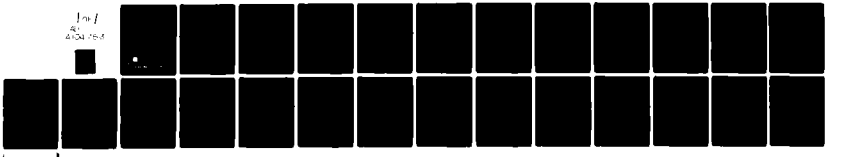
AD-A104 753 CARNEGIE-MELLON UNIV PITTSBURGH PA DEPT OF COMPUTER --ETC F/G 9/2
AN INCREMENTAL PROGRAMMING ENVIRONMENT,(U)
APR 80 P H FEILER, R MEDINA-MORA
UNCLASSIFIED CMU-CS-80-126 NL



END
DATE FILMED
10-81
DTIC

CMU-CS-80-126

(12)

# An Incremental Programming Environment

**Peter H. Feiler**
**Raul Medina-Mora**

**Department of Computer Science**
**Carnegie-Mellon University**
**Pittsburgh, Pa. 15213**

**April 1980**

DEPARTMENT
of
COMPUTER SCIENCE







**Carnegie-Mellon University**

# An Incremental Programming Environment

Peter H. Feiler
Raul Medina-Mora

Department of Computer Science
Carnegie-Mellon University
Pittsburgh, Pa. 15213

April 1980

## Abstract

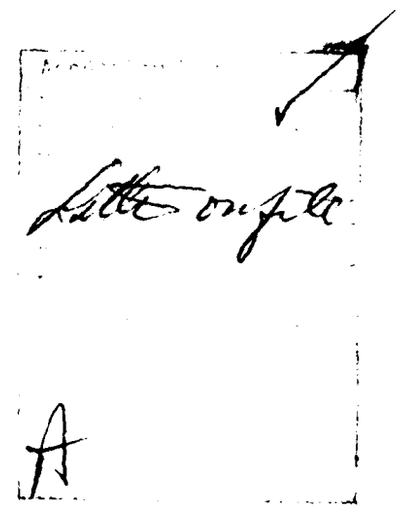
This document describes an Incremental Programming Environment (IPE) based on compilation technology, but providing facilities traditionally found only in interpretive systems. IPE provides a comfortable environment for a single programmer working on a single program.

In IPE the programmer has a uniform view of the program in terms of the programming language. The program is manipulated through a syntax directed editor and its execution is controlled by a debugging facility, whose actions are provided through commands of the editor. Other tools of the traditional tools cycle (translator, linker, loader) are applied automatically and are not visible to the programmer. The only interface to the programmer is the user interface of the editor.

This work is sponsored in part by the Software Engineering Division of CENTACS/CORADCOM, Fort Monmouth,NJ, and in part by the National University of Mexico.

# Table of Contents





# 1. Introduction

A Programming Environment supports programmers in the process of transforming specifications into working programs. Initially a programming environment consisted of a high level language and some simple tools. In order to improve the quality of programs, concepts such as modularization and data abstraction were incorporated into programming languages. The ADA language is a product of the evolution of these and other concepts.

In IPE (*Incremental Programming Environment*) we intend to go a step further by having the programming environment actively participate in the development and maintenance of software. This participation is accomplished by presenting a uniform and integrated system. The tools in the environment understand each other's objectives and collaborate towards a common goal. The environment has knowledge about the objects it manipulates and their current state. It is, therefore, able to respond to incorrect or undesirable user actions. It can also make the state available for inspection. A common program representation provides the means of communication among the tools.

IPE provides a comfortable environment for a single programmer working on a single program. It is, therefore, only a building block for a more general software development environment. Such a system is being designed and built at Carnegie-Mellon University in the *Gandalf project* [Notkin 79], and IPE is part of that project [Habermann 80a]. Gandalf, in addition to IPE's facilities, provides support for managing a project that involves the interaction of several programmers [Habermann 79], and for the manipulation of system compositions and version control [Habermann 80b].

IPE is an interactive programming environment that is based on *compilation technology* but provides facilities traditionally found in interpretive systems. By compilation technology we mean the translation of the source program into a lower level representation such as object or machine code.

The programmer has a uniform view of the program in terms of its source form. The program is manipulated through a syntax directed editor, and its execution is controlled by a control module that replaces the traditional debugger. The control module's commands are provided to the programmer as commands of the editor. Other tools are applied automatically at the appropriate times by IPE. These tools and their representation are not visible to the user, the only (thus uniform) interface is the user interface of the editor. During the construction and manipulation of the program, the programmer focuses on a small piece of the program at a time. The changes made cause the system to incrementally compile those pieces and incorporate them into the executable version of the program.

The language of choice for IPE should be a strongly typed algol-like language with facilities for

abstraction and modularization. A perfect choice would be ADA but as the language is still going through the final modifications and compilers do not yet exist for it, we have chosen instead GC, a type-checked variation of C that runs under UNIX[tm] at CMU [Feiler 79a]. Modularization and abstraction facilities have been added to GC through the system composition language.

The rest of the paper is organized as follows: in section 2 we describe traditional methods of programming in compiler systems and relate to them the approach taken in IPE. Section 3 contains a discussion of various design and implementation issues dealt with in IPE. Finally, in section 4 we draw some conclusions and describe the current state of the IPE implementation.

# 2. Program Modification Cycle

As a programmer goes from specification of programs to their implementation, many tools are normally used. The important issue here, however, is not the tools that are used, but the problems that the tools address. For example, a typical editor does not manipulate programs, but only manipulates text.

There are four problems that programming environment tools must help the programmer solve. First, there must be a method for the programmer to enter and modify programs. Second, there must be a way of ensuring the syntactic and semantic correctness of programs. Third, an executable form of the program must be created. And fourth, there ought to be a process that can help the programmer debug programs.

