl, W. E. Interprocedural data flow analysis in the presence of pointers, procedure variables, and label variables. In *Conf. Record of the Seventh Annual ACM Symposium on Principles of Programming Languages*, pp. 83-94 (1980).

[Zade83a] Zadeck, F. K. Incremental Data Flow Analysis in a Structured Program Editor. Ph.D. dissertation, Dep. of Mathematical Sciences, Rice Univ., Houston, TX (August 1983).