In the traditional methods these tools are largely independent of each other, have different user interfaces and use different representations for the programs. It is the intent of IPE to provide a uniform environment. For the programmer this means a unified user-interface for communication with the appropriate tools. The tools are integrated into one system and work with a common program representation. Some of the tools are not directly visible to the programmer, but are applied by the system automatically. The fact that the tools share a common representation permits the short-circuiting of the tools cycle.

## 2.1 Traditional Methods

Each of the programming environment problems mentioned in the previous section has been addressed many times since people started to program computers. Almost all the solutions, however, fit into a standard tools cycle. The variations on the tools used and the order of their use are slight, so the description below is representative of most traditional programming environments.

The traditional tools are the editor, the translator, the linker and loader, and the debugger. The order in which these tasks are usually applied is: enter/modify, compile, link and load, debug. This cycle is repeated until the program appears to be correct. The remainder of the section describes these tools in more detail.

### 2.1.1 The Editor

The editor is used in a programming environment as a way to initially enter and subsequently modify programs that are being developed in a particular programming language. The drawback with traditional editors is that they are general purpose text manipulators. The editor usually has no knowledge of whether the text being entered represents a program, a document, or a poem. If it has some idea, it is generally very limited. This ignorance inherently causes the need for parsers that

determine if the text which represents a program is syntactically correct. The parser is usually part of the translator, which makes the checking process very costly.

### 2.1.2 The Translator

The basic job of the translator is to transform program text into another representation of the program. In a compiler this representation is usually machine or assembly language, while in an interpreter it is often an intermediate language.

In traditional systems there are two tasks required for this transformation. First, the textual representation of a program must be analyzed for syntactic and semantic accuracy, a process that is necessary because of the inability of the editor to support a programming language. The resulting parse tree is then checked for consistency of the language semantics, which includes type checking, parameter checking, and operand coersion.

The second task is the actual translation of the program into a machine representation, such as assembly, object, or even micro code. This is done by a traversal of the parse tree, and is often combined with some semantic checking. In the process of generating code, optimizations are often applied at various points in order to improve the quality of the produced code. However, during program development, nonoptimizing code generators are often used.

### 2.1.3 The Linker and Loader

Programs, even small programs, are often broken into several smaller pieces. It is usually convenient to keep these pieces in separate files and work on them independently. The amount of program text to be checked is kept small. However, there are often links among the pieces that must be maintained. These are known as external references, and information about them is generated by the translator.

In a traditional system it is the job of the linker to resolve these external references so that previously translated program pieces can be combined to form a complete executable form of the program. Because no assumptions are made about other program pieces already being translated, linking is a separate pass in the tools cycle, requiring all translated program pieces to be processed for reference resolution.

The loader's job is simple once the linker has resolved the external references. All it has to do is actually place the program in memory so that it can be executed correctly. Traditionally, whenever any kind of change is made to a loaded program (*i.e.*, a piece is altered and recompiled), linking and loading has to be done again.

### 2.1.4 The Debugger

The debugger is used to help a programmer observe the execution of a program. The purpose of this is to detect erroneous program behavior and to locate its cause. A very simple form of debugging support is symbolic dumps. A more advanced form is the interactive debugger that allows programmers to dynamically set trace and break points at arbitrary locations, to inspect and alter values of variables and data structures, and possibly even to use conditional debug functions (*e.g.*, break whenever x = 0). While debuggers allowing these and possibly other actions are often available, most of them do not work at the source code level. Variable and procedure names can be referred to symbolically, but for the rest the programmer must deal with the machine representation.

Source level debuggers are mostly found in conjunction with interpreters. Where compiler systems provide source level debugging it is accomplished with a simple text line to machine representation mapping. The unit for break and trace points is a text line rather than a statement or expression. The compiler in such a compiler system in general does not optimize code because certain optimizations (*e.g.* code motion) make the mapping between the two representations difficult to maintain.

## 2.2 The IPE Approach

The approach to program development supported by IPE has some important advantages for both the programmer and the IPE implementor. Modularity restricts the accessibility at any point in the program. The programmer has controlled access to a subset of the program's procedures and objects, and thus can cause less damage to unrelated program pieces. Modularity also restricts the scope for semantic checking, making it much more simple and efficient.

The integration of tools and their automatic application at appropriate times provides a single and uniform user interface. The fact that a tool will reside in a well-defined environment, *i.e.* specific tools exist and they are applied at appropriate times, can be used to simplify the design and implementation of that tool. To enhance this cooperation all tools share a common program representation -- the syntax tree representation. The programmer also views the program modification cycle in a uniform way. He deals with his program in terms of language constructs. This avoids the usual necessity of constructing one representation in terms of another (*e.g.*, writing a syntactically correct program using a line editor).

The tools cycle in IPE, called *incremental program modification mechanism*, is very different from the traditional tools cycle. The programmer has a uniform view of his program in terms of source code. The program is manipulated through the editor and its execution is controlled by a control module (which replaces the traditional debugger). The control module's commands are presented to the programmer as commands of the editor. Other tools, such as the translator and the linker/loader,

are not visible to the user. They are automatically invoked to keep the tree representation and the (invisible) machine representation consistent.

### 2.2.1 Syntax Directed Program Editor

Modifications to the program can be made through the editor during construction of the program, and anytime the program execution is suspended. The editor is a constructive editor, that understands the syntax of the programming language, and constructs a program in terms of language constructs. Language constructs (such as variables, operators, expressions, different types of statements) can be added, modified, or removed. The programmer communicates with the editor in terms of language constructs.

The programmer constructs his program by inserting templates that represent different language constructs and then filling the "holes" of those templates with other templates. Since the editor knows which constructs are valid at any given point, it allows the programmer to insert a language construct only in a place where it is syntactically correct.

For example, instead of typing the character sequence for an if statement, the programmer calls on the template "if". The result is the insertion of

```
if (<expression>)
    <statement>
else
    <statement>
```

at the current program position, provided that this construct is syntactically correct in that context. The current program position is advanced to <expression> so that it can be similarly expanded. Note that the editor provides all the necessary keywords, separators, terminators, and all the other "syntactic sugar" required by the language like the parentheses around the <expression> construct that are required by the C language syntax. Problems such as misspelled or non-matching keywords cannot occur because the language constructs are inserted by the editor and not by the typist. The editor relieves the programmer from the worries of the syntax constraints imposed by the language.

The editor internally represents the program as a *syntax tree*. Each template corresponds to a node of a certain type in the tree. The holes of the template are the offsprings of the node. They will be filled in as subtrees representing the expansion of those holes. Figure 2-1 shows the representation of the "if" statement of the example above. To present the programmer the text of his program the editor uses an *unparser* that translates the syntax tree into human readable text. As part of its task this unparser does the pretty-printing of the program. Thus, the programmer actually constructs and manipulates a program tree without necessarily being aware of it.

The programmer interacts with the editor through language commands and editing commands. Language commands are used to construct new templates (*e.g.* the call for an "**if**" template). Editing commands are used for manipulating the program tree (*e.g.* delete a subtree). In order to provide a uniform interface, the programmer interacts with the control module through the editor interface. System commands are provided for those Control Module actions that cannot be supported with language or editing commands, like continuation of execution, display of procedure call nesting, etc.

The editor invokes the semantic checking routines while the programmer is constructing or modifying his program and informs the programmer of any semantic inconsistencies. The semantic checking goes on while the programmer is "thinking" at the terminal. When the programmer finishes the editing session, the program is syntactically correct and semantically checked, but it could be incomplete, that is, some "holes" may be left unexpanded. The editor does not enforce semantic correctness. It informs the programmer of errors and allows their correction. Code will not be produced for a program that is not semantically correct. Semantic correctness in this context means correctness with respect to the programming language semantics.

### 2.2.2 The Common Representation

The syntax tree constructed by the editor has been chosen as the common representation of programs. There are two types of nodes in the syntax tree: terminal nodes (or leaves) and non-terminal nodes.

*Terminal* nodes are used to represent variables, constants, some static language elements (*e.g.*, data type names), and unexpanded program constructs (placeholders or "holes") which will be substituted by the correct subtree in the process of constructing a program. The node for variables contains information about the symbol table. It also provides some space to put semantic information like the type of the variable and references to other occurrences of the same variable.

*Non-terminal* nodes describe subtrees of the program corresponding to control flow constructs and data definitions in the language (*e.g.* an **if** construct.) The information available at the node includes the type of language construct, references to the parent node, and to its offsprings. There are two classes of nonterminal nodes, one with a fixed number of offsprings, and the other with a variable number of offsprings (represented as a linked list for constructs like a **compound** statement). In addition, space for semantic information and mapping information from the code generator -- later used by the control module -- is provided.

Figure 2-1 illustrates the representation of the **if** statement example mentioned in section 2.2.1. The three offsprings are terminal nodes representing unexpanded nodes.

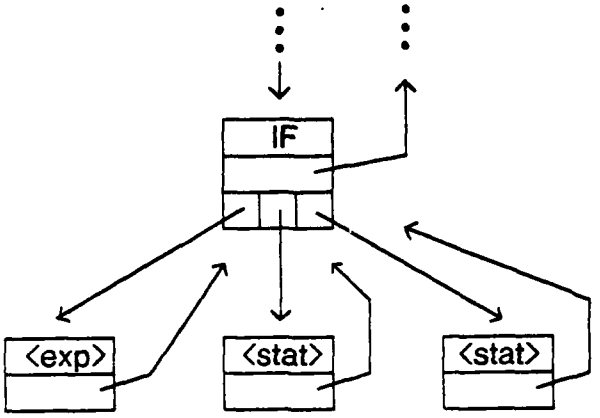



**Figure 2-1:** Representation of an IF Subtree

In the next sections we describe the remaining mechanisms and tools that are supported by IPE. They all interact with the common program representation. They are grouped into program translation, *i.e.* the update of the static program representation, and debugging, or analysis of dynamic program behavior.

### 2.2.3 Program Translation

In the process of constructing a program its semantic correctness is checked. The checking is invoked automatically on an incremental basis or whenever the programmer completes changes to a program piece (*e.g.* leaves the scope of a procedure with the display cursor). Semantic correctness is not enforced, that is, a programmer can construct a partially complete program. One example is the use of a variable before declaring it. However, code is only generated for semantically correct program pieces.

Similar to the semantic checking, code generation and link/loading of program pieces happen automatically. This activity is invisible to the programmer, with the exception of possibly noticing it in the system response. In order to keep the response time at a reasonable scale, the amount of code being regenerated, relinked and reloaded is kept small. When the programmer finishes a program change and the result is semantically correct, the procedure affected by the change is retranslated and its external references are resolved in the process. The machine code for the compiled procedure is replaced in the executable form through an incremental loading mechanism. Other program pieces do not have to be relinked or reloaded.

A detailed discussion of the translation step and the machine representation can be found in

section 3.2.

### 2.2.4 Debugging

One of the goals of the IPE project is to explore whether an IPE system can be supported by performing compilation but no interpretation. In many compiler systems to date the debugger often provides an interpreter for a limited subset of the language.

The debugger, embedded in the IPE system as part of the control module, uses the incremental program modification mechanism of the system to implement its actions. For example, the programmer inserts a breakpoint statement in the program tree using the constructive editor. This change to the program tree causes that program piece to be retranslated and the debug instruction to be reflected in the executable representation.

All debugging actions are performed through the editor's user interface. No separate command interpreter exists, and the programmer is unaware of the transition between the different tools provided by the IPE system.

As the above example shows, some debugging actions are implemented directly as language commands. This approach, however, may require an extension to the language (*e.g.* a breakpoint statement) and runtime support to transfer control between the user program and the IPE system. Other debugging actions are provided through system commands. Into this class fall the procedure call nesting display, the continuation of execution, and the execution of a temporary program statement. This last debugging action includes evaluation of an expression in context (*e.g.* a + b) and procedure invocation.

Since some debugging actions are implemented using the syntax tree representation, more sophisticated language-oriented debugging actions can be provided. One example is the runtime checking of assertions. This idea has existed for some time (*e.g.* in Euclid [Lampson 77]). Assertions are provided by the programmer as part of the program. They are in general limited to expressions that may include calls to functions without side-effects. We attempt to expand the expressive power of assertions beyond that of many programming languages. In order to express certain verification conditions, constructs such as quantifiers and previous values are introduced. Initially no code is generated for assertions, but at any time during program execution the programmer can request them to be enabled. Assertion code is then added in the machine representation, and the validity of the assertions is checked. This mechanism does not guarantee correct programs but is a small step in the direction of producing verified programs.

Other debugging actions make use of the data flow and control flow information in the program

tree. An example of a debugging action in this class is the tracing of a *crazy* variable, *i.e.* locating the place where a variable is assigned an erroneous value. This can be done by attaching a condition to a variable and having it automatically checked whenever the variable is changed.

In summary, quite sophisticated debugging actions can be provided without requiring a separate complex system such as an interpreter. Debugging actions do not impose any overhead when they are not enabled. However, limited debugging capabilities are always available, and the debugging actions can be enabled any time the program execution is suspended.

# 3. Design and Implementation Issues

In this section we discuss some technical issues that have arisen in the process of designing and implementing IPE.

## 3.1 The Editor

As mentioned earlier, the editor knows how to build programs in a specific programming language. The major advantages of this constructive approach are that programs are edited in terms of the programming language constructs, and that users cannot enter syntactically incorrect programs. Hence, the first few compiles just "to get the semicolons right" are no longer necessary. The whole process of translation should be much more efficient than the normal method of lexical and syntactic analysis, since the translation of character strings to lexemes and from sequences of lexemes to syntactic units are no longer necessary. Parsers and lexical analyzers are no longer needed !

The programmer can manipulate the program tree through language commands and editing commands. Language commands facilitate the building of program pieces in terms of language constructs by expanding the "holes" in the program templates. The "holes" in the templates are called *meta-nodes*, that represent nodes that haven't been expanded. Meta-nodes have a name (also helpful for display purposes). By default they get assigned the name of the set of language constructs (also refered to as language operators) that can be applied at that point (*e.g.* the set of statements). The programmer may rename such meta-nodes in order to remember the name and return to them at a later time, when he wants to expand them. After every language command the current tree position is advanced to the next meta-node. Editing commands provide a facility for more general tree manipulation.

As part of the language specification an *unparsing scheme* is given for every language construct. This provides an easy way to implement several pretty-print formats for the same programming language by simply changing some internal tables.

Figures 3-1 and 3-2 illustrate a sample editor session for the process of creating a "**for**" loop statement in C. The first column gives the commands typed by the user, the second column shows how the terminal display would look like after the command is executed and in the third column the syntax tree structure is presented.

Note that commands do not need to be typed fully, the programmer only needs to type those characters needed to unambiguously determine the command. Some commands have synonyms for convenience (*e.g.* PLUS has "+" as a synonym). Variables are preceded by a single quote to differentiate them from commands.

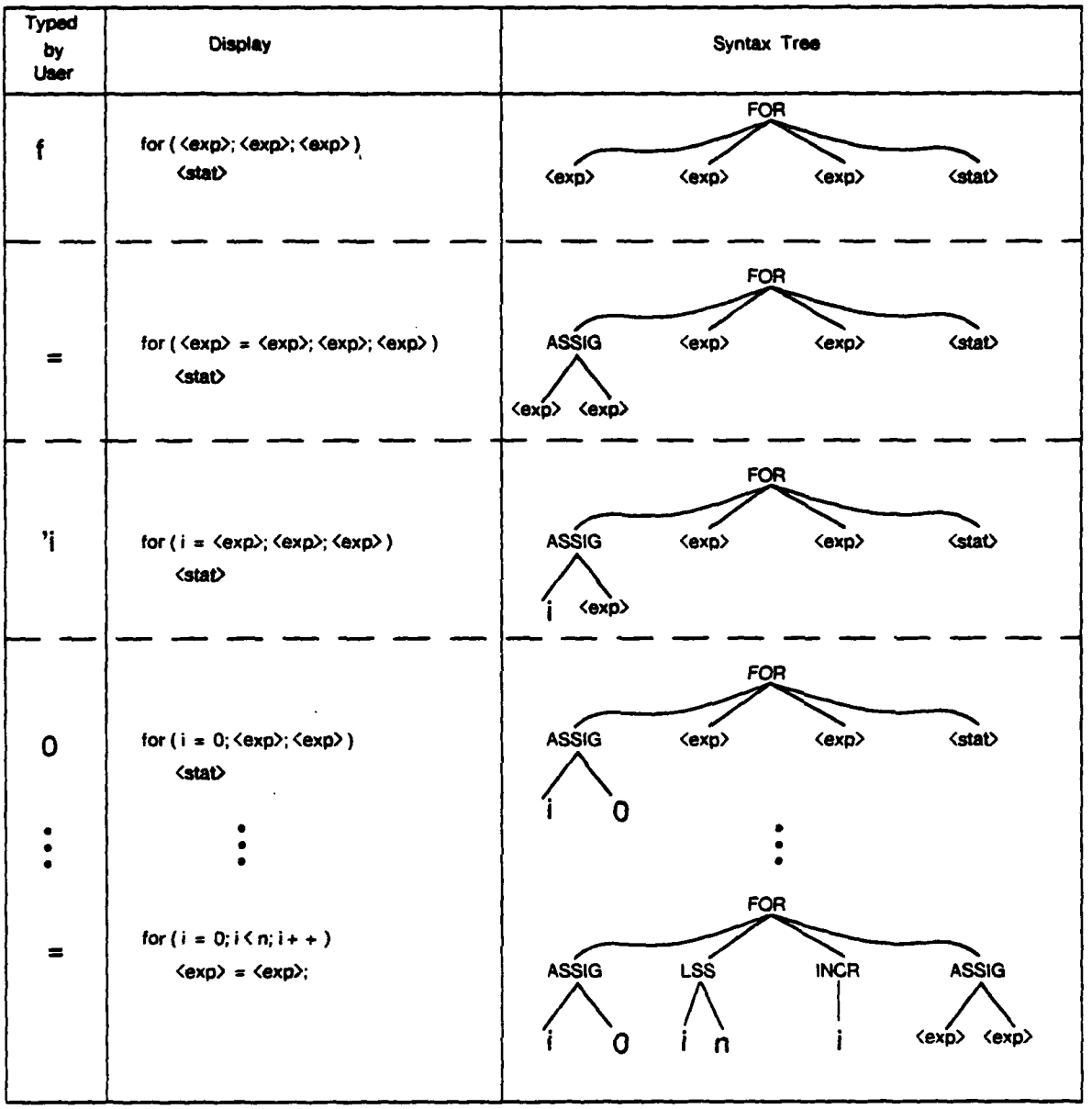



**Figure 3-1:** A Sample Editor Session

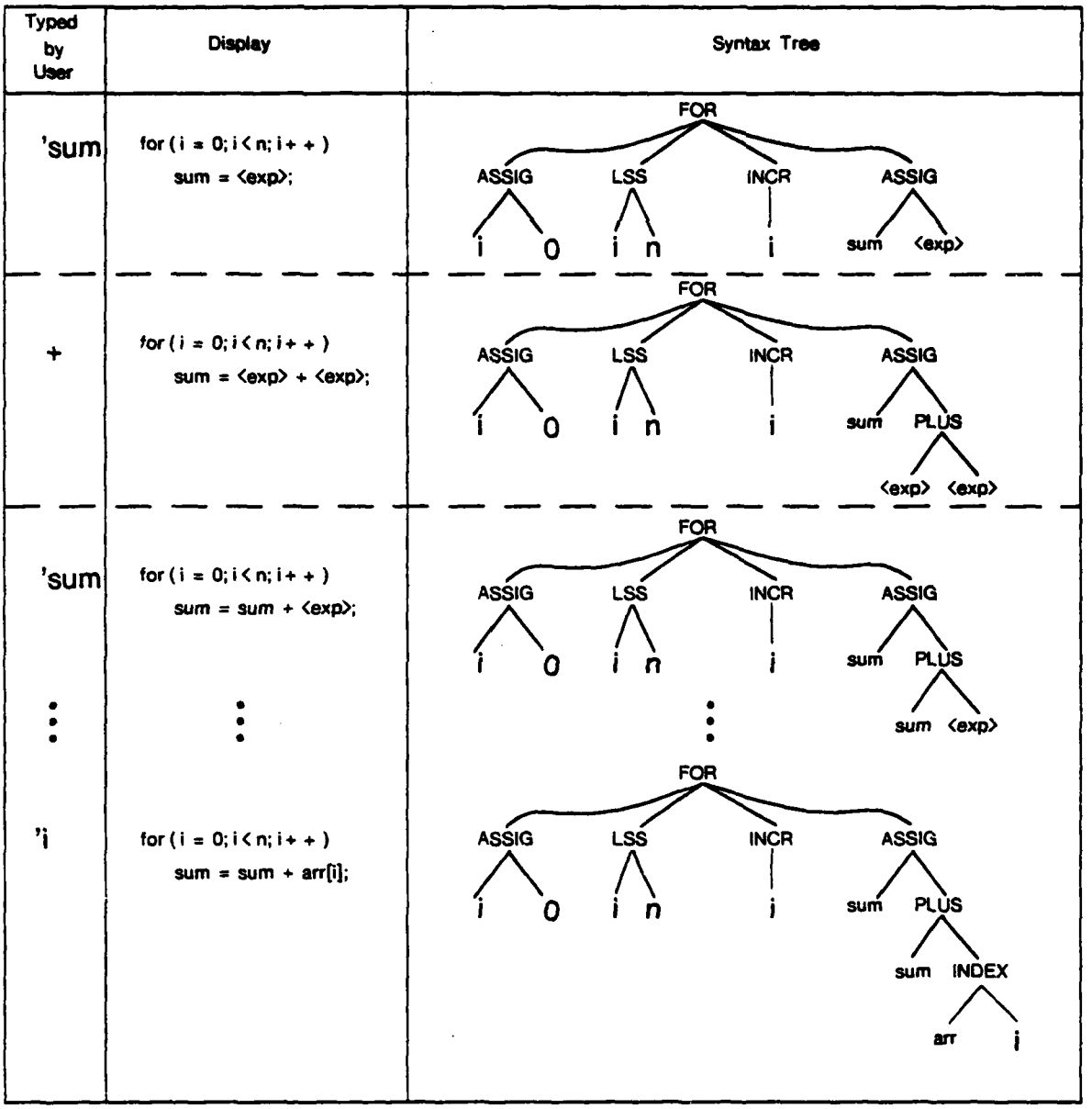



**Figure 3-2:** A Sample Editor Session (cont.)

Some of the editing commands available in the editor are:

**Delete.** Causes the deletion of the subtree that is at the current program position. A meta-node is then inserted on its place.

**Clip.** Removes a subtree from the program tree, but does not throw it away. This command prompts for a name under which the subtree can be referred to for future use. A meta-node is put in its place.

**Insert.** Prompts for the name of a previously clipped subtree and inserts it in the place of a meta-node as long as the root of the subtree is of the correct type (*e.g.* the insertion is syntactically correct).

**Write.** Saves the program tree by writing it into a file, so that it can be retrieved later. This command allows the programmer to save his program for later development.

**Read.** Retrieves a program tree from a file and places the current tree position at the first meta-node or at the root if there are no meta-nodes.

There are also commands to "move" around the tree: they move the current tree position (*e.g.* to the "father", to the "first son", to the next element in a list, or to some subtree specified by a pattern, etc). The current tree position is always highlighted in some form on the terminal display.

In addition there are other commands that provide on-line help facilities to inform the programmer about the available commands (*e.g.* which are the valid language commands at the current tree position?). In a sophisticated display device this information could be presented in a menu-like format.

As mentioned above, syntax directed editing has a major impact on the implementation of programming environments because it makes lexical analyzers and parsers obsolete. In addition it may also influence reference manual writing and even language design. The manual would no longer have to describe if a semicolon is a statement separator or a terminator and the language designer would not necessarily need to decide this. Emphasis will then be placed on the language constructs and the facilities provided by the language.

Some language ambiguities (*e.g.* the "dangling else" problem of some languages) disappear because with the constructive approach it is clear what the programmer means (*e.g.* to which **if** statement the **else** part belongs), and there is no need for a parser to try to figure it out.

Another traditional problem of programming languages has been the correct parenthesization of expressions given the operator precedences of the language. In this approach the programmer

constructs the expression tree directly. As a part of the unparsing mechanism, the editor provides the correct parentheses for the text form that is displayed on the terminal screen.

### 3.1.1 The Editor viewed independently from IPE

The editor can also be viewed separately from IPE. A *Syntax Directed Editor Generator* has been developed. It permits the automatic generation of a constructive editor for a formally described programming language. The syntactic formalism to describe languages expresses the language constructs, the sets of valid language constructs for every "hole" in the templates, the formatting (unparsing) information for the display of every construct, and the names of a set of routines, to be called by the editor that perform the semantic checking for every construct of the language and for the program as a whole.

A preprocessor is applied to such a description and it produces the internal tables that are used by the editor as its "knowledge" of the language. This process has been applied successfully to several very different types of languages: a subset of C on which IPE is currently based, to *Alfa* a non algol-like applicative language designed by Nico Habermann [Habermann 80c], and to the system composition and version maintenance language of Gandalf [Habermann 80b].

## 3.2 The Program Representation

The IPE system internally maintains two representations of a program, the tree representation and the *machine* or *executable* representation which is interpreted by the hardware during program execution. The first representation is the syntax tree of the program. One choice for the representation of the syntax tree was TCOL, especially $TCOL_{Ada}$ [Schatz 79]. It is a symbolic tree representation in text form, that has been accepted as one standard in the Ada language development. We decided against using that representation directly because of the conversion cost into an internal representation. We have chosen an internal representation that comes very close to an internal representation of $TCOL_{Ada}$, and we are able to generate a TCOL representation if necessary.

The second representation is generated automatically from the first through a translation step. This separation of a program into two representations allows a *host/target* approach, *i.e.* the execution of the program may be performed on a machine different from the machine running the IPE system as required by Stoneman [DoD 80]. Since all IPE tools primarily work on the tree representation, very little IPE support must be provided on the target machine. In order to incorporate a new target machine into IPE, the target machine runtime support and a code generator for the target machine on the host must be constructed. Therefore the dependency on the target machine is limited.

The changes made by the programmer in the tree representation of the program must be reflected in the machine representation. The behavior of the executing program must be consistent with what the programmer expressed in terms of the programming language. Since changes are made incrementally to the program tree, they also are reflected incrementally in the executable representation. However, the cost of this incremental program modification must be kept small, otherwise it will be noticeable to the programmer in the system response.

### 3.2.1 The Machine Representation

The structure of the machine representation has a strong influence on the cost of an incremental program modification. On one hand, a certain complexity in the supported structure, *e.g.* segmentation, permits a simple replacement mechanism. On the other hand, the representation should be supported directly on existing conventional machines without a software simulator.

The procedure has been chosen as the replacement unit of the program. Procedures have fixed entry points that refer to the actual location of the code. They are assigned when the procedures are specified. All procedure calls are made indirectly through these entry points. Global data is placed in an area that is separate from (and unaffected by) the replacement of program pieces.

Code generation is invoked on a semantically correct subtree representing the procedure to be compiled. Since the procedure is semantically correct, all called procedures must have been specified, *i.e.* entry points have been assigned to them, and all referenced global objects must have been declared, and therefore, space has been allocated for them. Therefore, all global references are known at code generation time. All local references within the procedure can also be resolved at code generation time. As a result, code can be produced with all external references resolved. A separate linking step is not necessary.

When the machine representation of a procedure is replaced, the new copy may not fit into the space of the old one. Therefore, it must be placed in some free memory space. Since memory space is often limited, released memory space must be recovered and the allocated space compacted. This may require moving the machine code for some of the procedures. In order to make this moving possible without requiring a relink of the moved program pieces, restrictions are imposed on the use of different addressing modes of the machine. References are kept invariant from the physical location of the code. all references within the moved code piece, *i.e.* local references, are relative to the code piece (*e.g.* PC relative), and all references outside the moved code piece, *i.e.* global references, are absolute. Thus, code can be compacted through a simple block-move operation. This garbage collection problem may not be relevant if an underlying machine with a more complex memory structure than linear memory space is available (*e.g.* a segmented machine).

A comparison with conventional compilation techniques, including separate compilation, shows that the cost of generating code for a modified program part is greatly reduced, thus keeping the response time small. Since the program is already in tree form, text does not have to be processed: lexical and syntax analysis is not done. Semantic checking is performed incrementally while the programmer modifies the tree through the editor, and the link-editing step is merged into the code generation phase. Therefore, the system effort to replace a program piece after the programmer made a change consists of code generation and loading of that program piece.

### 3.2.2 The Mapping

For debugging purposes, a mapping between the tree representation and the machine representation is required. This mapping allows the flow of execution in the machine representation to be traced in the tree representation. The mapping, however, does not need to be complete. Only certain tree nodes must have an exact mapping in order to determine the exact execution state in both representations (*e.g.* routine entry, return addresses, debugging stoppage points). For other program parts only an approximate mapping is needed. For example, signals from program execution that are not handled by a debugging action, are only indicated to have happened in a certain program region.

The mapping is produced by the code generator. It may be either locators for tree nodes in the machine representation or references to code pieces corresponding to tree nodes stored in the tree representation. Optimizations are applied by the code generator in order to improve the quality of the machine representation. However, they may affect the mapping of the program. Optimizations that transform the program tree cannot be employed at mapping points, and optimizations that cache program state in machine registers must update the cached values in memory at those points. Because the mapping is incomplete and the mapping points are sparse, optimized code is produced for large parts of the program, and debugging is still supported.

## 3.3 Program Execution

The IPE system supports incremental construction of programs, *i.e.* the building of program pieces until the program is in an executable state. A program is executable even though program pieces are not fully written yet. It is possible to start execution of a program that contains a procedure that is specified but whose body, *i.e.* implementation, is still missing. If that procedure is being called during execution, execution is suspended and the programmer can complete the body. Then the program execution may be resumed.

### 3.3.1 Continuation of Execution

Resumption of execution can be permitted if the program representations and the execution state of the program are in a consistent state. A consistent state means that the program shows the same behavior on resumption of execution after a change as it would when restarting the execution with the new executable version. Both the programmer's change to the logical structure of the program and the incremental update of the change in the executable representation may have side-effects on the execution state, *i.e.* the stack of activation records (dynamic control flow) and the state of objects. In the following paragraphs we describe different types of changes and their difficulties for continuation of execution.

Program changes to the control flow are easily dealt with if they do not involve active procedures. They are replaced by the incremental program modification mechanism and have no side-effects on the execution state. However, when replacing the executable representation of an active procedure, the placement of the code body may invalidate the return address on the activation stack and require an update. Similarly, garbage collection in the executable representation affects the return address of all active procedures that are being moved. Since return addresses are located on the activation stack, they are retrievable and can be updated.

Debugging actions that change the program representation are implemented using the incremental program modification mechanism. Since the debugger is an interactive dynamic facility, it is required that execution can be resumed after these kinds of changes. Program changes due to debugging actions do not affect the logical structure of the program, *i.e.*, the control or data flow although they may temporarily interrupt the control flow. The debugging actions, in general, examine and check the execution state (conditional debug statements, assertions, etc), display information (trace), and suspend execution (break). Thus, they do not invalidate the execution state.

Activation points in the different active procedures, *i.e.* return locations and current program counter, may be shifted relative to the beginning of the replaced procedure. Provided with the appropriate information in the program tree, the code generator can then determine the new location of an affected activation point when generating the mapping information.

User changes to the program are much harder to deal with. Consider the case where a statement (*e.g.* an assignment) is added to an active procedure in a place that has been executed. What are the effects on the execution state, because that statement has not been executed? For situations like this we take the approach that IPE works in cooperation with the programmer. Any potential problems are indicated, but the programmer makes the decision whether the execution state is consistent.

Another difficulty occurs when the change to the program affects the dynamic control flow. Procedure A calls procedure B. Both are on the activation stack and execution is suspended in B. The

change is to remove the call to B in A. What is the execution state from which execution can be safely resumed? Resumption from within B does not make sense. A possible solution, other than starting execution from scratch, is to restart execution at a safe point, (*e.g.* the call to the procedure with the change). This requires resetting the current execution state to the execution state at the time of the call. Depending on the definition of the execution state, this may be a hard problem. It is related to work done on recovery mechanisms (e.g. recovery blocks [Randell 75], stable variables [Liskov 80]). In IPE we take a conservative approach. Initially, there is no facility to unwind program execution other than starting it over. However, the programmer may explicitly specify and enable checkpoints to which the execution state can be restored. The cost for checkpoints is high, and the programmer should anticipate them.

### 3.3.2 Debugging

Debugging actions in IPE are implemented through editor functions and editor commands. This approach has several advantages. The debugger works on the tree representation, *i.e.* in terms of the source language. Conditional trace and breakpoints are actually recorded in the tree and are visible to the user. Their implementation in the machine representation is done by the code generator. Therefore the debugger has very limited knowledge of the machine representation. The evaluation of expressions in context is implemented in a similar fashion, by temporarily adding a subtree for the expression and generating code for it. Potentially the display of variable contents can be implemented as an expression evaluation with a printout of the result. This implementation, however, may be slow. Therefore, a special variable display function is implemented as an editor command in a more efficient manner.

The approach taken to implement the debugging actions also has some disadvantages. First of all the responsiveness of the debugger is dependent on the response time of the incremental program modification mechanism. Therefore that mechanism is the critical path of the whole IPE system. Some functions are more difficult to implement. One such function is single stepping, a mechanism found in many traditional systems. It requires a substantial amount of overhead to be performed correctly. As an alternative, however, IPE provides more sophisticated debugging actions, that are more tailored towards the actual debugging goal.

# 4. Conclusions

This paper presents IPE, an incremental programming environment, being designed and implemented by the authors at Carnegie-Mellon University. This system gives programming support to a single programmer working on a single program. Even though it is presented as a system in itself, IPE is part of Gandalf, a more general software development environment project. IPE will be extended to incorporate support for description of larger systems, multiple versions of programs, and management of several programmers on the same system.

We consider the major impact of IPE to be its integrated approach. The programming environment is now viewed as a single system rather than as a set of independent tools. There is a smooth transition between the tools, and some tools are not directly visible to the programmer, because they are automatically applied.

At all levels of the program development the programmer deals with language constructs. The programmer does not have to worry about the syntax constraints of the programming language anymore, nor has he to be aware of the different components of the environment. By making it easier for the programmer to construct and debug programs, it is very likely that the quality of his programs will be improved and the programming time will be reduced.

IPE is based solely on compilation technology. No software interpreter is provided. Compilation has the advantage that a high level program representation is transformed into the machine representation of a computer, not necessarily the same as the host machine. Therefore, IPE may be able to support several target machines with most of the IPE system residing on one host machine.

A syntax-directed constructive editor allows the programmer to build and change programs in terms of language constructs. The editor automatically provides keywords, separators, and terminators. This reduces the amount of typing and eliminates programming errors like missing semicolons. The editor uses its knowledge about the programming language and permits only syntactically correct programs to be constructed. Despite the fact that the editor has knowledge of the language, it is language independent in the sense that the editor is automatically generated from an editor kernel and a formal description of the language. The program is internally represented by a syntax tree, from which the text form is dynamically generated for display to the programmer. This syntax tree is the common program representation for all tools in IPE. Semantic checking, translation, and debugging work on the syntax tree.

The translation phase uses the fact that code is produced only for semantically correct program pieces to merge the linking step into the code generation. As the programmer is incrementally changing the tree representation of the program, the IPE system incrementally updates and maintains

an executable version by automatically applying the translation phase to program pieces and incorporating them on the target machine. The debugging facility of IPE also works on the syntax tree. It is implemented using the incremental program modification mechanism, *i.e.* incremental update, translate and load, and therefore, does not perform software interpretation. The code generator provides information for a mapping from the tree representation to the machine representation. Some optimization can be applied during code generation because only certain points in the tree require an exact mapping, even for debugging. Since the syntax tree is the primary program representation for the debugger, more sophisticated debugging actions in terms of the underlying programming language can be provided.

IPE is being implemented under the UNIX[tm] operating system running on a DEC VAX[tm]. At the time of this writing the state of implementation is the following. The editor generator has been implemented and an editor that supports a substantial subset of GC has been generated. Some of the editor commands are not available yet, *e.g.* pattern search. Semantic checking is performed as part of the translation step. Code is being generated for the supported subset of GC. The implementation of the incremental loader and a minimal debugging facility are in progress. To implement IPE we have used an incremental system that already incorporates some of the program development philosophy supported by IPE [Feiler 79b].

## References


[DoD 80] Department of Defense.
Requirements For Ada Programming Support Environments. "Stoneman" .
February 1980.

[Feiler 79a] Feiler, Peter H. and Medina-Mora, Raul.
The GC language.
Gandalf internal documentation.
1979.

[Feiler 79b] Feiler, Peter H.
IPC System Version 1.
Gandalf internal documentation.
1979.

[Habermann 79] Habermann, A. Nico.
A Software Development Control System.
CMU Computer Science Department.
1979.

[Habermann 80a] Habermann, A. Nico.
An Overview of the Gandalf Project.
CMU Computer Science Research Review 1978-79.
1980.

[Habermann 80b] *Habermann, A. Nico, and Perry, Dewayne E.*
Well-formed System Compositions.
CMU Computer Science Department.
1980.

[Habermann 80c] Habermann, A. Nico.
Notes on Programatics and its language Alfa. (Private Communication).
1980.

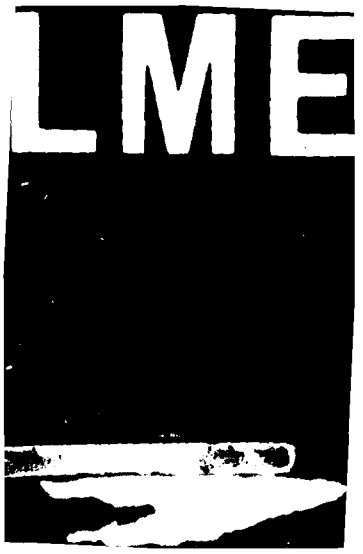
[Lampson 77] Lampson. B.W., Horning, J.J., London. R.L., Mitchell, J.G., and Popek, G.L.
Report on the Programming Language Euclid.
*SIGPLAN Notices* 12(2), January, 1977.

[Liskov 80] Liskov, Barbara.
Primitives for Distributed Computing.
Distinguished Lecture Series at CMU.
1980.

[Notkin 79] Notkin, David S., and Habermann, A. Nico.
Software Development Environment Issues As Related To Ada.
CMU Computer Science Department.
1979.

[Randell 75] Randell, Brian.
System Structure for Fault Tolerance.
*SIGPLAN Notices* 10(6), June, 1975.

[Schatz 79] Schatz, B., Leverett, B., Newcomer, J., Reiner, A., and Wulf, W.
*$TCOL_{Ada}$: An Intermediate Representation for the DoD Standard Programming Language.*
Technical Report, Carnegie-Mellon University, 1979.


LME